# Using Virtual Events for Edge-based Data Stream Reduction in Distributed Publish/Subscribe Systems

Philipp Zehnder, Patrick Wiener, Dominik Riemer
*FZI Research Center for Information Technology*
*Karlsruhe, Germany*
*Email: {zehnder, wiener, riemer}@fzi.de*

***Abstract*—Distributed publish/subscribe systems are an enabling technology for Industrial Internet of Things applications. While the number of sensors increases, network bandwidth becomes a bottleneck. Existing solutions typically aim to reduce network load either by pre-processing events directly on the edge or by aggregating events into larger batches. However, these approaches are rather static and do not adequately account for the application requirements of subscribers or the actual values of sensor measurements. This paper introduces methods for publish/subscribe systems to dynamically adapt payloads of events at runtime based on i) different data reduction and transformation strategies, ii) a wrapper solution around existing message brokers and iii) a semantics-based event schema registry. Consumers are able to subscribe to various quality levels and receive virtual events, that are reconstructed directly at the subscriber based on knowledge from the semantic model and dynamic decision rules. Our evaluation shows that the concept of virtual events can reduce the network load between publishers, the message broker and subscribers compared to multiple investigated compression techniques.**

***Keywords*-Publish/Subscribe Systems, Edge Processing, Semantic Web**

## I. Introduction

Publish/subscribe systems [1], [2] have become an established architectural pattern for loosely coupled distributed systems. In emerging application domains such as the Industrial Internet of Things (IIoT), settings in which thousands of sensors observe physical devices, are expected to be reality within just a few years. Such sensors produce information for machines, conveyors or environmental conditions in real-time, spanned over hundreds of production facilities even for a single company. Common approaches to integrate real-time data from heterogeneous sources based on the publish/subscribe pattern usually assume that events are sent from the originating device to a central middleware, where events are further processed and/or stored in order to make data accessible for advanced analytics applications. A steadily upcoming problem in such scenarios is the amount of required bandwidth and a high overhead required for synchronization of distributed messaging systems of such large-scale sensor networks [3]. In order to reduce bandwidth, *edge computing* [4], in which pre-processing algorithms are performed on incoming data streams close to the event source has evolved as a new architectural paradigm.

However, existing edge computing approaches are more concerned with moving analytical operations (such as pattern detection or anomaly detection) to the edge, making it hard to gain insights from raw data that might be required for yet unknown future application scenarios. In contrast, this work aims at another promising research direction, i.e., to rewrite events by reducing information (such as removing parts of the payload) directly at the event source and reconstructing events at the central messaging middleware, or even later at the subscriber, based on previously modeled background knowledge. A major advantage of such an approach is that *adaptive producers* can be used that are aware of low-interest situations (e.g., the base power consumption of an inactive machine) that can be reconstructed in the broker by using predictive models. We assume that the trade-off between very loosely coupled systems and a central coordination mechanism that also controls the behaviour of publishers based on a subscriber's demands can be a useful extension to future publish/subscribe systems. While early work on event reduction techniques in wireless sensor networks (WSN) has already been proposed [5], [6], a common framework for publish/subscribe systems, that leverages such techniques and automatically selects a suitable one is still missing. The selection of techniques should be based on the event payload's semantics and run-time monitoring information.

In this paper, we propose a methodological framework to optimize the network load of distributed publish/subscribe systems. First a motivating scenario from the smart building domain is illustrated in Section II. Then a framework to categorize techniques for runtime-based data reduction strategies (Section III) is presented. The data model (Section IV) contains *virtual events*, i.e., events reconstructed by subscribers without transmitting them. Based on those virtual events, we introduce a broker architecture in Section VI for semantic and content-aware processing, realized as a wrapper around existing broker technologies. Our evaluation in Section VII uses real-world data and shows that we can significantly reduce the amount of data sent over the network by reducing it directly at the edge.

978-1-7281-2365-3/19/$31.00 

## II. Motivation

To motivate our research, we present a use case from the smart building domain, where we show our approach with a Solar Thermal Climate System (STCS)[7] which uses solar heat to cool the building in summer and heat it in the winter. All parts of the system, for example the water tanks, the adsorption cooling machine, solar collectors, and also the rooms are equipped with sensors. The main purpose of the collected sensor data is to monitor and control the system's state in an optimal way. Edge devices are located throughout the building near sensors, producing data streams that are sent to a central message broker every second. Each sensor sends its state to ensure the system can act fast on changes. Based on this data, multiple algorithms can be performed. Indeed, for some use cases it might be sufficient to get events with a lower frequency, while others need an event every second.

Publish/subscribe systems, which often represent the backbone of such systems, realize a complete decoupling of data publishers and subscribers. Although such systems often try to transmit data as efficient as possible (e.g. with lightweight protocols such as MQTT), they are not aware of the semantic meaning of data and specific requirements of consumers. For instance, it would not be automatically detected when a machine is inactive, e.g, it is currently in idle mode. The sensors of this machine cannot measure anything and, therefore, no data needs to be transmitted. On the other hand, subscribers often do not need all available data, such as the raw data stream, for example, if a prediction based on aggregated values is performed. When the semantics of data and the requirements of subscribers would be better understood in a machine-processable way, the system could perform decisions, e.g., whether data is relevant and should be transmitted, on its own.

Therefore, the main goal of our work is to dynamically adapt data streams according to requirements of subscribers, resulting in a reduced consumption of network bandwidth, which makes it especially suitable for edge processing systems. In such edge processing environments, processing units are located close to the data producer, but often with limited computational capabilities. They can be used to perform some rather simple pre-processing with a low proximity to the data source. This further enables privacy-preserving techniques, meaning data which is confident and should not leave the local infrastructure can be filtered out.

