Spring 5-15-2019

Real-Time Reliable Middleware for Industrial Internet-of-Things

Chao Wang
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST.LOUIS

School of Engineering & Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Christopher Gill, Chair
Chenyang Lu, Co-Chair
Kunal Agrawal
Sanjoy Baruah
Jing Li

Real-Time Reliable Middleware for Industrial Internet-of-Things

by
Chao Wang

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2019
St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

The completion of this dissertation would have been impossible if I work alone by myself. I feel very grateful to have both Dr. Chris Gill and Dr. Chenyang Lu as my advisors in my Ph.D. training. I also thank Dr. Jonathan Turner for his guidance in my early year of this journey. To Dr. Jing Li I give my thanks, for technical discussions in many aspects of theoretical computer science research, as well as for her encouragements whenever I felt stuck along the way. I thank Dr. Kunal Agrawal for her technical feedbacks in improving my dissertation work, and I thank Dr. Sanjoy Baruah for his advice in how to position my research work in a broader context. I cherish those days working with them and learning from them.

I thank Byron Austin, a senior network engineer in Engineering IT, for his great help in configuring our edge-cloud test-bed for project FRAME. My thanks must also go to Chong Li, James Orr, Haoran Li, and Son Dinh, and members of the Applied Research Lab and the Cyber-Physical Systems Lab, and members in the departmental office, for all their techinal helps as well as daily accompany during my stay in the department.

Last but not least, I thank God the Father for granting me a chance to come to know Him, for giving me a willing heart to receive Jesus Christ His Son as my personal savior, and for His daily grace to sustain me. I often recalled that one Sunday morning, during a prayer meeting, an elderly lady of our church gave thanks on behalf of me, saying that the purpose that I come to the United States is to know Him. I truly believe so. With this dissertation I offer my thanksgiving.

Chao Wang

Washington University in Saint Louis

May 2019

Dedicated to my parents.

ABSTRACT OF THE DISSERTATION

Real-Time Reliable Middleware for Industrial Internet-of-Things

by

Chao Wang

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2019

Professor Christopher Gill, Chair

Professor Chenyang Lu, Co-Chair

This dissertation contributes to the area of adaptive real-time and fault-tolerant systems research, applied to Industrial Internet-of-Things (IIoT) systems. Heterogeneous timing and reliability requirements arising from IIoT applications have posed challenges for IIoT services to efficiently differentiate and meet such requirements. Specifically, IIoT services must both differentiate processing according to applications' timing requirements (including latency, event freshness, and relative consistency of each other) and enforce the needed levels of assurance for data delivery (even as far as ensuring zero data loss). It is nontrivial for an IIoT service to efficiently differentiate such heterogeneous IIoT timing/reliability requirements to fit each application, especially when facing increasingly large data traffic and when common fault-tolerant mechanisms tend to introduce latency and latency jitters.

This dissertation presents a new adaptive real-time fault-tolerant framework for IIoT systems, along with efficient and adaptive strategies to meet each IIoT application's timing/reliability requirements. The contributions of the framework are demonstrated by three new IIoT middleware services: (1) Cyber-Physical Event Processing (CPEP), which both differentiates application-specific latency requirements and enforces cyber-physical timing constraints, by prioritizing, sharing, and shedding event processing. (2) Fault-Tolerant Real-Time Messaging (FRAME), which integrates real-time capabilities with a primary-backup replication system, to fit each application's unique timing and loss-tolerance requirements. (3) Adaptive Real-Time Reliable Edge Computing (ARREC), which leverages heterogeneous loss-tolerance requirements and their different temporal laxities, to perform selective and lazy (yet timely) data replication, thus allowing the system to meet needed levels of loss-tolerance while reducing both the latency and bandwidth penalties that are typical of fault-tolerant sub-systems.

Chapter 1

Introduction

1.1 IIoT Overview

The term Industrial Internet-of-Things (IIoT) refers to a category of networked computing systems that observe and interact with a physical environment, in particular through systems of sensing data generated by embedded devices, especially for applications that need to maintain descriptive, predictive, and prescriptive analytics of the environment and the system therein. An IIoT system is thus a cyber-physical system (CPS) spanning both local and remote contexts.

An example of such an IIoT system is found in wind farms, which consist of several hundreds of wind turbines that produce electricity (Figure 1.1). Within each local context (referred to as an *edge*), a network of sensors and actuators are used to monitor and control a set of wind turbines, to improve operating efficiency and to maintain system reliability. For example, to adjust their orientation and speed according to the current windspeed and direction. With recent advancements in cloud computing, edge computing, and machine learning techniques,

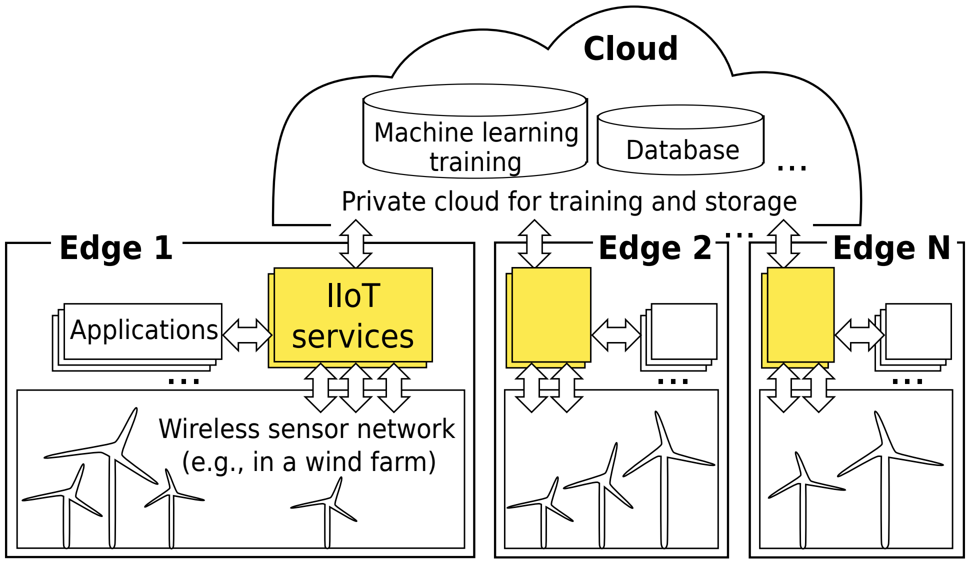

Cloud
Machine learning training
Database
...
Private cloud for training and storage
Edge 1
Edge 2
...
Edge N
Applications
IIoT services
...
Wireless sensor network (e.g., in a wind farm)

Figure 1.1: An Illustration of Industrial Internet-of-Things (IIoT).

large numbers of data collected from sensor networks can be used for local applications, for data storage in a cloud, and for sharing of data between edge systems [74].

To improve both performance and cost, increasing numbers of IIoT applications are deployed locally, that is, as edge systems. Moving applications to the edge of a larger cloud that hosts, e.g., latency-sensitive services, can reduce the application's response time to local stimuli, as well as can saving costs for cloud I/O and hosting. This dissertation focuses on IIoT services at the edge of a cloud. Modern edge computing platforms, such as Amazon AWS IoT Greengrass [1] and Microsoft Azure IoT Edge [56], use local IIoT services to perform computation and messaging, and results are either delivered to local applications or transported into the cloud. In this setting, facing increasingly high-volume and heterogeneous data traffic, an IIoT service can become a performance bottleneck. How to make IIoT services real-time, reliable, and efficient remains a real, challenging, and important research problem, which is the focus of this dissertation.

1.2 Research Challenges and Dissertation Contributions

IIoT applications typically have requirements for Quality-of-Service (QoS). Of particular interest are those pertaining to timing and delivery. In general, the research challenges are to have an IIoT service efficiently meet both types of QoS requirements while mitigating potential conflicts arising from that combination. Specifically, this dissertation address the challenges in three settings: (1) Cyber-physical event processing, where events are subject to time consistency criteria and latency requirements, and the workloads for processing events are nontrivial; (2) Messaging, where there is a lower per-message processing workload, but each message is subject to latency and/or loss-tolerance requirements; (3) Edge computing, where processing workload per data element varies, and in addition to needing to meet both latency and loss-tolerance requirements, it is critical that the IIoT service is efficient. The following describes each of these three topics in detail.

1.2.1 Temporal Requirements in Cyber-Physical Event Processing

Research Challenge: Cyber-physical event processing deals with events that carry data describing physical phenomenon [19, 46, 53]. An IIoT service takes a sensor data, tranforms it, and then delivers it to the applications that subscribe to the result. In IIoT applications such as wind turbine monitoring, events give a sequence of observations regarding physical states (e.g., windspeed, orientation, power consumption/generation, temperature, etc.), which may change over time. In addition, different types of events may have temporal dependencies on each other (e.g., for multi-sensor fusion [25, 61, 64, 68] or data alignment). In those cases, out-dated data should be marked or discarded. In addition, IIoT applications often have different levels of latency requirements (e.g., emergency response vs. logging), and it is critical that an IIoT service can serve each application according to its need. Moreover, an IIoT service may perform common processing (e.g., FFT and encryption) and complex processing (e.g., data fusion), and as different applications may require same processing of events, duplicated processing may waste computational resources. Finally, it is critical to properly leverage multi-core processors to improve processing throughput.

Dissertation Contribution: This dissertation describes a new real-time middleware called CPEP, for cyber-physical event processing, with the following four features: (1) *Enforcement of temporal validity and shedding* maintains temporal validity constraints, identifying and possibly removing out-dated data; (2) *Configurable processing operations* integrate both simple and complex event processing; (3) *Processing prioritization and sharing* ensure that higher-priority events are processed first and reduce the likelihood of starvation of lower-priority ones; (4) *Efficient concurrent processing* minimizes memory allocation for events and can scale up throughput with the number of CPU cores.

1.2.2 Reliability and Timing in Messaging

Research Challenge: Unlike cyber-physical event processing, an IIoT messaging service simply delivers a message of a certain topic to the applications that subscribe to the topic [59, 60, 66]. The per-message workload within the IIoT service is low, but at the same time the service must accommodate many more messages. In addition, IIoT applications have requirements for message latency and reliable delivery, and the needed levels of assurance are often combined in heterogeneous ways. For example, while monitoring applications may need hundreds of milliseconds bounds on latency and can tolerate a small number of consecutive message losses (e.g., by computing estimates using previous or subsequent messages), logging applications may only require sub-second latency but cannot tolerate any message loss. Besides latency differentiation, an IIoT service must properly differentiate messages' loss-tolerance levels, since fault-tolerant approaches in general tend to slow down a system [9, 13] or impose greater resource consumption [35, 63]. Finally, an IIoT service needs to account for both the discrepancy between traffic periods within an edge (e.g., tens of milliseconds) and those to a cloud (e.g., at least a significant fraction of a second), and the discrepancy between network latency within an edge (e.g., sub-millisecond) and that to a cloud (e.g., up to sub-second). Premature scheduling of cloud-bound traffic may delay edge-bound, latency-sensitive traffic.

Dissertation Contribution: This dissertation describes (1) a new fault-tolerant real-time messaging model, and (2) a new fault-tolerant real-time middleware called FRAME, for fault-tolerant real-time messaging. The new model gives a holistic description of how message publishers and a backup messaging broker may participate in recovering lost messages, and proves timing bounds for real-time fault-tolerant actions in terms of traffic/service parameters. The new middleware can leverage the proved timing bounds to support efficient and appropriate message differentiation to meet each of the latency and loss-tolerance requirements, and can mitigate latency penalties caused by loss recovery.

1.2.3 Efficiency in Reliable and Timely Edge Computing

Research Challenge: In general, an IIoT service should be able to handle both in-band processing (e.g., as in event processing) and large amounts of traffic (e.g., as in messaging). With the edge computing paradigm, the aforementioned challenges manifest themselves in terms of *serverless computation* (e.g., AWS Lambda [2]), where an IIoT service as an edge computing engine performs in-band computation for various input data traffic, and delivers computed results to applications of interest. It is critical to understand how the changes in traffic patterns/load and computational load may impact system performance; in particular, how they may impact the data-loss and latency performance. Furthermore, a practical systems solution is needed to meet required levels of data-loss and latency assurance despite load changes.

Dissertation Contribution: This dissertation gives both theoretical and practical contributions for the above challenge. In the context of the Primary-Backup Replication model [15], this dissertation formulates the relation between (1) frequency of data replication and (2) deadline to complete temporally assured replication. A novel selective, lazy data replication strategy is described, and is implemented in a middleware for adaptive real-time reliable edge computing (ARREC). The performance of ARREC is evaluated empirically, demonstrating that it can ensure needed data-loss tolerance levels while reducing both latency penalties for data computation/delivery, and network bandwidth consumption.

1.3 Dissertation Organization

The rest of the dissertation is structured as follows. Chapter 2 first presents the motivation and definition of cyber-physical event processing, followed by the design and implementation of CPEP the IIoT service middleware, showing how CPEP can address the aforementioned research challenge of meeting temporal requirements. The chapter concludes by a comprehensive evaluation of CPEP's empricial performance.

Chapter 3 brings into picture the fault-tolerance aspect of IIoT messaging services, and presents FRAME the fault-tolerant real-time messaging service middleware. The chapter begins with the background and motivation of such an IIoT messaging service, followed by an illustration of capabilities of FRAME using typical IIoT traffic configuration, and concludes with empirical validation of FRAME's performance.

Chapter 4 completes the scope of this dissertation by addressing the efficiency aspect of IIoT services. In the context of serverless computation in edge computing, the chapter describes patterns of processing and traffic in IIoT systems, and then presents ARREC the adaptive real-time reliable edge computing middleware. The timing analysis framework of ARREC is introduced, followed by a design of system architecture that leverages the timing bounds from the analysis. Finally, the chapter concludes by a description of an implementation of ARREC and empirical evaluation of its performance.

For clarity and to make a specific comparison, I describe both related work and system model in the same chapter of the specific IIoT service middleware. In Chapter 5, I give concluding remarks of this dissertation.

Chapter 2

Real-Time Cyber-Physical Event Processing

2.1 Introduction

Real-time event processing is essential for cyber-physical systems (CPS), such as Industrial Internet of Things [26, 28, 45] systems, which must perform operations on sensor data carried by events and must respond to stimuli with quick and correct actions (e.g., in milliseconds [42, 44]). For example, smart electric grid applications require latency to be less than 50 ms and processing operations are conducted near the edge of the network to the extent possible [44].

Multi-sensor fusion is required by many real-world applications such as position estimation, obstacle detection, and object tracking [25, 61, 64, 68]. By synthesizing data supplied by different sensors, multi-sensor fusion offers subscribers a more cohesive and reliable assessment of the environment. Such processing is typically multi-stage. For example, data from sensors (event suppliers) is first passed through one or more filters for noise reduction,

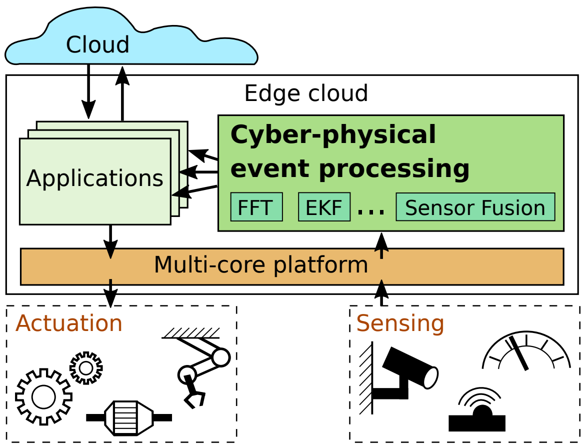


Figure 2.1: Cyber-physical event processing.

and then a Fast Fourier Transform (FFT) is applied to the result to obtain frequency domain representations. Results from different processing streams are then combined, producing an event that represents a broader-spectrum assessment for applications (event consumers).

Real-time cyber-physical event processing must support configurable complex operations, meet applications' latency requirements, enforce temporal validity of events, and leverage multi-core platforms, as Figure 2.1 illustrates. First, applications often perform simple common operations (e.g., FFT) as well as complex operations that may be realized by combining other common operations (e.g, a multi-sensor fusion realized by filters, FFTs, etc.). Second, a cyber-physical system must accommodate applications' different latency requirements, and should allow applications to share processing and data. Duplicating complex operations (or even portions of them) across application features wastes both communication bandwidth and computational resources, and re-implementing such operations for each application may unnecessarily increase software complexity and decrease software reliability. Third, cyber-physical applications are often subject to temporal validity constraints. For example, for automotive driving features such as adaptive cruise control, where data from sensors are fused to provide range estimates, the relevance of each sensor reading may decrease over time, and out-dated data should be discarded. Finally, to better serve the needs of real-time *edge* *computing* [45], an event processing service must efficiently work with streams of events in terms of memory allocation and throughput.

To address these needs, in this Chapter we introduce a real-time middleware for cyber-physical event processing (CPEP), with the following four features: *Configurable processing operations* integrate both simple and complex event processing; *Processing prioritization and sharing* ensure that higher-priority events are processed first and reduce the likelihood of starvation of lower-priority ones; *Enforcement of temporal validity and shedding* maintains temporal validity constraints, identifying and removing out-dated data; and *Efficient concurrent processing* minimizes memory allocation for events and can scale up throughput with the number of CPU cores.

We implemented CPEP atop TAO, a mature and widely used open-source middleware [38, 62] by adding the above capabilities to its Real-Time Event Service. We compared CPEP with the Apache Flink stream processing platform [3] and our empirical results show that CPEP can (through prioritization) better prevent higher-priority processing from incurring unnecessary delay, (through operation sharing) help reduce latency of lower-priority processing, and (through shedding) improve throughput of time-consistent events.

2.2 Related Work

Cyber-physical event processing is an essential part of modern Industrial Internet-of-things architectures [41], and in many use cases [28, 45] it is critical to minimize the time it takes to respond to stimuli. To this end, both messaging middleware (for example, Kafka [47]) and the Data Distribution Service (DDS [60]) have been deployed. Kafka provides fault-tolerance and load-balancing for delivery of time-stamped log messages, and provides an interface for implementing message processing, but does not differentiate messages according to consumers' priority levels. DDS provides QoS options for data (event) delivery, but does not process events. In contrast, CPEP both differentiates messages according to consumers' priority levels and processes event subject to time consistency.

Apache Flink is an open-source stream processing framework featuring high throughput, low latency event processing and windowing, and fault tolerance [3]. The Flink framework accepts multiple event streams and performs stream transformations according to a plan. The results are new event streams, which in turn can be used for further transformations or be delivered to event subscribers. Flink supports a distributed runtime environment, where JobManagers (masters) receive the processing plans from clients and then distribute them to TaskManagers (workers) for execution. Windowing in Flink is either time driven (e.g., every 30 seconds) or event driven (e.g., every 100 events) and is typically used for event aggregation. Flink does not support absolute or relative time consistency enforcement, both of which are critical to real-time cyber-physical event processing, nor does it differentiate stream processing. In contrast, CPEP supports both types of time consistency enforcement and can prioritize stream processing.

Time consistency has been studied in real-time databases [67, 73], where *absolute time consistency* means that the datum being used by a transaction still carries a timely measurement, and *relative time consistency* means that the data being used by a transaction are updated within a specified time interval [67]. In our CPEP architecture, we extend the definition of time consistency for real-time cyber-physical event processing, and provide a design and implementation to enforce time consistency and to shed invalid work.

Time consistency is needed by many real-world applications. For example, a fire detection system may deploy a rule that triggers an alarm when both smoke and a high temperature occur within a certain time interval [20]; in modern automotive systems, conflicting commands sent to the same set of actuators (e.g., throttle actuator and/or brake actuator) within a certain time interval may cause unsafe interactions [24]; Dominguez et al. also surveyed other important feature interactions in embedded systems [24].

In social network analysis [36], the popularity of a post, as well as the size of the involved community, is determined by scores that decrease over time. While social network analysis is typically conducted at the scale of seconds or even hours, in our CPEP middleware we enforce time consistency at scales as fine as milliseconds, a resolution required by many cyber-physical applications. The field of Complex Event Processing (CEP) [53] offers rich semantics for expressing stimuli using sets of events [19, 46]. GraphCEP [54] processes events for social network analysis, and implements timetables for updating the ranking of posts and comments according to the progress of time. GraphCEP maintains time consistency (at the timescale of hours and seconds) but does not share computation among processing streams. In contrast, CPEP maintains time consistency at the timescale of milliseconds, and supports sharing of computation between processing streams.

2.3 Cyber-Physical Event-Processing (CPEP) Model

Our event processing model consists of three kinds of components: suppliers, an event service, and consumers. Each supplier pushes *typed data items*, which we call *events*, to the event service; the event service processes the events according to a graph that defines the needed operations and their input/output events, as illustrated in Figure 2.2; a consumer subscribes to the output events of operations. Each supplier pushes events either periodically or sporadically. Each consumer is associated with a priority level. In practice, a supplier (consumer) may be mapped to a distinct sensor or other device, and multiple suppliers

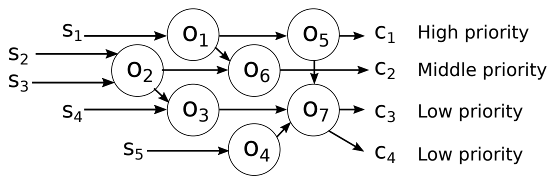


Figure 2.2: An example graph of event processing streams: s_i denotes a supplier; c_i, a consumer; o_i, an operator.

(consumers) may be mapped to a single device. The event service is executed within a single host.

In the following, we first define the processing within the event service, and then we define *absolute time consistency* and *relative time consistency* [67, 73][1], which identify the temporal validity of an event or a set of events, respectively.

2.3.1 Event Processing

Event processing in our model is configured as a directed acyclic graph, as illustrated in Figure 2.2, and paths along the edges in the graph define the data processing streams for each consumer. The nodes of the graph are event processing operators, such as FFT, and the edges denote the precedence relations between operators. For example, Figure 2.2 shows processing streams for four consumers, and the streams for consumer c_2 involve operators o_1, o_2, and o_6. Operator o_1 has three downstream operators (o_5, o_6, and o_7) and operator o_6 has two upstream operators (o_1 and o_2). A complex operation, such as multi-sensor fusion, may be built from a set of common operators. Execution of an operator produces an event. We call events that are pushed from one operator to another *internal events*, the events pushed from suppliers *supplier events*, and the events pushed to consumers *consumer events*.

[1]We extend the definitions to make them suitable for real-time cyber-physical event processing, as described in Sections 2.3.2 and 2.3.3, respectively.

The event service schedules operators to process events. An operator is ready for execution if its specified dependencies are satisfied, e.g., its upstream operators have completed processing and all of its input events have arrived. The event service adds ready operators to the execution schedule. After execution, the same events will never be used again by the same operator. This ensures that cyber-physical operations, such as multi-sensor fusion, do not (prematurely) process newly arriving data in combination with previously used data.

In Section 2.4, we describe how CPEP first prioritizes operators based on the consumers' priority levels and then schedules the operators using a fixed-priority preemptive scheduling policy. We assume that the configuration of processing streams is specified by domain experts developing a particular application.

2.3.2 Absolute Time Consistency

Event e_i is temporally valid at time t if t falls within the *absolute validity interval* of e_i, defined by

$$\text{abs}(e_i) = [t_b(e_i), t_e(e_i)), \tag{2.1}$$

where $t_b(e_i)$ and $t_e(e_i)$ respectively define the beginning and the end of the interval. Because only supplier events are associated with physical phenomena, we define an internal event's absolute validity interval to be the maximum overlap of all e_i's upstream supplier events' absolute validity intervals: Let $o(e_i)$ be the operator that produces e_i, and I_{e_i} be the set of events required by operator $o(e_i)$. We have

$$t_b(e_i) = \max\{t_b(u) \mid u \in I_{e_i}\}; \tag{2.2}$$

$$t_e(e_i) = \min\{t_e(u) \mid u \in I_{e_i}\}. \tag{2.3}$$

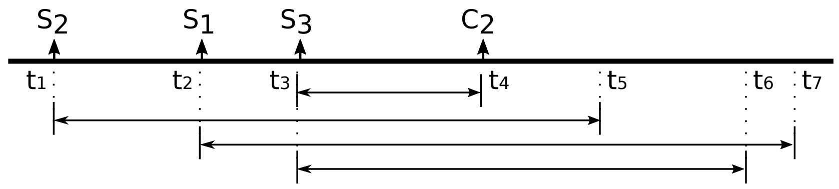


Figure 2.3: An example timeline of event processing for consumer c_2 in Figure 2.2. Each vertical arrow marks either the event creation times at the suppliers or the event arrival time at the consumer.

If e_i is from a supplier, $t_b(e_i)$ is defined to be the creation time of the event. For example, as shown in Figure 2.3, $[t_1, t_5)$, $[t_2, t_7)$, and $[t_3, t_6)$ respectively represent the absolute validity intervals of the events from s_2, s_1, and s_3, and an event for consumer c_2 is temporally valid as long as it would arrive at c_2 before t_5. We assume that each supplier event's absolute validity interval is also specified by domain experts developing a particular application.

2.3.3 Relative Time Consistency

Here we reuse the definition of absolute time consistency. In general, we say that sets of events required by an operator may have relative time validity constraints, and each such constraint describes a mutually dependent timed relation between the events in a set. Formally, we say that I_{e_i} is temporally valid, if given sets $Q_j \subseteq I_{e_i}$, $j > 0$, we have

$$|t_b(e_x) - t_b(e_y)| \leq R_{Q_j} \tag{2.4}$$

for every two events e_x and e_y in Q_j. We call R_{Q_j} the *relative validity interval* of the set Q_j. From Equation (2.4), equivalently, I_{e_i} is temporally valid if

$$|t_b(e_p) - t_b(e_q)| \leq R_{Q_j} \tag{2.5}$$

for all Q_j, where event e_p is the earliest created and event e_q the latest created in each Q_j. Equation (2.5) offers an efficient way to verify the relative time consistency at run-time, which we will discuss in Section 2.4.3. As with absolute time consistency, we assume that set Q_j and interval R_{Q_j} are specified by domain experts.

2.4 CPEP Design

CPEP processes cyber-physical events as follows. First, the graph of event processing streams is constructed from a configuration file, which specifies a list of the needed operators, with each item containing the operator type, the number of operators that immediately follow, and the indices to those operators. For each operator whose output event would be subscribed by a consumer, the operator is associated with a priority level mapped from the consumer's QoS specification. The event service assigns priority levels to the other operators by propagating upstream the priority levels of the consumer-facing operators, where each operator is assigned the highest priority level among its downstream operators. For example, the operators in Figure 2.2 would be partitioned into three priority groups (high: o_1, o_5; middle: o_2, o_6; and low: o_3, o_4, o_7). A supplier event's priority level is set to the highest priority level among the supplier-facing operators that would use it. With that priority assignment, the event service then reacts to the events pushed from suppliers, processes them according to the graph of event processing streams, and pushes the resulting events to consumers.

