In-ConcReTeS: **In**teractive **Con**sisten**c**y meets Distributed **Re**al-**T**im**e** **S**ystems, Again!

Arpan Gujarati, Ningfeng Yang
The University of British Columbia (UBC)
Vancouver, Canada
{arpanbg@cs,nxyang@ece}.ubc.ca

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)
Kaiserslautern, Germany
bbb@mpi-sws.org

***Abstract*—The problem of replica coordination is fundamental to building Byzantine fault-tolerant (BFT) distributed systems. Seminal BFT architectures for safety-critical real-time systems from the eighties and nineties relied on custom processors and networks, and are hence not readily usable today. Modern-day deployments on cloud platforms do not "scale down" to embedded platforms and are not designed around timeliness. Recent work on real-time BFT protocols focuses on simulations and reliability analyses. In short, there exist no easily programmable BFT libraries that can be conveniently retrofitted onto real-time applications with deadlines and that perform well on embedded platforms. We propose In-ConcReTeS, a BFT key-value store designed for building highly reliable control applications on commodity embedded platforms. At its core, In-ConcReTeS is a real-time friendly redesign and an efficient implementation of a BFT protocol used by seminal fault-tolerant architectures. We evaluated In-ConcReTeS using an inverted pendulum simulation and an automotive benchmark on a cluster of four Raspberry Pis connected over Ethernet. Our results show that, unlike Redis and etcd, In-ConcReTeS can repeatedly synchronize hundreds of key-value pairs, while tolerating faults, every tens of milliseconds.**

I. Introduction

Real-time software is central to autonomous systems, including autonomous vehicles such as airplanes, spacecraft and self-driving cars as well as safety-critical stationary systems such as nuclear reactors, power grids, and air traffic control. The software must remain *functionally correct*, both logically and temporally, for long periods of time. To this end, testing, verification, and certification efforts aim to ensure the absence of design flaws or timing-related bugs in safety-critical real-time software, whereas fault-tolerance mechanisms ensure correctness despite unpreventable hardware faults at runtime. This work focuses on the problem of timely fault tolerance in the presence of random, environmentally induced faults.

Provisioning redundant components (*replicas*) is a common approach to mitigating faults at runtime. However, in a system with redundant components, proper redundancy management is also necessary to ensure that **(i)** the system does not generate undesired outputs, **(ii)** the replicas remain synchronized over time, and **(iii)** if one or more replicas fail or diverge, the effects remain masked. Effective redundancy management is a complex problem, even more so in the context of real-time software because of strict timing requirements. For instance, with active replication, if *replica determinism* [57, 58] is desired, the fault-free replicated components have to deliver identical outputs in an identical order within a specified time interval.

The good news is that some of the most influential fault-tolerant architectures for safety-critical real-time systems, which were designed in the late eighties and early nineties, addressed this problem in depth. They proposed *hardware solutions* to ensure replica coordination in the presence of complex *Byzantine* faults. These refer to inconsistent broadcasts in distributed systems (*e.g.*, replica R_1 says to replica R_2, "my sensor reads 0.15" but to replica R_3 it says, "my sensor reads 35.33"). The understanding was that transient and permanent faults in the hardware could induce Byzantine faults in the worst case [23, 61, 62]. The result was a series of Byzantine fault-tolerant (BFT) architectures for extremely reliable real-time applications [31, 32, 34, 38, 42, 50, 68].

Today, few application domains use custom hardware, let alone for Byzantine fault tolerance. Prior solutions are hence not readily applicable. At the same time, real-time software has proliferated. For example, it is fairly easy to assemble a fully functioning drone using commodity off-the-shelf (COTS) hardware and open-source real-time software. However, there are currently no Byzantine fault tolerance solutions available that developers can retrofit onto existing real-time software when they desire to make their product highly reliable. Certainly, not everyone has the same resources as the automotive and avionics industries for research and development.