Since publishers must transform data streams, it changes the characteristics of the stream, which might affect the implementation of the algorithms subscribing to the streams. For example when the frequency of the data stream can be reduced some algorithms must adapt to that change. To avoid this, all reductions should be provided by the system without subscribers realizing them. Therefore, we introduce the term *virtual event*, explained later in more detail. A virtual event looks like the event sent by the publisher, but might be generated directly at the subscriber without actually being transmitted over the network.

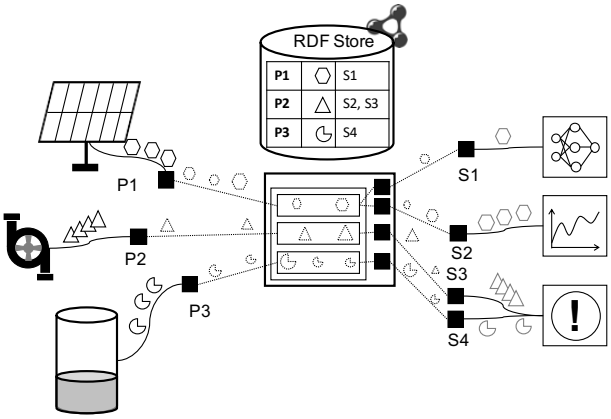


Figure 1. Dynamic reduction in publish-subscribe systems

Figure 1 illustrates the overall idea of our work. Centrally located is our message broker, which is responsible for transmitting events from producers to consumers. This broker also interacts with a semantic schema registry (based on a RDF triple store), describing the data on a specific topic and keeping a list of all subscribers of a specific topic and their individual requirements. Publishers (e.g. P1 - P3) are located with a close proximity to the data sources and transform data streams according to the requirements of the subscribers (e.g. S1 - S4). Subscribers take the information they get from the broker and reconstruct the virtual event, which can then be used by the algorithms, as depicted in the right part of the Figure.

## III. Event Stream Reduction Strategies

In many IoT-based data streaming scenarios, a major challenge is to handle both volume and frequency of events. A high number of heterogeneous event sources continuously produce events that must be transmitted, processed and stored. To be able to handle those large amounts of events, a common solution is to reduce the size. Multiple techniques are available to do this, ranging from simple compression techniques to more complex ones that also leverage from the semantics of the events and the context they are produced in. However, it is not sufficient to naively reduce data, because assessing the relevance of an event or part of its payload highly depends on the use case and the application domain. In this section, we show different kinds of data reduction strategies focusing on streaming data. An overview of the individual strategies is given in Figure 2.

In general, either the *event size* or the *frequency* of events can be reduced in a streaming scenario. For both reduction categories, different techniques exist, which are explained in more detail in this section. As events in data streams often contain redundant information that must not be transmitted,

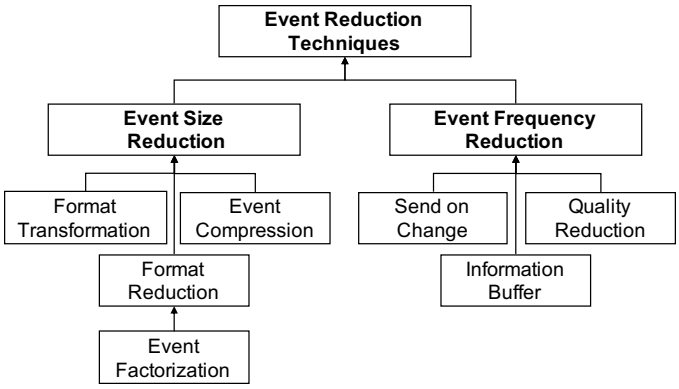


Figure 2. Hierarchy of reduction strategies

the following techniques to reduce the **event size** exist to filter out information directly at the event producer:

- **Format Transformation** Events can be serialized into different formats. The payload of the events remains the same as the representation of an event differs. Some format serializations result in larger event sizes than others. For example, XML usually comes with more boilerplate code than JSON. Further reductions can be achieved by using binary message formats to transmit events. A drawback of binary formats is that the content of those events is not human readable.
- **Format Reduction** In *Format Reduction*, only a subset of the raw event's payload is transmitted in order to reduce the event size. For this reduction technique, a sender must be able to decide which information is important and must be sent and which information is unimportant or can be reconstructed at the broker or receiver side. One example could be that individual parts of events are only transmitted once they change, leading to a reduction of redundant information in a series of events.
- **Event Factorization** A special case of *Format Reduction* is *Event Factorization*. In this case, data is factored out of the event and stored in a central instance, where receivers can access information and reconstruct the event. One example for this technique could be a central storage for static information like the location of buildings or some descriptions that will not change. A unique key can be created for such kind of information and stored in the central storage. Instead of transmitting all information, only the unique key must be sent.
- **Compression** Another frequently used technique is to compress the event before transmitting. Multiple compression techniques exist for that purpose [8]. The compression algorithm creates a binary representation of the event which must first be decompressed before it can be used.

