# FRAME: Fault Tolerant and Real-Time Messaging for Edge Computing

Chao Wang, Christopher Gill, Chenyang Lu

Department of Computer Science and Engineering
Washington University in St. Louis
Email: {chaowang, cdgill, lu}@wustl.edu

***Abstract*—Edge computing systems for Industrial Internet of Things (IIoT) applications require reliable and timely message delivery. Both latency discrepancies within edge clouds, and heterogeneous loss-tolerance and latency requirements pose new challenges for proper quality of service differentiation. Efficient differentiated edge computing architectures are also needed, especially when common fault-tolerant mechanisms tend to introduce additional latency, and when cloud traffic may impede local, time-sensitive message delivery. In this paper, we introduce FRAME, a fault-tolerant real-time messaging architecture. We first develop timing bounds that capture the relation between traffic/service parameters and loss-tolerance/latency requirements, and then illustrate how such bounds can support proper differentiation in a representative IIoT scenario. Specifically, FRAME leverages those timing bounds to schedule message delivery and replication actions to meet needed levels of assurance. FRAME is implemented on top of the TAO real-time event service, and we present empirical evaluations in a local edge computing test-bed and an Amazon Virtual Private Cloud. The results of those evaluations show that FRAME can efficiently meet different levels of message loss-tolerance requirements, mitigate latency penalties caused by fault recovery, and meet end-to-end soft deadlines during normal, fault-free operation.**

## I. 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 [1]–[4], where latency-sensitive applications run locally in *edge* servers, while computation-intensive and shareable tasks run in a private cloud that supports multiple edges (Fig. 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 differentiate levels of latency and loss-tolerance requirements. Without latency differentiation, latency-sensitive messages may arrive too late; without 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 consider 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.

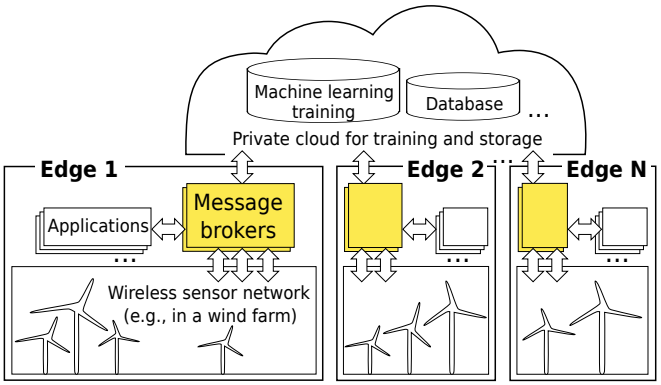



Fig. 1. An Illustration of IIoT Edge Computing.

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 [5], [6]. 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 [7], [8]. 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 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 [9], 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 II, we compare and contrast our approach to other related work. In Section III, we describe FRAME's fault-tolerant real-time model, using an illustrative IIoT scenario. The architectural design of FRAME is presented in Section IV, and its implementation is described in Section V. In Section VI, we present an empirical evaluation of FRAME. Section VII summarizes and presents conclusions.

## II. 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 [1], [10]. 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 [11] 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 [5], [12]. For distributed real-time systems, the TAO real-time event service [9] supports a configurable framework for event filtering, correlation, and dispatching, along with a scheduling service [13]. 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.

TABLE 1
Comparison of Related Middlewares and Standards.

| Middleware/Standard | Message-Loss Tolerance Strategies | | |
|---|---|---|---|
| | Pub. Resend | Local Disk | Backup Broker |
| Flink [20] | x | x | |
| Kafka [21] | x | x | x |
| Spark Streaming [22] | x | x | |
| NSQ[2] | | x | |
| DDS (Standard) [23] | | x | |
| MQTT (Standard) [24] | x | | |
| FRAME (This work) | x | | x |

