Simplifying CPS Application Development through Fine-grained, Automatic Timeout Predictions

Stefanos Peros, Stéphane Delbruel, Sam Michiels, Wouter Joosen, Danny Hughes

Simplifying CPS Application Development through Fine-grained, Automatic Timeout Predictions

STEFANOS PEROS, STÉPHANE DELBRUEL, SAM MICHIELS, WOUTER JOOSEN, and DANNY HUGHES, imec-DistriNet, KU Leuven

Application development for Cyber Physical Systems (CPS) is challenging, because the wireless network and the devices introduce latencies that vary continuously along with the load, status, or environmental conditions of the infrastructure. Reactive programming is well suited for the development of event-driven applications, yet current reactive programming frameworks require developers to predict event arrival time-boundaries at compile time, which is impractical, if not impossible, for CPS. Thus, there is a tradeoff between timeliness and completeness of complex event computations, e.g., operational efficiency in a manufacturing plant: Waiting too long until all individual events arrive can fail to produce a useful result, while not waiting long enough may lead to faults due to incomplete status information. In this article, we propose (a) a set of extensions to state-of-the-art reactive programming frameworks, which remove the burden of specifying timeouts at compile time by utilizing (b) Khronos, a middleware that automatically determines timeouts by taking into account variations in event arrival times due to the underlying infrastructure. Evaluation on a physical testbed shows that the extensions significantly decrease developer effort and that Khronos considerably improves timeliness under varying network configurations and conditions, while still satisfying the application's tolerance to missed events.

CCS Concepts: • **Information systems** → *Stream management*; • **Networks** → *Network monitoring*; • **Computer systems organization** → *Embedded and cyber-physical systems*; • **Software and its engineering** → *Middleware*;

Additional Key Words and Phrases: Cyber-physical systems, late arrivals, middleware, time management, event-driven programming, reactive programming

ACM Reference format:
Stefanos Peros, Stéphane Delbruel, Sam Michiels, Wouter Joosen, and Danny Hughes. 2020. Simplifying CPS Application Development through Fine-grained, Automatic Timeout Predictions. *ACM Trans. Internet Things* 1, 3, Article 18 (May 2020), 30 pages.
https://doi.org/10.1145/3385960

1 INTRODUCTION

Physical objects are being enhanced with computational capabilities to serve as components of larger distributed cyber physical systems (CPS) [13]. The possibility to remotely monitor CPS

This research is partially funded by the Research Fund KU Leuven and the FWO D3-CPS project.
Authors' addresses: S. Peros, S. Michiels, W. Joosen, and D. Hughes, imec-DistriNet, KU Leuven, Celestijnenlaan 200A, Leuven, Vlaams-Brabant, 3001, Belgium; emails: {stefanos.peros, sam.michiels, wouter.joosen, danny.hughes}@cs.kuleuven.be; S. Delbruel, imec-DistriNet, KU Leuven, Celestijnenlaan 200A, Leuven, Belgium, 3001, Belgium; email: stephane.delbruel@cs.kuleuven.be.

2577-6207/2020/05-ART18 $15.00
https://doi.org/10.1145/3385960

behavior in distributed infrastructures, by aggregating event data from various sensors over a wireless network, has digitally transformed several industries, including manufacturing, transport and logistics, and utilities [27, 41, 46, 50].

The development, monitoring, and maintenance of CPS applications is difficult due to the large number of devices, their dynamic network topology, heterogeneity, and their resource constrained nature [20, 41, 44]. Distributed reactive programming (DRP) is an emerging paradigm that aims to address these challenges by introducing the concept of remote signals. Signals are abstractions for asynchronous data streams whose dependencies can be specified by the application developer in a declarative fashion, forming a dependency graph. The reactive runtime elegantly handles signal updates by automatically propagating changes down the dependency graph, based on the desired consistency degree [37]. This relieves the application developer from the complexity of manual callback management, which is often the source of many bugs in event-driven applications [24, 36].

In CPS applications, (remote) signals often rely on sensors that measure a physical property of the environment, at a specified sampling period. However, packets can still be generated at different rates, referred to as packet inter-generation delay [47], due to the lack of a shared and accurate time-source along with device imperfections. This can lead to non-deterministic packet arrival times, even in the presence of emerging network technologies that provide deterministic network latency, e.g., Time-Sensitive Networking [62]. Packet inter-generation delay, together with the presence of varying network latency in state-of-the-art wireless network technologies, can result in non-deterministic packet arrival times at the gateway. In many cases, it is impossible to distinguish between a non-arrival due to a fault or due to delay, which can be crucial for detecting failures in distributed systems [22].

The detection of complex events may depend on the occurrence of multiple events, such as the arrival of sensor data, to compute a result. The problem lies in predicting time-boundaries for individual event arrivals, for which state-of-the-art solutions rely on the CPS application developer, hereinafter referred to as the developer. This is difficult, if not impossible, due to the heterogeneity and dynamism in the network, platform, and applications, along with the application developer's limited knowledge of the underlying infrastructure [11, 12]. For many CPS applications, complex events need to be computed in a timely fashion to produce useful output, e.g., detecting production line down-times in manufacturing. In other words, the quality of the result often depends on the data age of the input events, which can be too high when timeouts occur too late. However, when timeouts occur too soon, not all dependent events may have arrived, leading to incorrect results under incomplete information. Having clear control over the tradeoff between timeliness and completeness is of prime importance for CPS applications.

Current DRP frameworks offer developers limited support for timely interactions between event streams and for failure handling due to late event arrivals. More concretely, existing solutions require developers to specify timeouts at compile time, which is impractical, since event arrival times can differ due to the varying network latency and packet inter-generation delay [47]. Furthermore, these solutions force the event stream to be terminated when a timeout occurs, which is unsuitable for CPS applications that need to keep operating despite late event arrivals, e.g., monitoring production in a factory. Leased signals [44], for example, are language abstractions for data streams that enable the runtime environment to react to the expiration of a user-defined signal lease.

The main contributions of this work are (a) a novel set of language extensions for state-of-the-art reactive programming frameworks that remove the burden of specifying timeouts at compile time from the developer, built on top of (b) Khronos, a middleware that automatically predicts timeouts for sensor data event streams in CPS, based on the application's tolerance to missed events. Taken together, we present a novel solution that provides developers with reactive language

abstractions that lead to less complex and more concise programs, enabling them to easily trade off the timeliness (latency) with the completeness (quality) of the data without relying on their CPS infrastructure knowledge for the specification of static timeouts or other advanced configuration parameters at compile time.

The proposed language extensions are used to implement an industrial use-case example: monitoring the production process in a car safety equipment manufacturing plant. Our evaluation shows that the language extensions reduce program complexity in comparison to ReactiveX [1], a standard reactive programming framework. Finally, an extensive evaluation on a physical testbed shows that Khronos not only ensures constraint satisfaction in the presence of dynamism and heterogeneity, but that it also improves timeliness by dynamically setting timeouts, based on the observed status of the underlying network. The complete code-base of the reactive extensions,[1] Khronos, and the datasets used in the evaluation are open-source[2] to ensure the reproducibility of our work and promote collaboration.

The remainder of the article is structured as follows: Section 2 provides additional background, a formal specification of the system model, and the problem statement and explores the problem through the lens of a real-world industrial use-case. Section 3 provides an overview of the related work and the identified middleware requirements. Section 4 describes the design of the reactive extensions and the architecture of Khronos, along with the underlying prediction technique. Section 5 presents the implementation details of the extensions, the CPS network, and the middleware. Section 6 discusses the evaluation setup and results. Finally, Section 7 concludes the article.

2 BACKGROUND

Technological advancements in the area of Industry 4.0 have created new business opportunities based on integrating CPS with manufacturing, in the past known as Cyber-Physical Production Systems (CPPS) [33, 59, 63], which increase efficiency and reduce manufacturing costs [42]. Manufacturing plants are enhanced with actuators and sensors that measure various physical, time-varying properties, such as product displacement, machine temperature, liquid flow rate, and so on [51]. Sensor data are transmitted over the network to a control unit that takes actions to improve the operation of the manufacturing plant.

In reactive programming, time-varying values such as physical properties measured by sensors are represented by so-called signals, which encapsulate streams of discrete events [45]. Developers can aggregate over events produced by multiple signals using signal combinators, which result in a new output signal. The reactive program can be expressed by means of a dependency graph, where each node has incoming edges from nodes producing its input signal(s) and outgoing edges to nodes receiving its output signal(s). The key advantage of reactive programming is that the language runtime environment automatically propagates changes along nodes in the graph. This allows programmers to elegantly write event-driven applications without the disadvantages of manually managing callbacks (e.g., callback hell [25]), which are present in traditional programming solutions [14].

Typical network technologies used in industrial CPS applications are wireless mesh and star networks [19]. In mesh networks, messages may need to traverse the network across many hops, traveling through several devices before reaching their final destinations. Based on the underlying medium access control protocols, e.g., Carrier-Sense Multiple Access (CSMA) and Time Synchronized Channel Hopping (TSCH), per-hop latency varies from tens of milliseconds to several seconds. Variable latency is also prevalent in wireless star networks, such as LoRa and BLE [52,

[1] Available at: https://github.com/mazerius/rx-extensions.
[2] Available at: https://github.com/mazerius/khronos.

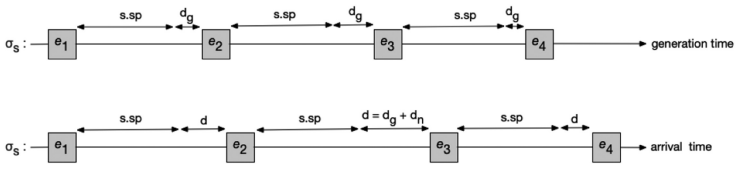

Fig. 1. Device data stream notation.

53]. In LoRA, latency varies with the spreading factor and payload size, while in BLE it is sensitive to radio interference. As a result, predicting packet arrival times in these networks and specifying corresponding timeouts in the application remains an open challenge.

State-of-the-art solutions rely on the developer's infrastructure knowledge to manually specify timeouts for each device data stream at compile time [44]. Static timeouts are hard to specify in advance, since the developer has limited knowledge of the infrastructure [11, 12] and are highly inflexible in the presence of dynamism, as their performance depends entirely on the current state of the network. Recent work [47] attempts to address this issue, focusing on probabilistic approaches to manage late event arrival times, yet still relies on the developer to specify the query frequency and further configuration parameters that directly impact their performance. In practice, static timeouts are determined using rules-of-thumb, such as a multiple of the sampling period or adding the average network delay [54].

Middleware can be used to interface between the CPS infrastructure and the applications [41], allowing abstraction and flexibility in application development that lead to long-term sustainable solutions. However, the diversity and heterogeneity of CPS architectures makes it hard for a general purpose middleware to keep track of the aforementioned tradeoff. A dedicated middleware is required that enables developers to easily manage the timeliness and completeness of events flowing from the CPS to their application, without requiring additional configuration from the developer by relying on his/her knowledge of the underlying infrastructure [26, 41].

Model and Problem Statement

In this section, we formally define the system model and problem statement that our solution aims to solve. Let $S = \{s | s \text{ is a data source in the CPS}\}$ be the set of all sensing devices in the underlying CPS infrastructure that transmit messages to the gateway. We refer to the sensing devices as data sources and to the generated messages as events. An *event* e is defined as a tuple of four attributes $e = \langle s, v, gts, ats \rangle$, where s is the data source that generated the event, v is the message payload (e.g., sensor data), gts is the timestamp at which the event was generated, and ats is the timestamp at which the event arrived at the gateway. We use the notation e^s as a shorthand notation to refer to an event that is generated by *data source* s. Every data source s emits a (possibly) infinite sequence of events, defined as the *device data stream* σ^s, where $\sigma^s = \{e^s\}$.

In our model, we focus on sensing devices that transmit periodically: A data source s has a sampling period sp, noted as $s.sp$. Despite a fixed sampling period, events arrive at the gateway at a non-constant rate due a varying *delay* d, where $d = d_g + d_n$ with d_g the packet inter-generation delay and d_n the network delay, as illustrated in Figure 1.

