# Adaptive Data Replication in Real-Time Reliable Edge Computing for Internet of Things

Chao Wang
*Department of Computer Science and Information Engineering*
*National Taiwan Normal University*
Taipei City, Taiwan R.O.C.
cw@ntnu.edu.tw

Christopher Gill Chenyang Lu
*Department of Computer Science and Engineering*
*Washington University in St. Louis*
St. Louis, USA
{cdgill, lu}@wustl.edu

***Abstract*—Many Internet-of-Things (IoT) applications rely on timely and reliable processing of data collected from embedded sensing devices. To achieve timely response, computing tasks are executed on IoT gateways at the edge of clouds, and for fault tolerance, the gateways perform data replication to backup gateways. In this paper, we report our study of data replication strategies and a real-time and fault-tolerant edge computing architecture for IoT applications. We first analyze how both embedded devices' storage constraints and data replication frequency may impose timing constraints on data replication tasks, and we investigate correlations between execution of data replication tasks and execution of edge computing tasks. Accordingly, we propose adaptive data replication strategies and introduce a framework for real-time reliable edge computing to meet the needed levels of data loss tolerance and timeliness. We have implemented our framework and empirically evaluated the proposed strategies with baseline approaches. We set up experiments using Industrial IoT traffic configurations that have requirements on data loss and timeliness, and our experimental results show that the proposed data replication strategies and framework can ensure needed levels of data loss tolerance, save network bandwidth consumption, while maintaining the latency performance.**

## I. Introduction

In Internet-of-Things (IoT) systems, computing at the edge of clouds is essential for latency-sensitive applications [1]. IoT edge computing platforms may leverage a trained machine learning model and perform local inference using data sent from embedded sensing devices. For fault tolerance, the edge computing host may replicate data to a secondary host to avoid a single point of failure and data losses. But replicating data at the rate of data arrivals is inefficient and will consume nontrivial network bandwidth. In this work, we present an adaptive data replication architecture for IoT edge computing that can meet applications' latency and data-loss requirements with efficiency.

A motivating example is seen in structural health inference applications under the Industrial Internet Reference Architecture [2]. In these applications, a trained inference model may be loaded at an IoT gateway for local event inference. Embedded sensing devices take records of local environmental status (e.g., structural vibration) and send out the data through an IoT gateway. An edge computing task at the IoT gateway then filters the data and/or infers events (e.g., structural damage). The processed data is sent to other local/remote application hosts for further analytics or control. To ensure data delivery, using the primary-backup approach [3] an IoT gateway may replicate data to a backup IoT gateway. Should the primary IoT gateway fail, the pending data may be processed by the backup edge computing tasks, and the embedded sensing devices may re-transmit data to the backup IoT gateway (Fig. 1).

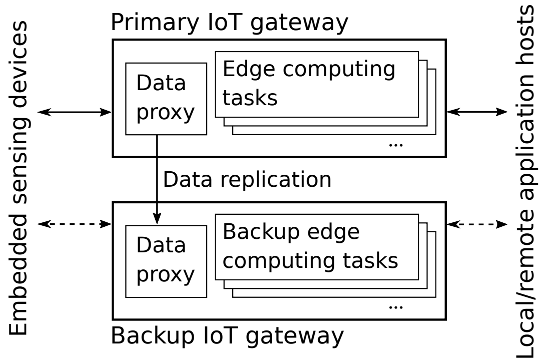


Fig. 1. Edge computing for Internet of Things.

To ensure real-time and reliable performance, research challenges arise from both platform and application aspects. From the platform aspect, embedded sensing devices have limited storage capacity to hold many data copies for re-transmissions, plus both the devices and IoT gateways do not have abundant network bandwidth for either data re-transmission or data replication. From the application aspect, applications may not tolerate more than a certain number of consecutive losses of data, plus applications may have end-to-end timing requirements in time scales of seconds to tens of milliseconds. With the platform constraints and application requirements, it is challenging for an IoT edge computing system to meet the requirements while being efficient. For example: *If a sensing device has less storage to save previous data, does the IoT gateway need to replicate its data more aggressively, and to what extent? If an application can tolerate losing every other data item, does the gateway need to replicate the new data item if it has successfully processed and delivered the last data item? Any constraint for the gateway to replicate data selectively?*

