Overload Protection of Cloud-IoT Applications by Feedback Control of Smart Devices

Manuel Gotin
Robert Bosch GmbH
Renningen, Germany
manuel.gotin@de.bosch.com

Dominik Werle
Karlsruhe Institute of Technology
Karlsruhe, Germany
dominik.werle@kit.edu

Felix Lösch
Robert Bosch GmbH
Renningen, Germany
felix.loesch@de.bosch.com

Anne Koziolek
Karlsruhe Institute of Technology
Karlsruhe, Germany
anne.koziolek@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
ralf.reussner@kit.edu

ABSTRACT

One of the most common usage scenarios for Cloud-IoT applications is Sensing-as-a-Service, which focuses on the processing of sensor data in order to make it available for other applications. Auto-scaling is a popular runtime management technique for cloud applications to cope with a varying resource demand by provisioning resources in an autonomous manner. However, if an auto-scaling system cannot provide the required resources, e.g., due to cost constraints, the cloud application is overloaded, which impacts its performance and availability. We present a feedback control mechanism to mitigate and recover from overload situations by adapting the send rate of smart devices in consideration of the current processing rate of the cloud application. This mechanism supports a coupling with the widely used threshold-based auto-scaling systems. In a case study, we demonstrate the capability of the approach to cope with overload scenarios in a realistic environment. Overall, we consider this approach as a novel tool for runtime managing cloud applications.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;

KEYWORDS

Cloud Computing, Internet of Things (IoT), Smart Devices, Feedback Control, Message Queues, Auto-Scaler, Performance

ACM Reference Format:
Manuel Gotin, Dominik Werle, Felix Lösch, Anne Koziolek, and Ralf Reussner. 2019. Overload Protection of Cloud-IoT Applications by Feedback Control of Smart Devices. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297663.3309673

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3309673

1 INTRODUCTION

The Cloud-IoT paradigm addresses the limitations of the Internet of Things (IoT) in terms of computation and storage capabilities by coupling it with Cloud Computing to enable a range of usage scenarios [3]. One of the most relevant ones in the areas of smart home, connected vehicles or Industry 4.0 is Sensing-as-a-Service (SensAAS), which aims to provide ubiquitous access to sensor data.

IoT platforms aim to connect the physical world with cloud solutions [8]. The basic responsibilities of an IoT platform are to offer an interface for devices to connect, receive and process their data and provide the data to connected applications. A common architectural design decision is to dispatch received data via message queues to connected applications.

Traditional runtime performance management leverages the capabilities of cloud computing by automatically scaling resources. These so called auto-scaling systems aim to provision resources in respect to the current resource demand as extensively surveyed in [11]. Typically, scaling decisions are performed horizontally by replicating resources.

However, auto-scaling systems may face limitations which render resource provisioning as prohibitive or inefficient. First, in order to avoid high operating costs the maximum number of provisionable resources is usually capped in practice. Second, dependencies to external services may impose a cap on the processing rate of the cloud application.

If the rate of received messages exceeds the rate of processed messages over a long period of time, critical infrastructure components like message queues eventually accumulate messages, resulting in a possibly long-lasting high delay for messages and may deplete the resources of the message broker. This may affect the Quality-of-Service (QoS) of Cloud-IoT Applications which aim to process recent data.

Modern message broker systems such as Pivotal RabbitMQ and Artemis ActiveMQ are able to cope with such situations by offering *backpressure* techniques like discarding messages or blocking producers based on time or resource criteria [1, 16]. However, by discarding expired messages data is lost and by blocking producers there is no feedback when they may send data again.

The contribution of this paper is a feedback control mechanism which considers the current processing rate of the cloud application in order to centrally adapt the send rate of devices. This allows

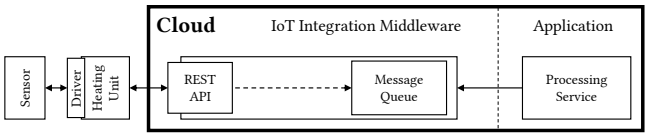

Figure 1: Architecture of the running example. Devices connect to the IoT integration middleware, which provides received sensor data to the application.

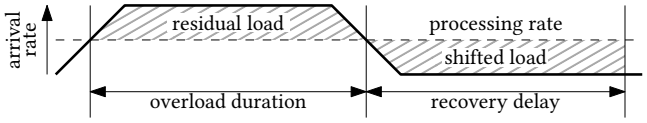

Figure 2: In an overload situation the rate of arriving messages exceeds the processing rate of the cloud application. This results in a *residual load* which has to be processed at a later point of time, inducing a *recovery delay*.

devices to cope with the increased send interval, e.g., by aggregating instead of discarding sensor data. Due to the highly dynamic environment we estimate the processing rate at runtime using message queue metrics in overload situations. We aim to stabilize the queue and the message processing delay by adapting the send rate until the overload situation has been resolved. In order to preserve the benefits of auto-scaling systems, we present a coupling technique for threshold-based auto-scalers. We support applications in which data can be aggregated and each message demands the same amount of resources. This results in a processing delay based on the degraded send rate but mitigates an uncontrollable message processing delay induced by a flooded queue.