Device data streams serve as input streams to the application, which consists of stream operators that process incoming events and inject the result to their output stream. In real-time stream processing, a timeout is associated with individual operations that can block the output stream indefinitely, such as waiting for an event from a sensing device to arrive [55]. More concretely, a *timeout* τ is associated with every event e, noted as τ^e, that determines how long the program should wait for e to arrive. The challenge lies in determining the value of τ in the presence of

varying d_g and d_n, which are not known *a priori* and can vary at runtime, to balance the tradeoff between two contradicting goals: completeness and timeliness.

The *completeness* $\gamma(\sigma^s)$ for a device data stream σ^s expresses the ratio of generated events that arrived on time from that stream. Formally, let $\delta_i^s = e_i^s.ats - e_{i-1}^s.ats$ be the difference between the arrival times of two consecutive events from σ^s. We define that an event e_i^s arrives *late* if $\delta_i^s > \tau^{e_i^s}$ and *on time* if $\delta_i^s \leq \tau^{e_i^s}$. By definition, if an event arrives late, then it is missed. To formally define the completeness of a stream σ^s, we first define a new stream ϕ^s, such that $\phi^s = \{f(e^s)\}$, where

$$f(e_i^s) = \begin{cases} 1 & \text{if } \delta_i^s <= \tau^{e_i^s} \\ 0 & \text{otherwise} \end{cases}. \tag{1}$$

In words, for every event generated by σ^s, the stream ϕ^s produces 1 if the event arrived on time and 0 otherwise. The completeness $\gamma(\sigma^s)$ for a device data stream σ^s is defined as

$$\gamma(\sigma^s) = \frac{\sum_{i=1}^{n}(\phi_i^s)}{n}, \tag{2}$$

where n is the index of the last event generated by σ^s.

The timeliness for a device data stream σ^s refers to how close the timeouts are to the event arrival times from that device stream. For that purpose, we define the *prediction error* $\epsilon(\sigma^s)$ for σ^s as

$$\epsilon(\sigma^s) = \frac{\sum_{i=1}^{n} |\tau^{e_i^s} - \delta_i^s|}{n}. \tag{3}$$

A large prediction error corresponds to low timeliness and vice versa. Note that in theory, σ^s is a potentially infinite sequence of events, which means that $n \to \infty$. In practice, the completeness and prediction error are computed over a finite window of past events w, such that $n - i = w$.

The problem at hand consists of two objectives: minimizing $\epsilon(\sigma^s)$ while keeping $\gamma(\sigma^s)$ within a user-defined budget. Users, typically developers, can specify this budget by means of a *completeness constraint* ρ for a device data stream σ^s, which indicates how important events from that stream are to the application. A completeness constraint ρ for σ^s is satisfied if $\gamma(\sigma^s) \geq \rho$. We can now define the optimization problem that Khronos aims to solve:

Problem 1 (Timeliness Optimization). *Given a device data stream σ^s and a user completeness constraint ρ for that stream, the Timeliness Optimization aims at finding a timeout τ^{e^s} for each event e^s s.t.*

$$\min \epsilon(\sigma^s), \tag{4}$$

$$\text{subject to } \gamma(\sigma^s) \geq \rho, \tag{5}$$

The next subsection discusses further these concepts in the context of a real-world industrial use case.

Industrial Context

This section describes an industrial use-case from the customization and packaging division of a Fortune 500 car safety equipment manufacturing plant to further illustrate the problem. An overview of the factory is shown in Figure 2, including the device communication links and the corresponding link latency. In this example, under normal operation, link latency varies between tens to hundreds of milliseconds, as shown by the intervals in Figure 2. The packaging plant has six production lines, labeled Line 1 to Line 6. These lines consist of machines that produce airbags and are equipped with sensors that periodically compute and transmit the item processing rate to the gateway, measured as the number of produced units per minute. The item processing rate of

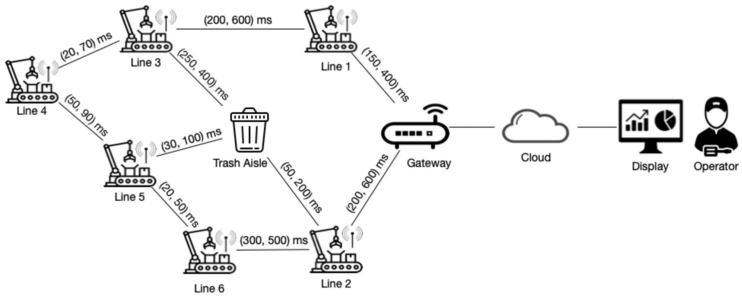

Fig. 2. Example: Industrial plant overview consisting of six production lines.

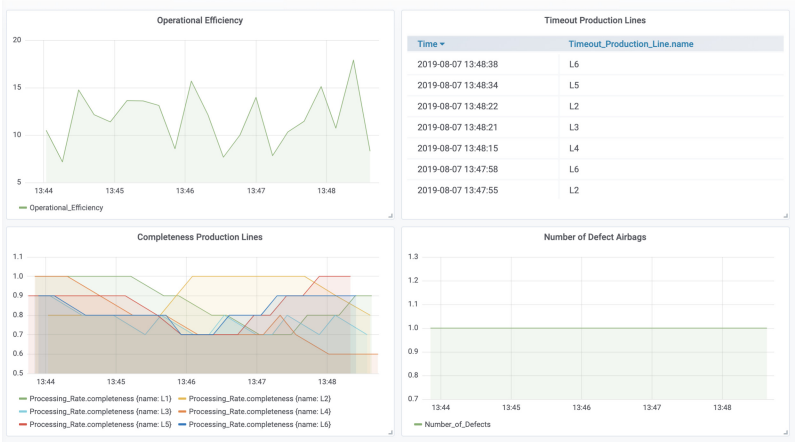

Fig. 3. Visual display monitoring the airbag manufacturing process.

a production line is an indicator of its operational efficiency and the average item processing rate across all production lines that manufacture airbags indicates the overall operational efficiency. Furthermore, these production lines are physically connected to a trash aisle, where defect airbags are deposited. The trash aisle is equipped with an object detection sensor, which enables the application to count the number of defect airbags due to a failure in the production process. The sensors in the manufacturing plant transmit messages at fixed sampling periods to the gateway, which are unknown to the developer, containing the recorded metadata as payload over the past time period. The metadata are forwarded to a back-end, e.g., the cloud, where the application computes the overall operational efficiency, which is shown on a display as illustrated in Figure 3, and observed by the operator, whose role is to monitor the manufacturing process. Furthermore, the application notifies the operator to shut down production when the number of defect airbags exceeds a certain threshold for a particular batch, as directed by the regulations on airbag manufacturing.

The application further requires that (1) the visual display is refreshed in a timely fashion, to provide a soft real-time view of the operational efficiency, and that (2) the human operator is informed based on correct information, to avoid shutting down production unnecessarily due to a network or sensor failure at the trash aisle. The developer can easily express the latter requirement using a high completeness constraint (e.g., $\rho = 1.0$), which causes large timeouts for the arrival of trash aisle sensor data. In this case, reaching the timeout strongly indicates that there is something exceptionally wrong with the sensor or the network, which the operator should investigate. The

former requirement can be accomplished by using lower completeness constraints, utilizing the fact that the item processing rates do not vary significantly during a single batch. This leads to shorter timeouts for the production line sensor data, and thus faster updates of the visual display, using the last received sensor data in case of a timeout.

The problem lies in determining the refresh period of the display, which is non-trivial, since packets can travel across different paths in the network, resulting in non-deterministic arrival times due to varying link latency and packet inter-generation delay, as explained in Figure 1. The information that depends only on events from a single sensor can be refreshed as soon as the event from that sensor arrives. However, this is not possible when the displayed information is a function of multiple events from different sources, in this case the overall operational efficiency, where all of the combined events need to correspond to the same time period in the real-world to ensure the quality of the result. Despite the small scale of the network, the recorded packet arrival times at the gateway show substantial variance, to the order of seconds. Unlike the typical Internet, the challenge in Wireless Sensor Network (WSN) technologies is largely due to the resource-constrained nature of the devices and the unreliability of the physical environment.

State-of-the-art solutions require the developer to specify a static refresh period for the display. In practice, typically an arbitrarily large refresh period is used to ensure completeness of the results at the expense of timeliness. As a result, the display lags behind in time, making it difficult for the operator to interpret the results and act in a timely fashion. Not only would the developers benefit from a solution that automatically determines event arrival timeouts, by having clear control through powerful language abstractions over the tradeoff between timeliness and completeness, but also the operator(s) can make timelier decisions based on the visualized information, improving the overall operation efficiency of the manufacturing plant. A critical overview of the state-of-the-art solutions is discussed in the next section.

3 RELATED WORK

Middleware and related frameworks are key components of complex systems, especially when dealing with the constraints of CPS. Due to the broader range of environments and related constraints on network resources found in CPS, the limited support offered by modern systems is not enough and a one-size-fits-all solution is not an option. This need for more specific solutions is raised by Mohamed et al. [41], who identifies that general-purpose distributed middleware are not flexible enough to tackle unique challenges in CPS. The challenges of middleware for CPS, including the support for real-time operations (e.g., decision making), autonomous operations, data integrity and correctness, have been addressed in the past as a subset of these in a generic form for a specific feature, or focusing on one for a more broader group of CPS applications. The authors [41] rely on past work to emphasize the specificity and diversity of a CPS ecosystem, and propose a more context-aware approach to consolidate the generic approach and limit the spread of the specific solutions.

Among these past works, Zhang et al. [61] explored the issues of real-time middleware used as platforms for distributed systems with time constraints when facing workloads with both aperiodic and periodic tasks. To tackle the lack of flexibility from existing systems, their contribution of configurable middleware components providing effective on-line admission control and load balancing for distributed computing platforms is an important step for CPS. However, the authors do not address timeliness challenges that occur due to the underlying network and its resource-constrained devices and middleware reconfiguration options to cope with its uncertainties and maintain real-time support.

In the context of distributed reactive programming, Myter et al. [44] proposed the concept of leased signals to deal with partial failures due to late event arrivals. Leased signals express a

semantic agreement between the signal and its subscribers: The time window during which an event is expected to be emitted by the signal. The runtime environment enables the application to react to the expiration of a lease by executing the application failure handling logic. Nonetheless, leased signals offer limited support for late event arrival management, since they rely on the developer to specify a static timeout for each signal at compile time. Furthermore, once a lease expiration has occurred, the signal is terminated, thus disallowing the application to react to future event arrivals from that signal's source.

Derler et al. [23] propose PTIDES, an actor-based programming model that infers upper bounds on the event propagation delay across actors. This enables correct event processing for timed models with discrete-event semantics, even in the presence of out-of-order events. However, PTIDES requires a model-time and a real-time delay function to be specified by the developer, including network latency bounds, which is unfeasible for non real-time CPS networks whose configuration can change at runtime. In contrast, Khronos computes timeouts for events relying only on its past observations of event arrival times, thus automatically adapting to changes in the CPS infrastructure without requiring the specification of delay models and/or bounds in advance.

Significant research efforts focus on the management of late event arrivals in Complex Event Processing systems, which can be categorized into punctuation, buffering, and speculation techniques. To date, several studies have investigated punctuation techniques to safely process a group of events that reside within the same time window [18, 32, 34, 35, 54, 54, 58]. A punctuation is a special event that is injected into a data stream to indicate the end of a subset of that stream. As a result, stream operators can safely process that subset of the stream with the knowledge that no future events will arrive out-of-order with respect to events belonging to that subset of the stream. However, most punctuation-based techniques rely on the developer and/or on *a priori* knowledge, e.g., transmission delay upper bounds, to insert punctuations, which limits their applicability in highly dynamic CPS scenarios. These techniques could benefit by integrating with Khronos to overcome this limitation while simultaneously providing guarantees on the amount of missed events.