Among fault-tolerance approaches, service replication has been studied for reliable distributed systems. Delta-4 XPA [6] coined the names *active/passive/semi-active* replication. In active replication, also called the state-machine approach [7], 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 [14], 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 [8]. A discussion regarding conflicts between real-time and fault-tolerance capabilities is available [15]. There are also recent studies for virtual machine fault-tolerance [16], [17] and for the recovery of faulty replicas [18]. 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 [19] 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 written to local hard disks; (3) *backup brokers*: like the primary-backup approach, message copies are transferred to other brokers; Table 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

[1] https://aws.amazon.com/greengrass/

[2] https://nsq.io

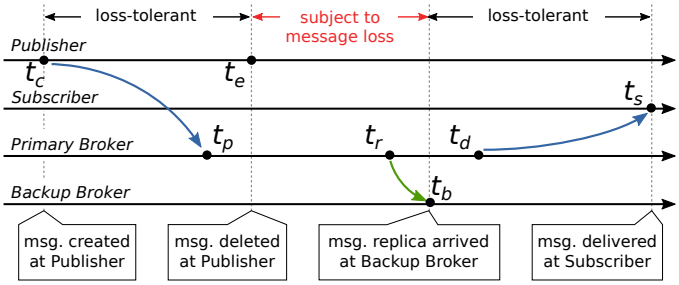



Fig. 2. Example timelines within the scope of message creation and delivery, and the relation between events happening in each component.

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.

## III. Fault-Tolerant Real-Time Messaging Model

In this section, we present the constraints for a messaging system to meet its fault-tolerant 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.

### A. 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 (Fig. 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.

### B. 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 failure, 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[3], 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 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 [12] 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.

### C. Temporal Semantics and Timing Bounds

As illustrated in Fig. 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 (e.g., due to a limited IIoT device capacity) 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

[3]For example, via PTP [25] and/or NTP [26] protocols; see Section VI-A for our experimental setup.

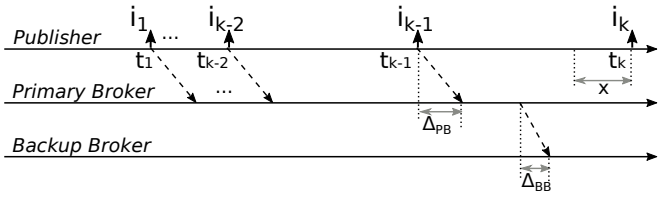



Fig. 3. Example Timelines for The Proof of Lemma 1.

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*

$$R_i^r \leq D_i^r = (N_i + L_i)T_i - \Delta_{PB} - \Delta_{BB} - x. \quad (1)$$

*Proof.* Without loss of generality, we consider a series of message creation times for topic $i$, as shown in Fig. 3. Adding $\Delta_{PB}$ 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*

$$R_i^d \leq D_i^d = D_i - \Delta_{PB} - \Delta_{BS}. \quad (2)$$

*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 $t_b$). 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 $\Delta_{BS}$, 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 $i$. □

TABLE 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 | $\infty$ | 0 | Edge |
| 5 | 500 | 500 | 0 | 1 | Cloud |

### D. 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 (1) and (2), and we schedule both activities using the Earliest Deadline First (EDF) policy [12]. 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.

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

$$D_i^d \leq D_i^r. \quad (3)$$

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 [1], 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 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.