978-1-7281-6602-5/20/$31.00 ©2020 IEEE
DOI 10.1109/IoTDI49375.2020.00019

In this paper, we present three major contributions:

1) *A holistic analysis for real-time reliable IoT edge computing.* We take into account the storage limitation of embedded sensing devices and propose a sufficient condition for the system to meet application-specific requirements, and we analyze when to perform data replication and its corresponding deadline. We point out that data may not need to be replicated for every arrival, and we show how changes in the frequency of replication would change the deadline of replication.
2) *ARREC: adaptive real-time reliable edge computing architecture.* We describe ARREC, an efficient edge computing architecture that may adaptively replicate data according to application's requirements for data loss and latency. Leveraging the facts that most IoT edge computing tasks have small execution times, ARREC is designed to postpones replication actions to the extent possible. Eventually, many replication actions may be safely skipped, because the primary IoT gateway may have already completed processing data and delivered the result to the subscriber.
3) *An efficient implementation and empirical evaluation.* We describe our implementation of ARREC within the mature TAO real-time event service [4] middleware, and we present an empirical validation of ARREC's performance using typical Industrial IoT traffic configurations that have requirements on data loss and timeliness. Our empirical results show that ARREC can meet both data-loss and latency requirements while saving network bandwidth consumption.

The rest of this paper is organized as follows: In Section II we present our system model and analysis for data replication. In Section III, we introduce the ARREC architecture and implementation, followed by our empirical evaluation in Section IV. We survey related work in Section V, and summarize this work and present conclusions in Section VI.

## II. System Model and Analysis

We focus on a service that performs edge computing in IoT gateways (Fig. 1), and we assume a publish-subscribe model. Each data publisher (embedded sensing devices) publishes data in terms of *topics* to an IoT gateway, with a *minimum inter-publishing time* $T_i$ for data topic $i$. The IoT gateway receives data from data publishers and the data will trigger an edge computing task. At the completion of the edge computing task, the gateway then delivers the processed results, also in terms of topics, to its subscribers (local/remote application hosts). We follow the primary-backup approach [3] and define two types of IoT gateways, the *Primary* that processes and replicates data, and the *Backup* that receives the replicated data. There is one Backup per Primary IoT gateway, each running on a different host. A network link connects both hosts, and data replication between the hosts will consume network bandwidth. The Primary is subject to processor crash failures with fail-stop behavior, and 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.

TABLE I
Example Data Topic Specification.

| Category | $L_i$ | $N_i$ | $D_i^p$ (ms) | $T_i$ (ms) |
|---|---|---|---|---|
| 1 | 0 | 1 | $\infty$ | 50 |
| 2 | 0 | 1 | 100 | 100 |
| 3 | 0 | 1 | 500 | 500 |
| 4 | 3 | 0 | 50 | 50 |
| 5 | 3 | 0 | 100 | 100 |

We assume the following requirements. For data topic $i$, each data publisher can keep $N_i$ 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* $L_i \geq 0$, saying that the subscriber cannot accept more than $L_i$ consecutive losses for data topic $i$. Further, each subscriber has a *latency requirement* $D_i^p$ for each data topic $i$ 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. Finally, we define a *relative replication deadline* $D_i^r$ for data topic $i$, or simply a replication deadline, to be the maximum allowable response time for the Primary to complete replicating the data.

Throughout the rest of this paper, we use example data topics in Table I for demonstration purpose, based on the following three observations from Industrial IoT systems:

1) *Data publishers have limited data storage for retransmission.* Often, data publishers are embedded sensing devices. Some other data publishers such as wireless base stations may have more capacity, but the capacity is amortized to the number of data topics they aggregate. For each topic category in our example, we chose the minimum feasible value of $N_i$ according to the constraints proved in Section II-B.
2) *Data topics may have moderate or no loss-tolerance requirements.* For event inference purpose, IoT data is often generated frequently, where intermittent losses of data may be compensated, for example, by estimation from previous or subsequent data. This is demonstrated by topic categories 4 and 5.
3) *Some data topics may require zero loss but have no latency requirement.* An example is topics used for logging purposes. This is demonstrated by topic category 1.