2.4.1 Prioritized Processing and Sharing

The top-level component for processing is an *EventProcessor*, and there is one EventProcessor per priority level, as is illustrated in Figure 2.4. An EventProcessor includes two sets of active objects [48]: a set of *worker threads* in charge of processing same-priority events, and

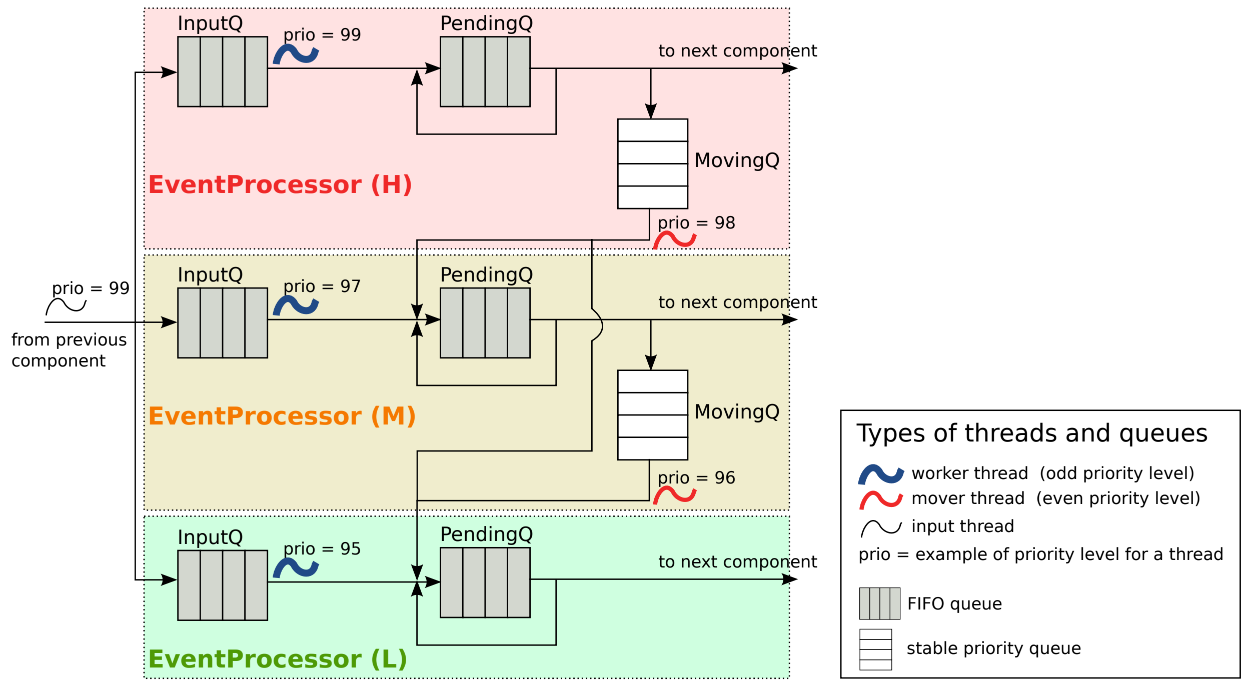


Figure 2.4: The CPEP Architecture (H: high priority, M: middle priority, L: low priority).

a set of *mover threads* in charge of sharing the events that carry results of processing across priority levels (e.g., $o_1 \rightarrow o_6$, $o_2 \rightarrow o_3$, and $o_5 \rightarrow o_7$ in Figure 2.2).

A worker thread executes each operator that is *ready* due to arrival of a supplier event and/or completion of its upstream operator(s), and will proceed to process the next supplier event only when there remain no such pending operators. A mover thread shares the processing result in a tuple that contains both a reference to the pending operator and a reference to the resulting event.

CPEP prioritizes processing of streams and enforces the following two properties: (1) any processing of a certain priority level will preempt any cross-priority sharing from the same (or a lower) priority level; and (2) any cross-priority sharing from a certain priority level will preempt any processing of a lower priority level. This is achieved by assigning adjacent thread-level priorities to the worker and mover threads and scheduling them using a fixed priority preemptive scheduling policy: starting from the EventProcessor of the highest priority level, we first assign all its worker threads the highest thread-level priority, and then assign all its mover threads the next thread-level priority. We then repeat the process for the EventProcessor at the next priority level, using the remaining thread-level priorities. An example priority assignment is shown in Figure 2.4.

Each EventProcessor has three queues. The *InputQ* buffers all supplier events of the same priority level as that of the EventProcessor. The *PendingQ* holds the tuples for the subsequent same-priority operators along the graph of processing streams, and the *MovingQ* holds the tuples for cross-priority sharing. If cross-priority sharing is needed, the current worker thread puts the corresponding tuple into the MovingQ. An idle mover thread then moves the tuple from the MovingQ to the PendingQ(s) of the destination EventProcessor(s), which is then processed by the worker thread of each destination EventProcessor.

2.4.2 Concurrent Processing and Replacement

To improve throughput and reduce latency, on a multi-core platform CPEP can deploy multiple same-priority worker threads and execute independent operations concurrently (including independent portions of a complex operation), and different-priority worker threads can concurrently work using different CPU cores when possible. Concurrent processing is available in the following two circumstances: (1) when there are multiple event arrivals, be it from suppliers or from some preceding operators (for example, via sharing), and (2) when an operator that is followed by multiple operators produces an event. In both cases, the PendingQ will be populated by multiple tuples. Concurrent processing is made possible in the following two ways: (1) *Collaborative*: idle worker threads can take pending operators from the PendingQ after others have populated it. For example, operators o_3 and o_4 in Figure 2.2 may be processed concurrently. (2) *Pipeline-like*: CPEP allows a new series of processing along the graph of streams to start before the completion of the current series. For example, processing for operators o_1 and o_2 in Figure 2.2 may start even before the completion of processing for operator o_6.

For each EventProcessor, we set both the number of worker threads and the number of mover threads to be equal to the number of CPU cores that are dedicated for processing, but do not pin them to particular CPU cores. This can improve resource utilization and reduce processing latency[2]; for example, threads of lower priority levels, upon being preempted, can migrate to available CPU cores.

With concurrent and prioritized processing, for an operator requiring multiple events (e.g., operator o_6 in Figure 2.2), it is possible that an upstream operator may produce a second copy of a previously delivered event while the operator is waiting for an event from another operator upstream. In this case, the worker thread taking the second event will replace the previous event by it. Such replacement occurs each time a new event arrival is available, until the needed event from the other upstream operator is available. The replacement only takes effect on the immediate operator that receives the event, and other worker threads processing operators downstream from it will keep using the event they took when processing that operator.

CPEP can be configured to also enforce event replacement in the InputQ, in which case the buffer length is at most the number of different supplier event types, and each new event arrival of the same type will replace the previous event as long as the previous one has not yet been dequeued. Without event replacement, lower-priority InputQ's buffer needs to be large enough to accommodate preemption. The buffer length can be determined from workload profiling.

[2]See related discussion for multi-processor global scheduling and partitioned scheduling [22].

For both time and space efficiency, internally CPEP maintains a single instance for each event creation, and all workers may access the same instance concurrently if they need it. The event is stored in a centralized structure, named the *EventStore*, where each event is typed according to the supplier/operator that produced it. To accommodate pipeline-like concurrency, the EventStore includes one ring buffer per event type, and a new event of that type is put into the ring's next slot. The ring size is bounded by the maximum number of temporally valid instances of that event type at any given time point; for example, the ring size is equal to one if the event's absolute validity interval is smaller than the event's inter-arrival time. Slots are reclaimed in a lazy fashion, and only if there is a new event creation but no available slot. When needed, the slot holding the oldest event is reclaimed.

2.4.3 Time Consistency Enforcement and Shedding

CPEP enforces both absolute time consistency and relative time consistency, and can be configured to have worker threads either mark or shed time-inconsistent events. With marking, CPEP simply labels such events and lets consumers decide what to do with them. With shedding, CPEP aborts any subsequent processing. Validation of both types of consistency is performed upon the invocation of each operator in the event processing graph. In addition, the absolute time consistency is also validated when the processing result is to be pushed to a consumer.

Absolute Time Consistency Validation

Given event e_i, to validate absolute time consistency, a worker thread compares the current time t against $t_e(e_i)$, i.e., the end time of the absolute validity interval. The worker thread reports a violation if $t > t_e(e_i)$. Upon a violation, if CPEP is configured to shed time-inconsistent events, the worker thread will update the value of $t_e(e_i)$ to the earliest end time among the absolute validity intervals of events on time-consistent upstream branches, and will remove the event references of the time-inconsistent upstream branch.

Relative Time Consistency Validation

Let $S = \{e_1, e_2, ..., e_k\}$ be a set of event types subject to a specified relative validity interval, and following Equation (2.5) we say that the relative time consistency is violated if the maximum time difference of any two events in S is larger than the specified interval. To validate such consistency, for each operator CPEP maintains an ordered list of timestamps, one timestamp per event type in S. When a worker thread invokes an operator with a new event, it first updates the list and then compares the time interval between the last and the first element in the list against the relative time consistency interval. The worker thread reports a violation if the latter interval is smaller.

The above design ensures the correctness of enforcement: for an operator, it is necessary to keep track of the timestamp of each required event, because event replacements may occur before all the needed events are available. For example, suppose we only keep track of the earliest timestamp, say t_1, and the latest timestamp, say t_2, with respect to S. Given the second arrival of an event type where its previous arrival has defined t_1. In this case, the second arrival will define t_2, but t_1 will become undefined since its previous definition was from the same event type that now defines t_2.

2.4.4 Discussion on Distributed Settings

In the current version of CPEP we describe a centralized service design, where a single service host processes all events. Extension to support distributed settings deserves further study but is beyond the scope of this dissertation. Here we discuss how the CPEP design may scale to support distributed settings. In particular, we focus on support for mapping the operators in the event processing graph onto multiple service hosts, to achieve distributed real-time cyber-physical event processing.

An ideal operator mapping would improve performance while preserving both latency differentiation and time consistency. In principle, (1) in terms of latency, where to map the operators involves a trade-off between inter-host communication delays and intra-host contention delays. Inter-host communication involves event transmission, and intra-host contention involves queueing and preemption. Therefore, for example, given an event processing graph with independent same-priority subgraphs, it may be advantageous to map operators of the same subgraph onto a single service host (thus minimizing inter-host communication) and operators of different subgraphs onto distinct service hosts (thus reducing intra-host contention). Further, it may be advantageous to map lower-priority operators onto multiple service hosts, provided that the amount of higher-priority interference on a service host outweighs the overhead of inter-host communication. (2) In terms of time consistency, events' validity intervals may drive mapping decisions, too. (3) In terms of memory efficiency, for multiple operators that share events, it may be advantageous to map them onto the same service host, to reduce additional event copies.

The operator mapping may be static, dynamic, or a hybrid. A static mapping may minimize the runtime overhead, with both the operators pre-allocated to each service host and the event routing pre-determined across service hosts. A downside is that the mapping may be pessimistic in the presence of aperiodic events. A dynamic mapping may reduce the pessimism at the cost of additional latency for runtime modules that must both decide where and how to route events and load operators if needed. A hybrid mapping also may be advantageous; for example, adopting static mapping for periodic events and dynamic mapping for aperiodic events.

2.5 CPEP Framework Implementation

In the architecture illustrated in Figure 2.4, we implemented the MovingQ using C++11's standard priority queue, and to preserve the ordering of same-priority items we customized the priority queue's Compare type to use the timestamp taken at insertion as a tie-breaker. We implemented the PendingQ using C++11's standard FIFO queue. For the configuration of event replacement, we implemented InputQ using C++11's standard FIFO queues to hold indices of each supplier event type. The indices are used to access a static storage for each supplier event type, and the size of the storage is equal to the number of different supplier event types. To enforce event replacement, each event arrival replaces the corresponding entry in the storage, and we push its index to the FIFO queue only if there is no such an index in the queue. For the configuration of no event replacement, we implemented InputQ using C++11's standard array as a ring buffer.

We protected the MovingQs, PendingQs, and InputQs with readers-writer locks to allow concurrent checks for non-emptiness of each queue. Each slot in the EventStore is also protected by a readers-writer lock to allow concurrent reads. To reduce priority inversion, we applied the pthread priority inheritance protocol to all worker threads and mover threads. At run-time, it takes $O(1)$ time to validate absolute time consistency, by comparing $t_e(e_i)$ against the current time. We maintain $t_e(e_i)$ by keeping track of the earliest end time for each upstream branch of $o(e_i)$.

The graph of event processing was implemented by an array of structs, with each struct including both data structures for a node in the graph and pointers that build the graph's topology. The size of the struct was 464 bytes, including padding. The construction of the graph needs two linear scans through the array of structs: one scan for propagating priority level information upstream, and another scan for propagating time consistency information

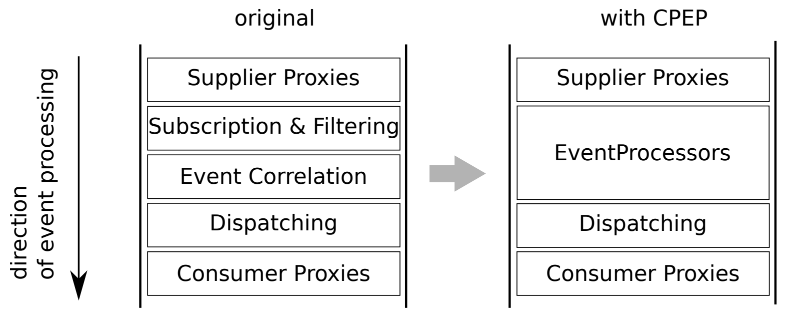


Figure 2.5: Implementation within the TAO event channel.

downstream. Notably, all the constructions will complete before the system starts processing events.

We implemented CPEP within the TAO real-time event service [38]. Event suppliers and consumers in TAO are connected via one or more *event channels*, each containing five modules, as shown in Figure 2.5. Event filtering is conducted at both the Subscription & Filtering module and the Event Correlation module, where the former filters events according to event's type and source ID, and the latter filters events according to correlation rules defined over event types. The Dispatching module dispatches events to the subscribed consumers. Prior to our work, the TAO real-time event service only supported simple correlations (logical conjunction and disjunction) over events' headers, with non-sharing filters built per consumer. In contrast, CPEP provides prioritized event processing, enforces time consistency, and enables sharing of operations for better performance.

In our implementation, we kept the original interfaces of the Supplier Proxies and the Consumer Proxies, so that suppliers and consumers can connect to the event channel as before. We replaced the Subscription & Filtering and Event Correlation modules with Event-Processors. We connected the Supplier Proxies to EventProcessors by a hook method within the push method of the Supplier Proxies module to put each event into the corresponding EventProcessor's InputQ. Worker threads dispatch their output events reactively.

2.6 Empirical Evaluation

Here we present six sets of experimental results. In Set 1 we compare CPEP against Apache Flink in terms of latency, throughput, and memory footprint; in Sets 2 to 5 we evaluate the effectiveness of CPEP in terms of prioritization, sharing, absolute time consistency shedding, and relative time consistency shedding, respectively; in Set 6 we present CPEP's overhead statistics. In all the experiments we enabled event replacement at InputQ; results of the configuration with no event replacement have been reported in our previous work [71].

Our test-bed consists of three hosts: on Host 1 we ran all event suppliers (Pentium Dual-Core 3.2 GHz, Ubuntu Linux with kernel v.3.19.0); on Host 2 we ran the CPEP event service and the Apache Flink server (Intel i5-4590 3.3 GHz four-core machine, Ubuntu Linux with kernel v.4.2.0); on Host 3 we ran all event consumers (Pentium Dual-Core 3.2 GHz, Ubuntu Linux with kernel v.3.13.0). We connected the three hosts via a Gigabit switch running in a closed LAN. Host 2 had two NICs, and we used one for inbound traffic from Host 1 and another for outbound traffic to Host 3. Out of its four cores, on Host 2 we used three cores in the first experiment set (a comparison study with Apache Flink), and in the remaining experiment sets we offloaded the inbound network IO and CPEP's Supplier Proxies to the fourth core, and the three cores were dedicated to event processing.

TAO's event channel was configured with the following parameters: we used the default factory, and we assigned *null* to `ECFiltering`, `ECSupplierFiltering`, `ECProxyConsumerLock`, and `ECProxySupplierLock`; *reactive* to `ECDispatching`; *reactive* to both `ECConsumerControl` and `ECSupplierControl`. Finally, we assigned zero to `ECConsumerControlPeriod` and zero to `ECSupplierControlPeriod`.

We assigned real-time priority levels to both worker threads and mover threads, with the highest priority level set to 99. We also assigned 99 as the priority level of the thread for Supplier Proxies. We did not use the PREEMPT_RT patch [29].

Events were supplied at different rates, and consumers of events were assigned different priorities. We used two graphs of processing streams, shown in Figures 2.6 and 2.7, and in each case we evaluated the performance under different degrees of system workload. The workload we implemented demonstrates standard multi-sensor fusion operations [25, 61, 64, 68], with the following four operators combined in different ways: Extended Kalman Filter (EKF) [76], Fast Fourier Transform (FFT) [30], Concatenation (CAT) (implemented using C++'s `memcpy` function), and the Advanced Encryption Standard (AES) [34]. We ensured that the operators are thread-safe by creating a distinct Kalman filter object per EKF operator and distinct FFTW matrices per FFT operator; the Libgcrypt library is by default thread-safe and we created a single Libgcrypt handler for all AES operators.

Because both absolute and relative time consistency enforcement use events' creation times to validate consistency, for event suppliers belonging to the same stream we chose to coordinate the phasing of event creations when evaluating absolute time consistency shedding (otherwise, the time difference between the creation times may dominate the latency and hence the shedding decision) and we chose not to coordinate event creations when evaluating relative time consistency shedding (otherwise, either all events would pass or all of them would be shed). Specifically, in the first four and the sixth experiment sets, we coordinated the event suppliers belonging to the same stream, and in the fifth experiment set we chose not to coordinate as we did in the others. We coordinated the phasing of all suppliers that are upstream from a common consumer event and that have the same event rate, by dispatching them all from the same timer expiration, rather than from individual independent timers.

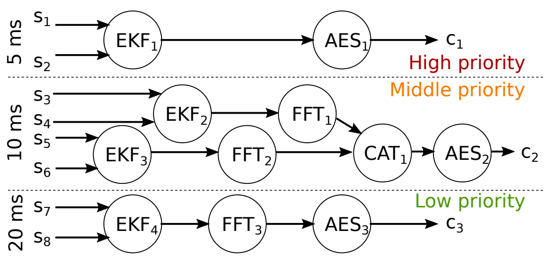
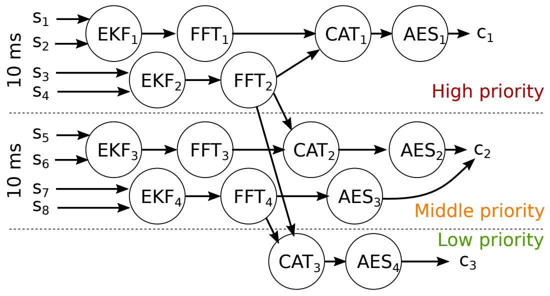
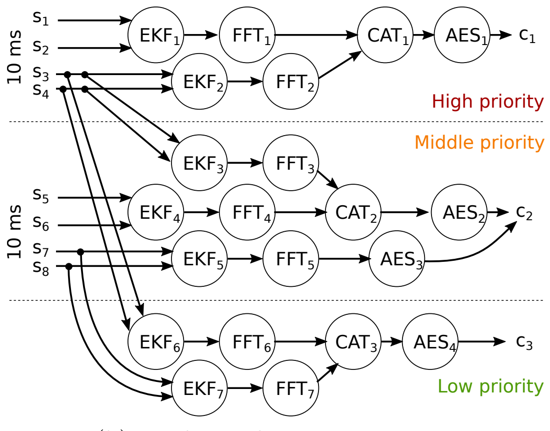


Figure 2.6: The graph of event processing streams for Experiment Sets 1, 2, and 5.



(a) With sharing operators.



(b) Without sharing operators.

Figure 2.7: The graphs of event processing streams for Experiment Sets 3, 4, and 6.

We measured the end-to-end latency, i.e., the time interval between the latest time a supplier pushed a required event and the time the consumer received the resulting event (e.g., $[t_3, t_4]$ in Figure 2.3). We synchronized our test-bed's hosts via PTPd [33], an open source implementation of the IEEE Std. 1588-2008 Precision Time Protocol [40]. Both the service host's clock and the consumer host's clock were synchronized to the clock of the supplier host. The synchronization error was within 0.05 milliseconds.

In each of the first four experiment sets, we ran each sub-case ten times and calculated the 95% confidence interval for each measurement; in the fifth experiment set, where we chose not to coordinate event suppliers, we ran each sub-case fourty times so that the 95% confidence interval converged. In each sub-case we sequentially ran three phases: warm-up, measuring, and dumping. In the warm-up phase we connected all event suppliers and consumers to CPEP and had them start pushing and receiving events; in the measuring phase we measured both the latency and the throughput of output events, and we kept all the measurements in memory, which were then saved to disk in the dumping phase. For CPEP, the warm-up phase took ten seconds. Apache Flink requires a longer warm-up time and we set it to 75 seconds. In both cases, the measuring phase spanned 100 seconds.

2.6.1 Experiment Set 1: Comparison with Apache Flink

Figure 2.6 shows the graph of event processing streams used in this experiment set. To cover different degrees of workload, we first deployed three copies of the high-priority streams, three copies of the middle-priority streams, and twelve copies of the low-priority streams, and then we increased the workload by deploying more copies of the middle-priority streams. Each supplier event carried a batch of one-byte datapoints[3]: each event supplied by s_1 and s_2 carried 512 datapoints (5 ms event inter-arrival time); s_3 to s_6, 1024 datapoints (10 ms); and s_7 and s_8, 2048 datapoints (20 ms). Each output event for a high-priority consumer carried 512 one-byte datapoints; each output event for a middle-priority consumer carried 1024 16-byte datapoints[4]; each output event for a low-priority consumer carried 2048 eight-byte datapoints.

[3]For example, in structural health monitoring[37], FFT may require 2048 samples to perform, and raw data may need to be transmitted to a base station if in situ processing is not sufficient.

[4]The FFT operator caused the increase in datapoint size, as the FFTW library transformed each byte of datapoints into an eight-byte real number (the single precision version of FFTW).

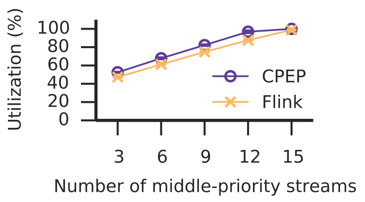
Table 2.1: Experiment Set 1: The 99th percentile latency (ms).

Priority	Service	Number of middle-priority streams				
		3	6	9	12	15
High	Flink	3.8 ± 0.1	5.9 ± 0.2	12.6 ± 0.4	52.6 ± 4.1	448.9 ± 171.7
	CPEP	0.8 ± 0.0	0.7 ± 0.0	0.7 ± 0.0	0.7 ± 0.0	0.7 ± 0.0
Middle	Flink	4.5 ± 0.1	6.4 ± 0.2	11.3 ± 0.4	28.9 ± 0.5	107.9 ± 18.1
	CPEP	1.6 ± 0.0	1.8 ± 0.0	2.2 ± 0.0	2.5 ± 0.0	3.0 ± 0.1
Low	Flink	5.2 ± 0.3	7.4 ± 0.2	15.5 ± 0.6	43.3 ± 1.3	679.8 ± 274.0
	CPEP	3.7 ± 0.3	4.8 ± 0.2	6.8 ± 0.6	10.6 ± 1.0	33.4 ± 0.2

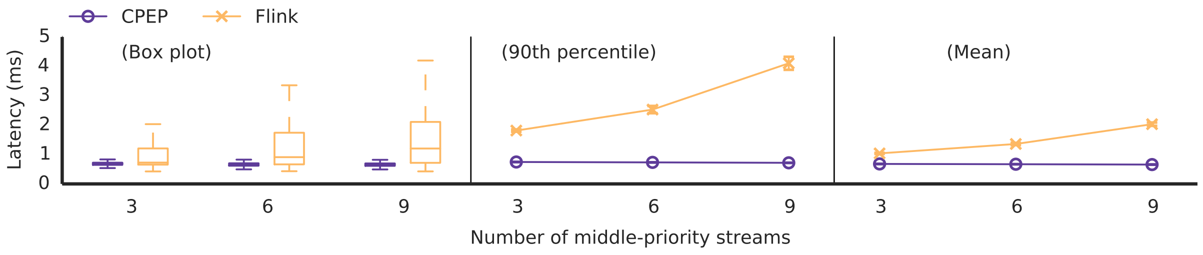
For a fair comparison to CPEP, in our Apache Flink (we simply call it Flink hereafter) Java application we used the same C++ libraries for the EKF, FFT, and AES operators, and invoked them via Java JNI. Since in CPEP we focus on single-host event processing, we configured Flink to run in a local environment, with a single Flink JobManager and TaskManager, respectively, and we ensured that they ran within the same JVM. To utilize multiple CPU cores, we set Flink's parallelism level equal to the number of CPU cores used for event processing (i.e., three), and also had the TaskManager use three task slots for concurrent execution. We implemented a socket data source to connect Flink with event suppliers, and a socket data sink to connect it with event consumers. We chose not to use Flink's Complex Event Processing library [32], because so far it does not support the removal of used events to avoid premature processing, which is required by cyber-physical event processing (see Section 2.3.1). We therefore implemented event processing using Flink's built-in data transformation functions. We also turned off checkpointing to reduce latency in Flink.

Latency Comparison

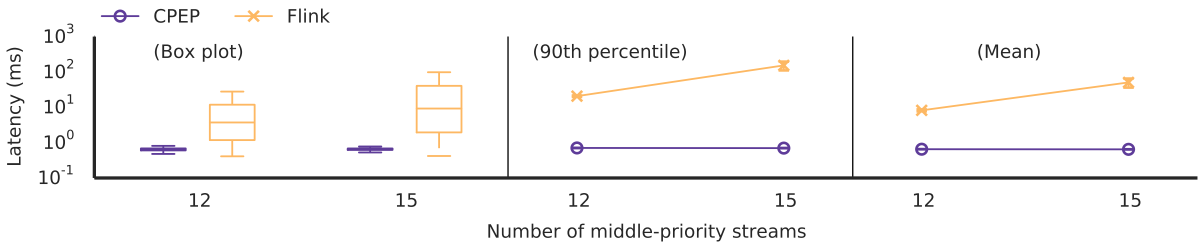
The latency results are shown in Figure 2.8 (high priority), 2.9 (middle priority), and 2.10 (low-priority) and Table 2.1; Figure 2.8(a) shows the CPU utilization normalized to the



(a) CPU utilization.