**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

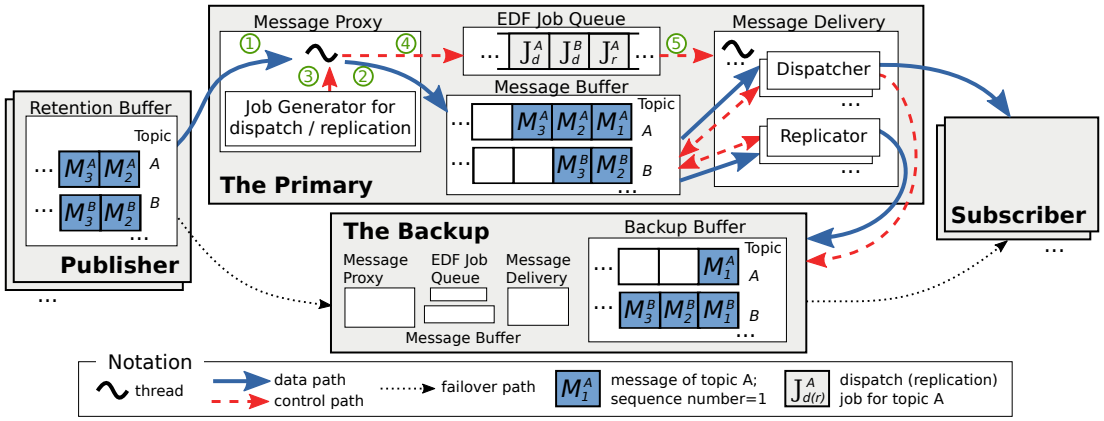



Fig. 4. The FRAME Architecture.

topic $i$. For example, if we are to meet a fault-tolerance requirement $L_i = 0$ (i.e., zero message loss), Equation (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 (1) and (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 VI.

**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 VI 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) 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) suggests a likely need for replication, unless $\Delta_{BS}$ 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 $\Delta_{BS}$, 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 $\Delta_{BS}$. Although an underestimated cloud latency at run-time might delay the cloud traffic (due to the use of EDF policy), edge computing clouds 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.

## IV. 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 Fig. 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.

### A. 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 $\Delta_{BS}$ 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 Fig. 4). At runtime, 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 $\Delta_{PB}$ 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 $\Delta_{PB}$ 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.

### B. 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.

TABLE 3
ALGORITHM FOR DISPATCH-REPLICATE COORDINATION.

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

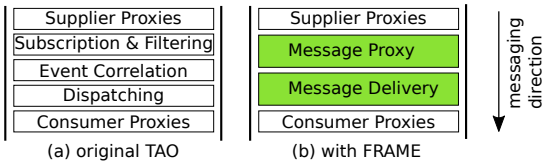



Fig. 5. Implementation of FRAME within TAO's Real-Time Event Service.

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.

## V. FRAME IMPLEMENTATION

We implemented the FRAME architecture within the TAO real-time event service [9], 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 our work, 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 Fig. 5(a). Fig. 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 to timestamp and compare deadlines to determine message priority. The Message Buffer, Backup Buffer, and Retention Buffer are all implemented as ring buffers.

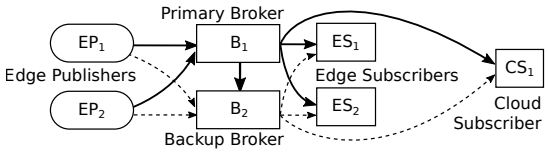



Fig. 6. Topology for empirical evaluation. Dotted lines denote failover paths.

## VI. 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 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, to cover a range of CPU utilization.

### A. Experiment Setup

Our test-bed consists of seven hosts, as shown in Fig. 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` [27]. We synchronized our local hosts via PTPd[4], an open source implementation of the PTP protocol [25]. 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`[5] that utilizes NTP [26], with synchronization error in milliseconds. The latency measurement for $\Delta_{BS}$ is dominated by the communication latency to AWS EC2, which was at least 20 milliseconds.

TABLE 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 | $\infty$ | 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 | $\infty$ | 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 $\pm$ 30.1 | 0.0 | 100.0 |
| 50 | 3 | 100.0 | 80.0 $\pm$ 30.1 | 0.0 | 100.0 |
| 100 | 0 | 100.0 | 73.2 $\pm$ 30.7 | 0.0 | 78.4 $\pm$ 13.3 |
| 100 | 3 | 100.0 | 79.3 $\pm$ 29.9 | 0.0 | 99.3 $\pm$ 0.5 |
| 100 | $\infty$ | 100.0 | 100.0 | 100.0 | 100.0 |
| 500 | 0 | 100.0 | 80.0 $\pm$ 30.1 | 0.0 | 100.0 |

We compared four 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 their arrival orders; (4) *FCFS-*, which is FCFS without dispatch-replicate coordination. In both FCFS and FCFS-, the Primary first performed replication and then dispatch.