We exploratorily show in a case study that the overload protection approach is able to cope with overload situations and that it can be successfully coupled with auto-scaling systems.

The remainder of this paper is organized as follows: Section 2 introduces a running example in the area of connected heating. Section 3 explores limitations of auto-scaling systems and derives challenges. We introduce our approach in section 4. In section 5, we present a case study for its evaluation. Section 6 gives an overview of related work. Section 7 discusses limitations and future work and section 8 concludes the results of the case study.

2 RUNNING EXAMPLE

Bosch offers many SensAAS services in the areas of smart home, connected vehicles or Industry 4.0. To quickly build and deploy such services, Bosch provides the Bosch IoT Cloud, a Platform-as-a-Service based on Cloud Foundry[1]. In this paper, the running example is based on a connected heating scenario, in which a varying number of connected heating control units periodically send sensor data. The architecture of the example is derived from a productively used system and closely resembles the architecture proposed in [4].

Components. Figure 1 shows the corresponding components in the running example. Every component is deployed on the cloud infrastructure except for the heating units.

The *Heating Unit* transfers temperature sensor data at a rate configured by the application developer. Each sensor data message contains the average temperature within the time interval determined by the configured rate.

The *IoT Integration Middleware* allows devices to connect, to receive the data from these devices, and to provide them to connected applications using a *Message Queue*. The *Processing Service* consumes messages out of this queue in order to process and finally persist them using an *External Database*. The External Database is shared across many applications and is not operated by the application developer.

[1]https://www.cloudfoundry.org/

3 CHALLENGES

Cloud applications are typically managed by auto-scaling systems at runtime. Essentially auto-scaling systems aim to cope with the current resource demand by (de-)provisioning resources. In practice, auto-scaling systems may face limitations in the following scenarios:

Cost-based Provisioning Constraints – In order to cap the maximum operating costs, auto-scaling systems are usually limited in terms of the maximal number of provisionable instances.

External Resource-based Constraints – Dependencies to external resources, e.g., a shared database system, can be a bottleneck in such a system. Scaling resources within the operational scope of a cloud application can be inefficient in such a case.

Figure 2 illustrates the issue associated with a limited processing rate. When the rate of incoming messages exceeds the processing rate, arriving messages cannot be processed immediately and have to be placed in a queue. We call this load *residual load*, which has to be processed at a later point of time, resulting in a *recovery delay*.

Based on these constraints we derive the following challenges in operating such an application:

Challenge 1 – Cope with overload situations in which the resource provisioning is saturated.

Challenge 2 – Cope with overload situations in which the provisioning decisions are inefficient.

Challenge 3 – Cope with a non-empty queue degrading the message processing delay.

4 APPROACH

The main objective of the approach is to mitigate overload situations of a cloud application deployed on a cloud infrastructure. In order to mitigate or resolve an overload situation, the send rate of each device is centrally adapted in respect to the current processing rate of the cloud application. This results in a lower message processing delay by reducing the *residual load* illustrated in figure 2.

It aims to support operating Cloud-IoT applications in sensing scenarios, which receive and process sensor data, which induces equal resource demand. A main requirement for this approach is a dedicated message queue which provides received data of devices to a connected application.

There are different factors that make an ideal send rate adaptation impossible in practice, e.g. monitoring and reconfiguration delays and performance uncertainties in the highly dynamic cloud

environment resulting in varying message processing rates. In the following we describe how our approach copes with those influences to systematically approach an appropriate arrival rate for the system.

4.1 Phases

The overload protection approach consists of three phases: an *idle phase*, an *overload protection phase* and an *overload recovery phase*.

The *idle phase* is the state in which the overload protection approach does not take any action. If the overload protection approach recognizes an overloaded situation of the cloud application, it transits to the *overload protection phase*. In this phase, the approach adapts the send rate of smart devices using the estimated processing rate of the cloud application. If the overload situation is resolved, the approach transits to the *overload recovery phase*. In this phase, the overload protection approach increases the send rate of the smart devices in order to smoothly recover the original send rate. If this results in an overload situation, it transits back to the *overload protection phase*. If the send rate is successfully restored, without inducing an overload situation, it transits to the *idle phase*.

4.2 Overload Situations

By providing received sensor data to the cloud application, the message queue is directly affected by an imbalance between the rate of received and processed messages. For this reason we use metrics of the message queue as indicators for overload situation. Modern message broker systems, such as RabbitMQ, support the monitoring of the followings metrics:

- Queue Length – Number of messages in the queue.
- Queueing Delay – Time the leading message has spent in the queue.
- Queue Arrival Rate – Number of messages entering the queue during a time unit.
- Queue Departure Rate – Number of messages leaving the queue during a time unit.