One option is to reuse one of the many cloud computing middleware that have surfaced in the last decade, which offer varying forms of fault tolerance [1, 2, 4, 11], or tap into research on general-purpose BFT software [9, 18, 19, 22, 39, 54, 67]. Unfortunately, since these systems are designed for best-effort, throughput-oriented applications, their performance suffers when deployed on resource-constrained embedded devices or when assessed in terms of predictability [45].

Recently, there have been some proposals from the real-time systems community for achieving Byzantine fault tolerance in real-time applications. Kozhaya et al. [40] and Gujarati et al. [29] discuss extensions to BFT protocols while considering bounded delays, whereas Roth and Haeberlen [60] discuss when Byzantine fault tolerance is overkill for real-time applications. Their primary contributions are stochastic analyses and simulations, which do not address practical deployment challenges. Li et al. [45] and Loveless et al. [47]

propose methods to reduce the latency of failure recovery in ZooKeeper [7] and distributed agreement in BFT state machine replication (respectively). However, Li et al.'s focus is low-latency edge computing, which is time-sensitive but not strictly real-time, and Loveless et al.'s proposal, unlike, say, ZooKeeper, cannot be easily retrofitted onto existing real-time software. Veríssimo et al. [65, 66] propose the *timely computing base* model for formally reasoning about applications with real-time requirements on environments with uncertain timeliness, but do not discuss or demonstrate its use for Byzantine fault tolerance. Their model requires programmers to refactor their applications, whereas we aim for a seamless integration.

This paper. We seek to develop a generic data store like ZooKeeper, Redis, or Cassandra that is easy to integrate in existing systems, but with a focus on periodic control applications, Byzantine fault tolerance, and real-time predictability on commodity embedded platforms.

Hence, we revisit the problem of Byzantine fault tolerance for real-time systems. Our aim is to answer the following questions. If a real-time application with a high degree of data exchange and communication is replicated, can the replicas be synchronized using a BFT protocol after every periodic iteration (like fault-tolerant architectures in the past)? What is a convenient synchronization API for application developers, and what are the system-level trade-offs involved?

Our main contributions are as follows (see Fig. 1 for an overview). First, we propose RT-EIGByz, a predictable real-time friendly implementation of a BFT distributed agreement protocol, specifically, an *interactive consistency* protocol [56], that was used in the design of seminal fault-tolerant architectures like MAFT [34] and SPIDER [50] (Section III). Second, we develop a distributed key-value store In-ConcReTeS, which interfaces with real-time applications using a time-aware API (to ensure data-age predictability), and transparently synchronizes any values written to the store across independent replicas using RT-EIGByz (Section IV). Third, we evaluate In-ConcReTeS against two widely used distributed datastores *Redis* [4] and *etcd* [1], using an inverted pendulum simulation and an automotive benchmark emulation (Section V). We show that In-ConcReTeS outperforms both baselines when interactive consistency is desired repeatedly, every 50 to 100 milliseconds, for hundreds of keys. We conclude with a summary of related work and our main findings (Sections VI and VII).

II. Background and Motivation

Our work on interactive consistency for real-time systems is motivated by prior work on fault-tolerant architectures [34, 50] for extremely reliable real-time systems, like flight control systems. We thus start by revisiting an example from one of these seminal architectures, MAFT [34], to understand the need for and the objectives of interactive consistency in detail.

Consider a flight-control system consisting of nine unique tasks responsible for processing the control laws for each of the three axes of rotation (roll, yaw, and pitch), system monitoring, and a validity check. These nine tasks (T_1 to T_9), their maximum execution times (E_1 to E_9, respectively), and their dependencies ($\Longrightarrow$) are illustrated in Fig. 2(a)(top).