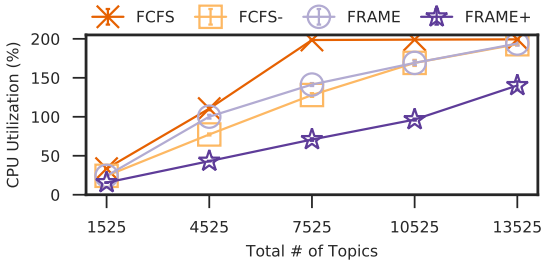
For each configuration we ran each test case ten times and calculated the 95% confidence interval for each measurement. We allowed 35 seconds for system initialization and warm-up. The measuring phase spanned 60 seconds. We injected a crash failure by sending signal `SIGKILL` to the Primary broker at 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.

### B. Message Loss-Tolerance Enforcement

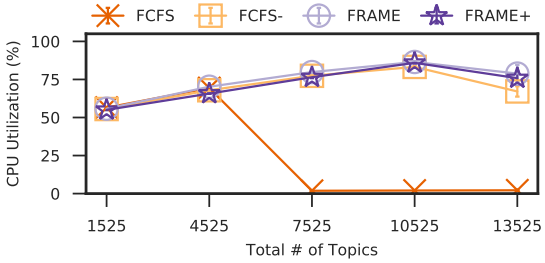
Table 4 shows the success rate of meeting loss-tolerance requirements under increasing workload. All four configurations had 100% success rate for 1525 and 4525 topics. FRAME outperformed FCFS after the workload reached 7525 topics and more, thanks to the selective replication of Proposition 1. FRAME only performed the needed replications (topic categories 2 and 5) and suppressed the others (topic categories 0, 1, and 3), saving more than 50% in CPU utilization for the Message Delivery module, compared with the result of FCFS

[4]https://github.com/ptpd/ptpd

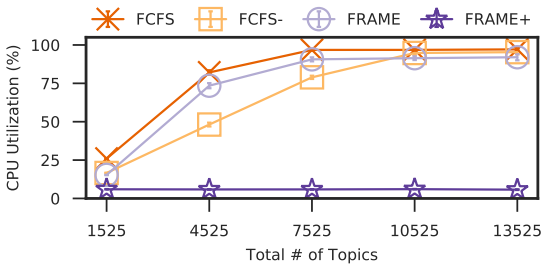
[5]https://help.ubuntu.com/lts/serverguide/NTP.html




(a) Message Delivery Module in the Primary.




(b) Message Proxy Module in the Primary.




(c) Message Proxy Module in the Backup.

Fig. 7. CPU Utilization for Each Configuration.

for the case with 7525 topics (Fig. 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 Fig. 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 perform any replication to its Backup, and loss tolerance was solely performed by publisher re-sending the retained messages. As shown in Table 4, FRAME+ met all loss-tolerance requirements in every case. Further, the replication removal saved CPU usage in the Primary broker host (Fig. 7(a)). The replication removal also saved CPU usage in the Backup broker host (Fig. 7(c)), because the Primary would send less traffic to it.

To evaluate the impact of dispatch-replicate coordination, we compared FCFS with FCFS-. FCFS- outperformed FCFS in loss-tolerance performance (Table 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. We evaluate this in the next subsection.

We further conducted a micro-benchmark to show that FRAME can keep the same level of loss tolerance despite

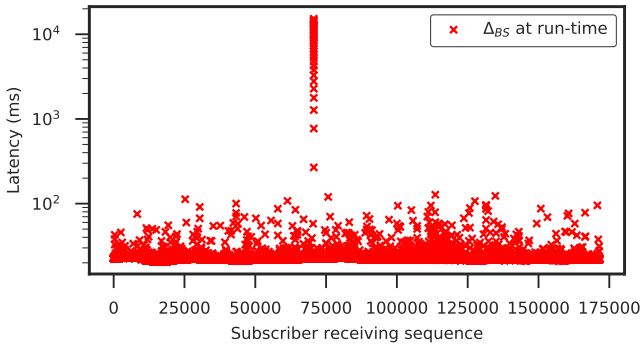



Fig. 8. Value of $\Delta_{BS}$ for a topic in category 5 through a 24-hour duration.