Typically, messages are processed in a First-Come-First-Serve (FCFS) manner. If the queue arrival rate exceeds the queue departure rate the queue eventually accumulates messages. Based on the FCFS policy, received messages must wait until preceding messages have been processed, which is reflected by the queueing delay. The QoS might be expressed by the queueing delay, subsequently a longer delay may lead to Service-Level-Agreement (SLA) violations. We consider a cloud application as overloaded, if the queue length or the queueing delay have surpassed a certain threshold, which is provided by the application developer. In the following we describe this threshold as T_M where M is the underlying metric. For example, $T_{QueueLength} = 1$ means that the cloud application is considered to be overloaded if the queue length is greater than 1.

4.3 Estimation of the Cloud Application's Processing Rate

We estimate the current processing rate of the cloud application to decide on a suitable send rate for the smart devices. This makes the approach more flexible, because we can accommodate cases in which we do not know the actual processing rate, e.g., when the rate is capped by external services. If the rate of incoming messages exceeds the processing rate of the cloud application, the message queue eventually gets filled with messages resulting in a non-zero length. In this case we assume that the cloud application processes messages at full capacity, which renders the queue departure rate as a proxy-metric for its processing rate. Additionally, we check, whether the estimated performance is smaller than the current queue departure rate, which indicates that the estimated processing rate is too small. Therefore, we refine the estimation in this case.

4.4 Calculation of the Adapted Send Rate of Devices

Based on the assumption that each sensor data demands the same amount of resources on the cloud application, we predict the currently supported send rate S' of each device at a time t using the cloud application's estimated processing rate $R_{CloudApplication}$ and the number of connected devices $N_{Devices}$ such that:

$$S'(t) := \frac{R_{CloudApplication}(t)}{N_{Devices}(t)}$$

We cap the send rate S' if it exceeds the default send rate $S_{default}$:

$$S'_{cap}(t) := \min\left(S'(t),\ S_{default}\right)$$

4.5 Send Rate Adaptation within the Phases

During the overload protection phase, the goal is to mitigate or resolve an overload situation. The calculated send rate of the devices is based on the processing rate of the cloud application. However, we reduce it further in this phase, in order to let a non-empty queue recover. This reduces the queueing delay, resulting in a lower message processing latency. For this reason we introduce a factor $k_{protect} < 1$ for reducing the send rate in the overload protection phase, therefore ensuring that the queue size decreases over time:

$$S'_{protect}(t) := S'_{cap}(t) \cdot k_{protect}$$

If an overload situation is resolved, we want to smoothly transit the adapted send rate back to the default send rate $S_{default}$. For this reason, we set the send rate $S'_{recover}$ during the recovery phase to $S'_{protect}$ at the beginning of the recovery phase. We then increase the send rate by a factor $k_{recover} > 1$ in iterations of length T:

$$S'_{recover}(t) := S'_{recover}(t - T) \cdot k_{recover}$$

This factor allows to handle an underestimation in the processing rate of the cloud application, since $S'_{recover}(t)$ converges to the actual processing rate of the cloud application.

4.6 Coupling with Auto-Scaling Systems

Cloud applications are usually operated with auto-scaling systems, which aim for provisioning resources to cope with the current resource demand. The overload protection approach adapts the send rate of devices and thus reduces the workload on the cloud application. Due to these complementary goals, we propose a coupling of both strategies in order to minimize interfering effects. When the overload protection approach recovers the queue and thus reduces both queue length and queueing delay, an auto-scaling system which uses these metrics may, for example, falsely assume an idle service and deprovision resources. To avoid such interference we

turn the auto-scaling system off during the overload protection and recovery phase and turn it on again, when the overload situation has been resolved. We propose two coupling techniques in order to decide when to activate the overload protection approach:

Overload-based. In this solution, a recognized overload situation is the sole trigger for activating the overload protection approach.

State-Aware. The auto-scaling system signals whether its scaling decisions are saturated to leverage its capabilities until it has reached its provisioning limit. That means that we extend the transition condition from the idle to the overload protection phase by additionally checking if the auto-scaling system has reached its provisioning limits.

4.7 Feedback Control Signaling

The adaptation of the devices should not induce additional load on the infrastructure or the cloud applications, because this could possibly nullify the advantages of the device adaptation. For this reason we propose to adapt the send rate of a device utilizing the acknowledgment of communication protocols like HTTP after sending sensor data by injecting the adapted send rate.

5 CASE STUDY FOR VALIDATION

The rationale of this case study is to investigate if the proposed approach is able to cope with the challenges defined in section 3. We study the running example in a productively used environment—the Bosch IoT Cloud—in order to include the influence of disturbance factors such as measurement errors, control delays and estimation errors of the processing rate. The structure of the case study is described in [18].

5.1 Research Questions

We refine the objectives of the case study into a set of research questions. The first research question refers to the qualities of the overload protection approach in a scenario without a simultaneously operated auto-scaling system.