(b) Moderate workload (CPU utilization = 40–80%).

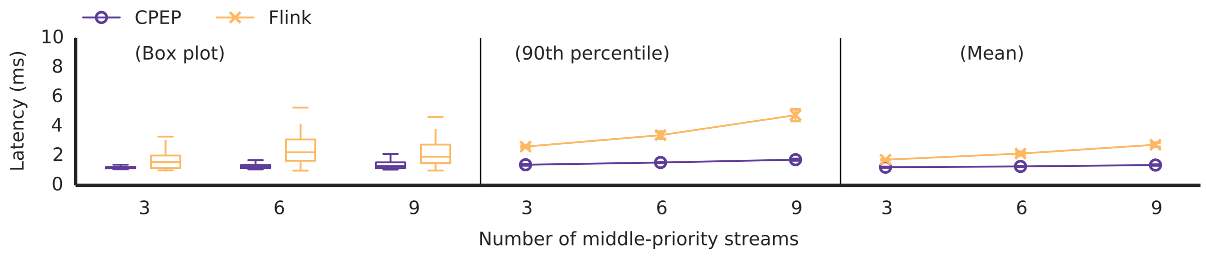


(c) Heavy workload (CPU utilization $>$ 80%).

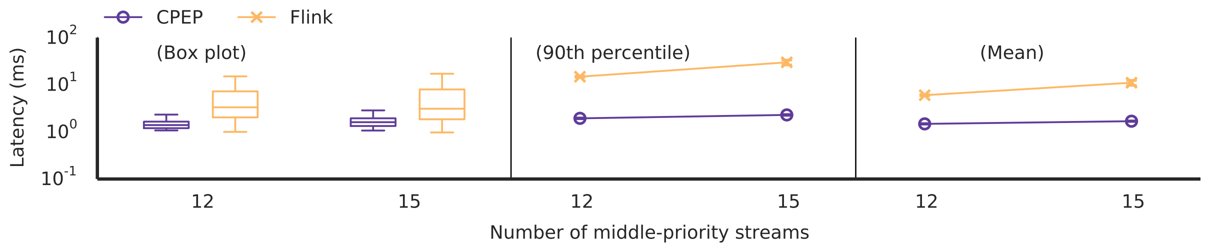
Figure 2.8: Experiment Set 1: Latency results of high-priority streams.

number of cores. The results show that CPEP outperformed Flink in terms of latency in all degrees of workload, and CPEP's latency performance followed the order of priority level. A major reason is that CPEP prioritizes event processing based on the consumer priority levels, while Flink does not. In CPEP, tasks of higher priority will preempt lower-priority tasks' execution, while in Flink tasks of higher priority may need to wait for the execution of tasks of lower priority.

Here we qualify the latency results by the corresponding inter-arrival times of events. In general, latency longer than the inter-arrival time implies that there are newer data available

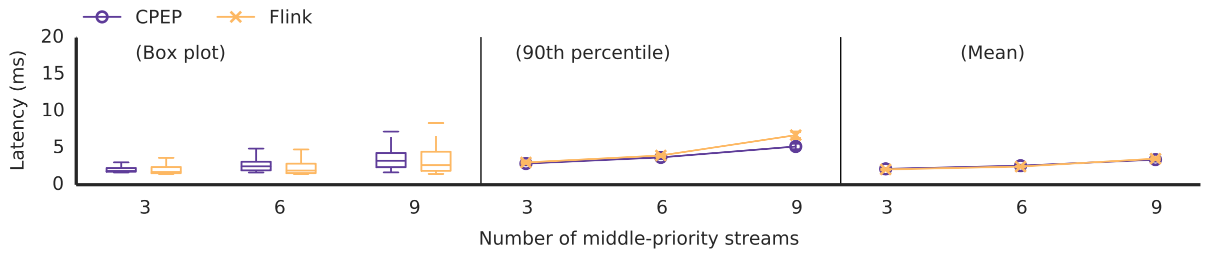


(a) Moderate workload (CPU utilization = 40–80%).

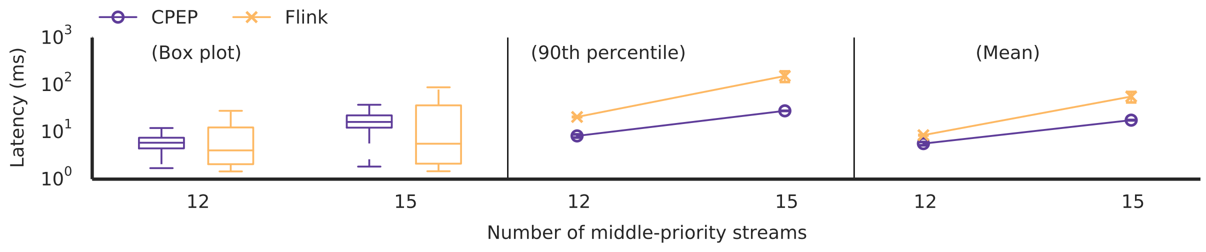


(b) Heavy workload (CPU utilization > 80%).

Figure 2.9: Experiment Set 1: Latency results of middle-priority streams.



(a) Moderate workload (CPU utilization = 40–80%).



(b) Heavy workload (CPU utilization > 80%).

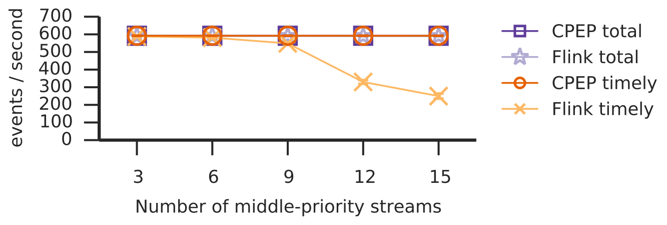
Figure 2.10: Experiment Set 1: Latency results of low-priority streams.

before a consumer has received the processed result. In practice, it is desirable to have latency even shorter than the inter-arrival time, and latency beyond the inter-arrival time may not be acceptable. Under moderate workload (CPU utilization = 40–80%), both CPEP and Flink had 90th percentile latency and mean latency shorter than the corresponding inter-arrival time of events; under heavy workload (CPU utilization $>$ 80%), however, the latency of Flink may exceed the inter-arrival time of events. The high-priority streams, for example, had about 10 ms mean latency (Figure 2.8(c)), which is twice the inter-arrival time.

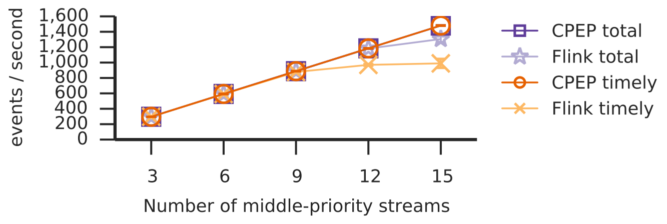
For the 99th percentile latency, as shown in Table 2.1, CPEP's latency was shorter than the inter-arrival time in every sub-case except for that of low-priority streams along with 15 middle-priority streams (33.4 ± 0.2 versus 20 ms); in this sub-case, the CPUs were nearly saturated, and the latency of Flink was also beyond the inter-arrival time of events (679.8 ± 274.0 ms versus 20 ms). For the low-priority streams, both the 90th percentile latency and mean latency of CPEP were similar to those of Flink, except for the aforementioned sub-case.

Throughput Comparison

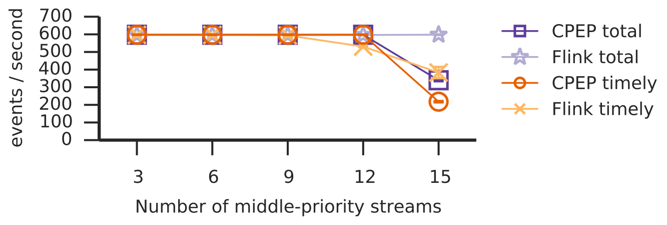
The throughput results for each priority level are shown in Figure 2.11. The plots labeled by *CPEP/Flink total* show the throughput in terms of total events received by a consumer, and the plots labeled by *CPEP/Flink timely* show the throughput in terms of events received with latency shorter than the inter-arrival time. For the high-priority consumers (Figure 2.11(a)), both CPEP and Flink produced events at rates close to the event rates at the suppliers (three high-priority streams), but Flink produced much less timely events as we increased the workload: for example, in the presence of 12 middle-priority streams, Flink only produced about 300 events per second, half of the ideal rate. For the middle-priority consumers (Figure 2.11(b)), Flink started to produce fewer events as we increased the workload. For the low-priority consumers (Figure 2.11(c)) we made the same observation for CPEP. In general,



(a) High priority



(b) Middle priority

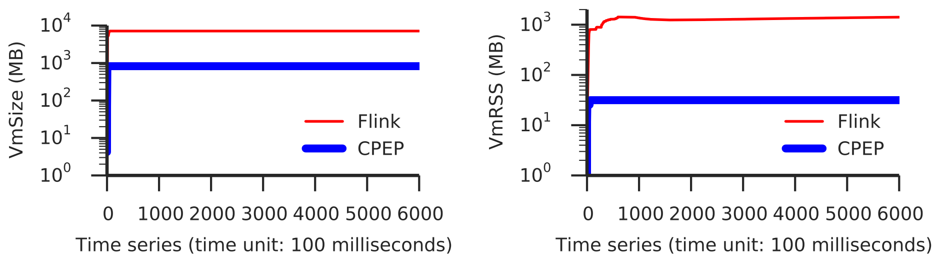


(c) Low priority

Figure 2.11: Experiment Set 1: Throughput results of each priority level.

CPEP and Flink behaved differently under heavy workload: in CPEP the greater latency accrued to events of the lowest priority level, thanks to the prioritized processing; in Flink the latency was distributed to all events, since Flink does not differentiate processing for different priority levels.

The prolonged latency also suggests that, in terms of absolute time consistency, many events may not be considered valid, hence both wasting CPU resources and unnecessarily increasing delay. Indeed, many more violations of absolute time consistency can occur, since the latency here only accounted for the duration between the last event creation and the event delivery (e.g., $[t_3, t_4]$ in Figure 2.3) while the absolute time consistency accounts for the duration



(a) VmSize. (b) VmRSS.

Figure 2.12: Experiment Set 1: Memory footprint comparison.

between the first event creation and the ultimate event delivery (e.g., $[t_1, t_4]$ in Figure 2.3). In Experiment Set 4, we evaluate CPEP's absolute time consistency enforcement and its shedding strategy. In the remaining five experiment sets, we do not use Flink and instead focus on evaluating CPEP's performance with different configurations, since CPEP clearly outperformed Flink in this comparison.

Memory Footprint Comparison

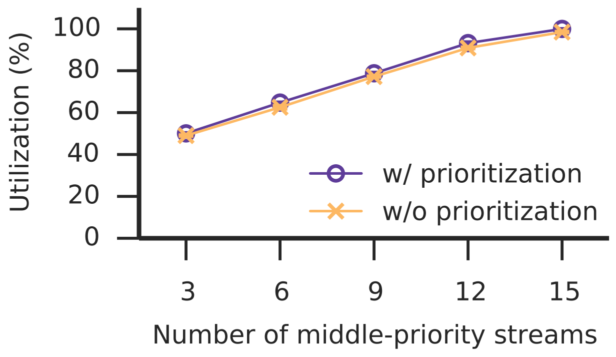
We empirically compared the memory footprint of Flink and CPEP by querying the kernel data structures via `/proc/[pid]/status`. We show in Figure 2.12 the result of virtual memory size (VmSize) and the virtual memory resident size (VmRSS). Compared with Flink's memory usage, CPEP reduced more than 88% of the max VmSize (from 7146.5 MB to 824.6 MB) and more than 97% of the max VmRSS (from 1424.8 MB to 31.5 MB). We also measured Java JNI's impact on the memory footprint, via the `pmap` utility. The sampling rate was 100 milliseconds and we took 600 samples. The VmRSS of the JNI task was always 48 KB, which is less than 0.005% of the total VmRSS.

2.6.2 Experiment Set 2: CPEP Prioritization

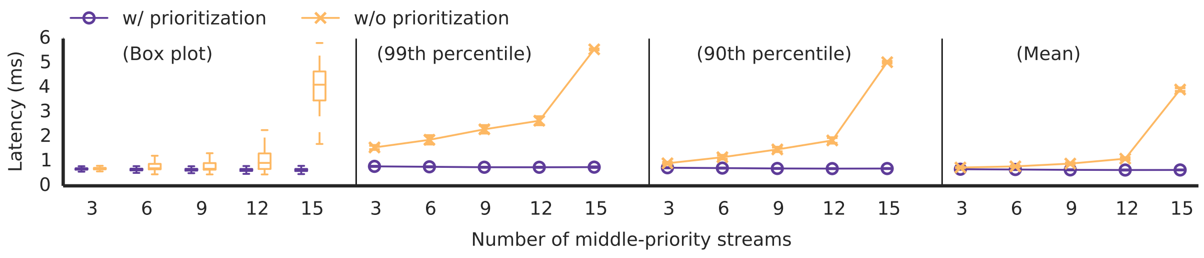
CPEP can be configured for either prioritized or non-prioritized processing. In this experiment set we compared the event latency of prioritized processing versus that of non-prioritized processing.

Figure 2.13 shows latency comparisons under different system workloads, with CPU utilization from around 45% to 95%, normalized to the number of cores used in processing event operations. As shown in Figure 2.13(b), prioritization maintained the latency of high-priority streams across different workloads. In contrast, without prioritization, the latency increased as the workload increased. Middle-priority streams exhibited similar behavior, shown in Figure 2.13(c). Low-priority streams exhibited the opposite behavior, shown in Figure 2.13(d), where prioritization led to higher latency for them than no prioritization did. This was because prioritization caused preemption of lower-priority processing. Nevertheless, the resulting latency was still less than half of the inter-arrival time of events (except for the sub-case of 15 middle-priority streams) and, as the next experiment set will show, sharing operations can further reduce the latency.

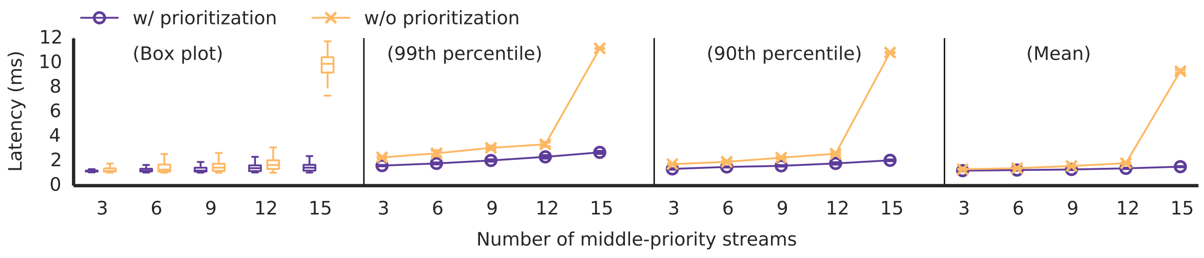
Figure 2.13 also shows that streams may have high tail latency even though the system was not heavily loaded (for example, the 99th percentile latency of high-priority streams in the sub-case of six middle-priority streams, with the normalized CPU utilization 63% (Figure 2.13(b))). This happened because event arrivals of different streams were independent of each other and sometimes arrived close in time and contended heavily with each other. A stream may experience a higher tail latency under a higher workload, because in this case the processing of the stream is more likely to be delayed due to such contentions.



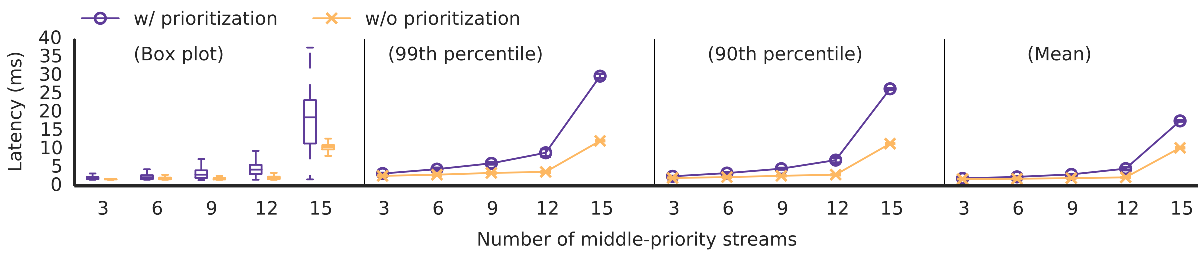
(a) CPU utilization.



(b) High priority.

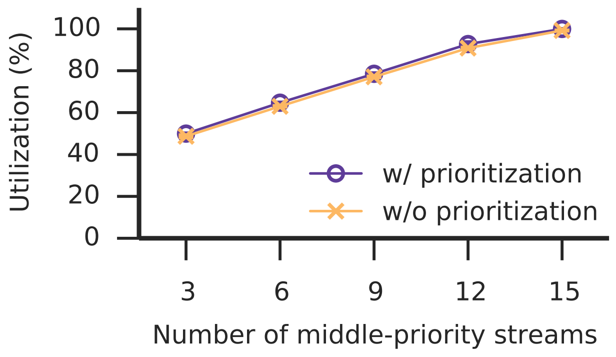


(c) Middle priority.

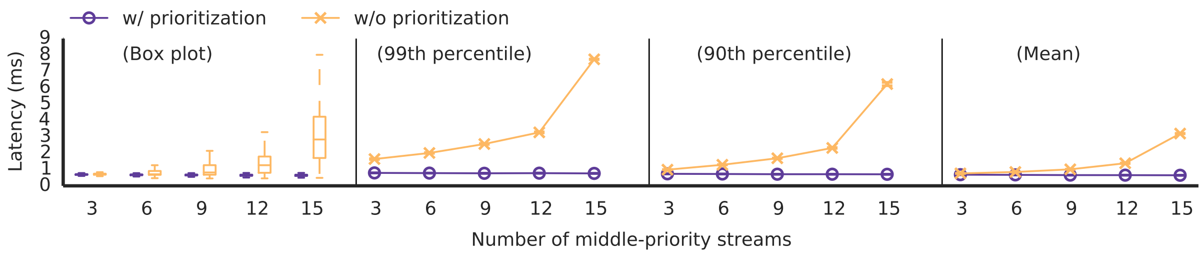


(d) Low priority.

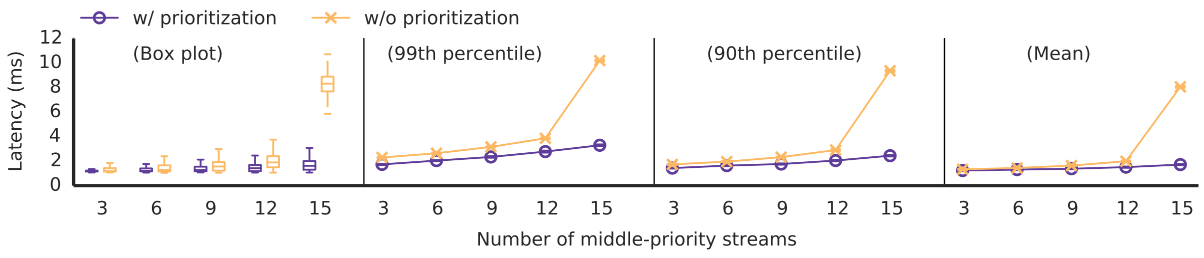
Figure 2.13: Experiment Set 2: Latency results.



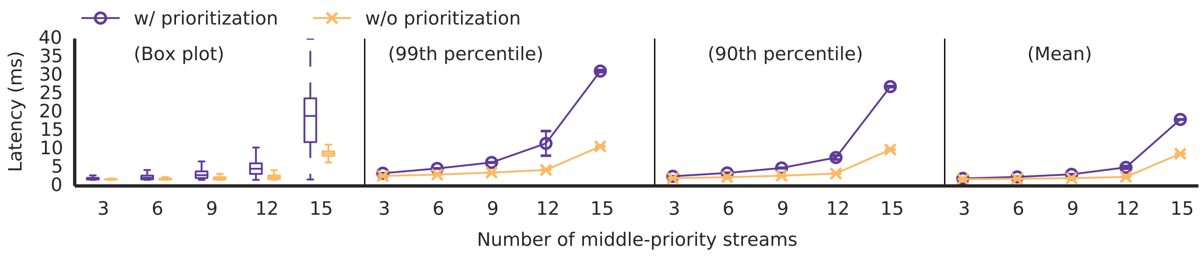
(a) CPU utilization.



(b) High priority.



(c) Middle priority.



(d) Low priority.

Figure 2.14: Experiment Set 2: Latency results (sporadic events).

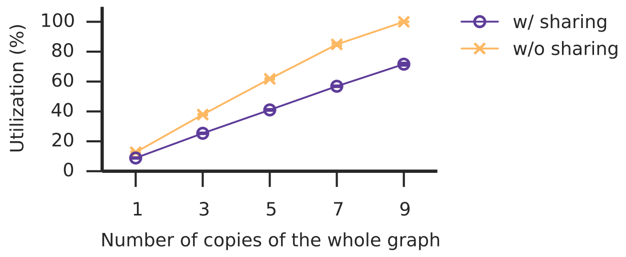
Results for Sporadic Events

We evaluated CPEP's performance with sporadic event streams, by adding a random time offset in the range of ± 2.5 ms to the inter-arrival time of each event supplier. The result is shown in Figure 2.14. The performance is similar to the case without random time offsets (Figure 2.13), confirming that CPEP works for both periodic and sporadic event streams.

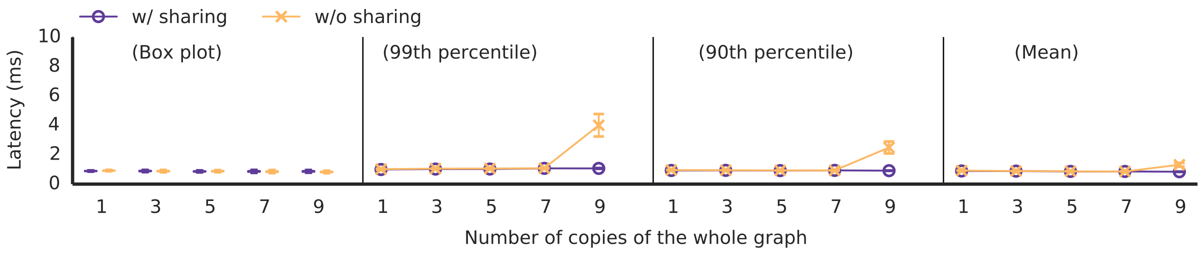
2.6.3 Experiment Set 3: Sharing Operators

In this experiment set we evaluated the latency performance of sharing operations. We also enabled prioritized processing. We configured the graph of event processing streams as shown in Figure 2.7. Each event supplied by s_1 to s_4 carried 512 datapoints; s_5 to s_8, 1024 datapoints. The non-sharing version was constructed from the sharing version by duplicating the shared operators (FFT_2 and FFT_4) and all their upstream operators (EKF_2 and EKF_4). All suppliers generated events with inter-arrival time of 10 ms. Because the streams share operations, here we varied workload by deploying copies of the whole graph.

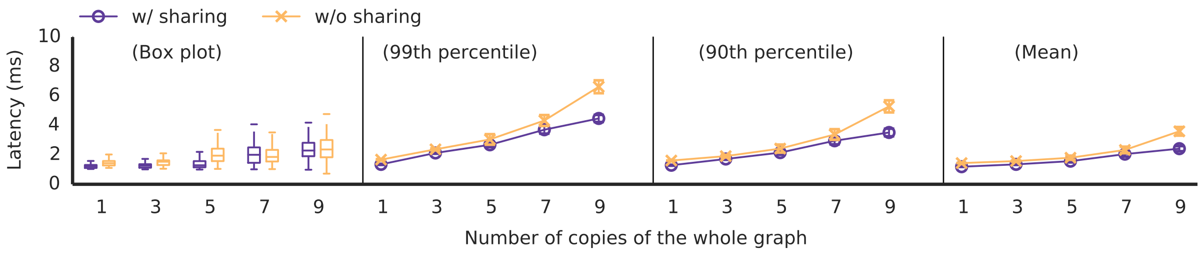
The latency results shown in Figure 2.15 confirm that sharing operations can help reduce event latency. In particular, lower-priority streams received higher reductions in latency, because the shared operations were done by the higher-priority counterpart. Figure 2.15(d) shows that as the workload increased, with sharing we may reduce the latency of lower-priority streams by more than 30% (e.g., the sub-case with seven copies of the whole graph), and with about the same mean latency the system can accommodate 40% more streams (e.g., from five to seven copies of the whole graph). Besides reducing latency, sharing also saved CPU utilization. Figure 2.15(a) shows that with five copies of the whole graph, sharing can save about 20% in CPU utilization.



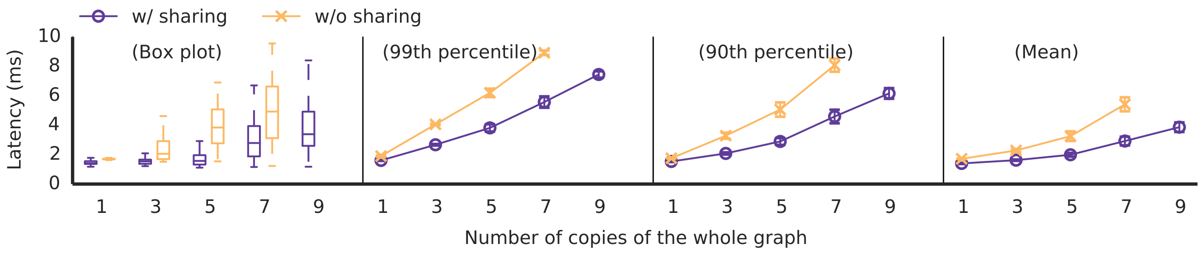
(a) CPU utilization.



(b) High priority.



(c) Middle priority.



(d) Low priority.