Besides reducing the size of individual events, it is also possible to reduce the frequency in which events are sent. We distinguish the following reduction techniques suitable for **frequency reduction**:

- **Send on Change** The event frequency can be reduced by not sending events at a fixed sampling rate (often provided by sensors in the IIoT domain), but rather only to send values once they change. This is especially efficient in scenarios where changes are rare, but it is still important to be notified once something changes. If complex event structures are used (e.g., nested properties), it must be decided which part of an event should be sent when only the value of a single property has changed. The next technique, *information buffer*, helps with that decision.
- **Information Buffer** An information Buffer is a buffer that is located at the receiver. The buffer stores all the information required to reconstruct an event. In such scenarios, event producers only send changes in the event payload as in the technique before, while the receivers are able to reconstruct the complete event. In both described techniques, the frequency is reduced in scenarios where values do not change frequently.
- **Quality Reduction** The third frequency reduction technique is to reduce the quality of the information. In some scenarios, receivers do not necessarily require data in a high resolution. It is sufficient to reduce the accuracy of information. In such use cases, it might be sufficient to not send all events, instead only send events with larger time intervals in between to reduce the frequency.

Later in this paper, we investigate and apply various of the explained techniques on data streams in more detail. Therefore, we combine the individual techniques and use a semantic-based model of the data. In addition to the previously shown techniques, which are rather generic, it often depends on the application domain how data can be reduced while still ensuring the correctness of the algorithm. One example is presented in [9] where the authors compress geo-related data. Since we use a semantic model that contains detailed semantic information about the events, our system is capable of also applying domain-specific reduction techniques.

## IV. Data Model

The model distinguishes between events containing actual observations and the metadata describing the content and context of the events.

### A. Virtual Events

All actual observations are modeled in the form of events. They are emitted by event sources, resulting in a continuous data stream. Events have, as described in [10], *header* and *measurement* properties. We further introduce *dimension* properties containing unique identifiers about assets in one

data stream, like the sensor_id and room in Figure 3. This allows to group data streams accordingly.

| | |
|---|---|
| Header | **timestamp:** "2018-01-23T11:20:17.0" |
| Dimension | **sensor_id:** "TSOS"<br>**room:** "room_11.1" |
| Measurement | **temperature:** 10.1<br>**humidity:** 0.5 |

Figure 3. Example event of temperature and humidity sensor

Figure 3 shows an example event containing measurements of a temperature and humidity sensor. As a header property it contains the timestamp. Measurement properties contain the actual observations that change over time. Two consecutive events of a data source often contain similar or even the same information. This can be leveraged by not sending redundant information over the network and reconstructing the event at the subscriber. We call such events *virtual events* because they are reconstructed based on a combination of previously and partially transmitted information. Events containing just partial updates are called *partial events*. They are the basis for virtual events because they contain all the necessary information to ensure the virtual events are as similar to the actual events as possible.

### B. Semantic Description of Events

For the semantic metadata, the Resource Description Framework (RDF) is used, which allows to add knowledge in a machine-processable way. In contrast to events, this information must not be processed very fast, it rather helps to make decisions on what data to transmit. Listing 1 gives a simple example for the temperature property of Figure 3. Although the example is not complete, it should give an idea how the semantic description looks like.

```
1  <prop> a :EventProperty
2      :runtimeName          'temperature'
3      :runtimeType          so:Number
4      :hasUnit              qudt:DegreeCelsius
5      : valueSpecification  <valSpec>
6
7  <valSpec> a so: QuantitativeValue
8      :step                 0.1
9      :minValue             0
10     :maxValue             50
```

Listing 1. Semantic Description of Temperature Property

The property has the runtime name *temperature* and is of type *so:Number*, the *so* refers to the *schema.org* [1] vocabulary. Further, the unit is defined to be degrees celsius, which is derived from the QUDT ontology[2]. The value specification defines the minimum value as 0 and the maximum value as 50. Observations of the temperature sensor can have values between those two numbers with a step size of 0.1. With this information certain assumptions can be made about the events, like values higher than 50 are outliers.

## V. Datatypes and Event Reduction

How to reduce values of events is either based on the property data type or the event property semantics. Based on this information, decision rules on when to transmit the data can be defined.

### A. Data Type Rules

In our system, we support mainly four primitive data types, i.e., Boolean, Number, Enums, and Strings. For each of those data types, a different kind of reduction algorithm can be performed:

- A **Boolean** has two possible values, only when it changes it must be transmitted.
- **Strings** are not easy to reduce because often they do not contain a lot of redundant information (e.g., manually entered maintenance protocols). The most effective way is to compress Strings before sending, as proposed in [8]. However, if further domain knowledge is present, this knowledge can be used to reduce String properties. A typical example are error messages of machines, they can be cached and must not be re-sent.
- **Enums** can be encoded as numbers in case they have a fixed set of members. In case that enum members can be added dynamically, a state with all values must be provided at both the publisher and subscriber.
- **Numbers** are the most promising data type for performing data reduction operations as approximations can be applied on them. Numbers can be reconstructed in a time series based on the previously sent values.

### B. Domain Property Rules

Information about data types only contains how data is represented, but nothing about its meaning. For instance, an event's occurrence time can be represented as a String (e.g. ISO 8601), or number (e.g. UNIX time). To identify this semantic meaning, the description of event properties further contains a field with the *domainProperty*. This knowledge can be leveraged to reduce event properties with a minimum loss of information. In the following, we show some examples:

- **Geo locations** are represented as numbers (e.g. 49.006889). With the domain property wgs84:lat[3], it is specified that the value represents a WGS 84 latitude geospatial projection. A subscriber performing geo

[1] https://schema.org/ [Last Accessed: March 15, 2019]

[2] http://www.qudt.org/ [Last Accessed: March 15, 2019]

[3] http://www.w3.org/2003/01/geo/wgs84_pos#lat

operations could specify that changes of altitude values are important, but the accuracy of latitude and longitude measurements are not relevant for the correctness of the algorithm.