RQ1 – To which degree are overload situations mitigated?

(i) How does the approach affect the message processing delay and the number of processed messages?
(ii) How is it influenced by the severity of the overload situation?

The second research question focuses on the coupling with auto-scaling systems:

RQ2 – To which degree are overload situations mitigated *in a coupled scenario.*

(i) How does the approach affect the message processing delay and the number of processed messages?
(ii) How is it influenced by the severity of the overload situation?
(iii) How is it influenced by the type of coupling?

5.2 Methodology

Experimental Setup. We implemented a lightweight IoT platform which provides a REST API for the smart devices. As illustrated in figure 3, we utilize this IoT Integration Middleware to retrieve the number of connected heating units and for adapting their send rate. We are able to retrieve message queue metrics by utilizing the capabilities of RabbitMQ. In this setup, we simulate the heating units as virtual entities, deployed as microservices on the Bosch IoT Cloud, which periodically send sensor data. The cloud application consists of a scalable microservice which has a configurable service time to simulate the calls to the external database. Furthermore it consumes messages out of the RabbitMQ message queue. All components are deployed on the Bosch IoT Cloud.

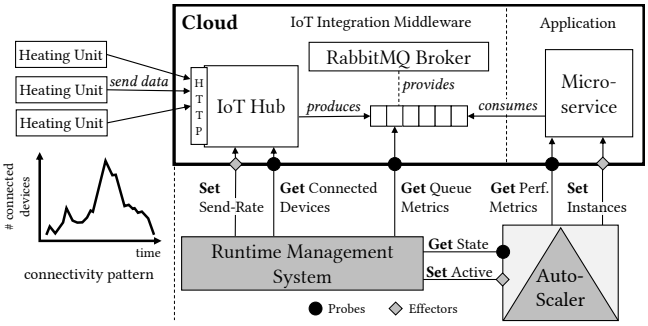

Figure 3: Experimental Setup. Devices, Middleware and Application are deployed on the Bosch IoT Cloud.

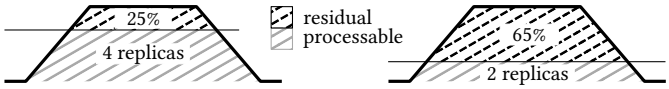

Figure 4: Example overload shares. For the same load, a provisioning of 4 replicas leads to an overload share of 25 %, 2 replicas lead to an overload share of 65 %.

Severity of overload situations. In this case study we consider overload situations which occurs due to provisioning caps. In order to quantify the severity of an overload situation we calculate the ratio of messages which can not be processed without queueing with maximum resource provisioning to the total number of produced messages. We call this ratio the *overload share.* Figure 4 illustrates this approach.

Evaluating the quality of the overload protection. One of the main goals of the overload protection approach is to reduce the message processing delay in overload situations. Additionally, it should stabilize the queue as a critical infrastructure component. For this reason we measure the average queueing delay, queue length and send interval during an experiment. Furthermore, we quantify the throughput in terms of the total number of processed messages during an experimental run, whereas each experiment run is observed for the same duration. Since our experimental setup is within a single cloud infrastructure, the message processing delay comprises the average queueing delay and the send interval but does not include the network latency as it is negligible. In order to derive the suitability of the overload protection we compare these values with values obtained in a similar setup without activated overload protection, which we declare as baseline. If not stated otherwise, the baseline describes a static and maximum provisioned system.

Workload. We introduce *connectivity patterns* to describe the changing number of connected devices over time. The workload on the system results from the number of connected devices and their send rate. Therefore the number of devices is given by the experimental setup whereas the send rate can be reconfigured during the experiment resulting in a dynamic workload. We select a pattern which we deem as challenging with some minor and a single major peak. It is illustrated in figure 3. Based on the concrete test setup it is stretched in terms of the number of devices and duration.

Auto-Scaling Setup. We use a threshold-based rules auto-scaling system, which is one of the most popular auto-scaling techniques in industry. We customize the auto-scaling system to offer an interface for (de-)activation and to provide the information, whether its provisioning decisions are saturated. In our setup we scale a single microservice which simulates external service calls by waiting for a configurable time. As shown in [7] message queue metrics are suitable in scaling I/O-intensive microservices and for this reason we rely on the queue length as the performance metric of the auto-scaling system. We configure the auto-scaling system for each experimental run as follows:

- **Performance Metric**: Queue Length
- **Upper Threshold**: 10
- **Lower Threshold**: 0
- **Minimal Instances**: 1
- **Maximal Instances**: `Experiment-based`

Based on our experience, this threshold combination achieves in a range of experimental setups a high degree of elasticity in terms of the measures proposed in [9].