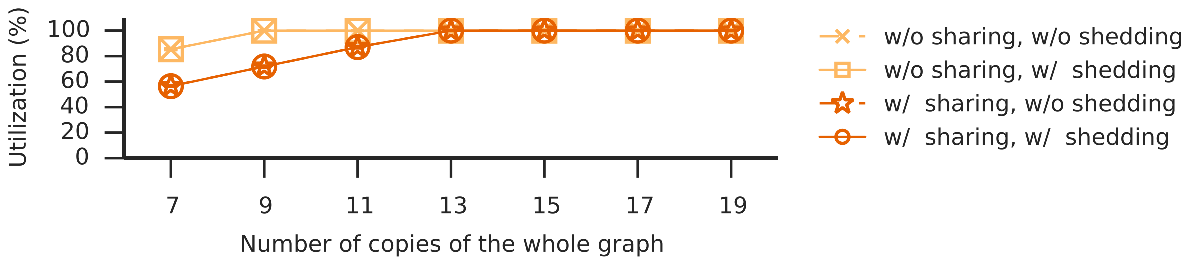
Figure 2.15: Experiment Set 3: Latency results; there was no output event of low priority for the case 'w/o sharing' in presence of nine copies of the whole graph.

For lower-priority streams, the result in Figure 2.15(d) also suggests that the savings in processing time due to sharing may outweigh the time spent waiting for higher-priority streams. For higher-priority streams, as shown in Figure 2.15(b) and (c), sharing processed events to lower-priority counterpart did not incur much overhead, either.

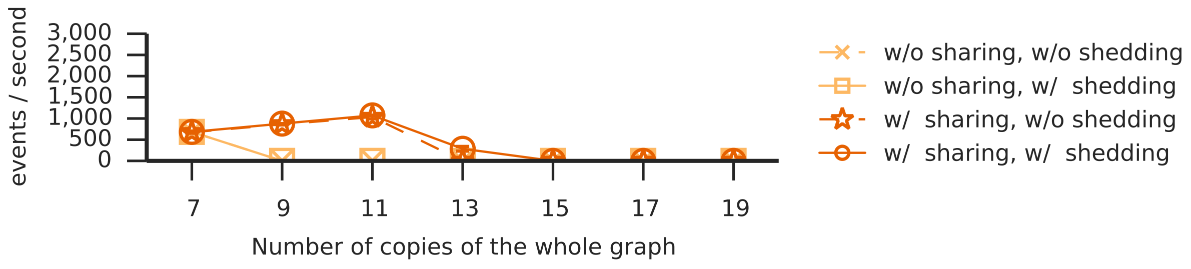
2.6.4 Experiment Set 4: Enforcing Absolute Time Consistency

In this experiment set we evaluated the performance of absolute time consistency shedding and sharing, both separately and combined, in terms of *timely-throughput*, i.e., delivery of events within their absolute validity intervals. Here we reuse the graph of processing streams as shown in Figure 2.7. We set the absolute validity interval to 10 ms.

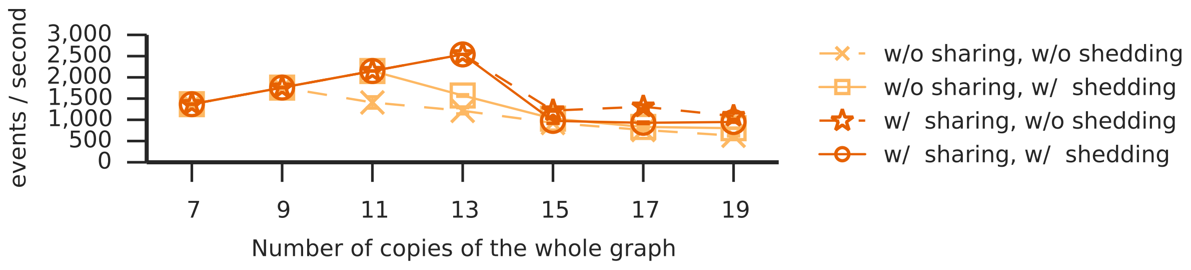
Figure 2.16 shows the result of timely-throughput in terms of events per second, where Figure 2.16(a) shows the CPU utilization. The benefit of shedding for timely-throughput was bounded by a range of system load, and for streams of higher priority the range shifted toward higher loads. This occurs because shedding did not take place under trivial loads, and because under heavy loads there is less slack time for streams to exploit. Since the absolute validity intervals approximate the supplier events' inter-arrival times, shedding occurred only when the total processing demand overloaded the CPUs. In particular, without sharing and with 11 copies of the graph of streams, the CPUs were saturated and shedding helped improve the timely-throughput of middle-priority streams (Figure 2.16(c)). With sharing and with 15 or more copies of the graph of streams, high-priority processing demand dominated CPU resources and therefore the timely-througput of middle-priority streams dropped. Under such conditions, not shedding middle-priority demand at operators may improve timely-throughput, because the incurred delay may lead to event replacements in InputQ, which act essentially as early removals of events that, if otherwise dequeued, would still become out-dated before processing completion.



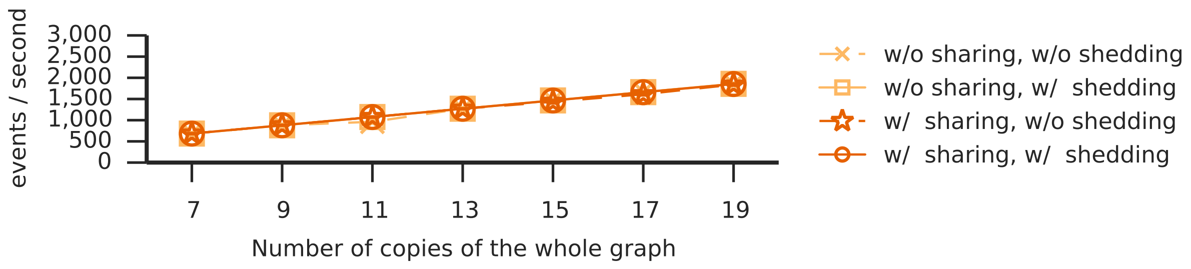
(a) CPU utilization.



(b) Low priority.



(c) Middle priority.



(d) High priority.

Figure 2.16: Experiment Set 4: Timely-throughput under different loads.

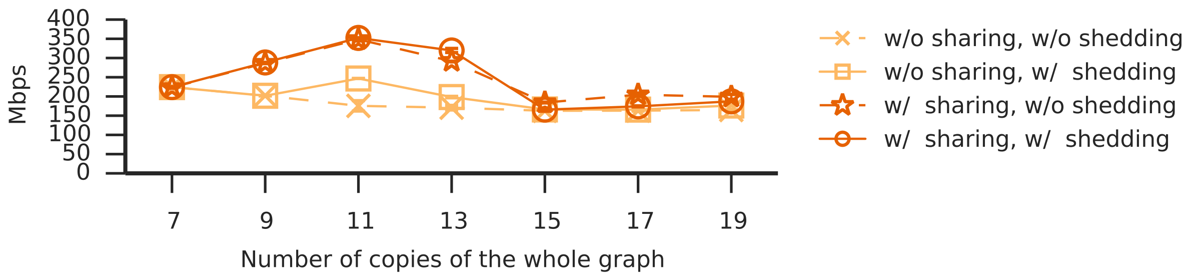


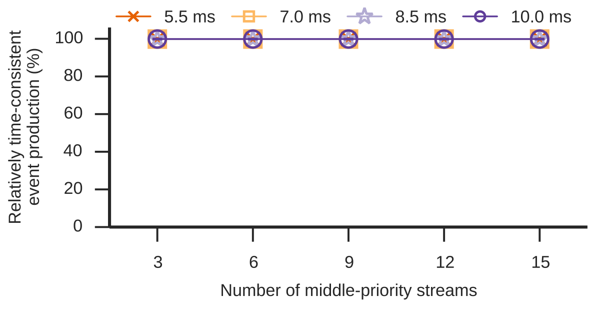
Figure 2.17: Experiment Set 4: Total timely-throughput (Mbps).

High-priority streams (Figure 2.16(d)), although already protected by prioritization, still benefited from shedding, because when the system is overloaded shedding can reduce the amount of intra-priority contention, i.e., the contention between the out-dated processing and the processing that works to meet the timing constraint. Low-priority streams (Figure 2.16(b)) benefited most from sharing, but may not benefit much from shedding because (1) shedding works best when the system is under heavy load, in which case the lower-priority streams may already suffer nontrivial preemption, and (2) the benefit of shedding would accrue first to higher-priority streams due to prioritization.

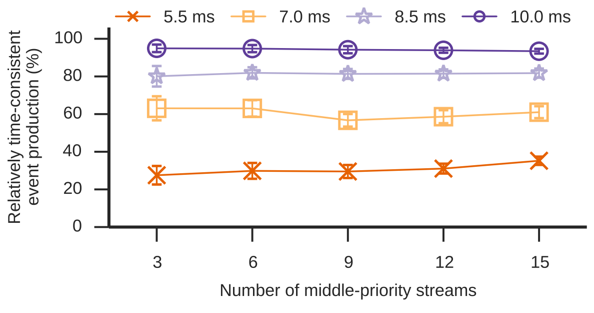
Figure 2.17 shows the timely-throughput and includes all three priority levels. It shows that without sharing, even though the CPUs had been saturated in the presence of nine copies of the graph, with the help of shedding the system can keep producing at 200 Mbps with up to 19 copies of the graph of streams; with sharing, we could still save about 20% CPU utilization in the presence of 11 copies of the graph.

2.6.5 Experiment Set 5: Enforcing Relative Time Consistency

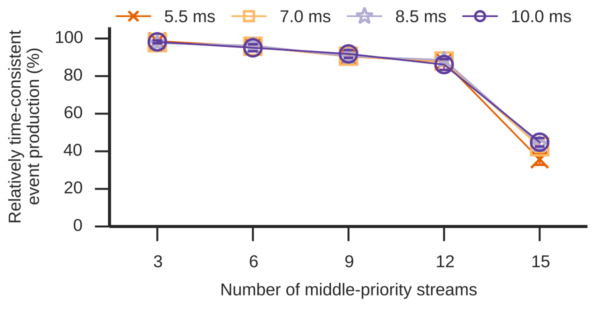
In this experiment set, we used the graph of event processing streams as shown in Figure 2.6, and we did not coordinate the suppliers for each stream. We set the relative validity interval to 5 ms for both high-priority and low-priority streams, and for the middle-priority streams



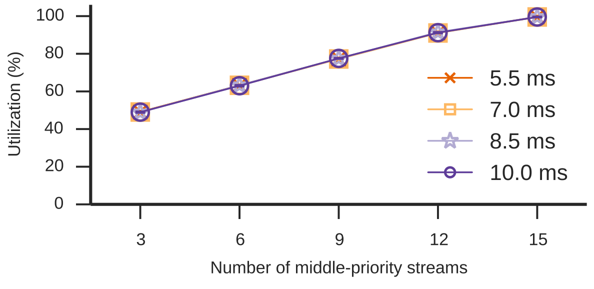
(a) High-priority events.



(b) Middle-priority events.

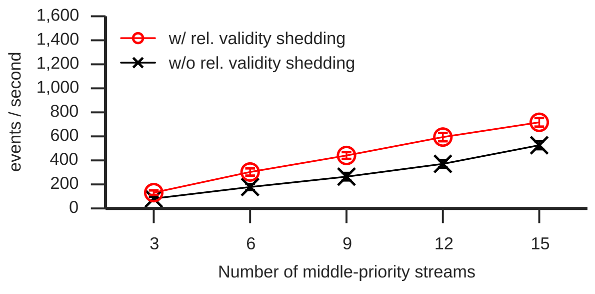


(c) Low-priority events.

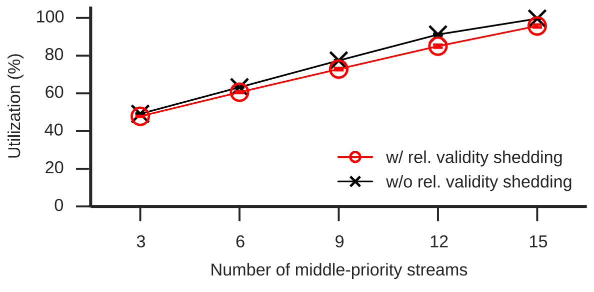


(d) CPU utilization.

Figure 2.18: Experiment Set 5: Percentage of relatively valid events with no relative validity shedding.



(a) Middle-priority throughput.



(b) CPU utilization.

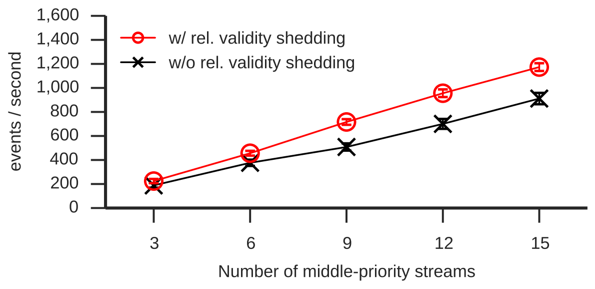
Figure 2.19: Experiment Set 5: Relative validity shedding with relative validity interval = 5.5 ms.

we evaluated four relative validity intervals: 5.5 ms, 7.0 ms, 8.5 ms, and 10.0 ms. In all cases, we enabled prioritized processing and did not enable absolute validity shedding. In the following, we first show the performance with no relative validity shedding, and then we show the performance improvement by enabling relative validity shedding.

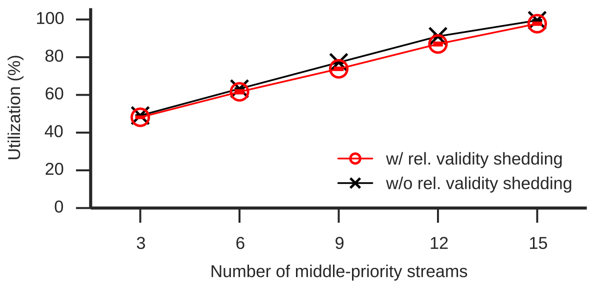
Effects of Violations of Relative Time Consistency

For the cases of no relative validity shedding, Figure 2.18(a) to (c) shows the percentage of relatively time-consistent events produced, and Figure 2.18(d) shows the CPU utilization, for each case of the middle-priority relative validity interval. First of all, Figure 2.18(a) shows that the high-priority streams were protected by prioritized processing; they were not affected by different middle-priority validity intervals, nor were they affected by the increase in the number of middle-priority streams. Figure 2.18(c) shows that, also due to prioritized processing, the low-priority streams had a decrease in the percentage of relatively time-consistent events produced as we increased the number of middle-priority streams.

Figure 2.18(b) shows the results of middle-priority streams. The middle-priority streams are subject to more relative time consistency violations, because they depend on four event types, two more than the high-priority streams and middle-priority streams (see Figure 2.6). Figure 2.18(b) shows that, with a longer relative validity interval, there was an increase in the percentage of relatively time-consistent event production. With the relative validity interval equal to 10 ms, close to the event inter-arrival time, the percentage was close to 100%. In general, the percentage did not change much as we increased the number of middle-priority streams, because (1) the contention from high-priority streams did not change, (2) relative time consistency depends on the event creation time, which is indifferent to the time of processing, and (3) the CPUs did not saturate.

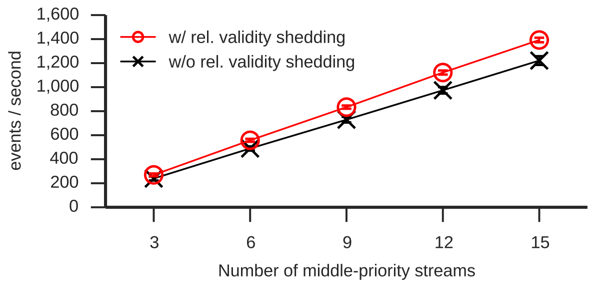


(a) Middle-priority throughput.

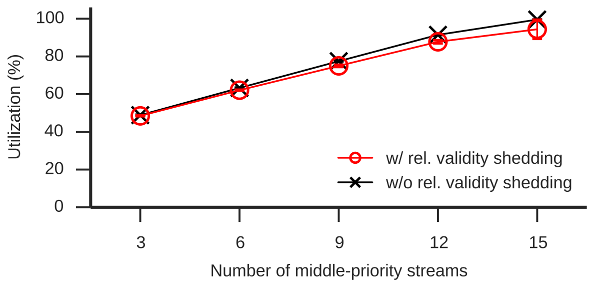


(b) CPU utilization.

Figure 2.20: Experiment Set 5: Relative validity shedding with relative validity interval = 7.0 ms.



(a) Middle-priority throughput.



(b) CPU utilization.

Figure 2.21: Experiment Set 5: Relative validity shedding with relative validity interval = 8.5 ms.

Figure 2.18(d) shows the CPU utilization normalized to the number of cores. With the result shown in Figure 2.18(b), this suggests that with no relative validity shedding, many more CPU cycles were wasted if we had a shorter relative validity interval.

Effects of Relative Time Consistency Enforcement and Shedding

Figure 2.19 to 2.21 show the result of relative validity shedding with each relative validity interval, respectively. In all cases, relative validity shedding produced a higher relatively time-consistent throughput and at the same time reduced CPU utilization. With a shorter relative validity interval, there was more relative gain in throughput by shedding, because in that case there were more violations of relative time consistency and thus shedding saved more CPU cycles for relatively time-consistent processing. Figure 2.20, for example, shows the results for relative validity interval = 7.0 ms, in which case shedding helped improve the throughput by about 200 events/second and saved about 5% in CPU utilization. It might not be possible to achieve the ideal throughput, which is $100 \times k$, where k is the number of middle-priority streams, because event suppliers were not coordinated and since without phase coordination the relevant events may spread out in time, violations of relative time consistency are more likely. With relative validity shedding, the throughput we had is much closer to ideal.

2.6.6 Experiment Set 6: Overhead Measurements

We evaluated CPEP's runtime overhead by showing both (1) queueing overhead, and (2) preemption overhead. For queueing overhead, we measured both enqueue and dequeue times for the InputQ, PendingQ, and MovingQ. We took the average of one million measurements of each operation, and the result is shown in Table 2.2. The recorded dequeue time for the PendingQ included the time spent in binding operator's parameters to the dequeued metadata. The time complexity of the MovingQ is logarithmic in the queue length. Regarding the number of queueing operations per event, each supplier event needs at most one enqueue and one dequeue for the InputQ, each operator along the event processing graph needs exactly one enqueue and one dequeue for the PendingQ, and each transition between operators of different priority needs exactly one enqueue and one dequeue for the MovingQ. Our implementation permits concurrent checks for non-emptiness of a queue via readers-writer lock (see Section 2.5).

For the preemption overhead, we used `trace-cmd` to measure the number of scheduling events at runtime for 100 seconds. We compared CPEP's prioritization (using Linux's `SCHED_RR`

Table 2.2: Experiment Set 6: Latency of event queues.

	InputQ	PendingQ	MovingQ
Enqueue	1.23 ns	34.86 ns	194.21 ns
Dequeue	1.03 ns	337.81 ns	177.33 ns

real-time scheduler with real-time priorities) against a baseline version that disabled CPEP's prioritization (using Linux's default CFS scheduler with no real-time priority). Our result shows a 40% reduction in the number of context switches (the `sched_switch` events), with prioritization enabled than with it disabled, from about 60 switches per millisecond to about 37 switches per millisecond. As the `SCHED_RR` real-time scheduler permits global push and pull operations, it resulted in an increase in the number of thread migrations (the `sched_migration` events) from about 2.5 migrations per millisecond under CFS to about 6.3 migrations per millisecond under `SCHED_RR`. The preemption overhead therefore involves a tradeoff between a reduction in context switches and a smaller increase in the thread migrations. A detailed comparison between `SCHED_RR` and CFS can be found in the literature [14].

2.7 Concluding Remarks

In this Chapter, we introduced the CPEP middleware for real-time cyber-physical event processing. CPEP features configurable operations, prioritization, time consistency enforcement, efficient memory management, and concurrent processing. We implemented CPEP within the TAO event service, and empirically evaluated it in comparison to Apache Flink, showing that CPEP outperforms Flink in terms of latency, throughput, and memory footprint. Our further experiments showed that CPEP can both reduce latency and improve timely-throughput, through prioritization, sharing, absolute time consistency shedding, and relative time consistency shedding.

Chapter 3

Fault-Tolerant Real-Time Messaging

3.1 Introduction

The edge computing paradigm assigns specific roles to local and remote computational resources. Typical examples are seen in Industrial Internet-of-Things (IIoT) systems [26, 41, 43, 45], where latency-sensitive applications run locally in *edge* servers, while computation-intensive and shareable tasks run in a private cloud that supports multiple edges (Figure 3.1). Both an appropriate configuration and an efficient run-time implementation are essential in such environments.

IIoT applications have requirements for message latency and reliable delivery, and the needed levels of assurance are often combined in heterogeneous ways. For example, emergency-response applications may require both zero message loss and tens of milliseconds end-to-end latency, monitoring applications may tolerate a small number of consecutive message losses (e.g., by computing estimates using previous or subsequent messages) and require hundreds of milliseconds bounds on latency, and logging applications may require zero message loss but may only require sub-second latency.

Such systems must be able to differentiate levels of latency and loss-tolerance requirements. With no latency differentiation, latency-sensitive messages may arrive too late; with no loss-tolerance differentiation, the system may demand excessive resources since it must treat every message with the highest requirement level. An edge computing system further needs to account for both the discrepancy between traffic periods within an edge (e.g., tens of milliseconds) and those to a cloud (e.g., at least sub-second), and the discrepancy between network latency within an edge (e.g., sub-millisecond) and that to a cloud (e.g., up to sub-second). Premature scheduling of cloud-bound traffic may delay edge-bound, latency-sensitive traffic.

It is challenging to differentiate such heterogeneous requirements for both latency and loss tolerance efficiently. Differentiating latency requirements alone at millisecond time scales is nontrivial; enabling message loss-tolerance differentiation adds further complexity, since fault-tolerant approaches in general tend to slow down a system. In particular, systems often adopt service replication to tolerate crash failures [9, 13]. Replication requires time-consuming mechanisms to maintain message backups, and significant latency penalties may be incurred due to system rollback upon fault recovery. Alternative replication methods may reduce latency at the expense of greater resource consumption [35, 63]. To date, enabling and efficiently managing such latency/loss-tolerance differentiation remains a realistic and important open challenge.

In this paper, we propose the following problem formulation to address those nuances of fault-tolerant real-time messaging for edge computing: each message topic is associated with a *loss-tolerance level*, in terms of the acceptable number of consecutive message losses, and

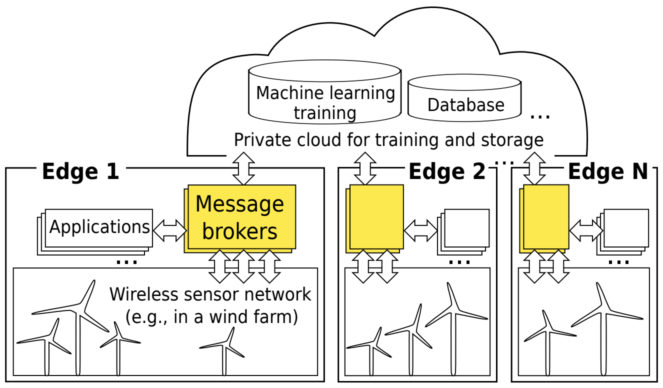

Machine learning training
Database
...
Private cloud for training and storage
Edge 1
Edge 2
Edge N
Applications
Message brokers
Wireless sensor network (e.g., in a wind farm)

Figure 3.1: An Illustration of IIoT Edge Computing.

an *end-to-end latency deadline*, and the system will process messages while (1) meeting designated loss-tolerance levels at all times, (2) mitigating latency penalties at fault recovery, and (3) meeting end-to-end latency deadlines during fault-free operation. In this paper, we focus on the scope of one edge and one cloud.

This paper makes three contributions to the state of the art in fault-tolerant real-time middleware:

- *A new fault-tolerant real-time messaging model.* We describe timing semantics for message delivery, identify under what conditions a message may be lost, prove timing bounds for real-time fault-tolerant actions in terms of traffic/service parameters, and demonstrate how the timing bounds can support efficient and appropriate message differentiation to meet each requirement.

- *FRAME: A differentiated Fault-tolerant ReAl-time MEssaging architecture.* We propose an edge computing architecture that can perform appropriate differentiation according to the model above. The FRAME architecture also mitigates latency penalties caused by fault recovery, via an online algorithm that prunes the set of messages to be recovered.

- *An efficient implementation and empirical evaluation.* We describe our implementation of FRAME within the TAO real-time event service [38], a mature and widely-used middleware. Empirical evaluation shows that FRAME can efficiently meet both types of requirements and mitigate the latency penalties caused by fault recovery.

The rest of this paper is organized as follows: In Section 3.2, we compare and contrast our approach to other related work. In Section 3.3, we describe FRAME's fault-tolerant real-time model, using an illustrative IIoT scenario. The architectural design of FRAME is presented in Section 3.4, and its implementation is described in Section 3.5. In Section 3.6, we present an empirical evaluation of FRAME. Section 3.7 summarizes and presents conclusions.

3.2 Related Work

Modern latency-sensitive applications have promoted the need for edge computing, by which applications can respond to local events in near real-time, while still using a cloud for management and storage [41, 65]. AWS Greengrass is a typical edge computing platform [1], where a *Greengrass Core* locally provides a messaging service that bridges edge devices and the cloud. Our model aligns with such an architecture. While there is recent work [6] on a timely and reliable transport service in the Internet domain using overlay networks, to our knowledge we are the first to characterize and differentiate timeliness and fault-tolerance for messaging in the edge computing domain.

Both real-time systems and fault-tolerant systems have been studied extensively due to their relevance to real-world applications [13, 52]. For distributed real-time systems, the TAO real-time event service [38] supports a configurable framework for event filtering, correlation, and dispatching, along with a scheduling service [31]. In this paper, we consider timing aspects of message-loss tolerance and show that our new model can be applied to address needs for efficient fault-tolerant and real-time messaging.

Among fault-tolerance approaches, service replication has been studied for reliable distributed systems. Delta-4 XPA [9] coined the names *active/passive/semi-active* replication. In active replication, also called the state-machine approach [63], service requests are delivered to all host replicas, and the responses from replicas are compared or suppressed and only one result is returned. In passive replication, also known as the primary-backup approach [15], only one primary host handles requests, the other hosts synchronize to it, and one of the synchronized hosts would replace the primary host should a fault occur. Semi-active approaches have been applied to real-time fault-tolerant systems to improve both delay predictability and latency performance [35]. A discussion regarding conflicts between real-time and fault-tolerance capabilities is available [58]. There are also recent studies for virtual machine fault-tolerance [21, 72] and for the recovery of faulty replicas [55]. In this paper, we follow directions established in the primary-backup approach.