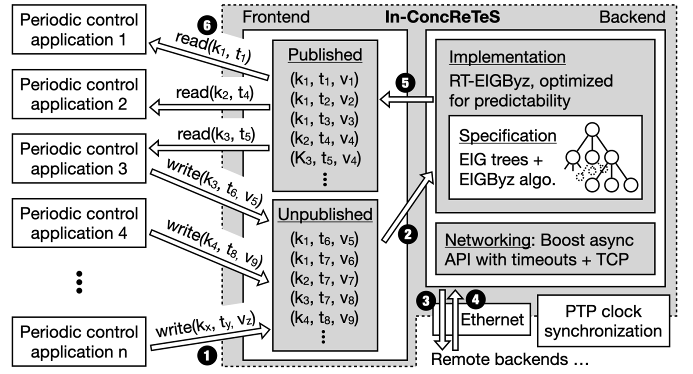


Fig. 1. Overview of our main contributions, RT-EIGByz and In-ConcReTeS. ① Application tasks periodically write critical state variables to In-ConcReTeS's *unpublished* store along with respective publishing times (which are absolute timestamps). RT-EIGByz periodically ② empties the unpublished datastore, ③ coordinates with replicas on remote nodes, ④ converges to a synchronized set of values (*i.e.*, which are agreed upon by all fault-free replicas), and finally ⑤ adds these to the *published* datastore. ⑥ Application tasks may access these values before the start of their next iteration.

MAFT assumes that any fault is possible, no matter how malicious, and therefore relies on modular redundancy and distributed agreement protocols to tolerate Byzantine faults. For instance, tasks T_1-T_8 in Fig. 2(a) have three active replicas each that function concurrently and synchronously. These are distributed among nodes $N_1 - N_6$. Task T_9 must be executed by all application processors and has six replicas.

At the end of each iteration, all replicas execute a distributed agreement protocol to agree upon replica-specific copies of application data. Each replica then locally runs a voting protocol to reduce the multiple copies of application data into a single copy. Fig. 2(b) illustrates the distributed agreement and voting processes for task T_1 and its replicas. Together, these add a "confirmation delay" of $\Delta^1_{sync} = \Delta^1_{agreement} + \Delta^1_{voting}$ time units between the finishing time of task T_1 and the starting time of its dependent tasks. For instance, in the schedule illustrated in Fig. 2(a)(bottom), the starting time of task T_8's replica on node N_4 is pushed back to $t = 7$ despite the application processor on N_4 becoming idle earlier (indicated by the box labeled ϕ between $t = 6$ and $t = 7$), because T_8 depends on task T_4 but T_4's data is not globally synchronized until $t = 7$. In fact, accounting for synchronization delays in complex real-time applications is one of the challenges our work addresses.

Notice that MAFT first executes what we refer to as an *interactive consistency* (IC) protocol (for distributed agreement). Its objective is to ensure that all replicas know about every other replica's local data, but *not* to select a representative candidate among these data items. The selection process instead is a separate post-processing step whose criteria may vary for each application. For example, in Fig. 2(b), a fault-tolerant midpoint voting strategy is used after the IC algorithm to select the final values. This distinction is important for real-time systems:

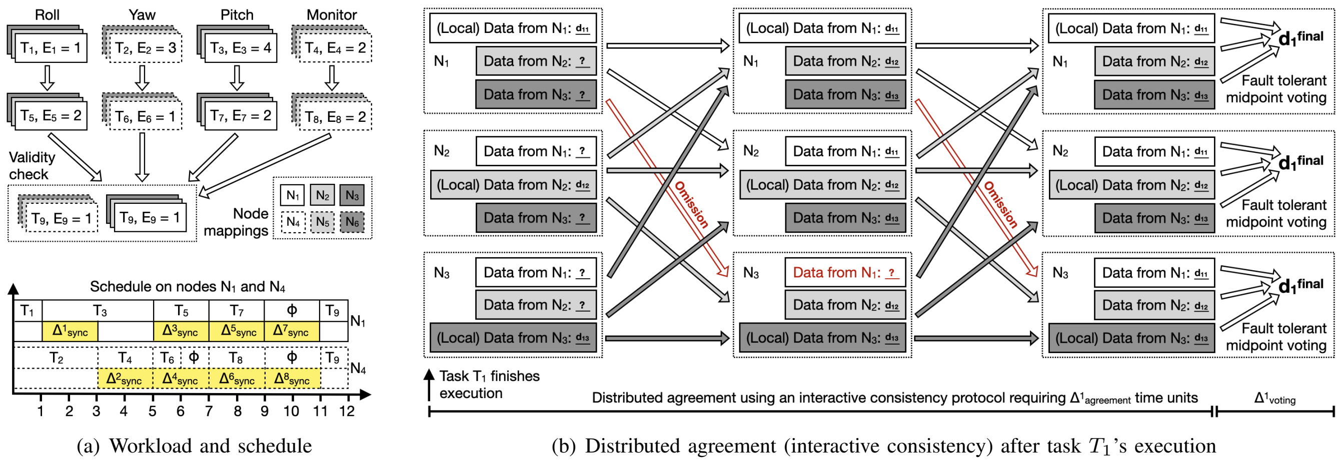


Fig. 2. Example from MAFT [34]. **(a)** (Top) Workload description with inter-task dependencies and replica to node assignments. (Bottom) Schedule of replicas of tasks $T_1 - T_9$ on nodes N_1 and N_4 along with the time spent on distributed agreement (synchronization) for each task. Since each node in MAFT contains an applications processor and an operations controller processor, the tasks and synchronization can proceed in parallel. ϕ denotes waiting time; for instance, since T_8 depends on T_4, it must wait for T_4's data to be synchronized before it can proceed. **(b)** Nodes $N_1 - N_3$ execute a two-round distributed agreement protocol to agree upon a final value of task T_1's data from replica-specific data items d_{11}, d_{12}, and d_{13}. The red arrow denotes an omission fault: messages from N_1 to N_3 are omitted in both rounds, but N_3 nonetheless finds out about data item d_{11} via N_2 in the second round.

application data commonly includes sensor inputs, control law parameters, and actuator commands, whose values can differ slightly across replicas owing to noise in data collection, arithmetic imprecision, and timing variations. Hence, separating the two steps allows MAFT to not require equality among non-faulty copies of data (a *de facto* assumption in most other forms of distributed agreement), and instead to use more general voting strategies for inexact answers [12].

Formal definition. The IC problem is defined as follows. Suppose there are N process replicas, p_0, p_1, …, p_{N-1}, who want to synchronize their data, $d_0, d_1, \ldots, d_{N-1}$ (respectively), with each other to remain consistent. Suppose each replica p_i also maintains a *decision vector* D_i whose k^{th} element denotes the data from replica p_k. Initially, of course, since each replica p_i is unaware of other processes' data items, only the ith element of D_i is updated, *i.e.*, $D_i[i] = d_i$, whereas all other elements are assumed to be empty, *i.e.*, $\forall k \neq i$, $D_i[k] = \bot$.

The goal of an IC protocol is to achieve three objectives. Each replica p_i must eventually update each value $D_i[k]$ to a value other than $\bot$ that it believes to be the data of replica r_k (*termination*). If replicas p_i and p_k are *correct* – *i.e.*, they remain fault-free during the IC protocol execution and latent faults that occurred in the past do not affect their current operations – then their decision vectors must be identical, *i.e.*, $D_i = D_k$ (*agreement*). Finally, if replicas p_i and p_k are correct, their decision vectors must also correctly reflect each other's data items, *i.e.*, $D_k[i] = d_i$ and $D_i[k] = d_k$ (*integrity*). For real-time systems, termination may additionally require that the IC protocol finishes its execution before its deadline.

III. Interactive Consistency

A. Algorithm Criteria

Assuring interactive consistency among distributed nodes in the presence of faults depends on several factors, including the degree and type of faults (*e.g.*, crash, corruption, or inconsistent broadcast), fault detection mechanisms in use (do checksums detect all "lies"?), network type (point-to-point or broadcast), and clock synchronization (are nodes tightly or loosely synchronized, or not synchronized at all?).