Configuration of the Overload Protection Approach. We configure the parameters of the overload protection approach based on our experience. For each experimental setup we use the following configuration:

- **Protection Multiplicator**: $k_{protect} = 0.98$
- **Recovery Multiplicator**: $k_{recover} = 1.1$
- **Overload Recognition Metric**: Queue Length
- **Overload Recognition Threshold**: $T_{QueueLength} = 1$ if not stated otherwise

Experiment Runs. We set the duration of each experiment to $t_{experiment} = 5.5$ minutes. Based on our experience, the Bosch IoT Cloud is a highly reactive cloud infrastructure which performs scaling decisions within a few seconds. For this reason the duration is sufficient to perceive important effects. Furthermore we configure each microservice instance to process a maximum of $5.5\frac{\text{msg}}{\text{sec}}$ and set the default send rate of a simulated thing to be $S_{default} = 2\frac{\text{msg}}{\text{sec}}$.

5.3 Results

To answer both research questions we execute a set of experiments with varying overload severities for different runtime management scenarios.

The first research question is adressed by using the overload protection approach in a static provisioning scenario, which we declare as `Overload Protection (only)`. For the second research question we focus on two types of coupling: the first type requires the auto-scaling system to exchange the information if scaling decisions are saturated, thus we describe this coupling as `state-aware`. The second type simply checks the overload condition to activate the overload approach. In this setup we increase the threshold for the overload recognition to $T_{QueueLength} = 250$ in order to leverage the capabilities of the auto-scaling system before activating the overload protection mechanism.

Metric	Baseline	A.-S.	O.P.	Coup.	Coup. (S.-A.)
Processed Messages	6453.6	6405.9	5855.0	5738.0	5499.1
Avg. Queue Length	545.2	654.3	2.2	120.2	25.2
Avg. Q.-Delay [sec]	21.5	25.1	5.0	13.2	6.4
Max. Q.-Delay [sec]	66.6	79.8	6.9	43.8	8.4
Avg. Send-Inter. [sec]	0.5	0.5	0.63	0.61	0.65
Max. Send-Inter. [sec]	0.5	0.5	1.27	1.12	1.26
Provisioned Instances	6.5	4.0	6.5	3.6	3.78

A.-S. = Auto-Scaling, O.P. = Overload Protection, Coup. = Coupled, S.-A. = State-Aware

Table 1: Metrics for different strategies across all overload shares.

In summary we compare the following approaches, whereas maximum instances describes the maximum number of provisionable instances depending on the overload share of the concrete experimental setup:

- **Baseline**: No overload protection, static provisioning with maximum instances.
- **Overload Protection (only)**: Overload protection, static provisioning with maximum instances.
- **Auto-Scaling (only)**: Auto-Scaler only, initial 1 instance and provisioning capped at maximum instances.
- **Coupled**: Overload protection coupled with Auto-Scaler. Activation by overload threshold.
- **Coupled (State-Aware)**: Overload protection coupled with Auto-Scaler. Activation by saturated provisioning and overload threshold.

Figure 5 show the results for each approach, whereas table 1 summarize it.

In the following we discuss each result for each setup independently, to finally conclude it:

Baseline. By statically provisioning instances without adapting the send rate the baseline reaches the highest number of processed messages. However, since it has no mechanism to mitigate overload situations it also results in a considerably high average queueing delay of 23.0 secs.

Auto-Scaling (only). This setup aims to reduce the operating costs of the cloud application by provisioning resources only if required. For this reason it is expected to be overall similar to *Baseline*. However, due to delays in monitoring and reconfiguration the auto-scaling system may degrade the number of processed messages within the experimental duration and may increase the queueing delay. The results of both setups are similar, which demonstrates the quality of the auto-scaling system and the infrastructure. The degradation of the queueing delay is on average 15.6 % whereas the processed messages are reduced by 1.1 %. Furthermore it reduces