A number of authors have considered the effects of buffering events and postponing their processing to ensure reliable results [28–30, 38]. While reliable, buffering techniques can introduce high memory overhead and latency, which can be detrimental in the context of real-time stream processing. The authors in Reference [39] focus on low latency event detection by reducing the stream processing delay through the adaptive parallelization of stream operators. Their solution does not consider the impact of large waiting times for event arrivals, which can negatively affect the event detection latency. Previous studies [28–30] have explored the relationships between result latency and accuracy for specific stream operators, striving to achieve a right balance by dynamically adjusting the buffer size at runtime based on the network delay [60]. However, these approaches are operator specific, which can be problematic for expressing complex CPS applications. In contrast, Khronos not only provides a generic solution for managing late event arrivals that does not target specific operators, but it also takes into account changes in the packet intergeneration delay.

A significant amount of research has explored speculation techniques to handle late event arrivals [16, 18, 40, 48]. Unlike buffering, speculation techniques do not stall the processing of events within the operator's input buffer, which can lead to significant latency improvements. However, the presence of late event arrivals causes the output stream to be rolled back and recomputed, which can incur high performance penalties both in terms of result quality and latency [43, 47]. Furthermore, previous work in this area assumes reliable data sources that can reproduce their events for rollback recovery [15, 31], which is typically not the case with resource-constrained sensing devices. Recent research efforts [43] are combining speculation with buffer-based

techniques to reduce result latency, but unlike Khronos they do not provide clear control over the result accuracy.

Rivetti et al. [47] address the lack of support of the above mentioned techniques in trading off result latency with quality by proposing ProbSlack, a probabilistic approach to deal with late event arrivals. Probslack adds a dynamic offset, based on its models of the packet inter-generation delay and network delay, to a user-specified query frequency that determines how often events are processed. However, device sampling periods can change at runtime in the context of dynamic CPS environments, which is why Khronos does not rely on the user to specify query frequencies in advance. Furthermore, ProbSlack relies on a user-specified period T to refresh its models for the two delays, which clearly impacts the performance of the approach. It is unclear from the article how users should configure this period to ensure low result latency while satisfying the result quality constraints in the presence of dynamism.

Requirements

In the context of the industrial use case and the related work, we identify five requirements for CPS middleware:

- **A**) The programming language should support fine-grained specification of application behavior not only for events that arrive on time, but also for events that arrive too late with respect to the application requirements, in a non-terminal fashion. The middleware should support the registration of completeness constraints for individual and groups of device data streams, through a set of provided services.
- **B**) The middleware should not rely on the developer's knowledge of the underlying infrastructure and require no further manual configuration after deployment.
- **C**) The middleware should adapt to changes in the CPS infrastructure to satisfy the application constraints in the face of network and application dynamism.
- **D**) The middleware should satisfy the application constraints for a wide variety of different infrastructures and application requirements.
- **E**) The middleware should provide CPS applications with context regarding the completeness and timeliness.

4 ARCHITECTURE

In this section, we describe the design of the proposed reactive extensions, along with the underlying prediction technique and architecture of Khronos.

4.1 Reactive Extensions

The proposed reactive extensions, described later in this section, are built on top of observable variables, observers and operators, as shown in Figure 4, because they provide a suitable basis of elegantly expressing application logic for handling emissions of a single event type [21, 37]. Traditionally, an observable represents a data stream that emits events in a synchronous or asynchronous fashion. Observers can subscribe to such events and be notified when (a) an event is emitted by the observable, (b) the stream is completed, and (c) the stream terminated due to an error. Operators can be attached to observables and be chained to one another, performing an operation on the input whenever an event is emitted and passing the result along the chain until it reaches the observer(s).

The traditional model of observables, as described above, terminates the stream once it has ended its lifecycle and otherwise forwards emitted events to its subscribers. The lifecycle of a stream can end without errors (completion) or due to an error. In our work, streams cannot complete, since the sensing devices generate messages as long as they are powered. The stream

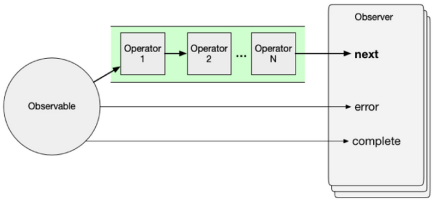

Fig. 4. Observables, operators, and observers in ReactiveX.

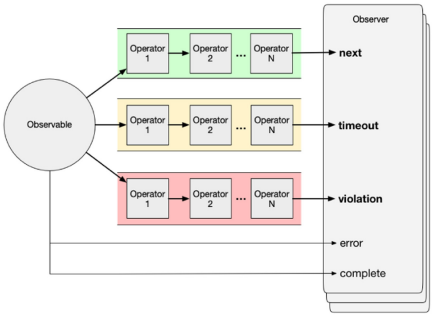

Fig. 5. Observables, operators, and observers in extended ReactiveX.

however can terminate with an error when the device goes permanently offline, e.g., due to depleted battery. A strong indicator of a device being permanently offline is the occurrence of a timeout event under a large completeness constraint (i.e., $\rho \rightarrow 1.0$ for the corresponding device data stream).

However, this is not expressive enough to describe CPS applications that need to deal with timeouts and completeness constraint violations in a non-terminal fashion, e.g., in the industrial use case example described in Section 2. Thus, we extend the traditional observable model with two additional non-terminal flows for `timeout` and `violation` events, as shown in Figure 5. In addition to notifying its observers when a new value is emitted, i.e., a `next` event, an observable emits a `timeout` and `violation` event to its subscribed observers when a timeout has occurred or a completeness constraint is violated, respectively. Developers can specify isolated chains of operators for each of the `next`, `timeout`, `violation` flows, improving expressiveness and enabling fine-grained execution of application logic, as specified by requirement **A)**.

4.2 Prediction Technique

Khronos automatically (re)computes the timeouts of event (sensor packet) arrivals, based on the application completeness constraint(s) for that device. These timeouts are determined using an approach similar to the Retransmission TimeOut (RTO) timer in the Transmission Control Protocol (TCP), a well-established transport layer protocol for communications over the Internet [56].

TCP's RTO is a durable solution that works on top of a wide, heterogeneous and dynamic infrastructure and also tackles the problem of determining timeouts for non-deterministic packet arrivals. Both RTO and Khronos are faced with a similar challenge: determining how long to wait for an event arrival before taking action, in the presence of varying network latency. In both cases, the tradeoff between timeliness and completeness depends on these timeouts. RTO's approach is simple and lightweight, since it uses exponentially weighted moving averages instead of storing past observations to compute the timeout in every step, which fits the resource-constrained model of CPS. In RTO, however, it is the sender who needs to determine how long to wait after sending a packet before deciding that the corresponding acknowledgement will not arrive (timeout). In the context of industrial monitoring, it is the receiver (application) that needs to decide how long to wait for a message from the sender, which transmits periodically. Since the application does not know exactly when/if the message is sent, in contrast to RTO, the sender's packet inter-generation delay also needs to be taken into account. Finally, CPS applications can have flexible completeness constraints, while RTO's design is limited to covering 99% of all packet arrivals.

In TCP, the RTO needs to determine how long to wait for the acknowledgment to arrive after a segment has been sent, before re-transmitting the segment. Short timeouts result in unnecessary

re-transmissions and possibly network congestion, while long timeouts negatively impact performance. RTO keeps track of two exponentially weighted moving averages: the smoothed round-trip time (SRTT) and round-trip time variance (RTTVAR), with smoothing factors $\alpha = 7/8$ and $\beta = 3/4$, respectively, as specified in RFC 6298 [49]. SRTT is the best current estimate of the round-trip time to the destination, and RTTVAR is the variance in round-trip times. The timeout is computed as $RTO = SRTT + K * RTTVAR$, where $K = 4$ based on the observation that typically less than 1% of all packets arrive more than four standard deviations too late. The role of K in this formula is to over-provision by adding that many times the variance to the mean to cover over 99% of packet arrival times [56].

Khronos approach

In the context of our use-case, Khronos needs to determine how long it should wait for each packet to arrive such that the application completeness constraints are satisfied for each device data stream without unnecessarily long timeouts. For each completeness constraint ρ for a device data stream σ^s, Khronos computes the smoothed arrival time $S(e_i^s)$ and the arrival time variance $\mathbb{V}(e_i^s)$, whenever a new event e_i arrives at timestamp t_i. $S(e_i^s)$ is the best current estimate for the next event arrival time and $\mathbb{V}(e_i^s)$ the variance in arrival times. These are computed by the formulas:

$$S(e_i^s) = \alpha S(e_{i-1}^s) + (1 - \alpha)R(e_i^s), \tag{6}$$

$$\mathbb{V}(e_i^s) = \beta \mathbb{V}(e_{i-1}^s) + (1 - \beta)|S(e_{i-1}^s) - R(e_i^s)|, \tag{7}$$

where $R(e_i^s)$ is the actual arrival time of the event at timestamp t_i and α, β the smoothing factors, set to the same values as in RTO, which are empirically derived. A timeout based on $S(e_i^s)$ alone is too inflexible for large variance in arrival times, which is accounted for by $\mathbb{V}(e_i^s)$. The timeout $\tau^{e_i^s}$ for the next event at timestamp t_{i+1} is computed as

$$\tau^{e_i^s} = S(e_i^s) + K_\rho * \mathbb{V}(e_i^s), \tag{8}$$

where K_ρ is the K value for completeness constraint ρ. As in RTO, K_ρ determines how sensitive the timeout is to packet arrival time variance. Intuitively, large values for K_ρ lead to large timeouts and thus larger prediction errors but fewer missed events. Unlike RTO, our model supports various completeness constraints ρ, for each of which we determine a corresponding value for K_ρ, as described later in this section. The computation cost of our approach is linear ($O(n)$), where n is the number of registered completeness constraints. Concretely, whenever a packet arrives from a device, Khronos performs, per completeness constraint for that device, five multiplications and five additions to compute the next timeout.

Determining K_ρ

In this section, we propose a methodology to determine a one-to-one mapping between K_ρ and a completeness constraint $\rho \in \langle 0.1, \ldots, 1.0 \rangle$. Once K_ρ is determined, timeouts can be directly computed according to the equations in Section 4.2 without the need for further configuration. The goal is to find the smallest K that satisfies ρ across all device data streams in the network both (a) under normal operation and (b) in the presence of external disturbances. For the former, we monitor the underlying network of sensing devices over a period of one week to collect a representative set of event arrival times over all the sensing devices, noted as O_n.

For the latter, we identify three types of disturbances in the context of our use case, which correspond to changing the device sampling period, network size and network latency respectively. These types of disturbances are common for our use case, but are also typical in the context of large CPS deployments. The sensing device sampling period is decreased and increased in a stepwise fashion. This change corresponds to a common scenario where device sampling periods

can decrease during peak operation to sample more frequently, and then increase during normal operation to save energy. The network size is reduced by turning off a part of the sensing devices, and then restored by powering them up. This corresponds to a common scenario in any real CPS deployment where devices can arbitrarily leave and (re)join the network later, e.g., due to interference causing temporary connectivity errors. Finally, the network latency is increased by reconfiguring the allocated bandwidth assigned to each of the sensing devices, which is often done in practice to ensure that the network has enough resources to support a large number of devices. For each disturbance type, we monitor the network over the period of 1 week, before and after introducing that disturbance, to collect a representative set of event arrival times, noted as O_d. The total set of observed event arrival times during the entire monitoring period is denoted as $O = O_n \cup O_d$. While in our case monitoring the network over a week was enough to collect a representative set of observations (around 200.000), this monitoring period depends on the device sampling periods.

After the monitoring period is finished, K is incrementally increased in small steps, starting from zero, until the resulting timeouts satisfy the given completeness constraint ρ over O, noted as $K_{\rho,min}$. Naturally, larger ρ impose stricter limits to the number of missed events, leading to larger K values. Finally, we overprovision to limit the impact of overfitting and to improve robustness by setting K_ρ equal to

$$K_\rho = 2 * K_{\rho,min}. \tag{9}$$

As a result, developers can deploy and use the middleware without the need of further configuration, as stated by requirement **C)**. A drawback of doubling $K_{\rho,min}$ is that it leads to large prediction errors in the extreme case where $\rho \to 1.0$, which can be impractical for some applications, e.g., with both strict time and certainty requirements in fault detection. The resulting K_ρ values are discussed in Section 5.3, and the correctness of the approach is evaluated in Section 6.