For instance, in the Controller Area Network (CAN) [20] with robust bit synchronization and fault detection mechanisms, atomic broadcast is guaranteed with very high probability [61], so interactive consistency can be achieved relatively easily using a single round of information exchange. On the other hand, if we consider asynchronous nodes, even a single crash failure can make distributed agreement impossible (FLP impossibility result) [26]. Roth and Haeberlen [60] provide a tutorial on how the complexity of the agreement protocol increases depending on the fault model and external factors.

We choose an interactive consistency protocol based on the following three criteria. First, we require Byzantine fault tolerance. Prior work suggests that hardware faults can manifest as Byzantine faults [23, 43]. Byzantine faults are often associated with malicious agents in distributed systems, which may lie. Since environmentally-induced faults occur randomly rather than due to malicious intent, one might argue in favor of using an IC protocol optimized for the weaker *authenticated Byzantine* fault model.[1] However, despite being a weaker threat model, distributed agreement for authenticated Byzantine faults is not necessarily more efficient because it requires the use of authenticators (such as checksums or message authentication codes), which can induce prohibitive overheads.

[1] Authenticated Byzantine faults are a subset of Byzantine faults [10] also resulting in inconsistent broadcasts, but where the faulty nodes cannot imperceptibly alter an authenticated message. For example, if replica R_1 says to R_2, *'my sensor reads 9.15'*, R_2 cannot lie to R_3, "R_1 says *'my sensor reads 0'*!", without getting caught. Since all messages are assumed to be cryptographically authenticated, R_3 can easily verify whether R_1 generated the message *'my sensor reads 0'*, and thereby find out that R_2 is faulty.

Our second criterion is that we require a protocol that is optimized for *synchronous networks*. Use of clock synchronization protocols such as the Precision Time Protocol (PTP) [25, 27], which can achieve sub-microsecond clock accuracy in a local area network, is common in distributed real-time systems. Also, the maximum time that any message is in transit can be upper-bounded analytically for real-time applications. For example, Loeser and Haertig [46] use network calculus [44] to upper-bound the transmission delay in Ethernet-based systems. Hence, the risk of complete network synchrony failure is sufficiently low, if not zero. An alternative is to consider a protocol that is optimized for *partially synchronous* networks [24, 65], wherein message delivery bounds may occasionally be violated in the presence of faults, *e.g.*, due to a fault-induced router reboot [28]. Unfortunately, algorithms designed for partial synchrony overcome temporary phases of asynchrony through additional rounds, but such an approach is counter-productive in real-time systems because data is time-sensitive, *i.e.*, delayed messages have zero utility. From a certification point of view, it is actually preferable to employ a simpler protocol that drops tardy messages rather than one that introduces additional complexity to ensure delayed message delivery in the wake of an unlikely loss of network synchrony.

The majority of BFT protocols is *leader-based*, *i.e.*, they mark one replica as primary and the remaining as secondaries. Even though secondary replicas may remain active throughout, their outputs are suppressed, unless the primary replica malfunctions. Leader-based protocols are vulnerable to performance degradation due to faulty leaders, which may become a reliability bottleneck in real-time applications [8]. For example, if the mechanism to switch the primary can take up to Δ_{switch} time units, but a critical data item in a high-frequency control loop needs to be synchronized among its replicas in less than Δ_{switch} time units, a primary failure can render the control loop unavailable for one or more iterations. Therefore, our third requirement is a *leader-less* BFT protocol.

The interactive consistency algorithm used in MAFT [56] satisfies all these criteria. Hence, we reuse it, while focusing on a specific version of that algorithm, EIGByz, by Borran and Schiper [13]. EIGByz is preferable because of its simplicity and because implementing it using real-time periodic tasks is straightforward, as observed by prior work [29].

B. Implementation

Readers may refer to EIGByz's pseudocode in the original paper by Borran and Schiper [13, Algorithm 1]. We explain our implementation in brief for a single instance of EIGByz, which solves a single instance of interactive consistency (as formally defined in the previous section). Henceforth, we refer to our implementation of EIGByz as RT-EIGByz.