### A. Need for Data Replication

We say that the data in the Primary is *covered* if either the publisher still has a copy of it, the Primary has replicated it to the Backup, or both. Should the Primary crash, all data elements that are not covered would be lost. For data elements of topic $i$, let $x_i(t)$ be the largest number of consecutive data elements that are not covered at time $t$. The system can meet the loss-tolerance requirement if the Primary can ensure $x_i(t) \leq L_i$ at all times.

For small $N_i$, the value of $x_i(t)$ is dominated by factors including the inter-publishing time of data, the execution time of each edge computing task, and the choice of scheduling policy. For example, if the Primary can ensure that the processed data is always successfully delivered to its subscribers before the next data arrival of the same topic, then it means $x_i(t) \leq 1$ at all times. In this case, if $L_i \geq 1$ then there is no need for data replication.

In general, for each topic $i$, $x_i(t)$ changes over time as data of that topic arrives at and departs from the Primary, and as each data publisher may replace its previous data copy for a new data creation. Data replication is used to reduce $x_i(t)$ and is needed only if $x_i(t)$ will otherwise exceed $L_i$. Further, data replication must complete before $x_i(t)$ actually exceed $L_i$.

An ideal, clairvoyant data replication strategy is to just perform the needed replications at the right time and complete each replication action in time, so as to make $x_i(t) \leq L_i$ at all times. In practice, we may alternatively consider performing data replications regularly for some predetermined conditions, and in this way we may also specify a safe deadline for each data replication action. In the following, we suppose that for each data topic $i$, the Primary is set to perform data replication once every $M_i$ arrivals for some predetermined $M_i \geq 1$.

## B. *Deadline for Data Replication*

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

**Lemma 1.** *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. □

In the following, we derive bounds on the replication deadline in terms of applications' requirements and platform parameters. Let $\delta_{PP}$ be the latency from a publisher to the Primary, $\delta_{PrB}$ 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.

**Lemma 2.** *For data topic $i$, set parameter $M_i \geq 1$ and let $y = L_i - M_i$. To prevent more than $L_i$ consecutive data losses, the replication deadline must satisfy the following bound:*

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

*Proof.* We consider a sequence of arrivals of data topic $i$, as shown in Fig. 2. Subtracting $\delta_{PP}$ 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 interval $(t_{k-1}, t_k - T_{FO})$. Without loss of generality, we suppose that the crash happened immediately after the data arrival at time $t_{k-1}$, and

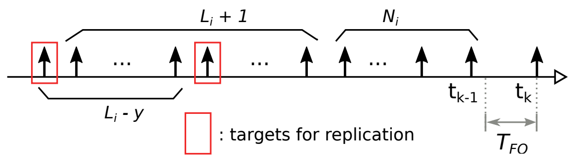


Fig. 2. An illustration for the proof of Lemma 2.

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 Fig. 2), and the replication must complete no later than time $t_{k-1}$. Therefore, the replication deadline must be smaller than or equal to $((k-1) - ((k-1) - (N_i - 1) - (y+2))T_i - \delta_{PP} - \delta_{PrB} = (N_i + y + 1)T_i - \delta_{PP} - \delta_{PrB}$.

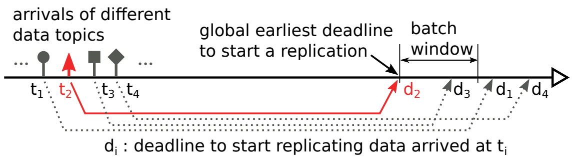
Now suppose that a crash happened at a time instant within $[t_k - T_{FO}, t_k]$. In this case, the publisher cannot detect the crash in time and would still send data that should have arrived at the Primary at time $t_k$, and that data will be lost. The publisher would send subsequent data to the Backup 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 2 implies that a shorter interval between replications (a smaller $M_i$) can permit a longer replication deadline. For example, suppose that $T_{FO} + \delta_{PP} + \delta_{PrB} = 15$ ms. For topic category 4 in Table I, setting $M_i = 3$ we have $D_i^r = 35$ ms, and setting $M_i = 1$ we have $D_i^r = 135$ ms. We note that setting $M_i = 1$ also gives the bound introduced in [5].