4.3 Middleware

The proposed middleware acts as a generic bridge between the underlying CPS infrastructure and the applications that run on top of them. The identified requirements that Khronos addresses are highlighted in Section 3. Khronos' architecture and key responsibilities of each component are discussed in Section 4.3.1. Next, the provided Application Program Interface (API), which enables CPS applications to specify completeness constraints for device data streams, is explained in Section 4.3.2.

4.3.1 Components. Figure 6 shows a complete overview of Khronos' architecture. Khronos acts as a generic bridge between external CPS applications and the gateways of the underlying CPS infrastructure. The middleware components are divided into three layers, based on their responsibilities: CPS Communication, Time Management, and Application Management. The rest of the section describes these responsibilities and the role of each component in greater detail.

CPS Communication. This layer is responsible for managing communication between Khronos and the underlying CPS network(s). The `Gateway Manager` is responsible for maintaining an overview of the underlying `Gateways` in the CPS, enabling communication between the CPS network and the middleware. It listens for published sensor data from each `Gateway` and forwards it to the `Data Parser` for parsing. Assigning the responsibility of parsing raw messages to a separate component improves extensibility for new message formats in the future. The parsed data are then passed to the `Network Monitor` and the `Time Management Layer`. The `Network Monitor` maintains an overview of network statistics, including the discovered devices and communication latency between Khronos and the `Gateway`(s). The former is necessary to verify that incoming requests refer to operational devices, while the latter is included in the resulting timeouts for incoming messages from sensing devices in the CPS. For industrial-scale networks with a

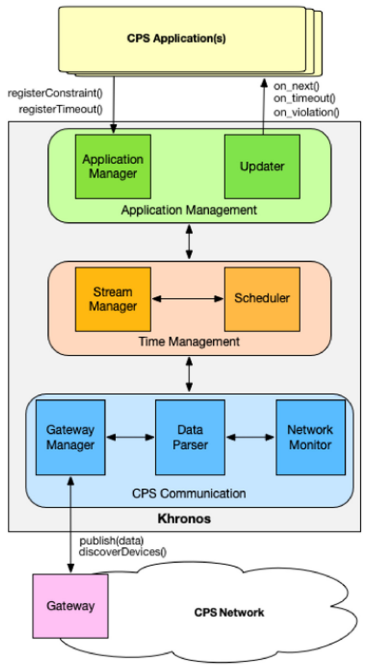

Fig. 6. Khronos component diagram.

large number of gateways, the modularity of this layer can be exploited to improve scalability, e.g., by distributing its components across the gateways and thus decoupling the CPS communication logic from the rest of the middleware.

Time Management. This layer is responsible for processing network statistics and packet arrival times to coordinate the callbacks of each CPS application. The `Stream Manager` maintains completeness statistics and determines the timeouts per device data stream for the completeness constraint(s) specified by the CPS application. This enables fine-grained automatic timeout predictions for each completeness constraint that is associated with an individual sensor data stream. The `Scheduler` coordinates the notifications to the application layer using the above timeout(s) through the `Application Management` layer, whenever a packet arrives or a timeout is exceeded, complying with requirement **F)**.

Application Management. This layer is responsible for communication between Khronos and external applications. The `Application Manager` provides applications with the API described in Section 4.3.2 and is responsible for registering the completeness constraint(s) and/or static timeouts to the `Scheduler`. The `Updater` is responsible for notifying the registered applications when to execute the `on_next`, `on_timeout`, and `on_violation` application logic, as coordinated by the `Scheduler`. Grouping the notification logic in a separate component improves modularity and simplifies support for multiple notification schemes.

4.3.2 Application Programming Interface. Khronos offers developers a simple API, which consists of two operations:

- `registerCompleteness(device, constraint, on_next, on_timeout, on_violation)`
- `registerTimeout(device, timeout, on_next, on_timeout)`

The registerCompleteness(..) method addresses requirement **B)** and takes five arguments: the device, the completeness constraint, and three callback methods. device:<String> identifies the CPS device data stream(s) and can be a unique identifier (e.g., serial number) or a wildcard (e.g., sensor type) that refers to a group of devices. constraint is the value for the completeness constraint, expressed as a fraction. Given the completeness constraint, the middleware updates the timeout for the next packet, whenever a packet has arrived from the corresponding device. on_next(value:<Sensor Data>, timeout:<Double>, completeness:<Double>) is the callback method that is invoked by the middleware whenever data from the specified device arrives on time. It takes as arguments the value of the arrived sensor data and the corresponding timeout and completeness. on_timeout(timeout, completeness) is the callback method that is invoked by the middleware whenever the timeout is reached and no sensor data have arrived. It takes as arguments the value of the timeout and the current completeness. on_violation(value, timeout, completeness) is the callback method that is invoked by the middleware whenever the completeness is below the constraint when the timeout is reached or a packet has arrived. It takes as arguments the value of the sensor data (if any) and the current timeout and the completeness. For example, a simple application can define on_next(...) to update the average temperature whenever new temperature data arrives, on_timeout(...) to count the number of occurred timeouts and on_violation(...) to spawn a pop-up alert window upon constraint violation.
registerTimeout(..) enables developers to register a static timeout for a sensor device data stream and takes four arguments: the device, the static timeout, and two callback methods, which contain application logic and are thus specified by the developer. device: <String> identifies the CPS device data stream(s) and can be a unique identifier or a wildcard that refers to a group of devices. timeout is the value for the static timeout for packet arrivals from the given device, expressed in time units (e.g., seconds). Given the timeout, the middleware recomputes the completeness for the given device whenever it receives a new packet from it. on_next(value, timeout, completeness) is the callback method that is invoked by the middleware whenever data from the specified device arrives on time. It takes as arguments the value of the arrived sensor data, the given timeout, and the current completeness, addressing requirement **F)**. on_timeout(timeout, completeness) is the callback method that is invoked by the middleware whenever the timeout for packet arrival from this device is reached. It takes as arguments the current timeout and completeness.

5 IMPLEMENTATION

This section discusses the key technologies that implement the reactive programming extensions, the underlying CPS network, and the middleware.

5.1 Reactive Extensions

ReactiveX is implemented in most modern programming languages [1]. In this article, we focus on extending RxJS, an implementation of ReactiveX in TypeScript, which is then compiled to JavaScript. JavaScript is a programming language often used for the development of front-ends that deal with asynchronous streams and events, thus an interesting choice for implementing the display functionality described in our industrial use case.

The extensions and modifications to RxJS are shown in Figure 7, where newly added classes, attributes and/or methods are highlighted in green. Note that the Observable<T> and the Subscriber<T> classes, with T a generic type, correspond to the Observable and the Observer in Figures 4 and 5, respectively. A Subject<T> is a special type of Observable<T> that allows events to be multicasted to many Subscriber<T> objects. The Orchestrator<T> is subclass of Subject<T> and is responsible for connecting to the middleware and for parsing and

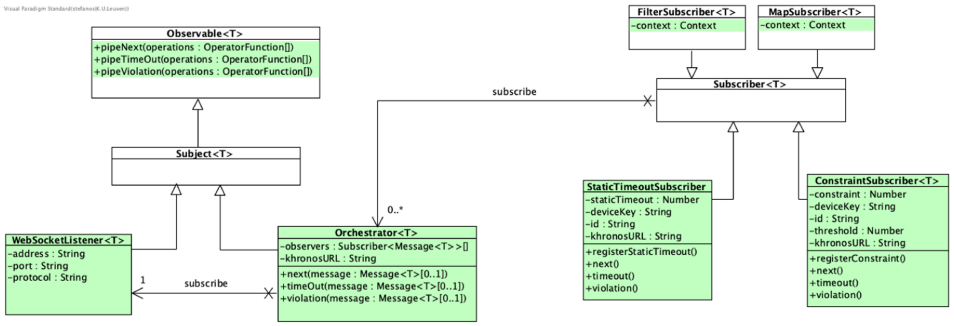

Fig. 7. Simplified class diagram of reactive extensions to RxJS.

forwarding incoming events to the corresponding subscribers. It creates and subscribes to a `WebSocketListener<T>` subject to connect to the middleware websocket, listening for incoming next, timeout, or violation event messages. These events are sent as JSON objects of type `Message<T>` with following attributes: the `id` of the data source related with the event, the `value` of type T, the `completeness` of the device data stream, the `timeout` for the next event, the `timestamp` of arrival, and the event `type` (next, timeout or violation).

Upon arrival of an event, the `WebSocketListener` notifies its subscriber (`Orchestrator`), who then filters and notifies the relevant subscribers based on the event `id` attribute. The event `type` is used by each chained `Operator` to determine if it should perform its operation and forward the result or just pass the message down the chain, as shown for `MapSubscriber<T>` and `FilterSubscriber<T>`. This enables operators to be associated only with a particular event `type`, which is passed to their constructors as an argument from the methods `pipeNext(...)`, `pipeTimeOut(...)` and `pipeViolation(...)`.

Finally, completeness constraints and static timeouts can be registered by creating `CompletenessConstraintSubscriber` and `StaticTimeoutSubscriber` objects, respectively. When a next, timeout or violation event occurs, the `next()`, `timeout()` or `violation()` functions of the subscribers are called, respectively, after the relevant operators have performed their operations on the input.

Use Case: Implementation

This section discusses the key elements for implementing the manufacturing plant monitoring application, described in Section 2, using the proposed extensions. The complete implementation is available online, along with the rest of the source code. The application requires: (a) visualizing the operational efficiency in a timely fashion and (b) notifying the operator when the number of defect airbags exceeds a threshold, to stop the production process, or when a timeout has occurred, to check the status of the sensor or the network.

First, to visualize the operational efficiency of the plant, six `ConstraintSubscriber` objects are created, one for each of the production line sensors. Listing 1 illustrates the initialization of such an observer for line L1, which registers a low completeness constraint (0.75) for its sensing device to achieve faster updates. This constraint is specified by the developer, and it is interpreted as follows: The middleware waits long enough for the sensor data of L1 to arrive, so that it is present in at least three out of four updates of the operational efficiency. When the sensor data are not present, the previously received value is used instead, leveraging the fact that the processing rate of the production lines varies slowly over time. These `ConstraintSubscribers` react to `next`, `timeout`, and `violation` events, produced by Khronos, by (a) updating the operational efficiency when an

```
var productionLine1 = new ConstraintSubscriber("3303/5702:00-17-0D-00-00-30-E7-2D|
    Item Processing Rate", 0.75, 0.99,
function(v){nextProductionLine(v,'L1')},
function(err) {console.error('L1 ' + err)},
function() {console.log('L1 Completed!')},
function(v){timeoutProductionLine(v,'L1')},
function(v){violationProductionLine(v, 'L1')});
```

Listing 1. Initializing a `ConstraintSubscriber` for sensor at L1 with a 0.75 completeness constraint, used for computing operational efficiency.

```
var trashAisle = new ConstraintSubscriber("3303/5702:00-17-0D-00-00-30-E7-2D|
    Object Detection", 0.9999, 0.99,
function(v){nextTrashAisle(v)},
function(err) {console.error('TA' + err)},
function() {console.log('TA Completed!')},
function(v){timeoutTrashAisle(v)},
function(v){violationTrashAisle(v)});
```

Listing 2. Initializing a `ConstraintSubscriber` for sensor at the trash aisle with a 0.9999 completeness constraint, used to detect faults in the manufacturing process.

```
var orchestrator = new OrchestratorSubject();
var chain = orchestrator.pipeNext(map(x => log(' Sensor data received: ', x)),map(
    x => storeValue(x)));
var chain = chain.pipeTimeOut(map(x => log(' Timeout event received: ', x)));
var chain = chain.pipeViolation(map(x => log(' Violation event received: ', x)));

chain.subscribe(productionLine1);
chain.subscribe(productionLine2);
chain.subscribe(productionLine3);
chain.subscribe(productionLine4);
chain.subscribe(productionLine5);
chain.subscribe(productionLine6);
chain.subscribe(trashAisle);
```

Listing 3. Subscribing the observers after chaining operators for `next`, `timeout`, and `violation` events.

event is present from each production line and (b) writing the result to a database, which is used by a third-party data visualization tool (e.g., Grafana [5]) to create and update the display. This functionality is realized by the `nextProductionLine(...)`, `timeoutProductionLine(...)`, and `violationProductionLine(...)` callbacks, shown in Listing 1.