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 $\Delta_{BS}$ for a topic in category 5 (Fig. 8)[6]. The setup value of $\Delta_{BS}$ 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 $\Delta_{BS}$.

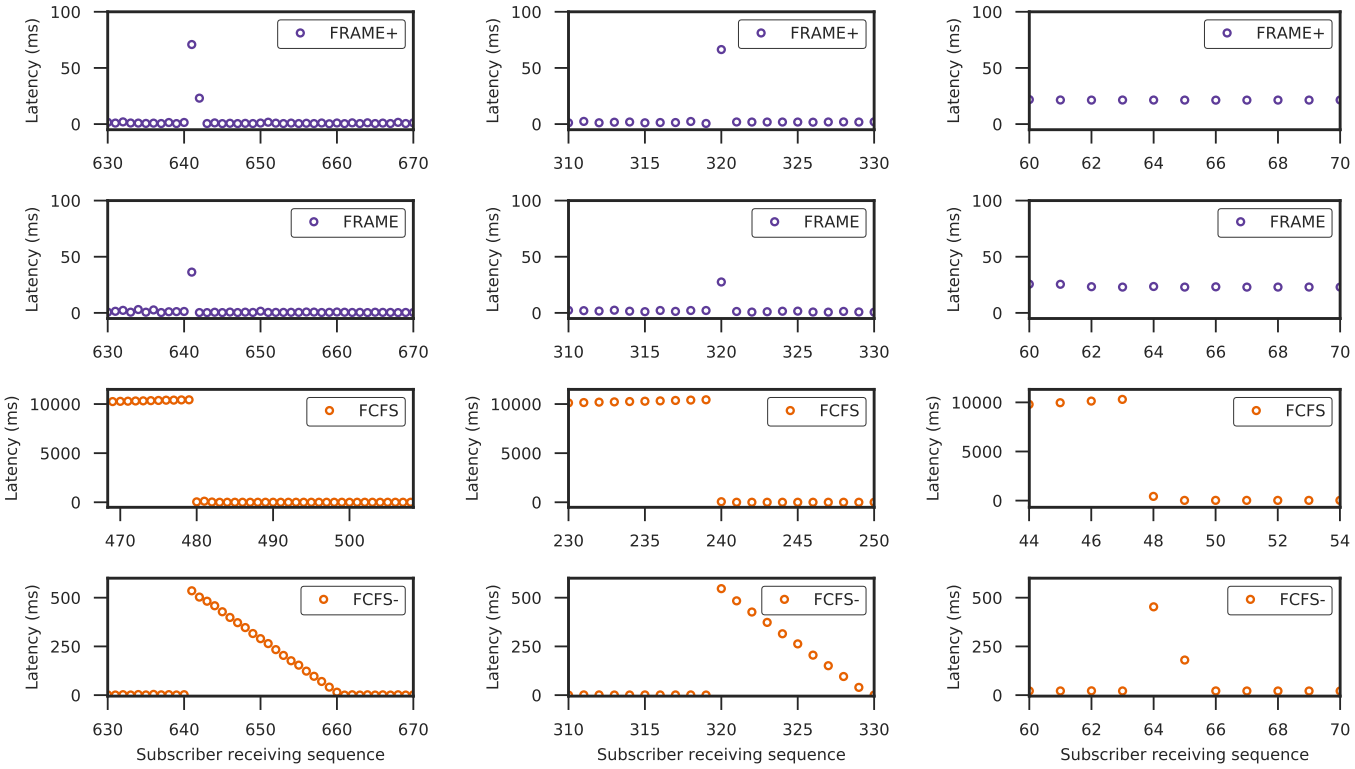
### C. Latency Penalties Caused by Fault Recovery

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 result is shown in Fig. 9 with each column presenting four configurations for a topic category.