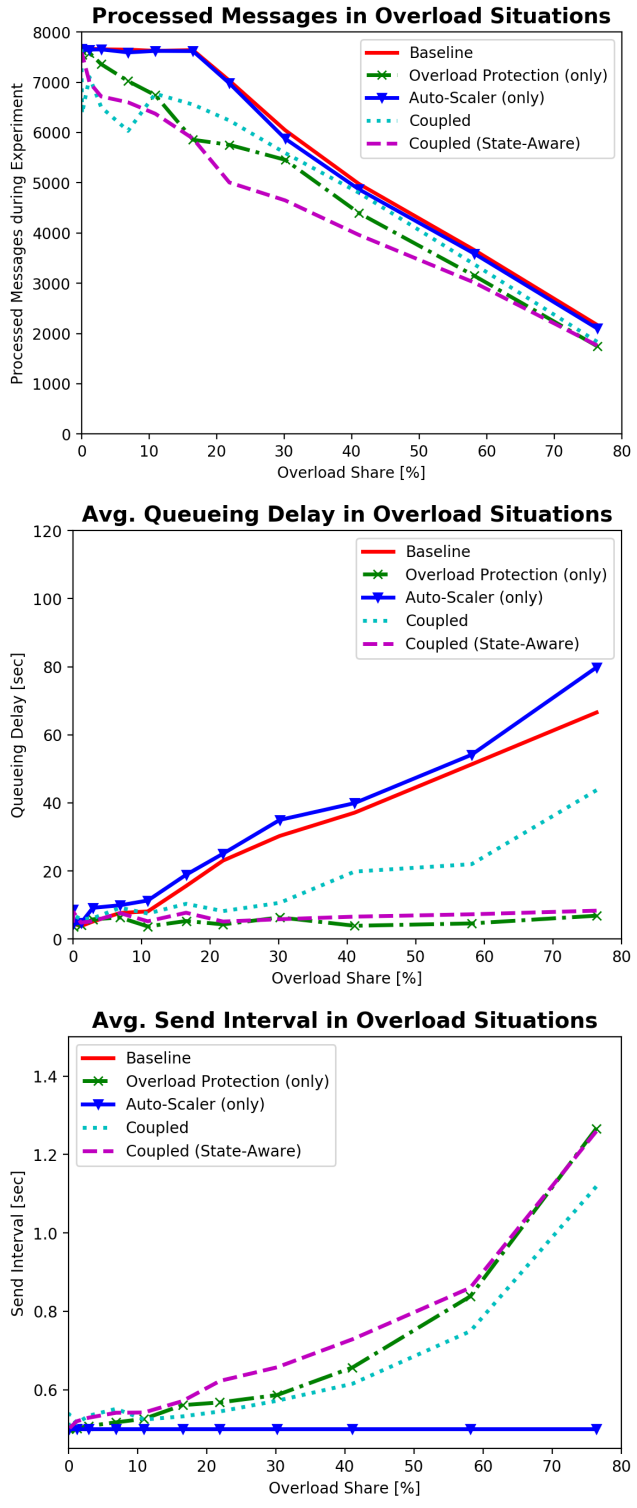

Figure 5: Average processed messages, queueing delay and send interval in different scenarios.

the provisioned instances by an average of 36.3 % resulting in lower operating costs.

Overload Protection (only). Relying on a static provisioning with the overload protection approach results in a consistently low average queueing delay of 5.0 secs with a reduction of processed messages of 11.1 %. Across all overload shares the maximum increase in the send interval is 0.76 secs and on average 0.14 secs, resulting in an overall low message processing delay.

Coupled. This scenario results in a reduced average number of processed messages by 11.9 % whereas the queueing delay is reduced by an average of 40.5 %. Across all overload shares the maximum increase in the send interval is 0.62 secs. The average queueing delay is with 13.7 secs between `Overload Protection (only)` and `Baseline`.

Coupled (State-Aware). Activating the overload protection after the auto-scaling system has reached its provisioning limits results in a consistently low average queueing delay of 6.53 secs with a reduction of processed messages of 16.7 %. The send interval is increased by a maximum of 0.75 secs and on average by 0.17 secs. Furthermore by utilizing an auto-scaling system it reduces the operating costs by reducing the provisioned instances by a total of 38.7 %, which is below the average instances of `Auto-Scaling (only)` by reducing the residual load.

Conclusion. The case study has shown, that the approach performs well in an isolated and coupled setup in preventing the message infrastucture from getting overloaded. We conclude the first research question by emphasizing the strong reduction of the queueing delay. In contrast, the reduction of the send rate is on average 18 %, resulting in a comparatively low message processing delay. Using a state-aware coupled setup offers a high degree of overload protection for all overload shares and reduces the costs compared to a static provisioned setup by efficiently utilizing an auto-scaling system.

5.4 Discussion

The presented case study has shown that the proposed overload protection is able to cope with the challenges in an isolated and coupled setup, consistently reaching a low message processing delay, expressed by a low and stabilized queueing delay with a comparatively low increment of the send interval, below 1 second on average. We have shown, that the approach results in a throughput reduction, since the residual load, as illustrated in figure 2, is reduced by increasing the send interval, which results in a lower number of messages to be processed. In order to quantify the costs of the throughput and send rate reduction the application's needs should be considered.

We have shown that a common type of auto-scaling system—threshold-based auto-scalers—can be combined with the overload protection approach. This is especially the case for situations where the auto-scaler discloses information about whether its provisioning is saturated or not, and to a lesser degree where the overload protection approach considers the overload recognition threshold T_{metric} only. Especially the state-aware coupling peforms nearly equally

well as a static provisioning with maximum instances, demonstrating the capability of this approach to successfully unify both runtime strategies with their complementary goals.

Altogether, we consider the presented approach as a novel tool for the application developer to manage the performance, costs and availability of the system in situations where a variable load on the system is expected. However, the application developer has to design a system that is able to cope with the send rate reduction, e.g. by letting the smart devices reconfigure their data processing behavior. Overall, this might affect the validity of the data, which we assume to be strongly dependent on the application's requirements. If an overload situation occur some applications may require current data, even if it is pre-processed, whereas other application require a high data resolution, even if the data is delayed. In case of economical provisioning limitations, the application developer should be aware, that the degradation of the QoS caused by a send rate reduction may be more expensive than raising these limits.