Second, to notify the operator, a `ConstraintSubscriber` object is created for the sensor at the trash aisle, with a very high completeness constraint (e.g., 0.9999), as shown in Listing 2. This observer reacts to incoming `next`, `timeout`, and `violation` events using the `nextTrashAisle(...)`, `timeoutTrashAisle(...)`, and `violationTrashAisle(...)` callbacks. The callback function `nextTrashAisle(...)` checks whether the measured number of defect items exceeds the threshold and if so notifies the operator. The latter two callbacks notify the operator that a `timeout` and a `violation` event occurred, respectively.

Finally, Listing 3 shows how the developer can chain operators to the `Orchestrator` subject for each of the `next`, `timeout`, and `violation` flows. Whenever an event arrives, a message is written to a log file for debugging purposes. Additionally, in the case of a `next` event, the `storeValue(...)` map operator stores the sensor value in a key-value map (dictionary), to be used in the future in

case of a timeout. The developer then subscribes the observers to the `chain` subject, so that they listen to each of the event streams. This illustrates how a non-trivial application, with multiple completeness constraints, can be implemented in a concise fashion using the proposed extensions, without requiring the developer to specify any timeouts at compile time. The resulting display of running the application is shown in Figure 3.

5.2 Network

In industrial CPS applications, like the industrial use case in Section 2, two widely used network technologies are Time Slotted Channel Hopping (TSCH) and Carrier-Sense Multiple Access (CSMA) mesh networks. In a TSCH mesh, all motes are precisely synchronized to tens of microseconds. Time is organized in slots that are allocated to motes in the network, allowing them to know in advance when to turn the radio on or off. Frequency bands are separated in channels, and communications are done using those different channels at different times, resulting in reliable, low-power communication. In CSMA networks, motes sense the shared medium before transmitting to verify the absence of other traffic. In wireless networks, CSMA is often enhanced with Collision Avoidance (CSMA/CA) to improve performance, where motes wait for a random period of time after sensing that the medium is not free, before retrying.

In the context of our use-case, a wireless mesh network is deployed to connect the underlying CPS devices. Mesh networks introduce increased complexity when dealing with network latency due to multi-hop communication. That is why we focus on a wireless mesh network and do not use LoRa or BLE for the implementation. The key technology used for the underlying wireless embedded network is SmartMesh IP (SMIP) [2], which is broadly used in industrial CPS applications, such as the manufacturing plant described previously in the use case.

By default, a SMIP network is a TSCH mesh but it can be easily reconfigured to a CSMA mesh through the `bbmode` setting. In the evaluation, we use this parameter to test our implementation on top of both a TSCH and a CSMA/CA wireless mesh. A SMIP network is a wireless, multi-hop mesh network that self-forms and self-maintains to guarantee high network reliability and ultra low-power. Due to this self-adaptation, several network parameters, including allocated bandwidth, latency, and hop-depth, can change over time without any system parameter reconfiguration, leading to non-deterministic packet arrival times.

For this article, a real-life testbed is built that consists of 34 physical devices, including the gateway. More concretely, there are 22 SmartMesh IP motes [3] (DC9003A-B), 11 VersaSense wireless devices [9] (Model P02), and one VersaSense Edge Gateway [8] (Model M01). The SmartMesh IP motes are not equipped with sensors: Their role is to act as routers that forward packets they receive across the network, enabling a widespread deployment with a large number of hops. The VersaSense wireless devices are built on top of SmartMesh IP and provide plug-and-play support: Up to four sensors or actuators, known as peripherals, can be connected on each VersaSense device. Each VersaSense device is also equipped with a built-in peripheral that measures the battery-level. As a result, there are in total 22 device data streams, each corresponding to a different peripheral. The VersaSense Edge Gateway, which is the network manager, acts as a bridge between the wireless sensor network and Khronos.

Table 1 shows the types of peripherals that are deployed in the testbed along with their quantities and default sampling periods. Each VersaSense device is equipped with at most one peripheral of the same type. These peripherals are fully self-identifying, requiring no further manual intervention. In the rest of the article, we use the term "device" to refer to a peripheral connected to a VersaSense device. It is uniquely defined by the peripheral identifier and the IPv6 address of the VersaSense device.

Table 1. Deployed Peripherals and Their Settings

Identifier	Peripheral Type	Quantity	Sampling
3302/5500	Sensor (Presence)	1	10 s
9803/9805	Sensor (Light)	3	120 s
3303/5702	Sensor (Temperature)	3	120 s
8040/8042	Sensor (Pressure)	3	60 s
9903/9904/2	Sensor (Thermocouple)	1	10 s
1010/9000	Sensor (Battery)	11	900 s

5.3 Middleware

We implemented Khronos on a Raspberry Pi 3, because it is a gateway-class device in terms of memory capacity and processing power, which is important for our industrial use-case. It is developed in Python v3.6 as a Representational State Transfer [6] (REST) server, using the `flask` [4] framework and implements the API that was described in Section 4.3.2. REST is a stateless communication protocol that separates the concerns of the client and server, enabling transparent communication between software systems. In this proof-of-concept implementation, Khronos communicates with applications using remote method invocation (RMI) and/or websockets. Pyro 4.6 is a python library that enables remote method invocation (RMI) on objects that are created and stored locally by client applications. These objects implement the callback methods discussed in Section 4.3.2: `on_next(...)`, `on_timeout(...)` and `on_violation(...)`. Clients application(s) register these objects to the Pyro name server, which provides them with a URI per object. Application(s) provide this URI as an argument to Khronos when calling the provided API methods, instead of directly passing the callback functions, leading to easier client-server integration. Khronos invokes each of the callbacks accordingly, based on whether or not constraint violation occurred, which then executes the corresponding method locally on the client machine. Alternatively, external applications can connect to a websocket that Khronos offers, which emits tagged events whenever a packet arrives or a timeout or constraint violation occurs.

The Raspberry Pi is in the same local area network (LAN) as the Versasense Edge Gateway. Khronos obtains the relevant network status information from the Versasense Edge Gateway through a CoAP [7] API, which is a client-server model similar to REST but designed for resource-constrained devices. Additionally, the VersaSense Edge Gateway listens for connections to a websocket [10], which enables full-duplex communication over a single TCP connection. Khronos connects to the websocket to receive the raw sensor data stream, which is processed by the rest of the middleware.

Resulting K_ρ

Khronos uses the technique discussed in Section 4.2 to automatically compute the timeouts for individual packet arrivals. K_ρ is used in formula (8) to determine the sensitivity to packet arrival time variance. Based on the described methodology, we determine K_ρ for a wireless TSCH mesh network. The same values can be used for Khronos on top of a CSMA/CA wireless mesh network, as shown in the experiments performed in Section 6. For other network technologies, such as LoRa and BLE, recomputation of K_ρ might be necessary, using the methodology described in Section 4.2. The resulting K_ρ values are shown in Table 2 for various completeness constraints ρ. Intuitively, since K_ρ determines the sensitivity of the timeout to change, the higher the completeness constraint, the larger the resulting K_ρ. For $\rho = 1.0$, K_ρ is in theory infinitely large so that packets always arrive on time. In practice, the results show that for $\rho = 1.0$, the ratio of missed events saturates at 0.003 or 0.3% for $K_\rho >= 300$.

Table 2. K_ρ Values for Different Completeness Constraints ρ

ρ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
K_ρ	0	0.1	0.6	1	1.2	1.4	2	2.8	4.6	300

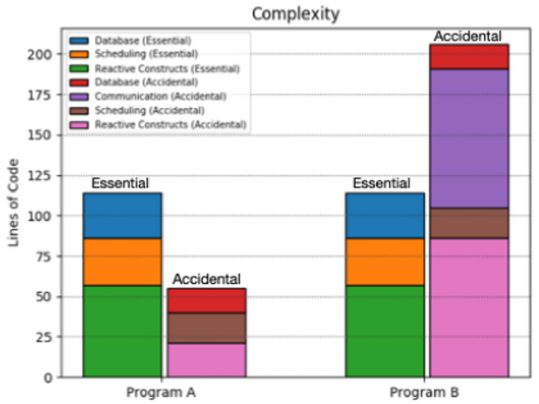

Fig. 8. Accidental and essential complexity percentages for RxJS with (Program A) and without (Program B) time management extensions.

6 EVALUATION

This section discusses the evaluation of the proposed language extensions and Khronos, focusing on three key aspects based on the requirements in Section 3: (1) the complexity of the resulting CPS application, (2) the performance of the resulting timeouts in the presence of application and network heterogeneity, and (3) the ability to adapt to network and application dynamism. The proposed extensions are evaluated by comparing two equivalent implementations of our industrial use case example, with and without the proposed extensions, discussed in Section 6.1. The performance of Khronos in the presence of heterogeneity and dynamism is evaluated by conducting an extensive set of experiments on a physical testbed. The evaluation metrics and the approaches against which Khronos is compared are discussed in Section 6.2. Finally, the empirical evaluation results are presented in Section 6.3 for each set of experiments, with a deeper discussion in Section 6.3.2.

6.1 Reactive Extensions

To evaluate the proposed reactive extensions, we implemented the industrial use case example, described in Section 2, using RxJS with and without the proposed time management extensions, resulting in programs A and B, respectively. We compare the two programs in terms of code size and complexity, using the notions of essential and accidental complexity [17]. As essential complexity, we consider the application-specific functionality: (a) computing the operational efficiency, which includes synchronizing the different sensor inputs; (b) updating the display, which involves writing to a database; and (c) the corresponding callback methods to handle incoming *next*, *timeout*, and *violation* events. As accidental complexity, we consider (a) the communication management between the application and the CPS, because it is overhead that is not a property of the problem at hand, (b) the creation of the reactive objects and their dependencies, as well as (c) helper functions to the scheduling (e.g., retrieving a message from the queue based on an identifier). The measured essential and accidental complexity of the two programs is presented in Figure 8.

Table 3. Lines of Code Comparison of Programs A (with Extensions) and B (without) Grouped in Terms of Functionality

				Reactive Constructs			
	Communication	**Database**	**Scheduling**	**Instantiation**	**Callbacks**	**Operators**	**Total LOC**
Program A	0	43	48	21	49	8	**169**
Program B	86	43	48	27	108	8	**320**
Percentage Change	−100%	0%	0%	−22.22%	−54.63%	0%	**−47.19%**

The results show that the proposed extensions reduce the accidental complexity of the program by 73.3%. This is largely due to eliminating (a) the communication overhead with the underlying CPS, which is now managed automatically by the extensions (the `Orchestrator` object), and (b) by eliminating the need for case splitting on the sensing device identifier inside the callback functions, by creating a `ConstraintSubscriber` object for each device data stream. The remaining accidental complexity is due to the import statements for the database and RxJS modules, the database and reactive object creation and the queue manipulation helper functions that are used for scheduling.

Table 3 provides an overview of the code size of the two programs, per block of functionality: communicating with the middleware, writing to the database, and scheduling the events to compute the operational efficiency. Additionally, we measured the lines of code for creating the needed dependencies between the reactive objects (observables and observers). The results show that using the proposed reactive extensions decreases the size of the resulting program by 47.19%. Thus, the proposed reactive extensions lead to programs that are less complex and more concise, which reduces the overall developer effort.