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

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




(a) Category 0, $T_i = 50$, $D_i = 50$. (b) Category 2, $T_i = 100$, $D_i = 100$. (c) Category 5, $T_i = 500$, $D_i = 500$.

Fig. 9. End-to-end latency of a topic before, upon, and after fault recovery.

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 (Fig. 9(b)), the Backup began processing at the 240th message.

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 Fig. 9(b), 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 (Fig. 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 (Fig. 9(c)).

### D. 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 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.

### E. 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-).

TABLE 5
SUCCESS RATE FOR LATENCY REQUIREMENT (%).

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

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

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+).

## VII. CONCLUSIONS

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.

## ACKNOWLEDGMENT

This research was supported in part by NSF grant 1514254 and ONR grant N000141612108.

## REFERENCES

[1] Industrial Internet Consortium, "Industrious Internet Reference Architecture," Jan 2017. [Online]. Available: https://www.iiconsortium.org/IIRA.htm

[2] K. Iwanicki, "A distributed systems perspective on industrial iot," in *2018 IEEE 38th International Conference on Distributed Computing Systems*, 2018, pp. 1164–1170.

[3] P. C. Evans and M. Annunziata, "Industrial internet: Pushing the boundaries of minds and machines," *General Electric Reports*, 2012.

[4] D. Kirsch, "The value of bringing analytics to the edge," *Hurwitz & Associates*, 2015.

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

[6] P. Barret, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Veríssimo, L. Rodrigues, and N. 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.

[7] F. B. Schneider, "Replication management using the state-machine approach," *Distributed systems*, vol. 2, pp. 169–198, 1993.

[8] A. S. Gokhale, B. Natarajan, D. C. Schmidt, and J. K. Cross, "Towards real-time fault-tolerant corba middleware," *Cluster Computing*, vol. 7, no. 4, pp. 331–346, 2004.

[9] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 184–200, 1997.

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

[11] A. Babay, E. Wagner, M. Dinitz, and Y. Amir, "Timely, reliable, and cost-effective internet transport service using dissemination graphs," in *2017 IEEE 37th International Conference on Distributed Computing Systems*, 2017, pp. 1–12.

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

[13] C. D. Gill, R. K. Cytron, and D. C. Schmidt, "Multiparadigm scheduling for distributed real-time embedded computing," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 183–197, 2003.

[14] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[15] P. Narasimhan, T. Dumitraş, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "Mead: support for real-time fault-tolerant corba," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 12, pp. 1527–1545, 2005.

[16] C. Wang, X. Chen, W. Jia, B. Li, H. Qiu, S. Zhao, and H. Cui, "Plover: Fast, multi-core scalable virtual machine fault-tolerance," in *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[17] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, pp. 161–174.

[18] O. M. Mendizabal, F. L. Dotti, and F. Pedone, "High performance recovery for parallel state machine replication," in *2017 IEEE 37th International Conference on Distributed Computing Systems*, 2017, pp. 34–44.

[19] J. Balasubramanian, A. Gokhale, A. Dubey, F. Wolf, C. Lu, C. Gill, and D. 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.

[20] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *arXiv preprint arXiv:1506.08603*, 2015.

[21] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: a distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[22] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. 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.

[23] Object Management Group. (2015) Data Distribution Service (DDS). [Online]. Available: http://www.omg.org/spec/DDS/

[24] O. Standard. (2019) Message queuing telemetry transport, version 3.1.1. [Online]. Available: http://mqtt.org

[25] IEEE, "Ieee standard for a precision clock synchronization protocol for networked measurement and control systems - redline," *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002) - Redline*, pp. 1–300, July 2008.

[26] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[27] B. H. Leitao, "Tuning 10gb network cards on linux," in *Proceedings of the 2009 Linux Symposium*. Citeseer, 2009.