- A **Lidar** measures the distance to objects in its surroundings with a laser sensor. The measurements are transmitted in an array of numbers. Each number represents the measure of the distance for one specific angle. Knowing that one position in the array always measures the same angle it must not be re-sent when the value not changes.
- **Power consumption** is also represented as a number. When the values fluctuate around zero it probably means that the machine is turned off and no values must be transmitted, even when the values change marginally.

### C. Decision Rule

For both data types and domain properties, it is possible to define decision rules, defining what data to send. As a default rule for each property, the data type rule is used, but if a domain property rule is available, this is used instead. An example decision rule is shown in Listing 2. This is the main decision rule we use in this work. It first calculates the prediction of a value $x_t^p$ of *property p* at *time t*, then it calculates the error of the prediction and the sensed value. If this error is higher than a threshold $\varepsilon$, it returns true, otherwise it returns false.

```
1  δ (x_t^p, P, e, ε)
2      x̂_t^p ← P(x_t^p)
3      err ← e(x_t^p, x̂_t^p)
4      IF (err >= ε)
5          RETURN true
6      ELSE
7          RETURN false
```

Listing 2. Decide whether a property value must be transmitted or not

## VI. Broker

The main objective of our broker is to dynamically transform data streams in a publish/subscribe system. Based on the semantic meta-information of the data and configurations provided by the user, the system decides how to transmit data. First of all, raw data is handed to the publisher via its API by the actual sensing device. For each of the event payload properties, the publisher decides, whether a given property must be sent to the broker or not. $x_t^p$ represents the value of *property p* at *time t*. In case a property is not sent, the subscriber is capable to reconstruct the sensed value based on a prediction $P$, performed on previously sent data. This results in $\hat{x}_t^p$, the value of a *property p* at *time t* of the virtual event. The configuration parameter $w$ defines how big the time window is, that is used for the prediction. A simple prediction would be to assume that a value did not change and is always equal to the last value. This approach has a window size of 1. All values of a *property p* within this window are referred to as $X_t^p$. On the broker, only reduced events are stored, which also reduces the storage cost of the broker and the network bandwidth needed within a distributed broker to synchronize.

| | |
|---|---|
| $x_t$ | event with all properties at *time t* |
| $x_t^p$ | actual value of event *property p* at *time t* |
| $\hat{x}_t$ | virtual event at *time t* |
| $\tilde{x}_t$ | partial event at *time t* sent and stored on broker |
| $P$ | prediction $P(x_t) = \hat{x}_t$ |
| $w$ | window size $\{w \in N \mid w > 0\}$ |
| $X_t^p$ | vector of last w values of *property p* |
| $e$ | error function $e(x_t^p, \hat{x}_t^p)$ |
| $\varepsilon$ | threshold value $\{\varepsilon \in R \mid \varepsilon >= 0\}$ |
| $\delta$ | decision rule, returns true or false |
| $\chi$ | semantic schema of event |
| $\zeta_{sub}^p$ | configuration for *subscriber sub (publisher pub)* of event *property p* |

Table I
Notations

### A. Approach

Figure 4 illustrates the technical overview referring to our motivating scenario. We have two publishers P1 and P2, both publishing data from two temperature sensors (T1 and T2), but located in different rooms. For simplicity reasons, the example only contains the sensor identifier, a timestamp, and the measured temperature value. On the right side, we have two subscribers (S1 and S2), both subscribing to different events with a different quality. In the middle of the Figure, a topic-based message broker and two instances of subscription translators (ST1 and ST2) are shown. Translators prepare events for the subscriber. Our approach is built as a wrapper around existing message brokers so that we do not rely on a specific technology. At step 1. the raw events are shown, how they are produced and handed to the message broker. At step 2., in the broker, it can be seen that for both streams the first event is transmitted fully, but the following events only contain the values that have changed. In the event stream of T2, the temperature changes quite frequently and for T1 it is almost constant. The first subscriber S1 only subscribes to P1. In S1, a condition monitoring application is realized, where an alarm should be triggered once a certain threshold is exceeded. Therefore, small changes are not relevant but more important changes should be detected. In this case, the system filters all changes out that are lower than 0.2 degree celsius, which results in step 3., sending one value less. At step 4., the reconstructed and materialized virtual events of the subscriber S1 can be seen. They again contain all the information, but some values differ slightly from the original values of sensor T1. But since small changes are not important, the algorithm still works correctly and the system could reduce the amount of