6.2 Setup

All experiments conducted in this evaluation are performed on top of data collected from the real-life testbed, as discussed in Section 5.2. The quantity and types of sensors are shown in Table 1. The experiments aim to extensively evaluate Khronos across two dimensions: heterogeneity and dynamism, both typical for CPS networks. Its performance is evaluated against the approaches described in Section 6.2.1, using the metrics discussed in Section 6.2.2. Unless specified otherwise, the experiments use the default topology of the testbed, which is shown in Figure 9.

6.2.1 Approaches. Khronos is compared against three state-of-the-art approaches [47, 54] that use a fixed timeout per device data stream:

DSP (Double Sampling Period). DSP sets the timeout for each packet arrival from a device equal to twice its sampling period. This leads to significantly large timeouts, ensuring high completeness at the expense of timeliness.

SPND (Sampling Period Network Delay). SPND sets the timeout for each packet arrival equal to the device sampling period plus the average network delay. This typically leads to smaller timeouts compared to DSP, at the expense of completeness.

STO (Static Timeout Oracle). STO is a theoretical approach that knows in advance all the packet arrival times from each device. STO computes a fixed timeout based on the completeness constraint for each device data stream. The timeout is equal to the smallest value that satisfies the given constraint for that device data stream across the experiment.

Khronos. For each completeness constraint, Khronos automatically computes timeouts for the next packet arrival from the corresponding device whenever a packet arrives, as discussed in Section 4.2.

DSP is an example that opts for high completeness, where fixed timeouts are set arbitrarily large enough and SPND opts for timeliness, where timeouts are equal to the device sampling period plus

Fig. 9. Default topology of the physical testbed, spanning throughout three building floors.

a fixed offset, e.g., the average network latency. In practice, only DSP, SPND, and Khronos can be used, since STO requires perfect knowledge of the future to compute the timeouts. However, STO is a reference benchmark as it demonstrates the best possible performance when using fixed timeouts under perfect information.

6.2.2 Metrics. We measure the performance of each approach for a device data stream σ^s using two evaluation metrics: the prediction error $\epsilon(\sigma^s)$, as defined in Section 2, and the constraint violation ν, which is defined next.

Constraint Violation (ν). This is the percentage (%) of packet arrivals for which the constraint is violated. In theory, to ensure that extremely large completeness constraints $\rho \rightarrow 1.0$ are always satisfied, the corresponding timeouts would be quasi-infinitely large. This is impractical, since timeouts should still occur within a finite amount of time. Thus, we slightly relax the definition of completeness constraint satisfaction in Section 2 by tolerating that $\gamma(\sigma^s) < \rho$ for at most 0.001% of event arrivals, where $\rho \in [0..1)$. In other words, a completeness constraint is satisfied when over 99.999% of the time, the measured completeness $\gamma(\sigma^s)$ for σ^s is greater or equal to the completeness constraint ρ, or equivalently while $\nu \leq 0.001\%$. For the extreme case where $\rho = 1.0$, the best approach is the one with the smallest constraint violation ν and the smallest prediction error (best-effort). Each approach is evaluated for completeness constraints $\rho \in \langle 0.1, \ldots, 1.0 \rangle$. Unless specified otherwise, by default the results are illustrated for a completeness constraint $\rho = 0.8$. The default values used for the most important gateway configuration parameters are shown in Table 4.

6.3 Results

This section provides an overview of the performed experiments and results, comparing Khronos against the approaches discussed previously in Section 6.2.1. The experiments evaluate Khronos across two dimensions: dynamism and heterogeneity.

Table 4. Default SMIP Network Manager Configuration Parameters [57]

Parameter	txpower	basebw	numparents	bbmode	bbsize	bwmult
Value	8	50000	2	0	1	1000

6.3.1 Dynamism. This section evaluates the capability of Khronos to satisfy application completeness constraints in the presence of network and application dynamism, as specified by requirement **C)**. From the use case and the literature study, we identify three sources of dynamism. First, applications can change device sampling periods over time to achieve their goal. Second, devices can leave (join) the network at any time, which results in a smaller (larger) network size, measured by the number of operational motes. Third, network parameters can be re-configured on the spot to impact the network latency. In the case of SMIP, the re-configuration requires a reset of the network manager to take effect. In the experiments to follow, devices are deployed across three floors of our departmental building, up to one floor away from the gateway, as is shown in Figure 9. The acquired results are shown for an arbitrarily selected device, but the same trend holds for all 22 sensing devices.

Impact of Network Size

In this experiment, we test the hypothesis that Khronos can consistently satisfy application completeness constraints in the presence of network size dynamism, by turning off and on 66.67% of the devices. Figure 10(a) shows the impact of changing the network size on the constraint violation and prediction error, over a period of five hours, for a sampling period of 10 s and $\rho = 0.8$. The left and right arrows in Subplot 3 indicate the two events: reducing and increasing the network size, respectively. Khronos reacts to the changes by increasing the smoothed arrival time variance (Equation (7) in Section 4.2), which in turn leads to larger timeouts and temporarily larger prediction errors (Subplot 2), but ensures that the constraint violation remains at 0% (Subplot 1). SPND is the only approach that violates the constraint throughout this experiment, as indicated by the pink line in Subplot 1. Overall, Khronos continuously satisfies the constraint, just like DSP and STO, but has a far smaller prediction error than DSP, shown in Subplot 2. Note that DSP's prediction error is proportional to the sampling period, which can be up to many orders of magnitude larger than Khronos' prediction error.

Performance with Dynamic Sampling Periods

In this experiment, we test the hypothesis that Khronos can consistently satisfy application completeness constraints in the presence of changing sampling periods, by re-configuring the devices. We evaluate the performance of each approach in two scenarios: stepwise increase and stepwise decrease of the sampling period.

Figure 10(b) shows the impact of increasing the sampling period over three days. The sampling period is increased from 60 to 120 s and from 120 to 240 s, shown by the arrival times in Subplot 3. Khronos reacts to both changes by increasing the smoothed arrival time variance (Equation (7) in Section 4.2), which in turn results in larger timeouts, leading to two peaks in the resulting prediction error (Subplot 2). The benefits of Khronos in terms of completeness are shown in Subplot 1: DSP and SPND both fail the constraint (Subplot 1) after the first change, in contrast to Khronos, which always has a constraint violation smaller than 0.001%. Additionally, Khronos is the only approach that achieves a consistently low prediction error compared to the alternative approaches. The large prediction errors of DSP, SPND, and STO clearly show the limitations of static timeouts, even in the presence of perfect knowledge of the future.

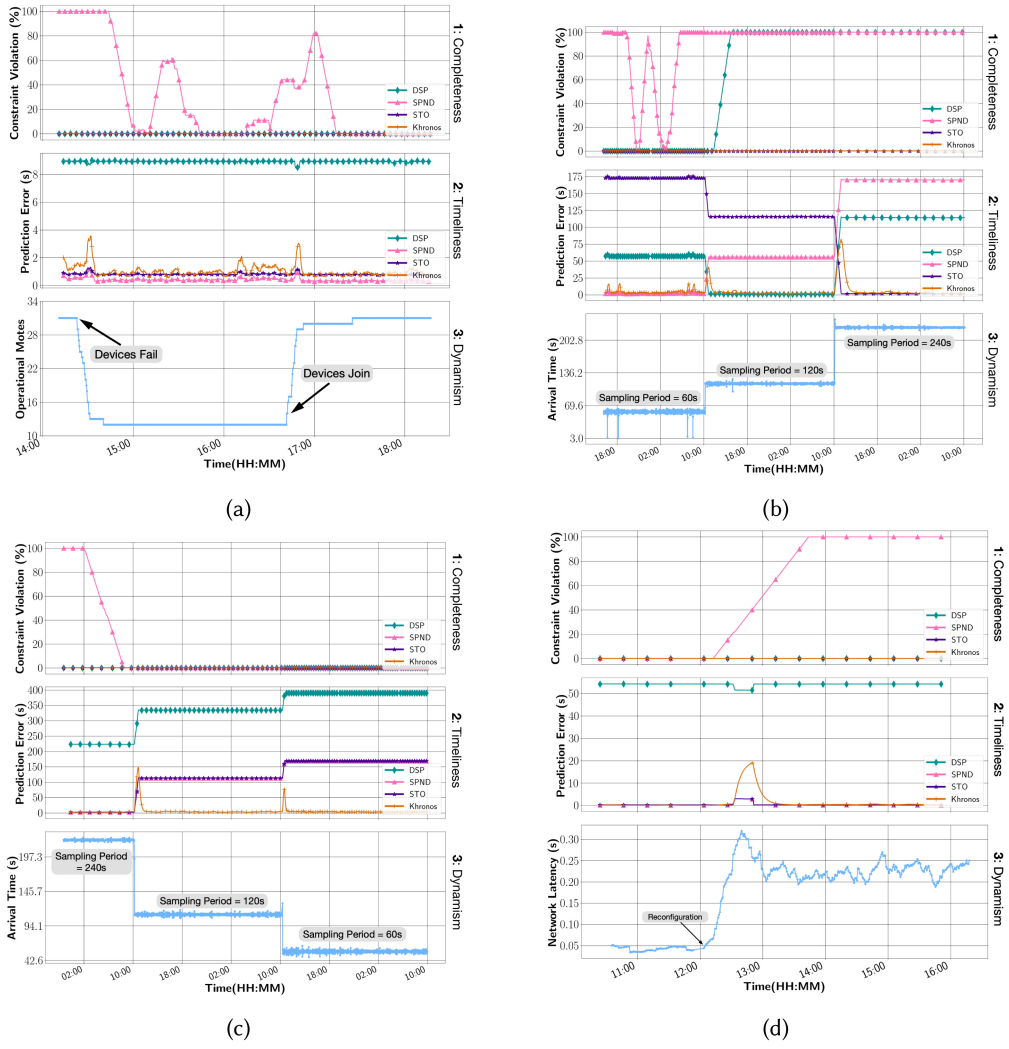

Fig. 10. Evaluation results for different dynamic scenarios. Timeplot for an arbitrary device, where $\rho = 0.8$, illustrating the impact of (a) changing network size, (b) increasing device sampling period, (c) decreasing device sampling period, and (e) increasing the network latency.

Figure 10(c) shows the impact of decreasing the sampling period over the course of 3 days. The sampling period is decreased from 240 to 120 s and from 120 to 60 s, shown in Subplot 3. Since SPND is defined by the initial sampling period and the sampling periods decrease, its constraint violation percentage decreases to 0% (Subplot 1). However, there is a clear penalty in timeliness for DSP, SPND, and STO, shown by the large prediction error in Subplot 2. Khronos is the only approach that consistently satisfies the constraint while at the same time resulting in drastically smaller prediction error, compared to the other approaches across the entire experiment.

Performance with Dynamic Network Latency

In this experiment, we test the hypothesis that Khronos can consistently satisfy application completeness constraints in the presence of network reconfiguration, leading to increased network

latency and variance. Figure 10(d) shows the impact of changing network latency by re-configuring the gateway. All devices are configured with a sampling period of 60 s, and $\rho = 0.8$ for each device data stream. The gateway (network manager) is restarted, indicated by the arrow in Subplot 3, with new configuration: `bwmult` = 100 and `basebw` = 1000. The new configuration leads to higher and more variable latency (Subplot 3), which Khronos detects and reacts by increasing the smoothed arrival time variance (Equation (7) in Section 4.2), which in turn results in larger timeouts and thus a temporarily larger prediction error (peak in Subplot 2), to ensure the completeness constraint remains satisfied (Subplot 1). SPND violates the constraint after network reconfiguration and DSP results in a prediction error that is proportional to the sampling period.

6.3.2 Heterogeneity. This section evaluates the capability of Khronos to satisfy application completeness constraints in the presence of network and application heterogeneity, as specified by requirement **D)**. Broadly, we identify two classifications for heterogeneity: on the network and on the application level. Networks can vary in their topology based on the use-case at hand. Similarly, networks can differ in their medium access control schemes, based on the application requirements (e.g., low-latency versus reliability). Sharing the medium without synchronization (CSMA) can lead to lower network latency, while time-synchronized channel-hopping (TSCH) minimizes packet collisions by allocating dedicated communication slots to each device. Finally, applications can require devices to sample at distinct rates while imposing different completeness constraints. The rest of this subsection compares the performance of the approaches for separate completeness constraints, network topologies, medium access control protocols and sampling periods.