## III. The ARREC Architecture

A strategy suitable for IoT edge computing follows our analysis in the previous section, which we refer to as *adaptive data replication*. Conventionally, with replication there is a trade-off between keeping needed levels of data loss-tolerance and saving network bandwidth, where less data loss means more bandwidth consumption. Adaptive data replication can mitigate the trade-off by safely skipping many replication actions and batching the rest. This is made possible by leveraging the proved deadlines for replication.

Adaptive data replication includes four steps: (Step 1): *mark* data for replication for every $M_i$ arrivals for data of topic $i$; (Step 2): *wait* until a certain time point in the future;



arrivals of different data topics
global earliest 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 $t_i$

Fig. 3. Illustration of adaptive data replication.

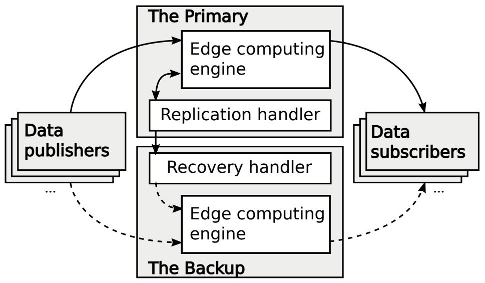


Fig. 4. ARREC System Architecture.

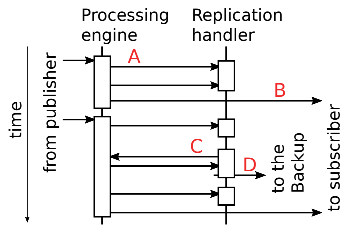


Fig. 5. Interactions between ARREC components.

(Step 3): *clear* the mark for data with which the edge computing task has completed and the result has been delivered to subscribers by then; (Step 4): *batch* marked data into a single data replication to the Backup.

Fig. 3 gives a timeline illustration. At Step 2, we set the time point to be the global earliest deadline to start a replication, where a deadline to start a replication is defined as the replication deadline minus the execution time of the replication action. The system does not perform data replication until then, and this allows the system to progress and thereby clear more marked data elements and save network bandwidth consumption. The replication action executes with the highest priority level and may preempt edge computing tasks.

We introduce a batch window for three purposes. Firstly, only data elements whose deadline to start a replication falls within the current batch window are included for replication in the current round, and this controls the size of a data batch. Secondly, the batch window bounds the frequency of replication actions: since all marked data elements in a given batch will be replicated in the current round, the global earliest deadline to start the next replication is lower-bounded by the window size. Thirdly, by using a batch window we allow more time for the system to progress and clear data marks consequently, and therefore the future batch windows may contain even fewer marked data elements for replication.

ARREC is designed to achieve efficient data replication while meeting applications' requirements. This is carried out via selectively grouping data for replication and replicate data batches in an adaptive and timely manner. The ARREC architecture is illustrated in Fig. 4. The system is pre-configured with the specifications from publishers and subscribers. In the Primary, upon each data arrival the *edge computing engine* creates an edge computing job. Before processing the data, the engine selectively creates a replication job, driven by the value $M_i$. The *replication handler* decides when to perform data replication. In the Backup, all replicated data elements are kept in a buffer, and upon fault recovery the *recovery handler* component then feeds those data elements to the edge computing engine. The engine schedules all jobs 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 paper we chose to use the earliest-deadline-first (EDF) scheduling policy as an example.

The proposed adaptive data replication strategy is carried out in the Primary via cooperation between the edge computing engine and the replication handler, as illustrated in Fig. 5. Upon each arrival of data topic $i$, the engine compares the value of $M_i$ with the number of data arrivals since the last replication. Once the number becomes larger than or equal to $M_i$, the engine will mark the arrival data, and the replication handler will update its timer for the next replication based on the marked data element's corresponding replication deadline (point A). In the meantime, the engine can perform needed edge computing (point B). When the timer expires, the replication handler will select all marked data elements for which the replication start time falls within a batch window (point C) and will replicate them in a batch (point D).