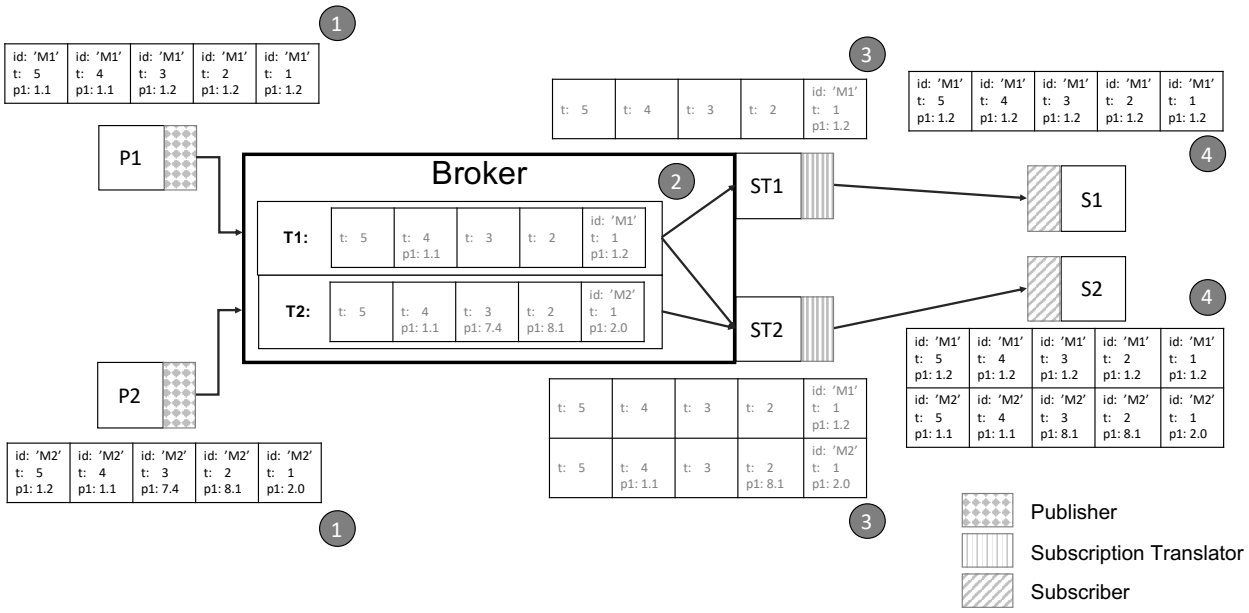


Figure 4. Example scenario illustrating the approach with two publishers and two subscribers

information that is transmitted.

In contrast, subscriber S2 makes a prediction for the future. Therefore, events from both sensors T1 and T2 are required. Previous research has shown that data can be reduced at edge nodes without affecting the accuracy of analytical approaches too much [6]. This means that, in many cases, we can further reduce the quality of events that are received from the broker. In this case we set the error threshold at 1.0 degree celsius. The resulting partial events can be seen at step 3. in the Figure. The subscriber S2 reconstructs the virtual events from the sent partial events and materialized them.

### B. Schema Registry

The schema registry is the central part that keeps all information about the whole system state. Once a new publisher is started, it registers its schema at the registry. The schema registry then creates a topic and returns it to the publisher, which then can start emitting events. For subscribers this is similar. Once they are started, they ask the schema registry for the topic they want to consume. The subscriber then opens a connection to the broker and receives the data. In addition, the schema registry does not only have an overview over the whole system, it is also capable of changing the configurations of publishers. A simple example underlining the advantage of such an approach is that if no subscriber subscribes to the events of a publisher, the corresponding publisher does not need to send any events. As soon as a subscriber starts subscribing, the schema registry notifies the publisher to start publishing. Even if the schema registry is a central control instance, it still can run distributed, which ensures that the schema registry is not a single point of failure. An example of a schema registry can be seen in Figure 1, where for each producer an entry is created, containing the schema of the events and the subscribers that are registered at a topic for subscription. The schema registry is responsible for the management of the system, while the actual data stream is handled by the message broker.

### C. Publisher / Data Producer

Publishers provide a simple API which can be used to transmit events to the broker. They receive the configuration $\zeta_{pub}$ from the schema registry. To publish an event $x_t$ at $time$ $t$, first the *reduceEvent* routine is called. This routine reduces the event according to the available reduction configurations. For each property it is defined in the configurations how granular the updates should be transmitted. The configuration also defines the desired precision of the values of the virtual events. Once the event is reduced within the publisher, it is transmitted to the broker. If the requirements of the subscribers change, the publisher is notified by the schema registry with the new configuration. It then starts to publish events according to the new configuration.

Listing 3 defines the *reduceEvent* routine. This routine is used by multiple algorithms, it takes an *event* $x_t$ at *time* $t$ as an input and removes unnecessary information according to the configurations $\zeta_{pub}$. These configurations are individual for all running instances of the routine. In more detail, publishers and subscribers define how they want to send or submit data and during runtime events are reduced accordingly. In line 2, we iterate over all properties that are part of an event's payload. Afterwards, in line 3 the

configuration $\zeta^p_{pub}$ of *property p* for the *publisher pub* is loaded. This configuration consists of a 4-tuple with the *decision rule* $\delta$, the *prediction function P*, the *error function e*, and the error value $\varepsilon$. The *decision rule* $\delta$ defines whether the current value of the property should be transmitted or not. This decision is based on the comparison of the predicted value calculated with the *prediction function P*. The *error function e* is used to calculate the difference between the predicted and the actual value and $\varepsilon$ defines how much the two values can differ. After the values are set, line 4 applies the *decision rule* $\delta$. If the return value is true, the value of the property is added to the partial event $\tilde{x}_t$. Once all properties of one event are processed, the newly created partial event $\tilde{x}_t$ is returned which contains all values that need to be transmitted to the broker.

```
1  reduceEvent( x_t )
2     FOR ALL  x_t^p  IN  x_t  DO
3        (δ,P,e,ε)  ← ζ^p_pub
4        IF (δ (x_t^p,P,e ,ε))
5           x̃_t^p .add(x_t^p)
6
7     RETURN x̃_t
```