A complementary research topic is fault-tolerant real-time task allocation, where a set of real-time tasks and their backup replicas are to be allocated to multiple processors, in order to tolerate processor failures while meeting each task's soft real-time requirements. The DeCoRAM middleware [8] achieved this by considering both primary and backup replicas' execution times and failover ordering, and thereby reducing the number of processors needed for replication. In contrast, the work proposed in this paper considers end-to-end timeliness of message delivery and tolerance of message loss, and via timing analysis can reduce the need for replication itself.

Modern messaging solutions offer message-loss tolerance in three ways: (1) *publisher retention/resend*: a publisher keeps messages for re-sending; (2) *local disk*: message copies are

Table 3.1: Comparison of Related Middlewares and Standards.

Middleware/Standard	Message-Loss Tolerance Strategies		
	Pub. Resend	Local Disk	Backup Broker
Flink [3]	x	x	
Kafka [4]	x	x	x
Spark Streaming [5]	x	x	
NSQ [59]		x	
DDS (Standard) [60]		x	
MQTT (Standard) [66]	x		
FRAME (This work)	x		x

written to local hard disks; (3) *backup brokers*: like the primary-backup approach, message copies are transferred to other brokers; Table 3.1 lists the usage of these strategies in modern solutions. We note that none of these solutions explicitly addresses the impact of fault tolerance on timeliness. In this paper, we introduce a timing analysis that gives insight into how publisher retention and backup brokers relate to each other, and we demonstrate a trade-off in applying those strategies. We chose not to examine the local disk strategy because it performs relatively slowly.

3.3 Fault-Tolerant Real-Time Messaging (FRAME) Model

In this section, we present the constraints for a messaging system to meet its fault-tolerance and real-time requirements. We first give an overview of a messaging model and its notation, followed by our assumptions and the requirements for fault-tolerant and real-time messaging. We then describe temporal semantics for such messaging and prove sufficient timing bounds to meet the specified requirements. We conclude the section with a discussion of how the timing bounds may be applied to drive system behaviors, using different system configurations as examples.

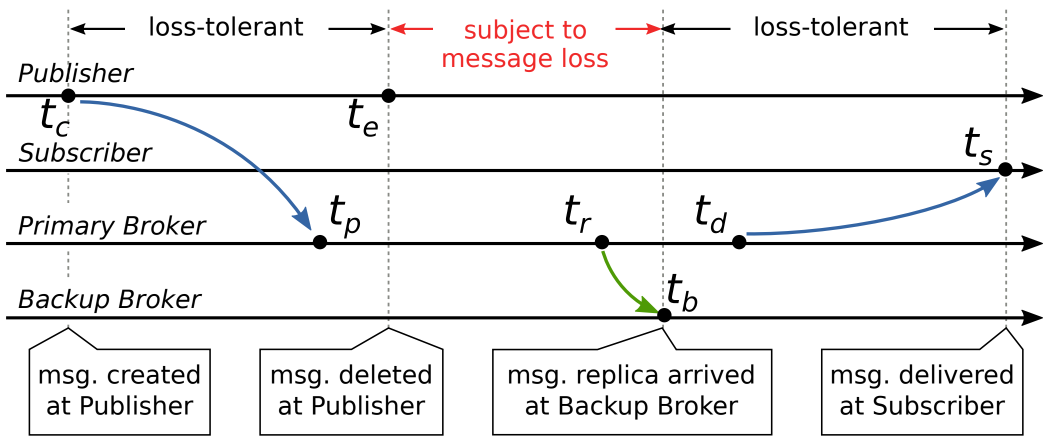


Figure 3.2: Example timelines within the scope of message creation and delivery, and the relation between events happening in each component.

3.3.1 Overview and Notation

We consider a common publish-subscribe messaging model, with *publishers*, *subscribers*, and *brokers*. Each publisher registers for a set of *topics*, and for each topic it publishes a *message* sporadically. A message is delivered via a broker to each subscriber of the topic. We define two types of brokers, according to their roles in fault tolerance. The broker delivering messages to subscribers is called the *Primary*, while another broker that backs up messages is called the *Backup*. The Backup is promoted to become a new Primary should the original Primary crash. The Primary and its respective Backup are assumed to be mapped to separate hosts. Each publisher has connection to both the Primary and the Backup, and it always sends messages to the current Primary. Each subscriber has connection to both, too. We use the term *message* interchangeably with *topic*.

Let I be the set of topics associated with a publisher. For each topic $i \in I$, messages are created sporadically with minimum inter-creation time T_i, also called the *period* of topic i. For each message, within the time span between its creation at a publisher and its final delivery at the appropriate subscriber, there are seven time points of interest (Figure 3.2): t_c

the message creation time at the publisher, t_p the message arrival time at the Primary, t_s the message arrival time at the subscriber, t_e the time at which the publisher deleted the message it had retained, t_r the time at which the Primary sent a replica of the message to the Backup, t_b the time the Backup received the message replica, and t_d the time the Primary dispatched the message to the subscriber. Let $\Delta_{PB} = t_p - t_c$ be the latency from the publisher to its broker, $\Delta_{BS} = t_s - t_d$ the latency from the broker to the subscriber, and $\Delta_{BB} = t_b - t_r$ the latency from the broker to its Backup.

3.3.2 Assumptions and Requirements

This study assumes the following fault model. Each broker host is subject to processor crash failures with fail-stop behavior, and a system is designed to tolerate one broker failure. We choose to focus on tolerating broker crash failures, since a broker must accommodate all message streams and is a performance bottleneck. Common fault-tolerance strategies such as active replication may be used to ensure the availability of both publishers and subscribers. The Primary broker host and the Backup broker host are within close proximity (e.g., connected via a switch). The clocks of all hosts are sufficiently synchronized[5], and between the Primary and the Backup there are reliable inter-connects with bounded latency. Publishers are proxies for a collection of IIoT devices, such as sensors, and aggregate messages from them.

For each topic i, its subscriber has a specific *loss-tolerance requirement* and *latency requirement.* A loss-tolerance requirement is specified as an integer $L_i \geq 0$, saying that the subscriber can tolerate at most L_i consecutive message losses for topic i. We note that such loss tolerance is specified because in common cyber-physical semantics (e.g., monitoring and tracking), a small number of transient losses may be acceptable as they can be compensated for, using

[5]For example, via PTP [40] and/or NTP [57] protocols; see Section 3.6.1 for our experimental setup.

estimates from previous or subsequent messages. A latency requirement is specified as an integer $D_i \geq 0$, defining a soft end-to-end latency constraint [52] of topic i from publisher to subscriber. For multiple subscribers of the same topic, we choose the highest requirements among the subscribers. Finally, we assume that each publisher can retain the $N_i \geq 0$ latest messages that it has sent to the Primary. During fault recovery, a publisher will send all N_i retained messages to its Backup. Let x be a publisher's fail-over time, which is defined as an interval beginning at a broker failure until the publisher has redirected its messaging traffic to the Backup.

3.3.3 Temporal Semantics and Timing Bounds

As illustrated in Figure 3.2, within the interval from t_c to t_s, a message may be loss-tolerant because either (1) it has a copy retained in the publisher (over time interval $[t_c, t_e]$) or (2) a replica of the message has been sent to the Backup (over time interval $[t_b, t_s]$). Nevertheless, there could be a time gap in between those intervals during which the message can be lost, because the publisher has deleted its copy and a replica has not yet been sent to the Backup (time interval (t_e, t_b)). Let $R_i^r = t_r - t_p$ be the response time for a job that replicates message i to the Backup, and $R_i^d = t_d - t_p$ the response time for a job that dispatches message i to the subscriber. Depending on the specifications of L_i and N_i, there are constraints on the response time of message dispatching and message replication. In the following, we prove an upper bound on the worst-case response time for replicating and dispatching, respectively.

Lemma 1. *Let D_i^r be the relative deadline for a replicating job for topic i. To ensure that the subscriber will never experience more than L_i consecutive losses of messages in the topic, it is sufficient that*

$$R_i^r \leq D_i^r = (N_i + L_i)T_i - \Delta_{PB} - \Delta_{BB} - x. \tag{3.1}$$

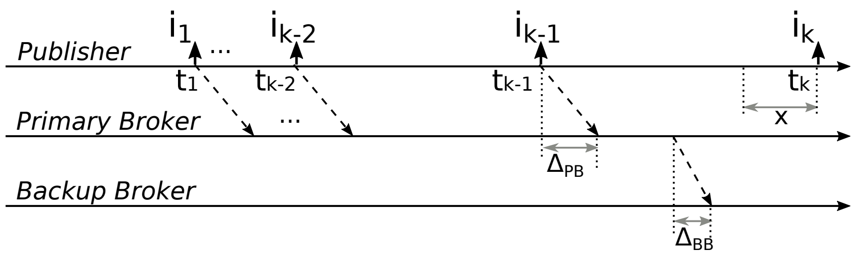


Figure 3.3: Example timelines for the proof of Lemma 1.

Proof. Without loss of generality, we consider a series of message creation times for topic i, as shown in Figure 3.3. Adding Δ_{PB} to each creation time, we have the release time of the replicating job for each message. Suppose that the Primary crashed at a certain time within $(t_{k-1}, t_k]$. We have two cases:

Case 1: Crash at a time within $(t_{k-1}, t_k - x)$*.* In this case, message i_k will be sent to the Backup instead, since the publisher has detected the crash of Primary. By definition, the publisher would send the latest N_i messages to the Backup once it detected failure of the Primary. Therefore, messages i_{k-1}, i_{k-2}, ..., through i_{k-N_i} would be recovered and are not considered lost. According to the requirement, topic i can have no more than L_i consecutive losses. Hence, message $i_{k-N_i-L_i-1}$ had to be replicated to the Backup before the Primary crashed, which means the response time of replicating the message must be smaller than $((k-1)-(k-N_i-L_i-1))T_i - \Delta_{PB} - \Delta_{BB} = (N_i+L_i)T_i - \Delta_{PB} - \Delta_{BB}$, supposing that, in the worst case, the crash happened immediately after the release of a replicating job for message i_{k-1}.

Case 2: Crash at a time within $[t_k - x, t_k]$*.* In this case, message i_k will be lost and then recovered after the publisher has detected the crash of the Primary. By definition, besides i_k, $N_i - 1$ earlier messages will also be recovered. The earliest message recovered by the publisher would be $i_{k-(N_i-1)}$. Similar to Case 1, message $i_{k-(N_i-1)-L_i-1}$ had to be replicated to the Backup before the Primary crashed, meaning that the response time of replicating the message must be smaller than $(T_i - x) + ((k-1) - (k - (N_i - 1) - L_i - 1))T_i - \Delta_{PB} - \Delta_{BB} = (N_i + L_i)T_i - \Delta_{PB} - \Delta_{BB} - x$.

Case 2 dominates, and hence the proof. □

Lemma 2. *Let D_i^d be the relative deadline for a dispatching job for topic i. For the topic to meet its end-to-end deadline D_i, it is sufficient that*

$$R_i^d \leq D_i^d = D_i - \Delta_{PB} - \Delta_{BS}. \tag{3.2}$$

Proof. We prove by contradiction. Let r be the current amount of time remaining before missing the end-to-end deadline, and $r = D_i$ at message creation. When message i arrives at the broker (time point t_p), we have $r = D_i - \Delta_{PB}$. Now, suppose that it would take longer than $D_i - \Delta_{PB} - \Delta_{BS}$ before the dispatch of message i (time point t_d). We will then have $r < (D_i - \Delta_{PB}) - (D_i - \Delta_{PB} - \Delta_{BS})$, i.e., $r < \Delta_{BS}$. By definition, the latency $[t_d, t_s]$ is at least Δ_{BS}, and therefore by the time the message reached the subscriber (time point t_s), we will have $r < 0$, i.e., a deadline miss. Thus, $D_i^d = D_i - \Delta_{PB} - \Delta_{BS}$ is an upper bound on the worst-case response time for dispatching message i. □

3.3.4 Enabling Differentiated Processing and Configuration

In the following, we give five applications of the timing bounds in Lemmas 1 and 2. We define deadlines for message dispatching and replication using Equations (3.1) and (3.2), and we schedule both activities using the Earliest Deadline First (EDF) policy [52]. Further, we propose a heuristic based on the fact that a dispatched message no longer needs to be replicated, and we show where the heuristic is useful.

Table 3.2: Example Topic Specifications.

Topic Category	T_i	D_i	L_i	N_i	Destination
0	50	50	0	2	Edge
1	50	50	3	0	Edge
2	100	100	0	1	Edge
3	100	100	3	0	Edge
4	100	100	∞	0	Edge
5	500	500	0	1	Cloud

Proposition 1. *(*Selective Replication*) It is sufficient to suppress the replication of topic i if a system can meet deadline* D_i^d *and*

$$D_i^d \ \leq \ D_i^r. \tag{3.3}$$

Following Proposition 1 we have a condition to judge whether there is a need for replication: $x + \Delta_{BB} - \Delta_{BS} > (N_i + L_i)T_i - D_i$.

As an illustration, we consider an IIoT scenario [41], where publishers are proxies for edge sensors, subscribers are either within an *edge* (e.g., in close proximity to publishers and brokers) or in a *cloud* (e.g., in AWS Elastic Compute Cloud (EC2)), and brokers are in closer proximity to publishers than to subscribers. We consider six categories of topic specification, as shown in Table 3.2. Categories 0 and 1 represent highly latency-sensitive topics (e.g., for emergency-response applications), with zero- and three-message-loss tolerance, respectively. Categories 2, 3, and 4 represent moderately latency-sensitive topics (e.g., for monitoring applications), with different levels of loss tolerance. $L_i = \infty$ means that all subscribers of the topic only ask for best-effort delivery. Category 5 represents weakly latency-sensitive topics (e.g., for logging applications), with zero-message-loss tolerance. The fifth column shows the minimum value of N_i that ensures D_i^r is non-negative.

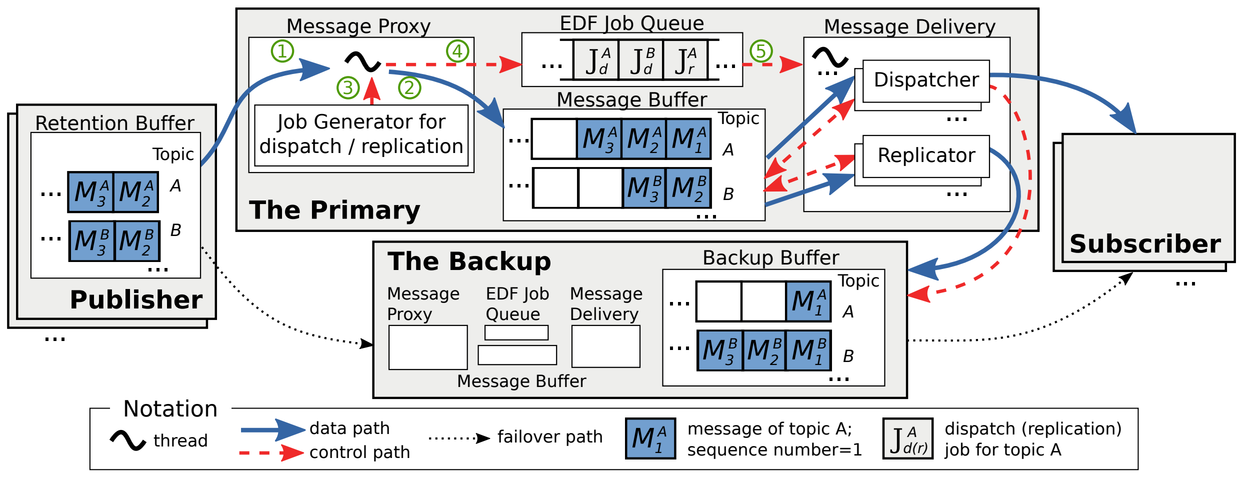


Figure 3.4: The FRAME Architecture.

1) Admission test: Lemmas 1 and 2 provide a simple admission test: both $D_i^r \geq 0$ and $D_i^d \geq 0$ must hold for any topic i. For example, if we are to meet a fault-tolerance requirement $L_i = 0$ (i.e., zero message loss), Equation (3.1) shows that we must enable publisher message retention. Otherwise, the message will be lost should the Primary crash immediately after a message arrival. In general, to satisfy $D_i^r \geq 0$, it follows that (1) if message period T_i is small, it then requires a larger value of $N_i + L_i$; and (2) a higher loss-tolerance requirement (i.e., a smaller L_i) requires a larger value of N_i.

2) Differentiating topics with heterogeneous latency (D_i) and loss-tolerance (L_i) requirements: Applying Equations (3.1) and (3.2), we have the following order over D_i^r and D_i^d, assuming $\Delta_{BS} = 1$ for subscribers within an edge and $\Delta_{BS} = 20$ for subscribers in a cloud, $\Delta_{BB} = 0.05$, and $x = 50$: $\{D_0^d = D_1^d < D_0^r = D_2^r < D_2^d = D_3^d = D_4^d < D_1^r < D_3^r < D_5^r < D_5^d\}$, indexed by topic category. There is no need for topic replication in category 4 since subscribers only ask for best-effort delivery. Applying Proposition 1, we can remove the need for replication in categories 0, 1, and 3, and only need replication for categories 2 and 5. This lowers system load and can help a system accommodate more topics. We give empirical validation of this in Section 3.6.

3) Leveraging publisher message retention: While assuming the minimum admissible value of N_i for each category allows one to study the most challenging case for a messaging system to process such a topic set, the value of N_i in practice may be tunable, for example, if a publisher is a proxy host for a collection of devices. Also, a fault-tolerant system is typically engineered with redundancies. Now, we increase the value of N_i by one for categories 2 and 5. We will have both $D_2^d < D_2^r$ and $D_5^d < D_5^r$, giving dispatching activities a higher precedence. Applying Proposition 1, we may further remove the need for replication in those categories as well. In Section 3.6 we will show the empirical benefit of such an increase in publisher message retention.

4) Differentiating topics with latency requirements non-equal to their periods: There can be messages that either have $D_i < T_i$ or $D_i > T_i$. Case $D_i < T_i$ applies to rare but time-critical messages, such as for emergency notification. In this case, without loss of generality we assume $T_i = \infty$ and $L_i = 0$. The admissible value of N_i is greater-than-zero, and Equation (3.3) suggests no need for replication as long as message delivery can be made in time. Case $D_i > T_i$ applies to messages with traveling time longer than their rate, such as in multimedia streaming. In this case, Equation (3.3) suggests a likely need for replication, unless Δ_{BS} is small.

5) Differentiating edge-bound and cloud-bound traffic: Traffic parameters within an edge and to a cloud are usually of different orders of magnitude. While edge-bound traffic periods may be tens of milliseconds, cloud-bound traffic periods may be a sub-second or longer. For network latency, we observed 0.5 ms round-trip time between a local broker and a subscriber connected via a switch, and 44 ms round-trip time between the broker and a subscriber in AWS EC2 cloud. Lemmas 1 and 2 capture the relation between these parameters. Cloud latency is less predictable, and we choose to use a lower-bound of Δ_{BS}, which can be obtained by measurement. Proposition 1 ensures the same level of loss tolerance even if at run-time there is an occasional increase in cloud latency. A loss-tolerance guarantee would break if a system chose to suppress a replication when it should not, but that will not happen as we use a lower-bound of Δ_{BS}. Although an under-estimated cloud latency at run-time might delay the cloud traffic (due to the use of EDF policy), in edge computing clouds are typically used for training and storage and do not have hard latency constraints. An over-estimation of cloud latency could be undesirable, however, as it could both preclude the use of selective replication and prematurely delay other traffic.

3.4 The FRAME Architecture

We now describe the FRAME architecture for differentiated fault-tolerant real-time messaging. The key criteria are (1) to meet both the fault-tolerant and real-time requirements for each topic efficiently, and (2) to mitigate both latency penalties during fault recovery and replication overhead during fault-free operation. The FRAME architecture, shown in Figure 3.4, achieves both via (1) a configurable scheduling/recovery facility that differentiates message handling according to each fault-tolerance and real-time requirement, and (2) a dispatch-replicate coordination approach that tracks and prunes a valid set of message copies and cancels unneeded operations.

3.4.1 Configurable Scheduling/Recovery Facility

During initialization, FRAME takes an input configuration and, accordingly, computes pseudo relative deadlines for replication, $D_i^{r\prime}$, and for dispatch, $D_i^{d\prime}$, with $D_i^{r\prime} = (N_i + L_i)T_i - \Delta_{BB} - x$ and $D_i^{d\prime} = D_i - \Delta_{BS}$. The content of the configuration includes values for N_i, L_i, T_i, and D_i, per topic i, and values for x and Δ_{BS} per subscriber. The computed pseudo relative deadlines $D_i^{r\prime}$ and $D_i^{d\prime}$ are stored in a module called the *Message Proxy* (see Figure 3.4). At run-time, for each message arrival, the Message Proxy first takes the arriving message and copies it into a *Message Buffer*, and then invokes its *Job Generator* along with a reference to the message's position in the Message Buffer. The Job Generator then creates job(s) for message dispatching (replicating). The Job Generator subtracts Δ_{PB} from $D_i^{r\prime}$ and $D_i^{d\prime}$, obtaining the relative deadlines D_i^r and D_i^d as defined in Lemmas 1 and 2, and then sets an absolute deadline for each dispatching (replicating) job to $t_p + D_i^d$ $(t_p + D_i^r)$. A replicating job will not be created if $D_i^d \leq D_i^r$, according to Proposition 1.

Scheduling of message delivery is performed using the EDF policy. This is achieved by pushing jobs into a queue called the *EDF Job Queue*, within which jobs are sorted according to their deadlines. A *Message Delivery* module fetches a job from the EDF Job Queue and delivers the message that the job refers to, accordingly. A job for dispatching (replicating) is executed by a *Dispatcher* (*Replicator*) in the module. A Dispatcher pushes the message to a subscriber, and a Replicator pushes a copy of the message to the Backup, where the message copy will be stored in a *Backup Buffer*. For a topic subscribed by multiple Subscribers, the Job Generator would create only one dispatching (replicating) job for each message arrival. A Dispatcher taking the job would push the message to each of its subscribers.

Fault recovery is achieved as follows. The Backup tracks the status of its Primary via periodic polling, and would become a new Primary once it detected that its Primary had crashed. Upon becoming the new Primary, the broker would first dispatch a selected set of message copies in its Backup Buffer. The dispatch procedure is the same as handling a new message arrival, except that jobs now refer to the broker's Backup Buffer, not its Message Buffer, and Δ_{PB} is increased according to the arrival time of the message copy. Only those message copies whose original copy have not been dispatched will be selected for dispatch.

Table 3.3: Algorithm for Dispatch-Replicate Coordination.

Type of Operation	Procedure
Dispatch	1. dispatch the message to the subscriber
	2. set *Dispatched* to `True`
	3. if *Replicated* is `True`, request the Backup to set *Discard* to `True`
Replicate	1. if *Dispatched* is `True`, abort
	2. replicate the message to the Backup
	3. set *Replicated* to `True`
Recovery (in the Backup)	1. if *Discard* is `True`, skip the message
	2. create a dispatching job for the message
	3. push the job into the EDF Job Queue

3.4.2 Dispatch-Replicate Coordination

During fault recovery, it would add both overhead to a system and latency penalties to messages if we did not differentiate message copies in the Backup Buffer. In FRAME, differentiation is achieved by maintaining a dynamic set of message copies in the Backup Buffer, and by skipping other copies during fault recovery. To be specific, during fault-free operation, once the Primary has dispatched a message, it will (1) direct its Backup to prune the Backup Buffer for the topic, and (2) cancel the pending job for the corresponding replication, if any. The coordination algorithm is given in Table 3.3. Flags (*Dispatched, Replicated, Discard*) are associated with each entry in the Message Buffer/Backup Buffer that keeps a message copy; for each new message copy, all flags are initialized to `False`. If a topic has multiple subscribers, the Primary would set the Dispatched flag to true only after the message has been dispatched to all the subscribers.

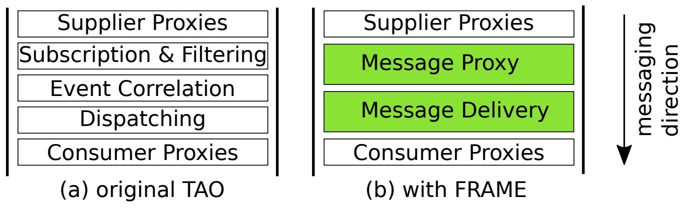


Figure 3.5: Implementation of FRAME within TAO's Real-Time Event Service.

3.5 FRAME Implementation

We implemented the FRAME architecture within the TAO real-time event service [38], where messages were encapsulated in events, publishers and subscribers were implemented as event suppliers and consumers, and each broker was implemented within an event channel. Prior to the work described in this dissertation, the TAO real-time event service only supported simple event correlations (logical conjunction and disjunction). In contrast, FRAME enables differentiated processing according to the specified latency and loss-tolerance requirements. An event channel in the original TAO middleware contains five modules, as shown in Figure 3.5(a). Figure 3.5(b) illustrates our implementation: we preserved the original interfaces of the Supplier Proxies and the Consumer Proxies, and replaced the Subscription & Filtering, Event Correlation, and Dispatch modules with FRAME's Message Proxy and Message Delivery modules.

We connected the Supplier Proxies to the Message Proxy module by a hook method within the push method of the Supplier Proxies module. The Message Delivery module delivers messages by invoking the push method of the Consumer Proxies module. We implemented Dispatchers and Replicators using a pool of generic threads, with the total number of threads equal to three times the number of CPU cores. We implemented FRAME's EDF Job Queue using C++11's standard priority-queue, and used C++11's standard `chrono` time library

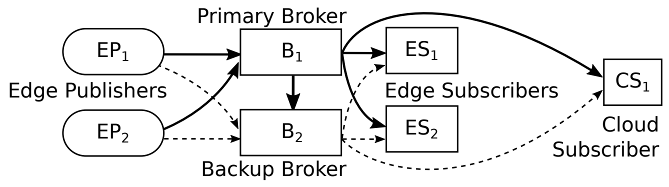


Figure 3.6: Topology for empirical evaluation. Dotted lines denote failover paths.

to timestamp and compare deadlines to determine message priority. The Message Buffer, Backup Buffer, and Retention Buffer are all implemented as ring buffers.

3.6 Experimental Results

We evaluate FRAME's performance across three aspects: (1) message loss-tolerance enforcement, (2) latency penalties caused by fault recovery, and (3) end-to-end latency performance. We adopted the specification shown in Table 3.2, with ten topics each in categories 0 and 1, and five topics in category 5. The timing values are in milliseconds. We evaluate different levels of workload by increasing the number of topics in categories 2–4. We chose to increase the workloads this way, as in IIoT scenarios sensors often contribute to the majority of the traffic load, and some losses are tolerable since lost data may be estimated from previous or subsequent updates. The payload size is 16 bytes per message of a topic. Publishers for categories 0 and 1 were proxies of ten topics, publishers for categories 2–4 were proxies of 50 topics, and each publisher for category 5 published one topic. Each proxy sent messages in a batch, one message per topic. The set of workloads we have evaluated includes a total of 1525, 4525, 7525, 10525, and 13525 topics.