5.5 Threats to Validity

In this section, we discuss the validity of the results of our case study according to the guidelines given by Runeson and Höst [18].

Internal validity. In our case study we observed the influence of a send rate adaptation on performance qualities of the cloud application. Across many experimental setups—both with and without coupling with an auto-scaling system—we experienced a similar effect on the overall system: the send rate adaptation reduces the load on the system which we derive mainly by observing the message queue. Since we operate the message queue in an isolated setup we exclude the possibility of an influence by an unknown factor.

External validity. The case study is built according to a typical architecture for cloud solutions in the area of SensAAS. We selected a connectivity pattern that allow us to observe the approach in minor and major overload situations. We deem the complexity of the pattern as sufficient to reveal the influence of disturbance factors on the quality of the adaptation. Furthermore we evaluated the coupling with a commonly used auto-scaling system in this case study. Since the smart devices in our case study are virtual entities, deployed as microservices on the cloud infrastructure, the network latency is expected to be much less than in a productive system. Because a higher network latency results in a delay of the adaptation, this may lead to a reduction of the responsiveness of our approach. In conclusion, based on this setup we expect a generalizability of the findings for similar cloud solution which receive and process data from smart devices.

Construct validity. By focusing on message queue metrics to derive the quality of the overload protection we measure a critical infrastructure component which is affected by overload situations. Furthermore, by relying on metrics provided by the message broker technology we observe metrics which directly reflect the performance characteristics of the queue.

Reliability validity. By describing the experimental setup in detail we expect the results to be similar when replicated. However, we consider the cloud infrastructure to not be easily replicable which may affect the quality of auto-scaling and the network latencies between the deployed components. Therefore, we conclude a qualitative reproducibility of the findings in this case study.

6 RELATED WORK

Current runtime management is focused mainly on auto-scaling systems, which are extensively surveyed in [11] and [17]. For this reason we focus on feedback control mechanism in Cloud-IoT scenarios. Furthermore, we discuss how our approach relates to control-theory-based methods and to other types of resource management in Cloud-IoT scenarios, namely edge and fog computing.

A feedback control mechanism is proposed in [15] to provide energy-efficient and network-friendly field sensing, by detecting conditions and accordingly controlling the sensing frequency. While it reduces the sensing rate it does not consider the state of the edge or cloud server which renders it unsuitable for overload protection.

In [14] a technique is presented to extend the operating life of power-constrained devices by an adaptive message rate. By using a message cache lengthy loss in sensor data is avoided. This approach can also cope with network outages by utilizing the cache in order to re-dispatch it on a restored network.

In [10], the authors present a congestion control method based on an improved Random Early Discard (RED) algorithm. This approach depends on probabilities to drop packets whereas we utilize the current applications processing rate to reduce the arrival rate.

An overview over various IoT congestion control algorithms used at transport layer is presented in [13] with a special focus on TCP. It concludes with the need for novel transport layer protocol, which is more in line with the requirements of devices. We deem our approach as a type of congestion control at application layer.

In [12], Maggio et al. give an overview of the space of decision-making strategies for self-adaptive systems. Particularly, control-theoretical self-optimization of systems has been proposed to manage the behavior and resources of a system under changes in the system's environment. For the presented approach, a formalization and analysis could give insight into the antagonizing mechanisms at play, namely the feedback control and the auto-scaling. While building these models is still tedious, there exists research on automating their extraction and application [5, 6]. We plan to look further into applying formal methods to gain insight into our system and how transient effects which we observed in our experiments, such as adaptation delays, impact their behavior.

Edge or fog computing [2, 19] is the concept of computing closer to the data source at the network edge instead of the centralized cloud, thus reducing processing latencies and required network bandwidth. This is also used to reduce the load on cloud solutions, like the presented approach, for example by aggregating or by preprocessing the data in a suitable way. However, we are not aware of a systematic method to change the send rate of devices dynamically at runtime that incorporates with existing edge or fog computing approaches. We propose to incorporate the method presented in this paper to utilize an additional degree of freedom in the way the load on the system is managed in a fog or edge computing scenario.

7 FUTURE WORK

In the current state, the approach aims to mitigate and recover from overload situations based on limitations of the processing rate of the cloud application occurring during run-time.

However, in practice provisioning limits are configured based on economic considerations. For this reason we see in future the inclusion of cost functions for provisioning and send rate reduction as an enabler for a more sophisticated coupling of both. This allows an application developer to balance the costs induced by a send rate reduction with the costs caused by the provisioning. The balancing can either take place at design- or run-time.

In the current approach we adapt the send rate equally for all devices. However, some applications may have smart things with different sensor data priorities, which we deem as a sensible extension for this approach.