Listing 3. Reduce events before transmitting them over the network

### D. Subscriber / Data Consumer

Subscribers can start subscribing at any time. Further they can define a quality level in the configuration parameters for the individual properties in the event schema. Each subscriber has its own event channel from the broker to the subscriber. For this channel, the events are transformed according to the needs of the subscriber. Algorithm *prepareSubscription* in Listing 4 calculates the partial events for the subscriber and sends them over the subscription channel. This algorithm runs directly on the broker and gets the partial events sent from the publisher as an input. This is why in line 2 of the algorithm the virtual event from the subscriber must be reconstructed with the *expandEvent* routine explained later in this section. The resulting event is then handed to the *reduceEvent* routine, where it is reduced according to the configuration of the subscriber. Note that $\tilde{x}_t$ and $\tilde{x}'_t$ are in general not the same, only in case the publisher and subscriber configuration are identical they contain the same value. $\tilde{x}'_t$ is then sent to the subscriber. This algorithm runs in ST1 and ST2 as illustrated in Figure 4. It can be seen that the events can further be reduced if the subscribers are not interested in the original quality. Our approach uses a checkpoint mechanism creating checkpoints from where the replay can be started. For simplicity reasons, this code is not included in the pseudo code of the algorithms.

```
1  prepareSubscription( x̃_t )
2     x̂_t ← expandEvent(x̃_t)
3     x̃'_t ← reduceEvent(x̂_t)
4     sendToSubscriber( x̃'_t )
```

Listing 4. Preparse the events for the specific subscribers

The *expandEvent* routine, shown in Listing 5, takes a partial event as an input and reconstructs a virtual event according to the configurations in the schema registry and the previously received events. In line 3, it extracts the event schema from the schema registry and loops over all properties from the schema. The *exist* function in line 3 checks whether the value of the property from the event schema is in the partial event or not. In case the value was transmitted in the event it is set in the virtual event. Otherwise, the prediction rule as defined in the configurations for that property is applied. The resulting value is then set as an event value for the virtual event as shown in line 6. At the end of the routine, the resulting virtual event is returned. It contains a mixture of transmitted values and predicted values that reflect the real measured values.

```
1  expandEvent(x̃_t)
2     FOR ALL χ^p IN χ DO
3        IF exists ( χ^p , x̃_t)
4           x̂_t^p ← x̃_t^p
5        ELSE
6           x̂_t^p ← P_ζsub (p)
7
8     RETURN x̂_t
```

Listing 5. Reconstructs a virtual event based on the configurations

The last algorithm is the *subscribeEvent* algorithm, running at the subscriber. As an input, it receives the reduced partial event produced by the *prepareSubscription* algorithm at the broker. This event is then expanded using the previously described *expandEvent* routine. $\hat{x}_t$ is the resulting virtual event that is consumed by the event processing algorithm. The degree to how much this value differs from the original measurement highly depends on the configuration of the publisher and subscriber. This ensures that our main goal cab be achieved, which is to enable different subscribers to get the data in different quality levels depending on their processing purpose of the subscriber.

## VII. Evaluation

In this section, we provide the results of our evaluation that are based on a prototypical implementation. We performed our evaluation on real-world data from an STCS system.[4]

[4]This dataset was provided by the Faculty of Management Science and Engineering of the Karlsruhe University of Applied Sciences.

### A. Reference Implementation

We implemented the system in Java as a wrapper around Apache Kafka[5]. We chose Kafka as it is highly scalable and capable of handling large amounts of events. On the producer side, we built a wrapper around the kafka-client producer. Before events are sent to the broker, they are reduced using the proposed algorithms presented in Section VI. On the server side, we implemented another component that receives the reduced events of the individual topics and prepares them for the subscriber. This means that our implementation is comprised of two reduction steps, one directly at the publisher and another one at the broker before events are transmitted to the subscriber. For the subscriber, we implemented a subscription interface that transforms the sent partial events into virtual events that can be processed by the program using the subscriber. To evaluate our approach, we added four logging stages into the system. Two stages are located at the publisher side (one for the raw events that should be send and one for the reduced partial events), another one at the broker, which logs the partial events for the subscriber and a final one directly at the subscriber to log the resulting virtual events after they have been reconstructed.

### B. Experimental Setup

Our data set consisted of approximately 1,000,000 Events. The size (serialized in JSON) is about 855 MB. For the evaluation, we ran an instance of Kafka in addition to three developed components, whereas all components are deployed as Docker containers. The server itself has 4 cores and 32 GB of memory. The publisher reads the data of the STCS system from a file. For sending events to the broker, we chose an $\varepsilon$ of 0 and a prediction where it is assumed that the value did not change. With these configurations, the virtual events can be reconstructed exactly as the sensed raw events. In our case the subscriber does not need events if the changes are small, so we set the $\varepsilon$ to 1 and we chose the same prediction rule as for the publisher. This means there is an error in the reconstructed virtual events but the algorithm using the subscriber only needs to detect significant changes. In our experiments, we created different serializations configurations and compared them. As serializations, we chose Apache Thrift[6], Apache Avro[7], JSON and JSON events compressed with GZIP. The drawback of Thrift is that it does not support serialization and deserialization of partial events as our approach relies on. Due to that reason, we only used it as a baseline and did not include it into our framework. JSON is a common format to serialize events and it is human-readable. For use cases where events do not have to be human-readable, we compressed it with GZIP and Avro to reduce the size of the serializations.

[5]https://kafka.apache.org/ [Last Accessed: March 15, 2019]
[6]https://thrift.apache.org/ [Last Accessed: March 15, 2019]
[7]https://avro.apache.org [Last Accessed: March 15, 2019]

### C. Results

We conducted two evaluations. First, we checked how good the data can be reduced by our approach, and second how much compute and memory overhead is required to calculate the the virtual events. Figure 5 shows the baseline of our experiments. On the left the size of the test data set serialized in JSON ( 885 MB), next to it the size of the data set compressed with gzip ( 390 MB) is shown. By using an Avro-based serialization, the size is 392 MB and with Thrift 505 MB.