3.6.1 Experiment Setup

Our test-bed consists of seven hosts, as shown in Figure 3.6: One publisher host has an Intel Pentium Dual-Core 3.2 GHz processor, running Ubuntu Linux with kernel v.3.19.0, and another has an Intel Core i7-8700 4.6 GHz processor, running Ubuntu Linux with kernel v.4.13.0; both broker hosts have Intel i5-4590 3.3 GHz processors, running Ubuntu Linux kernel v.4.15.0; one edge subscriber host has an Intel Pentium Dual-Core 3.2 GHz processor, running Ubuntu Linux with kernel v.3.13.0, and another has an Intel Core i7-8700 4.6 GHz processor, running Ubuntu Linux with kernel v.4.13.0; the cloud subscriber is a virtual machine instance in AWS EC2, running Ubuntu Linux with kernel v.4.4.0. We connected all local hosts via a Gigabit switch in a closed LAN. Both broker hosts had two network interface controllers, and we used one for local traffic and another for cloud traffic. In each broker host, two CPU cores were dedicated for Message Delivery, and one CPU core was dedicated for the Message Proxy.

We assigned real-time priority level 99 to all middleware threads, and we disabled `irqbalance` [51]. We synchronized our local hosts via PTPd [33], an open source implementation of the PTP protocol [40]. The publisher hosts' clock, the edge subscriber hosts' clock, and the Backup host's clock were synchronized to the clock of the Primary host, with synchronization error within 0.05 milliseconds. The cloud subscriber's clock was synchronized to the Primary's clock using `chrony` [69] that utilizes NTP [57], with synchronization error in milliseconds. The latency measurement for Δ_{BS} is dominated by the communication latency to AWS EC2, which was at least 20 milliseconds.

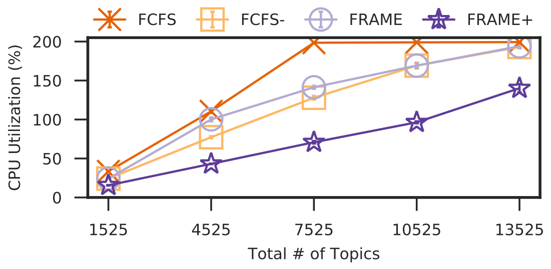
We compared four broker configurations: (1) FRAME; (2) *FRAME+*, where we set $N_i = 2$ for categories 2 and 5, to evaluate publisher message retention; (3) *FCFS (First-Come-First-Serve)*, a baseline against FRAME, where no differentiation is made and messages are handled in the order of their arrivals; (4) *FCFS-*, which is FCFS without dispatch-replicate coordination. In both FCFS and FCFS-, for each message arrival the Primary first performed replication and then dispatch.

Table 3.4: Success Rate for Loss-Tolerance Requirement (%).

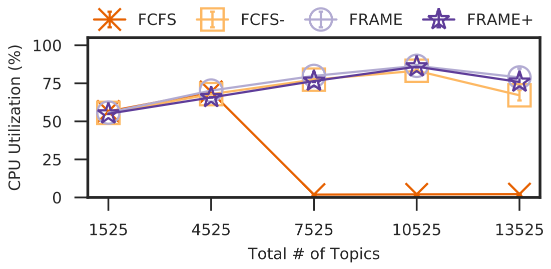
D_i	L_i	FRAME+	FRAME	FCFS	FCFS-
		Workload = 7525 Topics			
50	0	100.0	100.0	0.0	100.0
50	3	100.0	100.0	0.0	100.0
100	0	100.0	100.0	0.0	100.0
100	3	100.0	100.0	0.0	100.0
100	∞	100.0	100.0	100.0	100.0
500	0	100.0	100.0	0.0	100.0
		Workload = 10525 Topics			
50	0	100.0	100.0	0.0	100.0
50	3	100.0	100.0	0.0	100.0
100	0	100.0	100.0	0.0	100.0
100	3	100.0	100.0	0.0	100.0
100	∞	100.0	100.0	100.0	100.0
500	0	100.0	100.0	0.0	100.0
		Workload = 13525 Topics			
50	0	100.0	80.0 ± 30.1	0.0	100.0
50	3	100.0	80.0 ± 30.1	0.0	100.0
100	0	100.0	73.2 ± 30.7	0.0	78.4 ± 13.3
100	3	100.0	79.3 ± 29.9	0.0	99.3 ± 0.5
100	∞	100.0	100.0	100.0	100.0
500	0	100.0	80.0 ± 30.1	0.0	100.0

Note: 100% success rate for all with 1525 and 4525 topics.

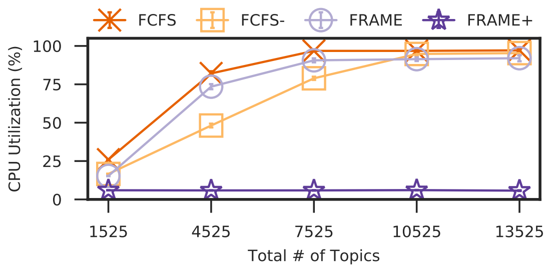
For each configuration we ran each test case ten times and calculated the 95% confidence interval for each measurement. We allowed 35 seconds for system initialization and warm-up. The measuring phase spanned 60 seconds. We injected a crash failure by sending signal `SIGKILL` to the Primary broker at the 30th second, and studied the performance of failover to the Backup. We also ran each test case without fault injection, to obtain both end-to-end latency performance at fault-free operation, and CPU usage in terms of utilization percentage, for each module of the FRAME architecture.



(a) Message Delivery Module in the Primary.



(b) Message Proxy Module in the Primary.



(c) Message Proxy Module in the Backup.

Figure 3.7: CPU Utilization for Each Configuration.

3.6.2 Message Loss-Tolerance Enforcement

Table 3.4 shows the success rate of meeting loss-tolerance requirements under increasing workload[6]. All four configurations had 100% success rate for 1525 and 4525 topics. FRAME outperformed FCFS after the workload reached 7525 topics and more, thanks to the selective replication of Proposition 1. FRAME only performed the needed replications (topic categories 2 and 5) and suppressed the others (topic categories 0, 1, and 3), saving more than

[6]The success rate for category 4 is always 100% because the category has no loss-tolerance requirement ($L_i = \infty$)

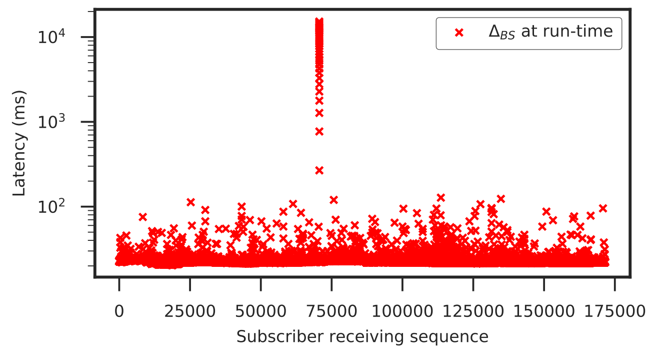


Figure 3.8: Value of Δ_{BS} for a topic in category 5 through a 24-hour duration.

50% in CPU utilization for the Message Delivery module, compared with the result of FCFS for the case with 7525 topics (Figure 3.7(a)). With FCFS, the Primary was overloaded: the threads of the Message Delivery module competed for the EDF Job Queue, and the thread of the Message Proxy module was kept blocked (implied in Figure 3.7(b)) each time it created jobs from arrivals of message batches.

To evaluate publisher message retention, we compared FRAME with FRAME+. Leveraging Proposition 1, with FRAME+ the Primary did not need to perform any replication to its Backup, and loss tolerance was solely performed by publisher re-sending the retained messages. As shown in Table 3.4, FRAME+ met all loss-tolerance requirements in every case. Further, the replication removal saved CPU usage in the Primary broker host (Figure 3.7(a)). The replication removal also saved CPU usage in the Backup broker host (Figure 3.7(c)), because the Backup did not need to handle additional traffic from the Primary.

To evaluate the impact of dispatch-replicate coordination, we compared FCFS with FCFS-. FCFS- outperformed FCFS in loss-tolerance performance (Table 3.4), because with FCFS- the Primary may replicate and deliver messages sooner since it did not coordinate with the Backup. But that way the Primary would miss opportunities to preclude latency penalties caused by fault recovery, however, and we evaluate the impact in the next subsection.

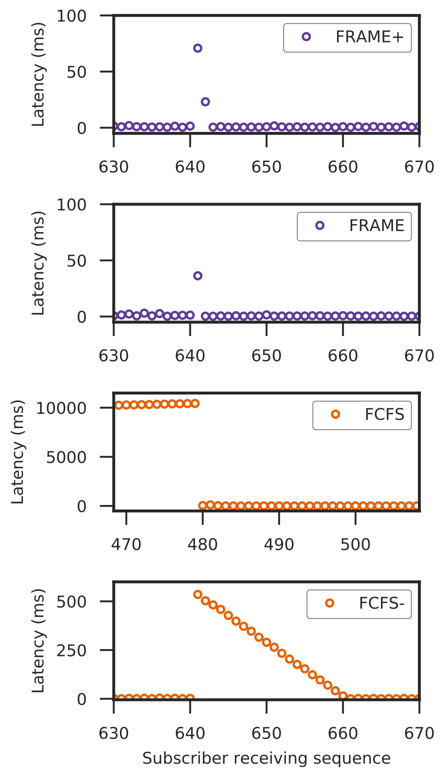


Figure 3.9: End-to-end latency before, upon, and after fault recovery (category 0, $T_i = 50$, $D_i = 50$).

We further conducted a micro-benchmark to show that FRAME can keep the same level of loss tolerance despite cloud latency variation. We ran the workload of 7525 topics non-stop for 24 hours, using the FRAME configuration, and we measured the run-time value of Δ_{BS} for a topic in category 5 (Figure 3.8)[7]. The setup value of Δ_{BS} for D_5^d was 20.7 ms, which was the minimum value from an one-hour test run. As a result, we observed no message loss throughout the 24 hours, despite changes in the value of Δ_{BS}.

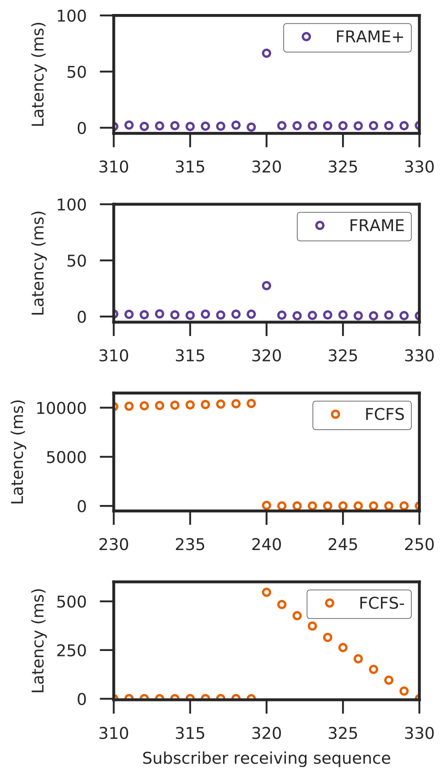


Figure 3.10: End-to-end latency before, upon, and after fault recovery (category 2, $T_i = 100$, $D_i = 100$).

3.6.3 Latency Penalties Caused by Fault Recovery

We evaluate the latency penalties in terms of the peak message latency following a crash failure. We set the size of the Backup Buffer to ten for each topic. Under the workload of 1525 topics, all four configurations performed well, and at higher workloads both FRAME and FRAME+ outperformed FCFS and FCFS-. In the following, we evaluate a series of end-to-end latency results under the workload of 7525 topics. We only show results of distinct messages, differentiated by their sequence numbers. Duplicated messages were discarded. The results are shown in Figures 3.9–3.11 with each column presenting four configurations for a topic category.

[7]The $+10^4$ ms latency spike occurred at around 8am on Thursday.

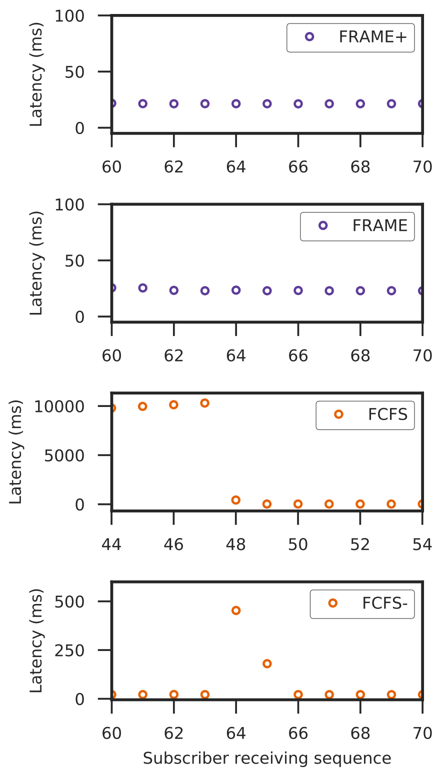


Figure 3.11: End-to-end latency before, upon, and after fault recovery (category 5, $T_i = 500$, $D_i = 500$).

In general, without dispatch-replicate coordination (demonstrated by FCFS-), the number of messages affected by fault recovery is lower-bounded by the size of the Backup Buffer, since at run-time steady state the Backup Buffer is full, and during fault recovery new message arrivals may need to wait. With the proposed dispatch-replicate coordination (demonstrated by FRAME+, FRAME, and FCFS), the amount of work is decoupled from the buffer size and is instead equal to the number of messages whose original copy has not yet been dispatched.

Both FRAME and FRAME+ met the loss-tolerance requirements (zero message loss); for FRAME, although the Primary did replication, the Backup Buffer was empty at the time of fault recovery (all pruned), suggesting the effectiveness of dispatch-replicate coordination; for FRAME+, the Primary did no replication according to Proposition 1. FRAME+ successfully recovered one message for each of categories 0 and 2 by publishers re-sending their retained message copies. The latency of FRAME+ during fault recovery was higher than that of FRAME, because with FRAME+ the Backup would process one additional message copy per topic in categories 2 and 5, and that caused delay.

For FCFS, the system was overloaded, messages were delayed (latency $>$ 10 seconds) and many of them were lost: 206 losses for a topic in category 0, 103 losses for a topic in category 2, and 20 losses for a topic in category 5. We observed that dispatch-replication coordination was in effect, as the Backup Buffer for those topics was empty at the time of fault recovery. After switching to the Backup, message latency sharply dropped. For example, for topic category 2 (Figure 3.10), the Backup began processing at the 240th message. Since the Backup Buffer was empty, there was no latency penalty for new arrivals.

For FCFS-, we observed that the Backup Buffer was full at the time of fault recovery, because there was no dispatch-replicate coordination. Therefore, there were large latency penalties since the Backup needed to process all message copies in the Backup Buffer. For example, shown in Figure 3.10, FCFS- had a peak latency above 500 ms, which was about 400 ms longer than the deadline. In contrast, FRAME had a peak latency below 50 ms. The latency prior to the Primary crash was low, because FCFS-, like FRAME and unlike FCFS, did not overload the system (Figure 3.7(a)). Finally, we note that while FCFS- processed messages in the Backup Buffer and caused great latency penalties, those messages were all out-dated and unnecessary, and all the needed messages were actually recovered by publishers re-sending their retained copies; for the topic in category 5, there was no message loss using FCFS- and the publisher re-sending was unnecessary, and the two latency spikes were due to overhead in processing unneeded copies (Figure 3.11).

Table 3.5: Success Rate for Latency Requirement (%).

D_i	L_i	FRAME+	FRAME	FCFS	FCFS-
		Workload = 4525 Topics			
50	0	100.0	99.9 ± 2.5E-2	99.9 ± 5.0E-2	100.0
50	3	100.0	99.9 ± 3.0E-2	99.9 ± 4.1E-2	100.0
100	0	100.0	100.0	100.0	100.0
100	3	100.0	100.0	99.9 ± 1.1E-3	100.0
100	∞	100.0	100.0	99.9 ± 1.9E-3	100.0
500	0	100.0	100.0	100.0	100.0
		Workload = 7525 Topics			
50	0	100.0	99.9 ± 4.4E-2	0.2 ± 0.1	99.9 ± 4.2E-2
50	3	100.0	99.9 ± 3.9E-2	0.2 ± 0.1	99.9 ± 6.3E-2
100	0	100.0	99.9 ± 8.8E-3	0.0	99.9 ± 1.4E-2
100	3	100.0	99.9 ± 5.6E-3	0.0	99.9 ± 1.3E-2
100	∞	100.0	99.9 ± 9.2E-3	0.0	99.9 ± 1.5E-2
500	0	100.0	100.0	0.0	100.0
		Workload = 10525 Topics			
50	0	100.0	99.9 ± 5.7E-2	0.2 ± 5.3E-2	99.8 ± 8.1E-2
50	3	100.0	99.9 ± 5.6E-2	0.2 ± 5.5E-2	99.8 ± 6.8E-2
100	0	99.9 ± 5.4E-2	99.9 ± 4.0E-2	7.2E-2 ± 0.1	99.9 ± 3.1E-2
100	3	99.9 ± 5.2E-2	99.9 ± 3.9E-2	7.2E-2 ± 0.1	99.9 ± 2.9E-2
100	∞	99.9 ± 5.0E-2	99.9 ± 4.3E-2	6.9E-2 ± 0.1	99.9 ± 3.1E-2
500	0	100.0	100.0	0.0	100.0
		Workload = 13525 Topics			
50	0	98.4 ± 2.9	85.4 ± 21.7	0.1 ± 0.1	99.4 ± 3.6E-1
50	3	98.4 ± 2.9	85.3 ± 21.7	0.2 ± 0.2	99.5 ± 2.3E-1
100	0	97.6 ± 4.4	83.7 ± 21.9	2.6E-4 ± 6.0E-4	98.3 ± 1.0
100	3	97.6 ± 4.4	83.8 ± 21.9	9.9E-4 ± 2.2E-3	98.3 ± 1.1
100	∞	97.6 ± 4.4	83.8 ± 21.9	6.6E-4 ± 1.5E-3	98.3 ± 1.1
500	0	98.6 ± 2.8	86.1 ± 21.8	0.0	100.0

Note: 100% success rate for all with 1525 topics.

3.6.4 Latency Performance During Fault-Free Operation

In addition to fault tolerance, it is critical that a system performs well during fault-free operation. Good fault-free performance implies an efficient fault-tolerance approach. Table 3.5 shows the success rate for meeting latency requirement D_i. All configurations performed well, except for FCFS at higher workloads, in which cases the system was overloaded as discussed in Section VI-B. This suggests that both the architecture and implementation are efficient, as even the FCFS configuration performed well as long as the system was not yet overloaded.

3.6.5 Key Lessons Learned

Here we summarize four key observations:

1. Applying replication removal as suggested by Proposition 1 can help a system accommodate more topics while reducing CPU utilization (FRAME v.s. FCFS).
2. Pruning backup messages can reduce latency penalties caused by fault recovery at a cost of nontrivial overhead during fault-free operation (FCFS v.s. FCFS-).
3. Following the first two lessons, combining replication removal and pruning can achieve better performance both at fault recovery and during fault-free operation (FRAME v.s. FCFS-).
4. Allowing a small increase in the level of publisher message retention can enable large replication removal and greatly improve efficiency (FRAME v.s. FRAME+).

3.7 Concluding Remarks

In this Chapter, we introduced a new fault-tolerant real-time edge computing model and illustrated that the proved timing bounds can aid in requirement differentiation. We then introduced the FRAME architecture and its implementation. Empirical results suggest that FRAME is performant both in fault-tolerant and fault-free operation. Finally, we demonstrated in an IIoT scenario that FRAME can keep the same level of message-loss tolerance despite varied cloud latency, and we show that a small increase in publisher message retention can both improve loss-tolerance performance and reduce CPU usage.

Chapter 4

Adaptive Real-Time Reliable Edge Computing

4.1 Introduction

Data processing at the edge of clouds is essential to Industrial Internet-of-Things (IIoT) systems [41]. An IIoT edge processing service receives data from local networked embedded devices, performs in-band processing, and delivers distilled information to IIoT applications. The service runs in resource-constrained edge platform and must be resilient to system failures. With the increasing deployment of IIoT devices for a wide range of IIoT applications, such data processing services must be able to meet a variety of applications' different fault-tolerance and real-time requirements with efficiency. For purposes of monitoring, inference, and prediction, IIoT applications typically can tolerate a few consecutive losses of data, as an application may compute estimates using previous or subsequent data. IIoT applications may require soft deadlines end-to-end, between data creation and final delivery of the processed result. For example, monitoring applications may require tens of milliseconds end-to-end latency, and late delivery is less useful to the application.

It is challenging for an IIoT edge processing service to remain efficient while meeting applications' latency and loss-tolerance requirements, especially as common fault-tolerance approaches often introduce significant overhead. IIoT devices often have limited storage capacity, which may limit re-transmissions end-to-end. State machine approaches [63] provide seamless, end-to-end data loss-tolerance but are resource-expensive as they run full copies of system replicas, while primary-backup approaches [15] are resource-efficient at a cost of reduced timeliness. During fault-free operation, recording and sending service states from a primary to a backup may delay normal data processing; during fault-recovery operation, replay from a previous state may delay processing of new data. It is thus nontrivial to achieve timeliness for both operations.

In this Chapter, we study adaptive real-time reliability techniques for IIoT processing services. We consider both resource constraints in edge computing systems and application requirements in data loss-tolerance and latency, and present three major contributions in this area of research:

1. *A timing analysis framework for data replication.* We extend the primary-backup service model by taking into account device-specific capacity for re-transmissions and application-specific requirements for loss-tolerance and latency. We analyze relative deadline for data replication to satisfy each level of loss-tolerance requirement, and show how the relative deadline inversely correlates to the frequency of replication.

2. *An architecture for $\underline{a}$daptive $\underline{r}$eal-time $\underline{r}$eliable $\underline{e}$dge $\underline{c}$omputing (ARREC).* We introduce a new software system architecture that can efficiently meet application-specific requirements for data-loss tolerance and latency. Based on our observations that (1) in-band

processing often has short execution times per data element, and (2) applications can tolerate a few consecutive losses of data, we introduce heuristics that perform selective lazy data replication to reduce latency impact on data processing, while keeping needed levels of data loss-tolerance assurance.

3. *An efficient implementation and empirical evaluation.* We describe our implementation of ARREC within the mature and widely-used TAO real-time event service [38] middleware, and present an empirical validation of ARREC's performance using typical IIoT workloads and data traffic patterns. Our empirical results show that ARREC can efficiently meet both latency and loss-tolerance requirements, during both fault-free operation and fault-recovery.

The rest of this Chapter is organized as follows: In Section 4.2, we survey related work. In Section 4.3, we present our system model and problem definition. In Section 4.4, we provide a new timing analysis for replication. In Section 4.5, we introduce the ARREC software architecture, and describe its implementation in Section 4.6. In Section 4.7, we present our empirical evaluation of ARREC. We summarize this work and present conclusions in Section 4.8.

4.2 Related Work

IIoT systems largely follow the edge computing paradigm, where applications respond to local events in near real-time [41, 65]. AWS Greengrass is a typical edge computing platform [1], where a *Greengrass Core* locally provides a processing service, and in-band processing is triggered by data from IIoT devices. Our approach aligns well with such an architecture. There is recent work [6] on a timely and reliable transport service, and other work [12] on fault-tolerant task allocation to meet different recovery time requirements. To our knowledge, we are the first to study real-time reliable in-band processing in the IIoT domain.

Stream processing is related to in-band processing, but usually with longer execution times per stream and therefore the service can accommodate less data traffic per service host. Modern stream processing services leverage checkpointing techniques for fault-tolerance and can be classified into two types, where in the micro-batch model [75], an event stream is segmented into micro-batches of duration in seconds. Within each micro-batch, a processing graph is partitioned into stages, and checkpoints are taken at barriers inserted between stages and at the end of the micro-batch. In the continuous operator model [17], each operator in a processing graph is a long-running task and intermediate results are directly transferred to the next operators. Checkpoints in this model are taken periodically and in a distributed manner, and together they form a consistent global snapshot [18], providing a rollback point for fault recovery.

Both stream processing models exhibit a performance trade-off between when a system is running fault-free and when it is recovering from failure. The micro-batch model induces less latency penalty for fault recovery, at the cost of more complex and time-consuming coordination during fault-free operation. The continuous operator model, in contrast, incurs less latency overhead when fault-free, but may take longer to recover from failure. The Drizzle project [70] offers an empirical comparison of two representative implementations: Apache Spark [75] (using the micro-batch model) and Apache Flink [17] (using the continuous operator model). Drizzle introduced a choice of grouping micro-batches to bound the coordination overhead under fault-free operation, defined as the ratio of time spent on scheduling to the overall execution time of the service. In contrast, in this Chapter we focus on a more performant IIoT processing service that works at time scales of tens of milliseconds, with large amounts of input data per service host and short execution times per data element, and is constrained by application-specific timing and loss-tolerance requirements. The approach proposed in this Chapter works efficiently in both fault-free and fault recovery operations.

In IIoT processing services, appropriate scheduling of both data processing and data replication activities is critical and challenging: a system should complete data replication in time to ensure needed levels of data loss-tolerance, while also making progresses in data processing to meet soft latency requirements. In essence, for both types of activity, a system must ensure timely completion of one type while allowing enough progress of the other. Research on real-time scheduling, such as *Zero-Slack* [23, 27, 39] and *Virtual-Deadline* [10, 11] strategies in mixed criticality systems [16] and the *earliest deadline zero laxity* scheduling algorithm (EDZL) [7, 49, 50], offer ways to schedule activities of different levels of assurance and ensure timely completion of the ones with the highest level. In the Zero-Slack strategy, for example, a critical time instant is pre-computed per task, and the system ensures that after that time instant, execution of high-criticality tasks will not be further delayed by low-criticality tasks. In contrast, in this Chapter, we observe that in IIoT processing services, data replication activities (which are more critical than data processing) can be safely postponed or even skipped. We prove deadlines for data replication, identify conditions of skipping replication, introduce a scheduling heuristic for lazy and selective data replication, and describe a new system architecture that leverages the heuristic. Our empirical validation shows that the resulting system can meet data loss-tolerance requirements while reducing latency impact on data processing.

4.3 System Model and Definition

In this section, we first present our service model for IIoT processing and our fault model. We then introduce IIoT real-time fault-tolerance specifications and our problem statement.