We implemented ARREC within the TAO real-time event service [4], where data elements are carried as events' payloads and publishers and subscribers are implemented as event suppliers and consumers. The edge computing engine, replication handler, and recovery handler are also implemented within the event channel. 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.

## IV. Empirical Evaluation

We evaluated two configurations of ARREC against two baseline configurations. The choice of $M_i$ for each data topic $i$ determines the deadline to start a replication, which in turn would affect both the global earliest deadline to start a replication and the number of data elements in each batch window. Since $M_i = 0$ means no replication, and $M_i > L_i$ has no guarantee to meet the loss-tolerance requirement of no more

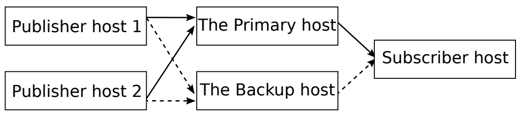


Fig. 6. Experimental topology.

than $L_i$ consecutive data losses, we evaluate configuration $M_i = 1$, denoted by `ARREC_all`, and configuration $M_i = L_i$, denoted by `ARREC_Li`, to cover the two extreme cases. The first baseline is `Retransmission-only`, in which the Primary performs no data replication at all and 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 case the Primary periodically replicates all data elements that has arrived since the most recent replication. `Periodic-50ms` is with replication period set to 50 ms, the shortest period of the topic specification in Table I. `Periodic-25ms` is with replication period set to 25 ms.

We used the topic specifications shown in Table I. For each topic we set the execution time of its processing load to be 0.1 ms. 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. The CPU utilization was between 145% to 200% of a single core's capacity, for all processing load in two CPU cores. We used one publisher to generate all data in topic category 5. 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 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 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 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 the next highest priority level 98, all with real-time scheduling policy `SCHED_FIFO`. We synchronized our local hosts via PTPd [1], an open source implementation of the PTP protocol [6]. 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

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

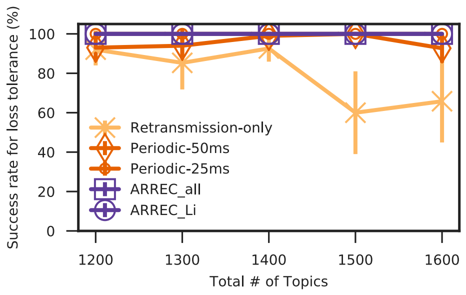


Fig. 7. Success rate for loss-tolerance requirement (%).

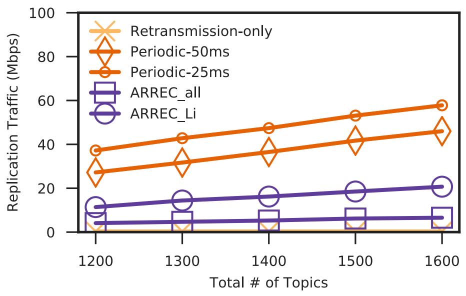


Fig. 8. Network bandwidth consumption for data replication.

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 over the latest 40 seconds.

## A. Data Loss-Tolerance Enforcement

Fig. 7 shows the success rate for meeting the loss-tolerance requirements, for topic category 1. 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.

## B. Network Bandwidth Consumption

Now we show that while meeting loss-tolerance requirements, ARREC may save network bandwidth consumption.

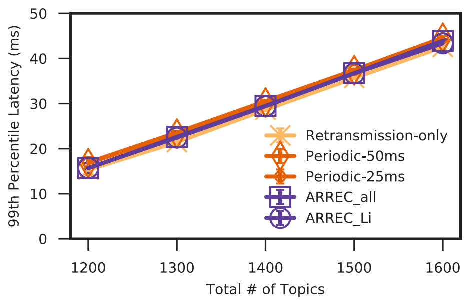


Fig. 9. 99th percentile latency.