Our approach assumes that data that arrives with a reduced send rate entails the same resource demand per received data packet. Therefore, the reduction in resource demand is reduced approximately with the same factor as the send rate. This may not be a valid assumption for all cloud solutions. To consider this, the approach needs a more sophisticated model of the resource demands in order to adapt the send rate accordingly.

One of the main limitations of the current approach lies in the adaptation of smart devices which send sensor data in an event-based manner. In such scenarios the reconfiguration of the smart devices is much more complex and should be integrated, for example, by configuring the device's threshold that has to be passed to send out data. This results in a more complex relationship between a send rate adjustment and its effect on the workload.

8 CONCLUSION

In this paper we proposed a feedback control mechanism to mitigate and recover from overload situations by adapting the send rate of smart devices. We have shown by the example of a case study that the overload protection approach is able to cope with overload situations, resulting in a low message processing delay, but a reduced number of processed messages. This is applicable both when coupled with auto-scalers as well as in isolation for statically provisioned systems. Furthermore it is able to cope with monitoring and reconfiguration delays and estimation errors in the processing rate of the cloud application. By estimating the processing rate of the cloud application and refining it in overload situations, the approach is able to dynamically adapt to changes in the environment. The presented approach supports application developers in runtime managing cloud applications in terms of reliability, costs and performance. We consider the overload protection approach as an additional tool in operating cloud solutions. In future work, we aim to extend the approach to allow cost optimization in terms of operating costs of provisioned resources and QoS degradation costs caused by adapting the send rate.

ACKNOWLEDGEMENTS

This work was partially supported by the German Research Foundation (DFG) as part of the Research Training Group GRK 2153: *Energy Status Data – Informatics Methods for its Collection, Analysis and Exploitation.*

REFERENCES

[1] Activemq.apache.org. 2018. Flow Control · ActiveMQ Artemis Documentation. *ThoughtWorks. https://activemq.apache.org/artemis/docs/latest/flow-control.html [Accessed September 17, 2018]* (2018).

[2] Flavio Bonomi, Rodolfo A. Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*, Mario Gerla and Dijiang Huang (Eds.). ACM, 13–16. https://doi.org/10.1145/2342509.2342513

[3] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. 2016. Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems* 56 (2016), 684–700.

[4] Cyril Cecchinel, Matthieu Jimenez, Sebastien Mosser, and Michel Riveill. 2014. An architecture to support the collection of big data in the internet of things. In *Services (SERVICES), 2014 IEEE World Congress on*. IEEE, 442–449.

[5] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2014. Automated design of self-adaptive software with control-theoretical formal guarantees. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 299–310. https://doi.org/10.1145/2568225.2568272

[6] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas B. Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir Molzam Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. 2015. Software Engineering Meets Control Theory. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, Paola Inverardi and Bradley R. Schmerl (Eds.). IEEE Computer Society, 71–82. https://doi.org/10.1109/SEAMS.2015.12

[7] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. 2018. Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 157–167.

[8] Jasmin Guth, Uwe Breitenbücher, Michael Falkenthal, Paul Fremantle, Oliver Kopp, Frank Leymann, and Lukas Reinfurt. 2018. A Detailed Analysis of IoT Platform Architectures: Concepts, Similarities, and Differences. In *Internet of Everything*. Springer, 81–101.

[9] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. 2016. Ready for rain? a view from spec research on the future of cloud metrics. *arXiv preprint arXiv:1604.03470* (2016).

[10] Jun Huang, Donggai Du, Qiang Duan, Yi Sun, Ying Yin, Tiantian Zhou, and Yanguang Zhang. 2014. Modeling and analysis on congestion control in the Internet of Things. In *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 434–439.

[11] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing* 12, 4 (2014), 559–592.

[12] Martina Maggio, Henry Hoffmann, Alessandro Vittorio Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. 2012. Comparison of Decision-Making Strategies for Self-Optimization in Autonomic Computing Systems. *TAAS* 7, 4 (2012), 36:1–36:32. https://doi.org/10.1145/2382570.2382572

[13] Neelesh Mishra, Lal Pratap Verma, Prabhat Kumar Srivastava, and Ajay Gupta. 2018. An Analysis of IoT Congestion Control Policies. *Procedia Computer Science* 132 (2018), 444–450.

[14] Keith E Nolan, Mark Y Kelly, Michael Nolan, John Brady, and Wael Guibene. 2016. Techniques for resilient real-world IoT. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2016 International*. IEEE, 222–226.

[15] Keigo Ogawa, Kenji Kanai, Masaru Takeuchi, Jiro Katto, and Toshitaka Tsuda. 2018. Edge-centric field monitoring system for energy-efficient and network-friendly field sensing. In *Consumer Communications & Networking Conference (CCNC), 2018 15th IEEE Annual*. IEEE, 1–6.

[16] Pivotal. 2018. Time-To-Live Extensions - RabbitMQ. *Pivotal. https://www.rabbitmq.com/ttl.html [Accessed September 17, 2018]* (2018).

[17] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. 2018. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 73.

[18] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131.

[19] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198