The results for each approach are shown as error-bar charts, where the bar height is equal to the mean across the 22 peripherals and the min and max values are, respectively, the lower and upper bound of the error range.

Meeting a Range of Completeness Constraints

In this experiment, we test the hypothesis that Khronos consistently satisfies a range of different completeness constraints ρ. The testbed is deployed across three building floors and each device is up to one floor away from the gateway, as shown by Figure 9. Devices are configured with their default sampling periods, shown in Table 1, and data have been collected over 7 days. During this period, over 4 million packets arrived at the gateway across all devices.

Figure 11(a) illustrates the constraint violation for each ρ, which is defined earlier in Section 6.2.2. STO by design never violates any constraint and is thus omitted from the figure. Khronos and DSP never violate the constraint, while SPND fails to satisfy $\rho >= 0.6$. For $\rho = 1.0$, Khronos violates the constraint only in 0.32% of the events, over 10 times less than DSP.

Figure 11(b) illustrates the prediction error of each approach, computed for different ρ. Overall, DSP has the highest prediction error, with a mean of 447 s and a max value of 850 s. Khronos' prediction error is in the same order as that of SPND and STO for $\rho \in [0..1)$, and slightly better than DSP for $\rho = 1.0$. The results show the Khronos satisfies all completeness constraints at least as well as DSP, with a prediction error comparable to that of SPND, almost two orders of magnitude less than DSP.

Performance in Heterogeneous Network Topologies

In this experiment, we test the hypothesis that Khronos satisfies completeness constraints for different network topologies. We compare the performance of the approaches for two different deployments. In topology A, the entire testbed is deployed within 1 m of the gateway. In topology B, devices are deployed across a building, up to two floors away from the gateway. For each topology, data have been collected over 72 hours, and devices are configured with their default sampling

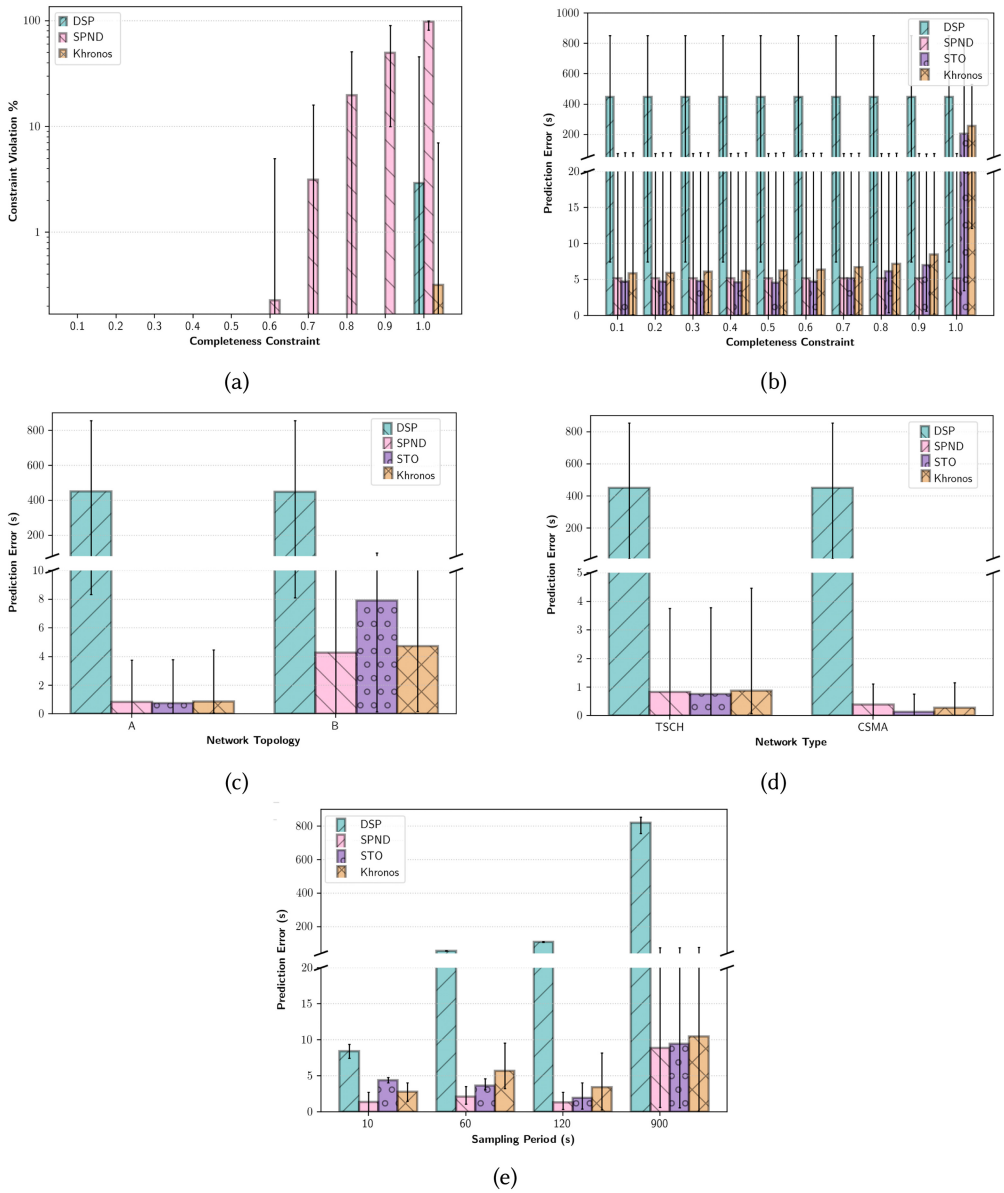

Fig. 11. Evaluation results for different heterogeneity scenarios. (a) Constraint Violation % and (b) average prediction error, measured in seconds, per-approach for $\rho \in \langle 0 \ldots 1 \rangle$. Average Prediction Error per-approach for $\rho = 0.8$, across (c) different network types, (d) different network topologies, (e) different sampling periods.

periods, shown in Table 1. During this period, around 2 million packets arrived at the gateway across all devices.

The constraint violation percentage (ν) of each approach is shown in Table 5. The results show that Khronos does not violate the constraint in either topology, unlike SPND (27.8% in topology A and 42.8% in topology B) and DSP (0.045% in topology B). Figure 11(c) illustrates the prediction error of the different approaches per-topology, for $\rho = 0.8$. In topology A, Khronos has a prediction

Table 5. Constraint Violation (%) Per-approach for Topology A and B, Where Completeness Constraint $\rho = 0.8$

Approach	Topology A	Topology B
DSP	0%	0.045%
SPND	27.8%	42.8%
STO	0%	0%
Khronos	0%	0%

Table 6. Constraint Violation (%) Per-approach for a TSCH and CSMA/CA Wireless Mesh Network, Where $\rho = 0.8$

Approach	TSCH	CSMA/CA
DSP	0%	0%
SPND	27.8%	40%
STO	0%	0%
Khronos	0%	0%

Table 7. Constraint Violation (%) Per-approach for Different Device Sampling Periods and Constraint $\rho = 0.8$

Approach	10 s	60 s	120 s	900 s
DSP	0%	0%	0%	0%
SPND	21.5%	20.3%	25.16%	16.18%
STO	0%	0%	0%	0%
Khronos	0%	0%	0%	0%

error in the same order as SPND and STO, while in topology B its prediction error is almost half of STO and around two orders of magnitude less than DSP.

Performance with Heterogeneous Medium Access Control Protocols

In this experiment, we test the hypothesis that Khronos can satisfy completeness constraints for networks with different medium access control protocols. We compare the performance of the approaches for two different medium access control protocols: TSCH and CSMA/CA. Data are collected over 72 hours, during which around 2 million packets are received at the gateway. All devices are deployed within one meter of the gateway (Topology A).

The constraint violation percentage (v) of each approach for a TSCH and CSMA/CA wireless mesh is shown in Table 6. The results show that Khronos does not violate the constraint in either topology, unlike SPND (27.8% in topology A and 40% in topology B). Figure 11(d) illustrates the prediction error per-approach for TSCH and CSMA/CA. While both DSP and Khronos satisfy the constraint, Khronos scores similarly to SPND and STO with a prediction error of 0.87 s (TSCH) and 0.27 s (CSMA/CA), drastically less than DSP's mean of 450 s.

Performance with Heterogeneous Sampling Periods

In this experiment, we test the hypothesis that Khronos can satisfy completeness constraints for different device sampling periods. We compare the performance of each approach for four sampling periods: 10, 60, 120, and 900 s. The testbed is deployed across three floors of our departmental building, as shown in Figure 9. For each completeness constraint, data has been collected over a course of seven days and devices are configured with their default sampling periods, shown in Table 1. During this period, over 4 million packets arrived at the gateway across all devices.

The constraint violation percentage (v) of each approach for different device sampling periods is shown in Table 7. The results show that Khronos and DSP always satisfy the constraint $\rho = 0.8$, while SPND fails it 21.5%, 20.3%, 25.16%, and 16.18% of the time for a sampling period of 10, 60, 120, and 900 s, respectively.

Figure 11(e) illustrates the prediction error per-approach for TSCH and CSMA/CA. Khronos' prediction error is slightly above SPND and comparable to STO, while satisfying $\rho = 0.8$ in contrast to SPND. DSP satisfies the constraint at a far larger cost, with a prediction error proportional to the sampling period and at least two orders of magnitude higher than Khronos for a sampling period larger than 10 s.

Discussion. The evaluation results show that Khronos succeeds in tackling the problem stated in Section 2, by yielding the smallest prediction error while ensuring completeness constraint satisfaction when compared to the alternative approaches. Khronos accomplishes this through dynamic timeouts for event arrivals, computed using Equation (8) in Section 4.2, which react to changes in the observed arrival times. When the interval between consecutive arrival times is close to constant, the second term in the equation is small, which also leads to smaller timeouts. On the contrary, when the interval between consecutive arrival times varies substantially, the second term in Equation (8) increases aggressively based on the value of K_ρ, which is proportional to the completeness constraint ρ. While this ensures, as the results have shown, that Khronos satisfies $\rho \in \langle 0, \ldots, 1 \rangle$ under various conditions, it does so at the expense of timeliness for $\rho = 1.0$, which can be seen in Figure 11(b). Indeed, the stricter the completeness constraint, the further timeouts are on average from the corresponding event arrival times. While this is a desirable property to ensure consistent constraint satisfaction, for $\rho = 1.0$ it leads to timeouts up to hundreds of seconds larger than the corresponding arrival times, which unlike our use case, can be unacceptable for applications that do not tolerate both missed events and large timeouts. Finally, while extensive, our evaluation is not exhaustive: depending on the underlying CPS network technology, other (dynamic) parameters can be present that influence event arrival times. In LoRa networks, for example, the time between consecutive event transmissions depends on the payload size and the spreading factor [52]. The payload size can change at runtime by the application, while the spreading factor can change based on the distance of the device from the gateway. Our approach is agnostic of the underlying network technology, since it relies purely on monitoring the network for event arrival times. Thus, it naturally generalizes to other types of networks and applications, such as smart cities using LoRa real-world deployments.

7 CONCLUSION

CPS are increasingly integrated with critical physical processes, including manufacturing, healthcare, and smart grids, enabling advanced monitoring and control to improve operational efficiency. Reactive programming simplifies the development of event-driven CPS applications: It offers powerful abstractions that encapsulate event streams, in addition to an execution environment that automatically propagates updates based on their dependencies. Current reactive solutions require CPS application developers to manually specify timeouts at compile time, which can lead to inefficiencies and incorrect results as event arrival times vary due to network and packet intergeneration delay. Reacting in a timely manner to changes while operating over complete information is a crucial research challenge.