The results are shown in Fig. 8. With a payload size of 512 bytes per data element, configuration `ARREC_all` may save 33–49 Mbps in replication traffic, which is about an 88% reduction, compared with configuration `Periodic-25ms`. Periodic replication consumed much network bandwidth because the traffic is close to a replica of normal data traffic passing through the primary IoT gateway, especially when the period of replication is short. In addition, 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 2) would allow more pending replications to be skipped. Configuration `ARREC_Li` outperformed 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.

## *C. Latency Performance*

We evaluated the latency performance before a fault occurs. Here we show the result for category 4, which has the shortest deadline (50 ms). Fig. 9 shows the 99th percentile latency, which represents a tail latency performance. Configuration `Retransmission-only` gave the baseline performance, and therefore we may see that the proposed adaptive data replication (ARREC_all and ARREC_Li) has no serious latency overhead. Overall, the 99th percentile latency all stayed within the 50 ms deadline.

Finally, we measured replication overhead in terms of CPU utilization of each configuration under increasing workload. Configuration `Retransmission-only` gives the baseline CPU utilization, 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.

## V. Related Work

A recent related work on Industrial IoT messaging architecture [5] shows that a system may meet applications' real-time and fault-tolerance requirements by proper scheduling of both message dispatch and message replication. We generalized that approach to include edge computing tasks, in which case messaging is considered as a task with trivial workload. Increase in workload per data item implies a higher likelihood of data losses when a fault occurs because, following Little's Law [7] and given the same arrival rate, there will be more data pending to be processed. Other related work includes a timely and reliable transport service in the Internet domain [8], and work on fault-tolerant task allocation to meet different recovery time requirements [9].

In IoT 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 while allowing enough progress of the other. In this viewpoint, studies on scheduling mixed criticality systems [10] offer related ideas. In this paper, we observe that in the IoT edge computing domain, data replication activities in some cases can be delayed and after sometime may be safely skipped. For lazy replication, a related idea appears in the *earliest deadline zero laxity* scheduling algorithm (EDZL) [11], [12], [13].

While in this paper we focus on IoT edge computing on resource-constraint platforms and at time scales of tens of milliseconds, we note that stream processing models [14], [15] perform well for cloud computing platforms. In particular, the micro-batch model (e.g., Apache Spark [14]) 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 (e.g., Apache Flink [15]) incurs less latency overhead when fault-free, but may take longer to recover from a failure. The Drizzle project [16] offers an empirical comparison of two models and introduced a choice of grouping micro-batches to bound the coordination overhead under fault-free operation.

## VI. Concluding Remarks

We presented adaptive data replication for IoT edge computing with platform constraints and application requirements. The empirical evaluation suggests that the ARREC architecture can efficiently meet the requirements for real-time reliable IoT edge computing. Using the proposed adaptive data replication, it is favorable to assign parameter $M_i$ with a smaller value, because it permits a longer replication deadline and many more data elements can be safely skipped from replication.

## Acknowledgments

This work was supported in part by MOST grant 109-2222-E-003-001-MY3, NSF grant 1514254, and ONR grant N000141612108.

## References

[1] 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.

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

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

[4] 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.

[5] C. Wang, C. Gill, and C. Lu, "Frame: Fault tolerant and real-time messaging for edge computing," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 976–985.

[6] 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.

[7] J. D. C. Little, "Proof for the queuing formula: L= $\lambda$w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[8] A. Babay, E. Wagner, M. Dinitz, and Y. 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.

[9] A. Bhat, S. Samii, and R. R. Rajkumar, "Recovery Time Considerations in Real-Time Systems Employing Software Fault Tolerance," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 23:1–23:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8980

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

[11] S. K. 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.

[12] T. P. Baker, M. Cirinei, and M. Bertogna, "Edzl scheduling analysis," *Real-Time Systems*, vol. 40, no. 3, pp. 264–289, Dec 2008. [Online]. Available: https://doi.org/10.1007/s11241-008-9061-6

[13] J. Lee and I. Shin, "Edzl schedulability analysis in real-time multicore scheduling," *IEEE Transactions on Software Engineering*, vol. 39, no. 7, pp. 910–916, 2013.

[14] 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.

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

[16] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 374–389.