4.3.1 Service Model and Fault Assumption

We focus on a local IIoT service that performs in-band data processing, and we call such a service an *edge processing core*, or simply an edge core. An edge core receives data from publishers, invokes processing operations on the data, and then delivers processed data to subscribers. We define two types of edge cores, the *Primary* that processes and replicates data, and the *Backup* that receives the replicated data. We assume there is one Backup per Primary edge core, each running on a different host. Each data publisher publishes data of a set of *topics*, with an inter-publishing time no smaller than T_i for data of topic i.

We assume that the Primary performs in-band processing for each data topic. This is modeled as a set of processing tasks, denoted by τ^p. For data of topic i, task $\tau_i^p \in \tau^p$ is the in-band processor for it, and the execution time is e_i. An arrival of data of topic i triggers a job release of task τ_i^p. At the completion of the job the Primary will deliver the processed data to its subscriber.

We assume that Primaries are subject to processor crash failures with fail-stop behavior. Upon a crash failure, the Backup will be promoted to become a new Primary. The new Primary will resume the service by re-processing/re-delivering the replicated data, and all publishers will then send data to the new Primary. We also assume that our target system uses reliable inter-connects between hosts running publishers, subscribers, the Primary, and the Backup: these inter-connects do not fail or partition. We define a set of replication tasks, denoted by τ^r, and a set of fault-recovery tasks, denoted by τ^f. For topic i, replication task $\tau_i^r \in \tau^r$ represents the work of replicating data belonging to topic i to the Backup, and fault-recovery task $\tau_i^f \in \tau^f$ represents the work by the Backup to process the replicated data belonging to topic i at the time of recovery, which is essentially a re-processing/re-delivery of data that was being handled by the Primary.

4.3.2 Requirements and Problem Statement

We assume that each publisher can keep the N_i latest data elements that it has sent to the Primary, and will send them to the new Primary as part of fault recovery. Each subscriber has a loss-tolerance requirement per data element of interest, specified as $L_i \geq 0$ for data type i, saying that the subscriber cannot tolerate more than L_i consecutive data losses. Further, each subscriber has a latency requirement D_i^p for each data topic it subscribes to. The requirement specifies a soft end-to-end deadline, between the time a publisher sent the data and the time the processed data arrived at its subscriber.

Conceptually, the problem is to schedule all tasks in $\tau \equiv \{\tau^p \cup \tau^r \cup \tau^f\}$ such that all constraints are met. A plausible approach, which we refer to as *eager data replication*, is to schedule jobs of τ_i^r for execution at their earliest possible times, so that each data element arriving at the Primary may have a copy in the Backup as soon as possible. Eager data replication, however, may incur significant delay to the data processing for τ^p, because frequent traffic from the Primary to the Backup could consume a significant portion of the CPU resources. In this Chapter, we explore *lazy data replication*, to study how both postponing executions of τ_i^r, and grouping pending replications to some extent, may help improve overall performance. In the following sections, we first discuss timing constraints for data replication, and then describe our IIoT middleware solution that leverages such constraints as an effective guide to schedule data replications.

4.4 Analysis for Data Replication

In this section, we introduce an analysis framework and thereby answer two questions: (1) Under what conditions is replication needed? (2) Before what time must a replication job complete?

We address each of these questions in the following subsections, respectively. For now, we assume that a system can meet these conditions if need be. In the next section, we introduce our design of the ARREC architecture that is capable of doing so.

4.4.1 Need for Data Replication

We say that a piece of data is *uncovered* if it has arrived at the Primary but has neither been replicated to the Backup, nor had a copy kept at the publisher. Should the Primary crash, all uncovered data would be lost. Intuitively, keeping a large number of data copies at a publisher for re-transmission could reduce the need for data replication to the Backup, but such an approach is neither ideal nor practical. It is not ideal because it will cause a burst in both re-transmission and re-processing. It is not practical because data publishers in IIoT systems are typically embedded devices that have limited data storage for such purposes.

Data replication is needed but may be performed sparingly. For example, let data topic i be the one of interest, and suppose $L_i = 3$ and $N_i = 0$. The Primary may need to replicate data from topic i only if the number of consecutive uncovered data elements of topic i has reached some threshold, e.g., three, though in which case the Primary must complete replication before the next data arrival from topic i; otherwise, a crash failure before the completion of replication would violate the loss-tolerance requirement.

In general, the need for replication depends on the number of consecutive uncovered data elements. A simple condition for the data of topic i is to replicate once every L_i data arrivals, and the system can thus ensure needed data-loss tolerance as long as it has succeeded in replicating data to the Backup. In the rest of this Chapter, we suppose that the Primary replicates data from topic i once every M_i arrivals, and we analyze the relation between M_i and the deadline to complete such a replication action.

4.4.2 Deadline for Data Replication

First, we prove a constraint between applications' requirements and platform parameters:

Lemma 3. *For data topic i, to prevent more than L_i consecutive data losses, L_i and N_i cannot be both zero.*

Proof. We prove by contradiction. Assuming that both $L_i = 0$ and $N_i = 0$. If a crash happened immediately after a data arrival, it would be impossible to ensure no data loss: the data did not have a copy kept at the publisher for re-transmission, and the Primary was unable to replicate data in time. The system would have at least one data loss. $\square$

We define a *relative replication deadline* for data topic i, or simply a replication deadline, denoted by D_i^r, to be the maximum allowable response time for the Primary to complete replicating the data. In the following, we derive bounds on the replication deadline in terms of applications' requirements and platform parameters. Let δ_{PP} be the latency from a publisher to the Primary, δ_{PrB} be the latency from the Primary to the Backup, and T_{FO} be a publisher's fail-over time, which is the interval between when the Primary crashed and when the publisher is able to send its data to the new Primary.

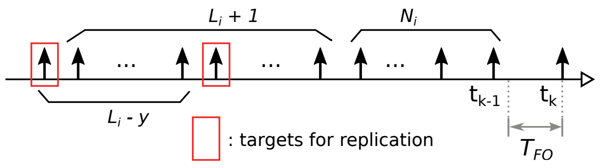


Figure 4.1: An illustration for the proof of Lemma 4.

Lemma 4. *For data topic i, set $M_i = L_i - y \geq 1$. To prevent more than L_i consecutive data losses, the replication deadline must satisfy the following bound:*

$$D_i^r \leq (N_i + y + 1)T_i - T_{FO} - \delta_{PP} - \delta_{PrB}. \tag{4.1}$$

Proof. We consider a sequence of data arrivals of type i, as shown in Figure 4.1. Subtracting δ_{PP} from each data arrival time, we have the data sending time at the publisher. Suppose that the Primary crashed at a time within $(t_{k-1}, t_k]$. There are two cases to prove:

Suppose that a crash happened within $(t_{k-1}, t_k - T_{FO})$. Without loss of generality, we suppose that the crash happened immediately after the data arrival at time t_{k-1}, and thus data arriving at time t_{k-1} will be lost. Later data will not be lost, because the publisher will be able to detect the Primary failure before time t_k and will send them to the Backup instead. By definition, all the latest N_i data will be recovered via publisher re-transmission.

There will be more than L_i consecutive data losses if there were at least $L_i + 1$ consecutive uncovered data elements when a system crashes. To avoid this, and since the Primary triggers replication only once every $L_i - y$ arrivals, in the worst case the last attempt of replication that must succeed would be the one made for the data that has arrived at time $t_{(k-1)-(N_i-1)-(y+2)}$ (e.g., the rightmost box in Figure 4.1), and the replication must complete no later than time t_{k-1}. Therefore, the replication deadline must be smaller than or equal to $((k-1) - ((k-1) - (N_i - 1) - (y+2))T_i - \delta_{PP} - \delta_{PrB} = (N_i + y + 1)T_i - \delta_{PP} - \delta_{PrB}$.

Now suppose that a crash happened at a time instant within $[t_k - T_{FO}, t_k]$. In this case, the publisher cannot detect the crash in time and would still send data that should have arrived at the Primary at time t_k, and that data will be lost. The publisher would send subsequent data to the Backup and so they will not be lost. The worst case is that the crash happens immediately after time $t_{k-1}+T_i-T_{FO}$, and therefore the replication deadline must be smaller than or equal to $(T_i-T_{FO})+((k-(k-(N_i-2)-(y+2))-1)T_i-\delta_{PP}-\delta_{PrB} = (N_i+y+1)T_i-T_{FO}-\delta_{PP}-\delta_{PrB}$.

□

Lemma 4 shows that a more frequent replication interval (a smaller M_i) can permit a longer replication deadline. For example, if we configure the Primary to replicate once every L_i data arrivals, a replication deadline would be $(N_i+1)T_i-T_{FO}-\delta_{PP}-\delta_{PrB}$; if instead we configure the Primary to replicate at every data arrival, then setting $y = L_i - 1$ the replication deadline would be $(N_i + L_i)T_i - T_{FO} - \delta_{PP} - \delta_{PrB}$, which is $(L_i - 1)T_i$ longer. For the case $L_i = 0$, according to the definition of M_i, a system must replicate at every data arrival, and thus setting $y = -1$ the replication deadline would be $N_iT_i - T_{FO} - \delta_{PP} - \delta_{PrB}$. In Section 4.7, we describe our empirical evaluation of the those alternatives.

The following Lemma suggests that the Primary may feasibly perform even less-frequent replication for data topic i:

Lemma 5. *For $N_i > 0$, one may set $M_i = \max\{L_i, 1\} + 1$, in which case the replication deadline must satisfy the following bound:*

$$D_i^r \leq N_iT_i - T_{FO} - \delta_{PP} - \delta_{PrB}. \tag{4.2}$$

Proof. Similar to the proof for Lemma 4, the worst case is that the Primary crashed within $[t_k - T_{FO}, t_k]$. Setting $M_i = \max\{L_i, 1\} + 1$ means that now we make a replication once every

$L_i + 1$ data arrivals (or once every two arrivals, if $L_i = 0$), and therefore the last replication job must complete before a crash. With publisher re-transmission, in the worst case the last replication job was for the data arrival at time t_{k-N_i}, and the deadline must be upper-bounded by $((k-1) - (k - N_i))T_i + (T_i - T_{FO}) - \delta_{PP} - \delta_{PrB} = N_iT_i - T_{FO} - \delta_{PP} - \delta_{PrB}$. □

Finally, the following Lemma shows that Lemma 5 gives the maximum allowable value of M_i:

Lemma 6. *To prevent more than L_i consecutive data losses, M_i cannot be larger than $L_i + 1$ for $L_i > 0$.*

Proof. We prove this by showing that replicating every $L_i + 2$ arrivals or longer does not work. Replicating at such an interval implies that there are at least $L_i + 1$ data arrivals between two replications. The loss-tolerance requirement will break if the earliest data that would be recovered by publisher re-transmission happens to be the one for which we made a replication, because in this case at least $L_i + 1$ data arrivals before that replication will be lost. □

4.5 The ARREC Architecture

We now provide a high-level overview of the ARREC architecture for adaptive real-time reliable edge computing, and then describe how ARREC's components coordinate to enforce loss-tolerance and latency requirements.

ARREC is designed to achieve efficient data replication while meeting applications' requirements. This is carried out via simple heuristics that selectively pick and group data for replication and replicate groups of data in a lazy yet timely manner. Postponing replication actions can effectively remove needs for actual execution, because typical IIoT in-band

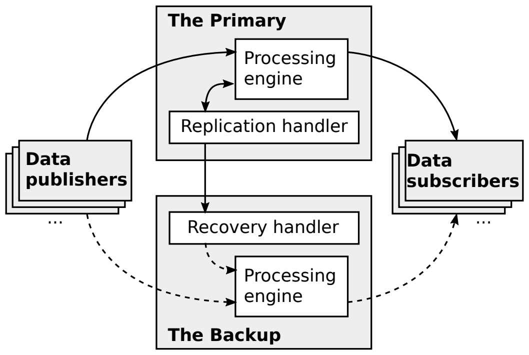


Figure 4.2: ARREC System Architecture.

processing has short execution times (e.g., performing inference or prediction) and processed and delivered data can be exempted from replication. In ARREC, we leverage the timing constraints on replication (see Section 4.4.2) to postpone replication actions to the extent possible, and thus improve overall efficiency of resource utilization, reduce latency impact on in-band processing, and at the same time keep needed levels of data loss-tolerance assurance for each application.

The ARREC architecture is illustrated in Figure. 4.2. The system is pre-configured with the specifications from publishers and subscribers. Upon each data arrival the *processing engine* component creates a processing job. Before processing the data, the processing engine selectively creates a replication job, driven by the value M_i (see Section 4.4.1). The *replication handler* component decides when to perform data replication. All replicated data elements are kept in a buffer in the Backup, and upon fault recovery the *recovery handler* component then feeds those data elements to the processing engine. The processing engine schedules all jobs of τ_i^p and τ_i^f according to absolute deadline, defined as the arrival time of data of topic i plus D_i^p minus the elapsed time since the data sending time at the publisher. In this Chapter we chose to use the earliest-deadline-first (EDF) scheduling policy as an example.

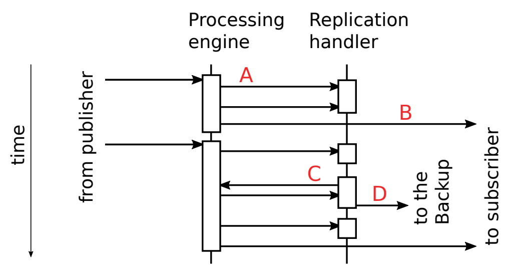


Figure 4.3: Interactions between ARREC components.

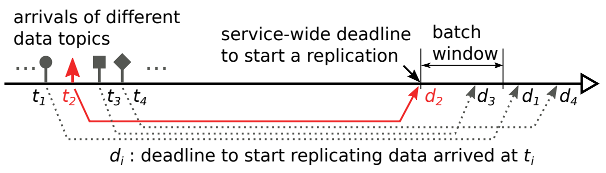

arrivals of different data topics
service-wide deadline to start a replication
batch window
t_1
t_2
t_3
t_4
d_2
d_3
d_1
d_4
d_i : deadline to start replicating data arrived at t_i

Figure 4.4: Illustration of lazy data replication.

4.5.1 Selective Lazy Data Replication

Selecting data for replication is carried out via cooperation between the processing engine and the replication handler, as illustrated in Figure 4.3. Upon each arrival of data topic i, the processing engine compares the value of M_i with the number of data arrivals since the latest replication, and marks data for future replication if the number becomes larger than or equal to M_i (Step A), in which case the replication handler in turn will update its timer for lazy replication based on the marked data element's corresponding replication deadline (Figure 4.4): the timer expiration time is set to the earliest among the deadlines to start replicating data, where a deadline to start replication is defined as the replication deadline minus execution time of the replication action. In the meantime, the processing engine can perform needed in-band data processing (Step B). When the timer expires, the replication handler will select all the marked data elements for which the replication start time falls within a batch window (Step C) and will replicate them in a batch (Step D).

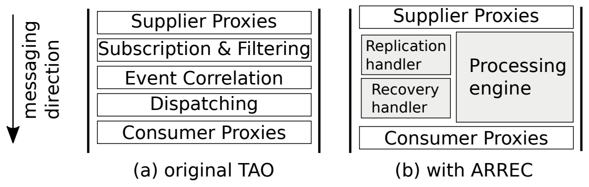


Figure 4.5: Implementation of ARREC within TAO's Real-Time Event Service.

4.5.2 Fault Detection and Recovery

The Backup periodically polls the Primary, and will become the new Primary should the Primary crash. The new Primary will schedule both the data from the previous Primary and the newly arriving data according to their processing deadlines. In IIoT in-band processing the failover overhead includes both the amount of processing work that needs to be re-performed and the subsequent latency impact to newly arriving data elements. It is thus critical to identify and skip redundant re-processing. Through the lazy replication heuristic, ARREC would create fewer data copies in the first place. In addition, ARREC can optionally perform a quick recovery by skipping data that has been delivered in the previous Primary. To enable quick recovery, the Primary attaches the sequence numbers of delivered data to the replication batch, and the Backup can use such information to identify and skip unneeded data copies.

4.6 Implementation

We implemented ARREC within the TAO real-time event service [38], where data elements are carried as events' payloads and publishers and subscribers are implemented as event suppliers and consumers. The processing engine, the replication handler, and the recovery handler are also implemented within an event channel. An event channel in the original TAO middleware contains five modules, as shown in Figure 4.5(a). Figure 4.5(b) illustrates our implementation: we preserved the original interfaces to event suppliers (i.e., the Supplier Proxies module) and to event consumers (i.e., the Consumer Proxies module), and replaced the Subscription & Filtering, Event Correlation, and Dispatch modules with ARREC's components.

We connected the Supplier Proxies module to the Message Proxy module by a hook method within the push method of the Supplier Proxies module. The Message Delivery module delivers messages by invoking the push method of the Consumer Proxies module. We implemented the processing engine using one thread serving as an input proxy on a dedicated CPU core, and a pool of generic threads serving as processing workers on a set of dedicated CPU cores, with the total number of threads equal to ten times the number of CPU cores for processing. We implemented the replication handler as a highest-priority thread, to prevent it from being delayed by data processing, and allocated it to the CPU cores for processing, and we used C++11's standard `chrono` time library to timestamp data.

4.7 Empirical Evaluation

In this section, we present our empirical evaluation of ARREC's performance from two perspectives: (1) loss-tolerance enforcement: can a system meet each application's required level of data loss-tolerance and is the approach efficient? (2) latency performance: what is replication's impact on the processing latency during fault-free and fault-recovery operations, respectively?

4.7.1 Experiment Design and Setup

We evaluate two configurations of ARREC against two baseline configurations. The `ARREC_all` configuration performs lazy data replication, and the `ARREC_Li` configuration in addition performs replication selectively, once per every L_i arrivals, for data topic i. The first baseline is `Retransmission-only`, in which the Primary performs no data replication at all and

Table 4.1: Topic Specification for Empirical Evaluation.

Category	Usage Example	L_i	N_i	D_i^p (ms)	T_i (ms)
1	logging	0	1	∞	50
2		0	1	100	100
3		0	1	500	500
4	monitoring	3	0	50	50
5		3	0	100	100

solely relies on re-transmissions from data publishers to the Backup for data-loss tolerance. Comparison against this baseline shows the overhead of data replication. The second baseline is `Periodic`, in which the Primary performs periodic, service-wide replication by replicating all data that has arrived since the most recent replication.

We present empirical results for a selection of topic specifications, based on the following three observations about IIoT systems:

1. *Data publishers have limited data storage for re-transmission.* Often, data publishers are embedded devices. Some publishers may have more capacity, such as wireless base stations that aggregate data, but in those cases the capacity is amortized to the number of data topics they aggregate.

2. *Most of the data topics may have moderate or no loss-tolerance requirements.* The majority of data generated in IIoT systems are from embedded sensors for monitoring purposes. Some intermittent losses of data may be compensated, for example, by estimation from previous or subsequent data.

3. *Some data topics may require zero loss but have no latency requirement.* An example is topics used for logging purposes.

Accordingly, for our empirical evaluation, we used the topic specifications shown in Figure 4.1. For each topic category, we chose the minimum feasible value of N_i (number of data kept at

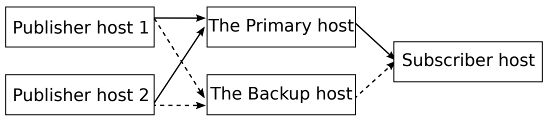
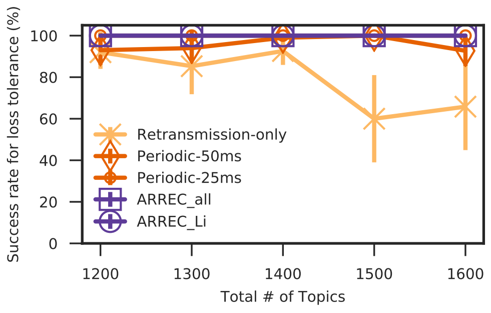


Figure 4.6: Experimental topology.

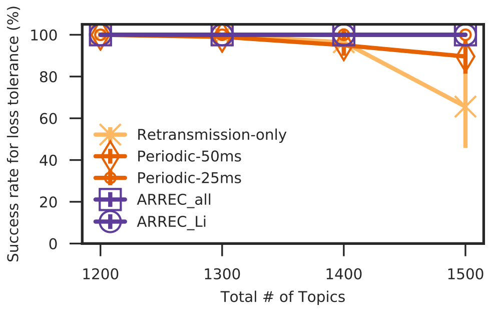
a publisher for re-transmission) for each value of L_i (tolerable number of consecutive data losses): for topics with $L_i = 0$, N_i must be greater than zero to tolerate a Primary host crash immediately after a data arrival. We loaded our system by feeding 50 topics for categories 1 and 4 each, and 100 topics for categories 2 and 3, and we gradually increase the number of topics in category 5, from 900 to 1300, to evaluate the performance of our system under a range of workloads. The total number of topics processed by the system is thus from 1200 to 1600. For each topic we controlled the execution time of its processing load to be 0.1 ms.

We evaluated two IIoT traffic patterns: (1) large group transmissions, by which we evaluated the impact of bursty data arrivals from a large publisher, and (2) small group transmissions, by which we evaluated the impact of many connections from multiple smaller publishers. For large group transmissions, we used one publisher to generate all data in topic category 5; for small group transmissions, we used multiple publishers to generate data in topic category 5, with ten topics per publisher. For the rest of the topic categories, we created publishers with ten topics per publisher.

Our test-bed consists of five hosts, as shown in Figure 4.6: One publisher host has an Intel Pentium Dual-Core 3.2 GHz processor, running Ubuntu Linux with kernel v.3.19.0, and another has an Intel Core i7-8700 4.6 GHz processor, running Ubuntu Linux with kernel v.4.13.0; both Broker hosts have Intel i5-4590 3.3 GHz processors, running Ubuntu Linux kernel v.4.15.0; one subscriber host has an Intel Pentium Dual-Core 3.2 GHz processor, running Ubuntu Linux with kernel v.3.13.0. We connected all hosts via a Gigabit switch in a closed LAN. In both the Primary host and the Backup host, two CPU cores were dedicated



(a) Large group.



(b) Small group.

Figure 4.7: Success rate for loss-tolerance requirement (%).

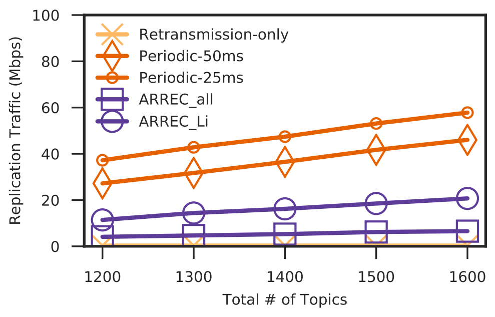
for both processing threads in a processing engine and the replication thread, and one CPU core was dedicated for the input proxy thread. We assigned both the replication thread and the input proxy thread the highest priority level 99 and worker threads with the next highest priority level 98, all with real-time scheduling policy `SCHED_FIFO`. We synchronized our local hosts via PTPd [33], an open source implementation of the PTP protocol [40]. The clocks of the publisher hosts, the subscriber host, and the Backup host were synchronized to the clock of the Primary host, with synchronization error within 0.05 milliseconds. We injected a crash failure by sending signal `SIGKILL` to the Primary broker at the 40th second, and studied the performance of failover to the Backup. We used the `iftop` tool to measure the average rate of network bandwidth consumption of the latest 40 seconds.

4.7.2 Message Loss-Tolerance Enforcement

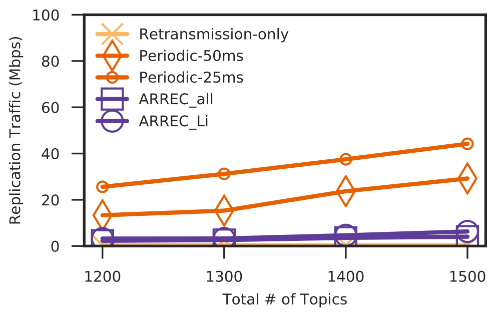
Figure 4.7 shows the success rate for meeting the loss-tolerance requirements, for topic category 1, for configurations `ARREC_all`, `ARREC_Li`, `Periodic-50ms` (with replication period set to 50 ms, the shortest period of the topic specification in Table 4.1), `Periodic-25ms` (replication period = 25 ms), and `Retransmission-only`. For each configuration, we ran each workload twenty times and calculated the average percentage of meeting the loss-tolerance requirement for each category, along with the 95% confidence interval.

Configurations `ARREC_all`, `ARREC_Li`, and `Periodic-25ms` met the requirements under each degree of workload, while both configurations `Periodic-50ms` and `Retransmission-only` occasionally failed to meet the loss-tolerance requirement. Topic category 1 is a challenging case, because the processing for data with no latency requirement may be delayed by some other more urgent data processing, as a result of the use of an EDF scheduling policy. For a certain topic in category 1, there could be multiple data waiting to be processed, and they would be lost upon a system crash. We observed a 100% success rate for all the other categories, as also due to the use of an EDF scheduling policy, all deadlines may be met as long as the system has not yet been saturated. For the case of small group transmissions, the system was saturated with 1600 topics.

The periodic replication strategy may fail to replicate those data in time, because it is not aware of the timing constraint on replication for each data. Doing replication at a shorter periodic may work, but its success relies on trial-and-error tuning to find a suitable period for each specific set of topics. With a shorter period also comes a higher overhead in resource consumption.



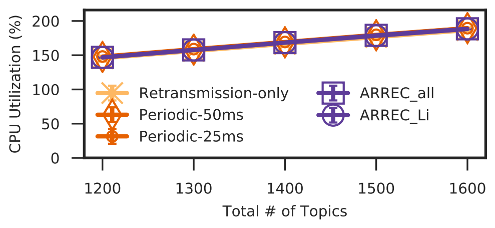
(a) Large group.



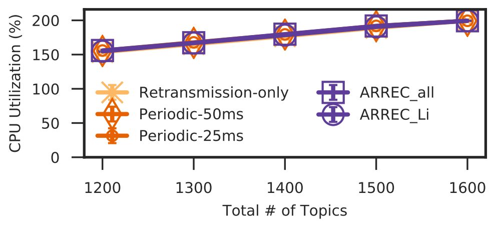
(b) Small group.

Figure 4.8: Network bandwidth consumption for replication traffic from the Primary to the Backup.