This article introduced a novel set of reactive programming extensions that enable CPS application developers to easily trade off timeliness versus completeness in their applications. These extensions utilize Khronos, a novel middleware that supports the specification of completeness constraint(s) per-device data stream, shielding the developer from manually specifying packet arrival timeouts or further configuration parameters. This is achieved by monitoring the CPS infrastructure and automatically specifying timeouts for event arrivals while satisfying the specified completeness requirements. Khronos relies on a single configuration parameter K_ρ, which controls the sensitivity to variance in event arrival times and can be determined empirically for any completeness constraint ρ by following the methodology proposed in this article.

The proposed reactive extensions are evaluated by comparing two equivalent implementations of an industrial use case example, written in RxJS with and without these extensions. The results indicate that using the proposed extensions reduces the accidental complexity of the program, decreasing the total lines of code by over 45%. Additionally, Khronos is evaluated on top of a physical testbed of 34 devices, connected through a state-of-the-art wireless mesh network that supports two medium access control protocols: TSCH and CSMA/CA. The experiments are performed in the presence of various sources of heterogeneity and dynamism in the underlying CPS. Overall, the results show that Khronos improves upon state-of-the-art approaches, resulting in up to two orders of magnitude smaller timeouts while never violating the application completeness constraints. Together, these results suggest that the combination of the proposed novel language extensions and Khronos provide an end-to-end solution that enables CPS application developers to easily write reliable distributed applications with flexible completeness requirements. A natural progression of this work is to analyze techniques that adapt the value of K_ρ at runtime, which will eliminate not only the need for pre-deployment configuration (e.g., by applying machine learning techniques) but can also improve the prediction errors when no missed events are tolerated.

REFERENCES

[1] ReactiveX. Retrieved December 12, 2019 from http://reactivex.io/languages.html.

[2] SmartMesh IP. Retrieved December 12, 2019 from https://www.analog.com/en/products/rf-microwave/wireless-sensor-networks/smartmesh-ip.html.

[3] DC9003A-B SmartMesh IP Eval/Dev Mote (Chip Antenna). Retrieved December 13, 2019 from https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/dc9003a-b.html.

[4] Flask (Full Stack Python). Retrieved December 13, 2019 from https://www.fullstackpython.com/flask.html.

[5] Grafana https://grafana.com/.

[6] Representational State Transfer (REST). Retrieved December 13, 2019 from https://en.wikipedia.org/wiki/Representational_state_transfer.

[7] RFC 7252 Constrained Application Protocol (CoAP). Retrieved December 13, 2019 from http://coap.technology/.

[8] VersaSense Edge Gateway (SmartMesh IP). Retrieved December 13, 2019 from https://www.versasense.com/pdf/VersaSense-M01.pdf.

[9] VersaSense Wireless Device (SmartMesh IP). Retrieved December 13, 2019 from https://www.versasense.com/pdf/VersaSense-Pxx.pdf.

[10] WebSocket. Retrieved December 13, 2019 from https://en.wikipedia.org/wiki/WebSocket.

[11] Sven Akkermans, Stefanos Peros, Nicolas Small, Wouter Joosen, and Danny Hughes. 2018. Supporting IoT application middleware on edge and cloud infrastructures. In *Proceedings of the 10th Central European Workshop on Services and Their Composition.*

[12] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. 2015. Automatic deployment of distributed software systems: Definitions and state of the art. *J. Syst. Softw.* 103 (2015), 198–218. DOI: https://doi.org/10.1016/j.jss.2015.01.040

[13] Radhakisan Baheti and Helen Gill. 2011. Cyber-physical systems. *Impact Contr. Technol.* 12 (2011), 161–166.

[14] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 452 (Aug. 2013). DOI: https://doi.org/10.1145/2501654.2501666

[15] Andrey Brito, Christof Fetzer, and Pascal Felber. 2009. Multithreading-enabled active replication for event stream processing operators. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems* (2009), 22–31.

[16] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2008. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the 2nd International Conference on Distributed Event-based Systems.* ACM, 265–275.

[17] Frederik P. Brooks and No Silver Bullet. 1987. Essence and accidents of software engineering. *IEEE Comput.* 20, 4 (1987), 10–19.

[18] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-performance dynamic pattern matching over disordered streams. *Proc. VLDB Endow.* 3, 1–2 (2010), 220–231.

[19] Antonio Cilfone, Luca Davoli, Laura Belli, and Gianluigi Ferrari. 2019. Wireless mesh networking: An IoT-oriented perspective survey on relevant technologies. *Fut. Internet* 11, 04 (2019), 99. DOI: https://doi.org/10.3390/fi11040099

[20] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. 2017. First-class reactive programs for CPS. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'17)*. ACM, New York, NY, 21–26. DOI: https://doi.org/10.1145/3141858.3141862

[21] Christophe de Troyer, Jens Nicolay, and Wolfgang de Meuter. 2018. Building IoT systems using distributed first-class reactive programming. 185–192. DOI: https://doi.org/10.1109/CloudCom2018.2018.00045

[22] X. Defago, P. Urban, N. Hayashibara, and T. Katayama. 2005. Definition and specification of accrual failure detectors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*. 206–215. DOI: https://doi.org/10.1109/DSN.2005.37

[23] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. 2008. *PTIDES: A Programming Model for Distributed Real-Time Embedded Systems*. Technical Report UCB/EECS-2008-72. EECS Department, University of California, Berkeley.

[24] Jonathan Edwards. 2009. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. ACM, New York, NY, 925–932. DOI: https://doi.org/10.1145/1639950.1640058

[25] Jonathan Edwards. 2009. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. ACM, New York, NY, 925–932. DOI: https://doi.org/10.1145/1639950.1640058

[26] Marisol García-Valls and Roberto Baldoni. 2015. Adaptive middleware design for CPS: Considerations on the OS, resource managers, and the network run-time. In *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware (ARM'15)*. ACM, New York, NY, Article 3, 6 pages. DOI: https://doi.org/10.1145/2834965.2834968

[27] Sabina Jeschke, Christian Brecher, Tobias Meisen, Denis Özdemir, and Tim Eschert. 2017. *Industrial Internet of Things and Cyber Manufacturing Systems*. Springer International Publishing, Cham, 3–19. DOI: https://doi.org/10.1007/978-3-319-42559-7_1

[28] Yuanzhen Ji, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2016. Quality-driven disorder handling for concurrent windowed stream queries with shared operators. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 25–36.

[29] Yuanzhen Ji, Jun Sun, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2016. Quality-driven disorder handling for m-way sliding window stream joins. In *Proceedings of the IEEE 32nd International Conference on Data Engineering (ICDE'16)*. IEEE, 493–504.

[30] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-driven continuous query execution over out-of-order data streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 889–894.

[31] Boris Koldehofe, Ruben Mayer, Umakishore Ramachandran, Kurt Rothermel, and Marco Völz. 2013. Rollback-recovery without checkpoints in distributed event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. ACM, 27–38.

[32] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. 2010. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM, 1081–1092.

[33] Paulo Leitão and Stamatis Karnouskos. 2015. *Industrial Agents: Emerging Applications of Software Agents in Industry*. Morgan Kaufmann.

[34] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. ACM, 311–322.

[35] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: A new architecture for high-performance stream systems. *Proc. VLDB* 1, 01 (2008), 274–288.

[36] Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. Deprecating the observer pattern. EPFL-REPORT-148043. 18 pages.

[37] Alessandro Margara and Guido Salvaneschi. 2014. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 142–153.

[38] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2014. Meeting predictable buffer limits in the parallel execution of event processing operators. In *Proceedings of the 2014 IEEE International Conference on Byig Data (Big Data)*. IEEE, 402–411.

[39] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable low-latency event detection with parallel complex event processing. *IEEE IoT J.* 2, 4 (2015), 274–286.

[40] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: Supporting consumption policies in window-based parallel complex event processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 161–173.

[41] Nader Mohamed, Jameela Al-Jaroodi, Sanja Lazarova-Molnar, and Imad Jawhar. 2017. Middleware challenges for cyber-physical systems. *Scalable Comput.: Pract. Exper.* 18, 4 (2017), 331–346.
[42] L. Monostori, B. Kádár, T. Bauernhansl, S. Kondoh, S. Kumara, G. Reinhart, O. Sauer, G. Schuh, W. Sihn, and K. Ueda. 2016. Cyber-physical systems in manufacturing. *CIRP Ann.* 65, 2 (2016), 621–641. DOI: https://doi.org/10.1016/j.cirp.2016.06.005
[43] Christopher Mutschler and Michael Philippsen. 2014. Adaptive speculative processing of out-of-order event streams. *ACM Trans. Internet Technol.* 14, 1 (2014), 4.
[44] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2017. Handling partial failures in distributed reactive programming. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'17)*. ACM, New York, NY, 1–7. DOI: https://doi.org/10.1145/3141858.3141859
[45] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2019. Distributed reactive programming for reactive distributed systems. *arXiv preprint arXiv:1902.00524*.
[46] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. 2010. Cyber-physical systems: The next computing revolution. In *Proceedings of the Design Automation Conference*. 731–736. DOI: https://doi.org/10.1145/1837274.1837461
[47] Nicolo Rivetti, Nikos Zacheilas, Avigdor Gal, and Vana Kalogeraki. 2018. Probabilistic management of late arrival of events. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS'18)*. ACM, New York, NY, 52–63. DOI: https://doi.org/10.1145/3210284.3210293
[48] Esther Ryvkina, Anurag S. Maskey, Mitch Cherniack, and Stan Zdonik. 2006. Revision processing in a stream processing engine: A high-level design. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 141–141.
[49] Y. Shafranovich. 2011. *Computing TCP's Retransmission Timer*. RFC 6298. RFC Editor. Retrieved from https://tools.ietf.org/html/rfc6298.
[50] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. 2011. A survey of cyber-physical systems. In *Proceedings of the International Conference on Wireless Communications and Signal Processing (WCSP'11)*. IEEE, 1–6.
[51] Sabrie Soloman. 2010. *Sensors Handbook* (2nd ed.).
[52] R. B. Sørensen, D. M. Kim, J. J. Nielsen, and P. Popovski. 2017. Analysis of latency and MAC-layer performance for Class A LoRaWAN. *IEEE Wireless Commun. Lett.* 6, 5 (Oct. 2017), 566–569. DOI: https://doi.org/10.1109/LWC.2017.2716932
[53] Michael Spörk, Carlo Alberto Boano, and Kay Römer. 2019. Improving the timeliness of bluetooth low energy in noisy RF environments. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks (EWSN'19)*. ACM, 12.
[54] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible time management in data stream systems. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 263–274.
[55] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. DOI: https://doi.org/10.1145/1107499.1107504
[56] Andrew S. Tanenbaum, David Wetherall, et al. 2014. *Computer Networks*. Harlow, Essex: Pearson.
[57] Linear Technology (Ed.). 2016. *SmartMesh IP Embedded Manager CLI Guide*. Linear Technology.
[58] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.* 15, 3 (2003), 555–568.
[59] Lihui Wang, Martin Törngren, and Mauro Onori. 2015. Current status and advancement of cyber-physical systems in manufacturing. *J. Manufact. Syst.* 37 (2015), 517–527.
[60] Nikos Zacheilas, Vana Kalogeraki, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. 2017. Maximizing determinism in stream processing under latency constraints. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, 112–123.
[61] Y. Zhang, C. Gill, and C. Lu. 2008. Reconfigurable real-time middleware for distributed cyber-physical systems with aperiodic events. In *Proceedings of the 28th International Conference on Distributed Computing Systems*. 581–588. DOI: https://doi.org/10.1109/ICDCS.2008.96
[62] L. Zhao, P. Pop, and S. S. Craciunas. 2018. Worst-case latency analysis for IEEE 802.1Qbv time sensitive networks using network calculus. *IEEE Access* 6 (2018), 41803–41815. DOI: https://doi.org/10.1109/ACCESS.2018.2858767
[63] K. Zhou, Taigang Liu, and Lifeng Zhou. 2015. Industry 4.0: Towards future industrial opportunities and challenges. In *Proceedings of the 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'15)*. 2147–2152. DOI: https://doi.org/10.1109/FSKD.2015.7382284

Received August 2019; revised December 2019; accepted February 2020