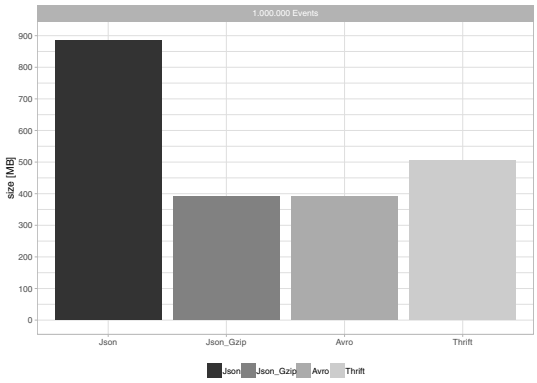


Figure 5. Baseline without reduction

It can be seen that compressing the data reduces the original data size by over a half. Next we performed multiple tests with our approach, leading to the results illustrated in Figure 6. For each serialization, we show the data size at the publisher, the amount transmitted from publisher to broker, from broker to the subscriber, and the reconstructed event at the subscriber. The size of the data at the publisher and subscriber is the same for each of the serializations, meaning the data was correctly reduced and reconstructed again.

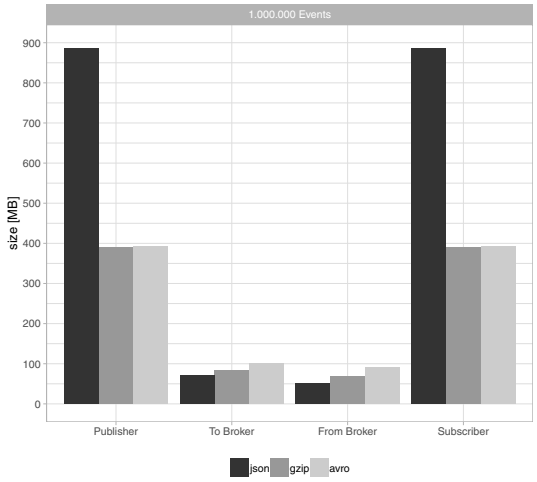


Figure 6. JSON with our reduction

In the middle of the chart it can be seen that reducing the events significantly reduces the event size. For JSON, it is less then a tenth of the original size. This big reduction can be explained by the low system inertia of the STCS. The data was sampled every second, but for example the room temperatures does not change that frequently. But since the virtual events are reconstructed at the end we do not lose any information. Also, when there are fast changes in the system we can detect them immediately. The Figure also shows that when the subscriber requires data with less quality it can be further reduced.

So far we demonstrated how our approach is capable of reducing data that must be transmitted over the network. But this benefit comes at the cost of additional memory and CPU usage to calculate the virtual events. To measure this impact, we turned off the logging mechanisms we used in our first evaluation to get the size of the events and measured the performance. We have five containers in our setup, all running on one virtual machine. The first container contains Kafka and the second one Zookeeper. The publisher (third container) reads data from a file and sends it to the broker. The fourth container is the wrapper around Kafka preparing the events for the subscriber, the fives and last one. The measurements are obtained from the Docker statistics API.

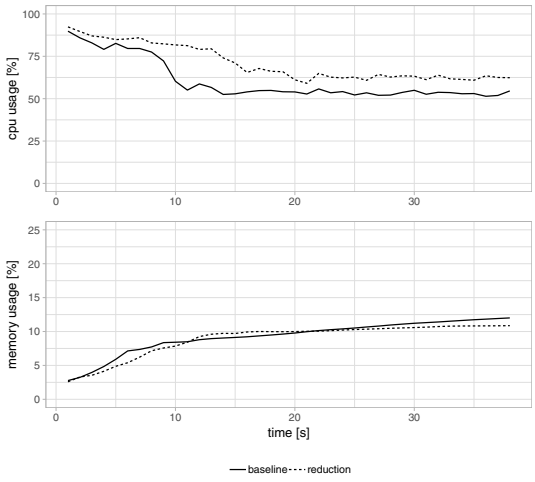


Figure 7. Computation performance of our approach

The results of the performance measurements are shown in Figure 7. We ran each setup 10 times and averaged the measures. The measurements started recording once the publisher started publishing events and stopped once all events had been sent by the publisher. As a baseline, the publisher was sending the raw events to Kafka and the subscriber read those events from Kafka. We compare our approach to this baseline, where we reduce the events before they are transmitted and reconstruct them directly at the subscriber. The results in the Figure summarize the metrics of all five containers. In the top chart, the CPU consumption is shown. Here it can be seen that our approach uses slightly more CPU than the baseline. The reason for that is that more operations for reducing and reconstructing the events are performed. It can be seen that this overhead is constant and only marginally exceeds the baseline. The memory consumption can be seen at the bottom of Figure 7. Over time our approach uses less memory, which is most likely due to the fact that we reduce the amount of transmitted data. This also means that less data needs to be stored on the message broker, e.g., Kafka in our setup. This shows that the overhead of CPU usage is slightly higher and the memory usage is comparable on this dataset.

Altogether, it can be said that our approach works especially well on data that does not change with a high fluctuation, while only adding little computational overhead. When there is a high fluctuation in the data the values are still transmitted correctly, but the reduction rate can be lower.