We evaluated resource efficiency of ARREC, showing that while meeting loss-tolerance requirements, ARREC consumed less network bandwidth, compared with the periodic replication baselines. The results are shown in Figure 4.8. With a payload size of 512 bytes per data element, configuration `ARREC_all` may save 33–49 Mbps in replication traffic, i.e., about an 88% reduction, compared with configuration `Periodic-25ms`. This result demonstrates the benefit of lazy replication: it gives more time for a system to process and deliver data before performing replication, and the system can skip spurious data. We observed that configuration `ARREC_all` saved more bandwidth than configuration `ARREC_Li`, although the latter only selects data for replication once every L_i arrivals. The reason is that the longer replication deadline permitted by configuration `ARREC_all` (see Lemma 4) would allow more pending replications to be skipped. Configuration `ARREC_Li` outperformed



(a) Large group.

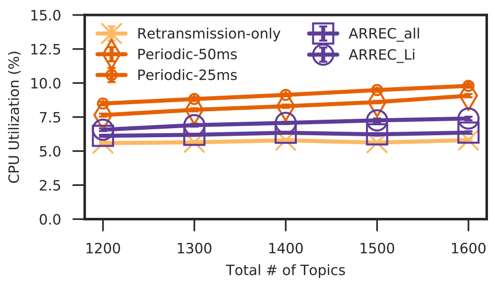


(b) Small group.

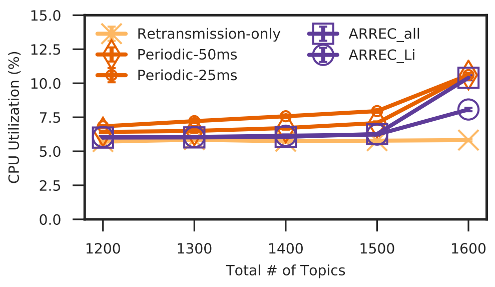
Figure 4.9: CPU% accounted for both the processing threads and the replication thread in the Primary.

the periodic replication baselines, because the use of a batch window (40 ms in this case) allows data with a longer replication deadline to be exempted from the current round of replication. Finally, our results also show that configuration `Periodic-25ms` took more network bandwidth than configuration `Periodic-50ms`, because with a shorter period the system had less chance to skip replication.

Figure 4.9 shows the system utilization of each configuration under increasing workload. Configuration `Retransmission-only` gives the baseline CPU%, i.e., with the processing threads only, as the replication thread is not active in this configuration. Comparing that against all the other configurations, we observed that the replication thread took at most 5% CPU utilization, and the addition did not grow in proportion to the increase in workload. We also measured the overhead of maintaining a group of pending replications, which accounted for less than 2.5% CPU utilization.



(a) Large group.



(b) Small group.

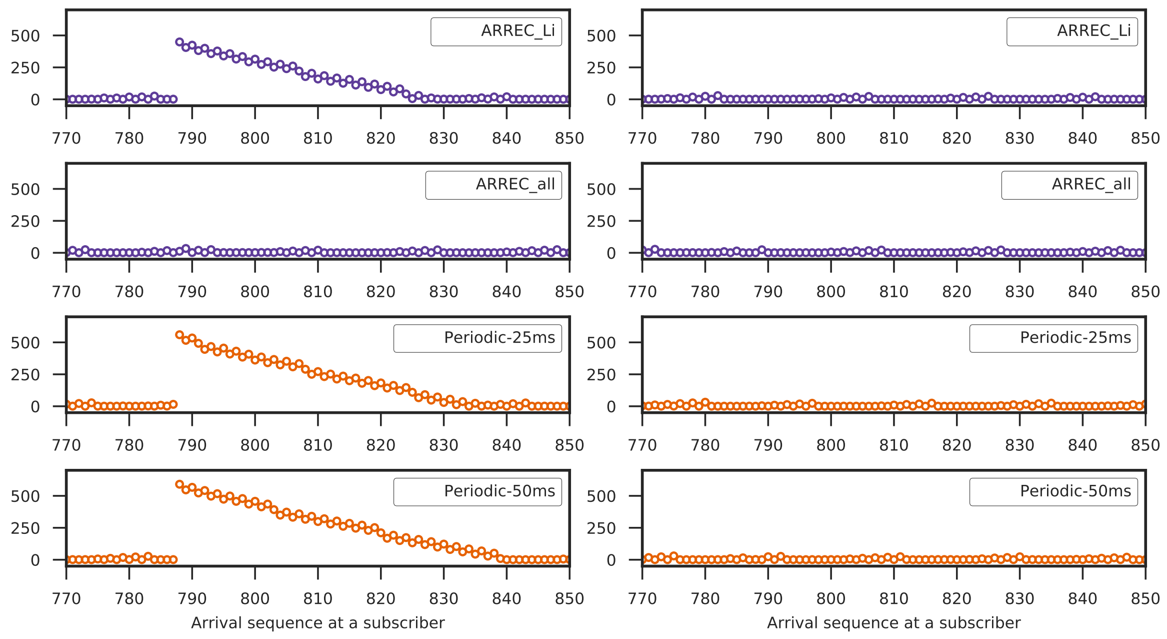
Figure 4.10: CPU% for the input proxy thread in the Backup.

Figure 4.10 shows the CPU utilization due to the recovery handler in the Backup, which was executed by the input proxy thread. The result for configuration `Retransmission-only` shows that the overhead due to fault detection (with a 10 ms polling period) accounted for about 5% in CPU utilization. The results for other configurations also show the overhead of handling data replication from the Primary. Configurations `ARREC_all` and `ARREC_Li` both had lower overhead than periodic replication, because of the use of lazy replication. The overhead of configuration `ARREC_all` was lower than that of configuration `ARREC_Li`, because the former permits a longer deadline before actually replicating data. Finally, Figure 4.10 also shows that when using periodic replication, a shorter period caused more work to be performed at the Backup (comparing configurations `Periodic-50ms` and `Periodic-25ms`), because more data was replicated from the Primary. This again suggests the benefit of leveraging temporal laxity in replication deadlines, as the ARREC design explicitly does.

We also empirically evaluated the performance of ARREC with small group transmissions as described in Section 4.7.1. In this case, ARREC also outperformed the periodic replication approaches, in terms of success rate for loss-tolerance (Figure 4.7(b)), network bandwidth consumption (Figure 4.8(b)), and CPU utilization (Figure 4.9(b)). Compared with the case of large group transmissions, we have three observations: (1) small group transmissions resulted in higher load to the system in the processing threads (compare Figures 4.9(a) and (b)); (2) the additional load due to replication remained the same (compare Figures 4.9(a) and (b)); and (3) the network bandwidth consumption was reduced (compare Figures 4.8(a) and (b)). With small group transmissions, there were more data publishers, and the connection overhead increased. As data arrivals became less bursty, there was less data within each period of replication, and therefore comparing with the periodic replication approaches, the relative benefit of lazy replication was less apparent.

4.7.3 Mitigation of Overhead Latency

We evaluated latency during and after fault recovery. In each topic category, we took a sequence of data arrivals at a publisher, gouging the change of the end-to-end latency over time. Figure 4.11 show the latency performance with and without the proposed quick recovery strategy, in topic category 4, for a workload of 1400 topics. Notably, configuration `ARREC_all` demonstrated almost no latency penalty for fault recovery, even with no aid from the proposed quick recovery strategy (Figure 4.11(a)), since many pending replications were skipped and thus little workload needed to be re-processed during fault recovery. For other configurations, the increase in latency was due to a combination of the effects from the need to re-process data in the Backup buffer and the new arrivals with shorter deadlines during re-processing. The quick recovery strategy removed the redundant need for re-processing, and thereby



(a) Without quick recovery. (b) With quick recovery.

Figure 4.11: Latency (ms) during and after fault recovery (large group).

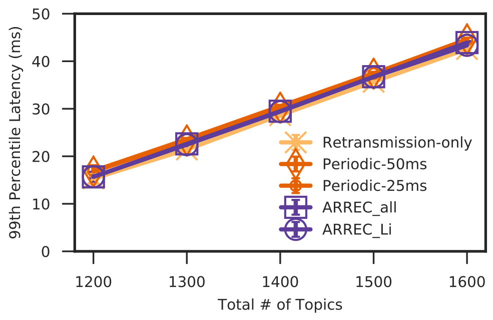


Figure 4.12: 99th percentile latency (large group).

effectively reduced the combination of effects that would otherwise slow down the system (Figure 4.11(b)). We observed similar results for other topic categories.

Finally, we evaluated the latency performance before a fault occurs. As we primarily consider soft latency requirements, we looked at the tail latency performance. In particular, we measured the 99th percentile latency for a topic in each category. Figure 4.12 shows

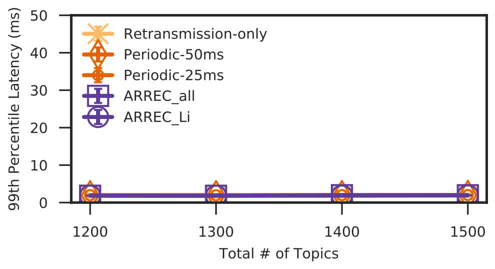
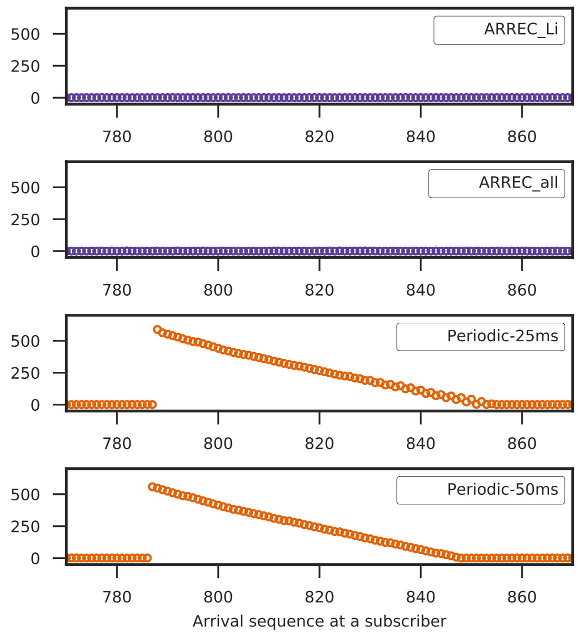


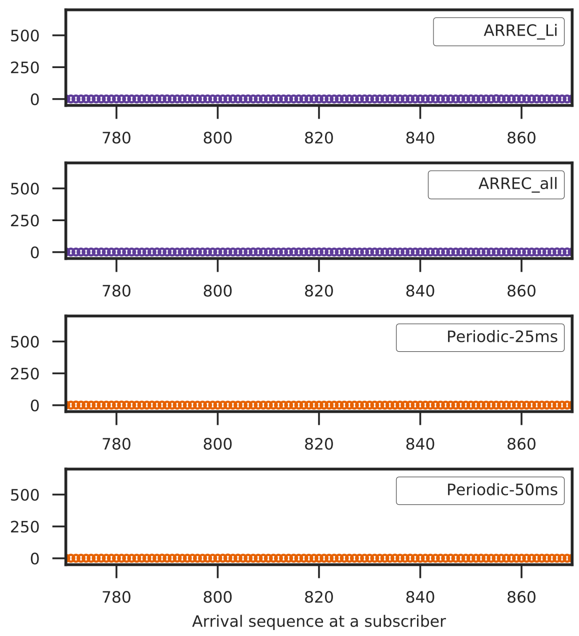
Figure 4.13: 99th percentile latency (small group).

the result for category 4, which has the shortest deadline (50 ms). Again, configuration `Retransmission-only` gave the baseline latency to evaluate the impact caused by each replication configuration. The results show that replication caused only a slight impact to the tail latency performance. Configurations `ARREC_all` and `ARREC_Li` always outperformed periodic replication baselines, because they both performed replication less frequently and replicated less data in each round. Configuration `ARREC_all` further outperformed configuration `ARREC_Li`, because it replicated least-frequently and with the least data in each round of replication. Overall, the 99th percentile latency stayed within the requirement.

For small group transmissions, the latency performance is shown in Figures 4.13 and 4.14, respectively. For fault recovery latency, compared with large group transmissions, we observed that configuration `ARREC_Li`, like configuration `ARREC_all`, did not have a latency burst right after failover. With small group transmissions, the data arrivals were less bursty and thus it is more likely that the replication handler may skip more data replications, since the processing engine would be able to finish some data processing and delivery earlier. Periodic replication approaches may not benefit much from this, as they are indifferent to replication deadlines and do not use a batch window to further postpone some replication actions. Finally, all configurations performed well in terms of tail latency (Figure 4.13), because there were fewer bursty data arrivals.



(a) Without quick recovery.



(b) With quick recovery.

Figure 4.14: Latency (ms) during and after fault recovery (small group).

4.8 Concluding Remarks

In this Chapter, we focused on support for IIoT services that efficiently perform in-band data processing and meet applications' requirements for data loss-tolerance and latency. Based on the observations that (1) IIoT in-band processing typically has a short execution time, and (2) IIoT data traffic often includes topics with different loss-tolerance and latency requirements, we introduced ARREC, a new middleware design and implementation that performs in-band data processing while also performing adaptive and lightweight data replication to a backup host. Our empirical evaluation shows that (1) ARREC can meet needed levels of data loss-tolerance with efficient resource consumption, while reducing latency overhead that would otherwise manifest during fault recovery; (2) concerning the frequencies of selecting data elements to replicate, it is favorable to select at a higher frequency because, thanks to short execution times of processing per data element, the proposed lazy replication heuristic can effectively skip many selected data elements; and (3) ARREC is performant for both large and small groups of IIoT data transmissions to the service, where large group transmissions may save the service's CPU utilization and small group transmissions may reduce tail latency of data processing.

Chapter 5

Conclusions

The advancement in IIoT systems and edge computing pose new research challenges in the areas of cyber-physical, real-time, and fault-tolerant computing. This dissertation has studied an IIoT data service framework that connects the devices that generate data and the applications that make use of them, and performs appropriate in-band data transformation. Such an IIoT data service framework must meet different types and levels of application-specific requirements pertaining to timing and reliability. Accordingly, this dissertation has presented three related middleware services, called CPEP, FRAME, and ARREC. CPEP addresses the need for real-time cyber-physical event processing, with a focus on both reducing latency according to the specified event priority, and shedding and sharing cyber-physical processing loads for efficiency. FRAME provides fault-tolerant real-time messaging, where both message loss-tolerance requirements and latency requirements are efficiently handled, from a holistic view of timing aspects, with consideration of platform parameters such as devices' retransmission capacity as well as transmission latency with the edge system and to the cloud. Finally, ARREC offers an adaptive middleware solution for heterogeneous in-band data processing that is typical in IIoT systems. The use of lazy group data replication effectively reduces the overhead of ensuring loss-tolerance, resulting in an IIoT data service that is real-time, fault-tolerant, and efficient. These three research contributions provide evidence that an IIoT service may continue meeting applications' demand of timely and reliable data supply, even though the underlying system may experience failures and failovers.

References

[1] Amazon. *AWS IoT Greengrass*. 2019. URL: `https://aws.amazon.com/greengrass/`.

[2] Amazon. *AWS Lambda*. 2019. URL: `https://aws.amazon.com/lambda/`.

[3] Apache. *Apache Flink: Scalable Stream and Batch Data Processing*. 2019. URL: `https://flink.apache.org`.

[4] Apache. *Apache Kafka*. 2019. URL: `http://kafka.apache.org`.

[5] Apache. *Apache Spark Streaming*. 2019. URL: `https://spark.apache.org/streaming/`.

[6] Amy Babay, Emily Wagner, Michael Dinitz, and Yair Amir. "Timely, reliable, and cost-effective internet transport service using dissemination graphs." In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 1–12.

[7] Theodore P. Baker, Michele Cirinei, and Marko Bertogna. "EDZL scheduling analysis." In: *Real-Time Systems* 40.3 (Dec. 2008), pp. 264–289.

[8] Jaiganesh Balasubramanian, Aniruddha Gokhale, Abhishek Dubey, Friedhelm Wolf, Chenyang Lu, Chris Gill, and Douglas Schmidt. "Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2010, pp. 69–78.

[9] P. A. Barret, Andrew M Hilborne, Peter G Bond, Douglas T Seaton, Paulo Veríssimo, Luís Rodrigues, and Neil A Speirs. "The Delta-4 extra performance architecture (XPA)." In: *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*. IEEE. 1990, pp. 481–488.

[10] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems." In: *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE. 2012, pp. 145–154.

[11] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. "Mixed-criticality scheduling on multiprocessors." In: *Real-Time Systems* 50.1 (2014), pp. 142–177.

[12] Anand Bhat, Soheil Samii, and Ragunathan (Raj) Rajkumar. "Recovery Time Considerations in Real-Time Systems Employing Software Fault Tolerance." In: *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Ed. by Sebastian Altmeyer. Vol. 106. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 23:1–23:22. DOI: 10.4230/LIPIcs.ECRTS.2018.23.

[13] Kenneth P. Birman. *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. Springer Publishing Company, Incorporated, 2012.

[14] Bjorn B. Brandenburg. "Scheduling and Locking in Multiprocessor Real-Time Operating Systems." PhD thesis. The University of North Carolina at Chapel Hill, 2011.

[15] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. "The primary-backup approach." In: *Distributed systems* 2 (1993), pp. 199–216.

[16] Alan Burns and Robert Davis. "Mixed criticality systems-a review (the eleventh edition)." In: *Department of Computer Science, University of York, Tech. Rep* (2018), pp. 1–77.

[17] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. "Lightweight asynchronous snapshots for distributed dataflows." In: *arXiv preprint arXiv:1506.08603* (2015).

[18] K Mani Chandy and Leslie Lamport. "Distributed snapshots: Determining global states of distributed systems." In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), pp. 63–75.

[19] Gianpaolo Cugola and Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing." In: *ACM Comput. Surv.* 44.3 (June 2012), 15:1–15:62.

[20] Gianpaolo Cugola and Alessandro Margara. "TESLA: A Formally Defined Event Specification Language." In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. 2010, pp. 50–61.

[21] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. "Remus: High availability via asynchronous virtual machine replication." In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco. 2008, pp. 161–174.

[22] Robert I. Davis and Alan Burns. "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems." In: *ACM computing surveys (CSUR)* 43.4 (2011), p. 35.

[23] Dionisio De Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. "On the scheduling of mixed-criticality real-time task sets." In: *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*. Citeseer. 2009, pp. 291–300.

[24] Alma L. Juarez Dominguez. "Detection of Feature Interactions in Automotive Active Safety Features." PhD thesis. University of Waterloo, 2012.

[25] Nour-Eddin El Faouzi, Henry Leung, and Ajeesh Kurian. "Data Fusion in Intelligent Transportation Systems: Progress and Challenges–A Survey." In: *Information Fusion* 12.1 (2011), pp. 4–10.

[26] Peter C. Evans and Marco Annunziata. "Industrial Internet: Pushing the Boundaries of Minds and Machines." In: *General Electric Reports* (2012).

[27] Tom Fleming, Huang-Ming Huang, Alan Burns, Chris Gill, Sanjoy Baruah, and Chenyang Lu. "Corrections to and Discussion of "Implementation and Evaluation of Mixed-criticality Scheduling Approaches for Sporadic Tasks"." In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.3 (2017), p. 77.

[28] FogHorn. *Industries Served and Use Cases - Foghorn Systems*. 2017. URL: `https://www.foghorn.io/industries/`.

[29] The Linux Foundation. *The Real Time Linux Collaborative Project*. 2017. URL: `https://wiki.linuxfoundation.org/realtime/start`.

[30] Matteo Frigo and Steven G. Johnson. *FFTW*. 2017. URL: `http://www.fftw.org`.

[31] Christopher D Gill, Ron K Cytron, and Douglas C Schmidt. "Multiparadigm scheduling for distributed real-time embedded computing." In: *Proceedings of the IEEE* 91.1 (2003), pp. 183–197.

[32] GitHub. *Complex Event Processing for Flink*. 2017. URL: `https://github.com/apache/flink/tree/master/flink-libraries/flink-cep`.

[33] GitHub. *PTP Daemon*. 2019. URL: `https://github.com/ptpd/ptpd`.

[34] GnuPG. *The Libgcrypt Library*. 2017. URL: `https://gnupg.org/software/libgcrypt`.

[35] Aniruddha S Gokhale, Balachandran Natarajan, Douglas C Schmidt, and Joseph K Cross. "Towards real-time fault-tolerant CORBA middleware." In: *Cluster Computing* 7.4 (2004), pp. 331–346.

[36] Vincenzo Gulisano, Zbigniew Jerzak, Spyros Voulgaris, and Holger Ziekow. "The DEBS 2016 Grand Challenge." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 2016, pp. 289–292.

[37] Gregory Hackmann, Fei Sun, Nestor Castaneda, Chenyang Lu, and Shirley Dyke. "A holistic approach to decentralized structural damage localization using wireless sensor networks." In: *Real-Time Systems Symposium, 2008*. IEEE. 2008, pp. 35–46.

[38] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. "The Design and Performance of a Real-Time CORBA Event Service." In: *ACM SIGPLAN Notices* 32.10 (1997), pp. 184–200.

[39] Huang-Ming Huang, Christopher Gill, and Chenyang Lu. "Implementation and evaluation of mixed-criticality scheduling approaches for periodic tasks." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE. 2012, pp. 23–32.

[40] IEEE. "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems - Redline." In: *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002) - Redline* (July 2008), pp. 1–300.

[41] Industrial Internet Consortium. "Industrious Internet Reference Architecture." In: (Jan. 2017).

[42] Real-Time Innovations. *Connext DDS at a Glance: Understanding the Software Framework that Connects the Industrial IoT.* White Paper. Real-Time Innovations, 2017.

[43] Konrad Iwanicki. "A Distributed Systems Perspective on Industrial IoT." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS).* IEEE. 2018, pp. 1164–1170.

[44] Kedar Khandeparkar, Krithi Ramamritham, and Rajeev Gupta. "QoS-Driven Data Processing Algorithms for Smart Electric Grids." In: *ACM Trans. Cyber-Phys. Syst.* 1.3 (Mar. 2017), 14:1–14:24. DOI: 10.1145/3047410.

[45] Daniel Kirsch. "The Value of Bringing Analytics to the Edge." In: *Hurwitz & Associates* (2015).

[46] Gerald G. Koch, Boris Koldehofe, and Kurt Rothermel. "Cordies: Expressive Event Correlation in Distributed Systems." In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems.* ACM. 2010, pp. 26–37.

[47] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: a Distributed Messaging System for Log Processing." In: *Proceedings of the NetDB.* 2011, pp. 1–7.

[48] Greg R. Lavender and Douglas C. Schmidt. "Active Object: an Object Behavioral Pattern for Concurrent Programming." In: *Proc. Pattern Languages of Programs,* 1995.

[49] Jinkyu Lee and Insik Shin. "Edzl schedulability analysis in real-time multicore scheduling." In: *IEEE Transactions on Software Engineering* 39.7 (2013), pp. 910–916.

[50] Suk Kyoon Lee. "On-line multiprocessor scheduling algorithms for real-time tasks." In: *TENCON'94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994.* IEEE. 1994, pp. 607–611.

[51] Breno Henrique Leitao. "Tuning 10Gb network cards on Linux." In: *Proceedings of the 2009 Linux Symposium.* Citeseer. 2009.

[52] Jane W. S. Liu. *Real-Time Systems.* 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[53] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[54] Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. "GraphCEP: Real-Time Data Analytics Using Parallel Complex Event and Graph Processing." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems.* ACM. 2016, pp. 309–316.

[55] Odorico Machado Mendizabal, Fernando Luís Dotti, and Fernando Pedone. “High performance recovery for parallel state machine replication.” In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 34–44.

[56] Microsoft. *Azure IoT Edge*. 2018. URL: `https://azure.microsoft.com/en-us/services/iot-edge/`.

[57] David L Mills. “Internet time synchronization: the network time protocol.” In: *IEEE Transactions on communications* 39.10 (1991), pp. 1482–1493.

[58] Priya Narasimhan, TA Dumitraş, Aaron M Paulos, Soila M Pertet, Carlos F Reverte, Joseph G Slember, and Deepti Srivastava. “MEAD: support for Real-Time Fault-Tolerant CORBA.” In: *Concurrency and Computation: Practice and Experience* 17.12 (2005), pp. 1527–1545.

[59] NSQ. *A Realtime Distributed Messaging Platform*. 2019. URL: `https://nsq.io`.

[60] Object Management Group. *Data Distribution Service (DDS)*. 2015. URL: `http://www.omg.org/spec/DDS/`.

[61] Daniel Piri. “Sensor Fusion for Nanopositioning.” MA thesis. Austria: Vienna University of Technology, 2014.

[62] Douglas C. Schmidt. *The ADAPTIVE Communication Environment (ACE)*. 2017. URL: `http://www.cs.wustl.edu/%5C%7Eschmidt/ACE.html`.

[63] Fred B Schneider. “Replication management using the state-machine approach.” In: *Distributed systems* 2 (1993), pp. 169–198.

[64] Abu Sebastian and Angeliki Pantazi. “Nanopositioning With Multiple Sensors: A Case Study in Data Storage.” In: *IEEE Transactions on Control Systems Technology* 20.2 (2012), pp. 382–394.

[65] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. “Edge computing: Vision and challenges.” In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[66] OASIS Standard. “MQTT version 3.1.1.” In: (2014).

[67] John A. Stankovic, Sang Hyuk Son, and Jörgen Hansson. “Misconceptions About Real-Time Databases.” In: *Computer* 32.6 (1999), pp. 29–36.

[68] Ciza Thomas, ed. *Sensor Fusion and Its Applications*. Sciyo (Publisher), 2010.

[69] Ubuntu. *Time Synchronization*. 2019. URL: `https://help.ubuntu.com/lts/serverguide/NTP.html`.

[70] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. “Drizzle: Fast and adaptable stream processing at scale.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 374–389.

[71] Chao Wang, Christopher Gill, and Chenyang Lu. “Real-time middleware for cyber-physical event processing.” In: *Quality of Service (IWQoS), 2017 IEEE/ACM 25th International Symposium on*. IEEE. 2017, pp. 1–6.

[72] Cheng Wang, Xusheng Chen, Weiwei Jia, Boxuan Li, Haoran Qiu, Shixiong Zhao, and Heming Cui. “PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance.” In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association. 2018.

[73] Ming Xiong, Rajendran Sivasankaran, John A. Stankovic, Krithi Ramamritham, and Don Towsley. “Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics.” In: *Real-Time Systems Symposium, 1996., 17th IEEE*. IEEE. 1996, pp. 240–251.

[74] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. “A survey on the edge computing for the Internet of Things.” In: *IEEE access* 6 (2018), pp. 6900–6919.

[75] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. “Discretized streams: Fault-tolerant streaming computation at scale.” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 423–438.

[76] Vincent Zalzal. *KFilter - Free C++ Extended Kalman Filter Library*. 2008. URL: `http://kalman.sourceforge.net`.