## VIII. Related Work

This section presents related work relevant for our approach. While there already exist different technologies that implement publish/subscribe systems, our approach relies on topic-based publish/subscribe systems, where subscribers make use of content-based filtering (e.g., by expressing quality restrictions). This is similar to content-based publish/-subscribe systems, where subscribers can define individual restrictions. Such systems must ensure that the right data is sent to the correct subscribers. In general, to reduce the computation overhead, some approaches exist that try to optimize this. One such an approach is [11]. They propose a popularity-based routing approach to optimize the data flow within the broker. Another approach is presented in [3], where they use software defined networking (SDN) to filter events directly on network devices. Within this approach, it is possible to filter out events earlier before they are transmitted to the subscriber. In addition, we build a wrapper around topic-based systems. By using existing solutions we can leverage efficient routing within the systems and events are pre-processed and filtered out near the publisher to ensure that only relevant information are transmitted to the subscriber.

There also exist message brokers designed for edge processing systems, for example EMMA [12]. They optimize the QoS during runtime by monitoring the latency to detect the proximity between the message broker, publisher, and subscribers. Their message broker is based on MQTT and capable of a dynamic rescheduling of clients and brokers. Our concepts could be combined with such an approach to reduce the amount of events transmitted over the network and automatically reschedule when the situation changes.

Another related area is the field of WSN. In such networks, sensing devices are distributed and connected via a wireless network. Each of these devices often has a limited amount of energy, meaning they must optimize the

transmission of gathered information. In [5] and [6], the authors propose an approach where individual sensors can decide whether they transmit data or not. The receiver is capable of reconstructing the measurements although they are not actually being sent. This results in a more energy efficient local computation compared to transmitting raw data to a central node. We used similar ideas to reduce numerical values in our approach, but also handle other datatypes. Furthermore, we allow data subscribers to dynamically define the required quality to access event streams.

The serialization of events can also significantly influence the event size as described in [13]. In addition, several techniques exist to perform specific compression algorithms, for example to use shared dictionaries to compress events in publish subscribe systems [8]. In general, while our approach focuses on reducing the content of event streams, serialization and compression techniques can be integrated in our system to further improve the reduction of the event size.

## IX. Conclusion and Future Work

In this paper, our goal was to use static and dynamic information of events and measurements to enable data stream reductions in publish/subscribe systems. More specifically, we showed how subscribers can define their required data quality and how the system uses this information to optimize the flow of events. First, we introduced a framework that classifies different techniques to reduce events in terms of event size and frequency. Then, we presented a semantic model of events. Furthermore, we introduced the concept of a virtual event, which is an event that must not be completely transmitted but can be reconstructed by the system with or without loss of quality. Our third contribution is a concept and architecture for a wrapper around existing topic-based message brokers. In the evaluation, we were able to show that our approach is better than only using compression techniques or binary formats. On the one side our approach adds little overhead in CPU power, which will be later distributed on edge devices. The advantages of our approach are especially important when events contain a lot of redundant information. On the other side, in case of high fluctuation of measured values within events, we do not gain too much with our reduction. This is because we used rather simple prediction techniques. As a consequence we plan to integrate more advanced prediction algorithms. So far, the configuration parameters must be defined for the individual properties, in the future, we aim to investigate if those configurations could be learned by the system and adapted during runtime in the future.

## References


[1] H. Shen, "Content-Based Publish/Subscribe Systems," in *Handbook of Peer-to-Peer Networking*, X. Shen, H. Yu, J. Buford, and M. Akon, Eds. Boston, MA: Springer US, 2010, pp. 1333–1366.

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[3] S. Bhowmik, M. A. Tariq, J. Grunert, and K. Rothermel, "Bandwidth-efficient content-based routing on software-defined networks," in *Proc. of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, 2016, pp. 137–144.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[5] Hongbo Jiang, Shudong Jin, and Chonggang Wang, "Prediction or Not? An Energy-Efficient Framework for Clustering-Based Data Collection in Wireless Sensor Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 1064–1071, Jun. 2011.

[6] N. Harth and C. Anagnostopoulos, "Quality-aware aggregation predictive analytics at the edge," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017, pp. 17–26.

[7] S. Müller, P. Wiener, A. Bürger, and J. Nimis, "IoT for All: Architectural Design of an Extensible and Lightweight IoT Analytics Platform," in *Proceedings of the 2017 International Conference on Industrial Engineering and Systems Management (IESM), Saarbrücken, 2017*, V. F. Bousonville T., Melo T., Rezg N., Ed., Saarbrücken, 2017.

[8] C. Doblander, T. Ghinaiya, K. Zhang, and H. Jacobsen, "Shared dictionary compression in publish/subscribe systems," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA*, 2016, pp. 117–124.

[9] N. Meratnia and A. Rolf, "Spatiotemporal compression techniques for moving point objects," in *International Conference on Extending Database Technology*. Springer, 2004, pp. 765–782.

[10] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications Company, 2010.

[11] P. Salehi, K. Zhang, and H. Jacobsen, "Popsub: Improving resource utilization in distributed content-based publish/subscribe systems," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS '17, Barcelona, Spain*, 2017, pp. 88–99.

[12] T. Rausch, S. Nastic, and S. Dustdar, "Emma: Distributed qos-aware mqtt middleware for edge computing applications," in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*. IEEE, 2018, pp. 191–197.

[13] K. Maeda, "Performance evaluation of object serialization libraries in XML, JSON and binary formats," in *Digital Information and Communication Technology and It's Applications (DICTAP), 2012 Second International Conference On*. IEEE, 2012, pp. 177–182.