Recall from Section I that real-time predictability is one of our primary objectives. That is, the latency of system operations must be bounded, if the workload is bounded. At the same time, our system must be able to repeatedly synchronize the data of periodic real-time tasks among replicas every tens of milliseconds. That is, replica coordination latencies must be extremely low, on a consistent basis.

To achieve these objectives, we need to address two main challenges not considered in prior work. EIGByz collects all its data in a complex Exponential Information Gathering (EIG) tree data structure. Given that we know exactly how and when specific nodes in the tree are accessed, how can RT-EIGByz lay out the tree in memory for extremely efficient reads and writes? How can we implement RT-EIGByz's networking layer such that it is timely and at the same time tolerates slow, or crashed replicas without compromising the progress of the algorithm? We address these challenges while walking through the high-level structure of the algorithm (which is derived from EIGByz). To disambiguate RT-EIGByz from EIGByz, we number the optimizations that are specific to our implementation.

Data structures. The number of children of each node in the EIG tree is equal to the number of replicas N participating in the IC protocol, and the number of levels in the tree is equal to one plus the number of rounds R in the IC protocol.

① In the case of real-time systems, both N and R will typically be determined at design time for predictability. Hence, RT-EIGByz ensures that each replica p_i statically allocates a tree object EIG_i at the start, which has enough memory to accommodate the intermediate data for all rounds, removing the need for dynamic memory allocation.

② RT-EIGByz lays out EIG trees as a set of one-dimensional arrays, which greatly reduces memory copying overheads when exchanging data between replicas. In particular, each tree EIG_i consists of $R+1$ one-dimensional arrays and the xth array $EIG_i[x]$ consists of N^x elements (tree nodes), such that $EIG_i[0][0]$ is the root node, and $EIG_i[x+1][z \cdot N^x + y]$ is the zth child of any node $EIG_i[x][y]$, for $z \in [0, N-1]$.

EIGByz requires each node in the tree to consist of a data item and a label. The root node of replica p_i's tree EIG_i consists of its local data d_i and an empty label. The subsequent nodes store data items that are received from other replicas via message exchanges, either directly or indirectly, and labels denoting the provenance of their data items (respectively). ③ However, RT-EIGByz does not require nodes to store label information. Instead, it uses the path to a node from its root as a proxy for its label. For example, suppose that

- p_a tells p_b *"my local data is d_a"*,
- p_b tells p_c *"p_a's local data is d_a"*, and
- p_c tells p_i *"p_b told me that p_a's local data is d_a"*.

Replica p_i stores d_a in the cth child of the bth child of the ath child of the root node of its tree EIG_i. Path $a \rightsquigarrow b \rightsquigarrow c$ is therefore sufficient to indicate the provenance of this data. Since a replica never ends up indirectly informing itself about its own data item, only cycle-free paths are considered *valid*, *i.e.*, paths such as $a \rightsquigarrow b \rightsquigarrow a$ are *invalid*.

Runtime protocol. EIGByz derives its simplicity from the EIG tree data structure. That is, given the aforementioned definitions, the runtime protocol is fairly straightforward.

In the *initialization* stage, each replica p_i stores its local data d_i in the root node of its tree EIG_i.

This is followed by the *communication* stage consisting of R rounds of information exchange. In each round $r \in [1, R]$, each replica p_i sends all elements in the rth level of its tree, *i.e.*, $EIG_i[r-1]$, to all replicas, including itself. Each replica p_i then waits for a pre-configured network delay, after which it expects to have received analogous messages from other replicas. After waiting, each p_i copies the entire message received from every replica p_j into the $(r+1)^{\text{th}}$ level of its tree with an offset of j slots, at $EIG_i[r][j \cdot N^{r-1}]$.

④ RT-EIGByz's array layout greatly simplifies the computations involved in the communication step: since all elements in $EIG_i[r-1]$ are stored in a contiguous region of memory, sending is efficient; and since messages received from peers are copied entirely (without breaking them down) to specific locations in $EIG_i[r]$, receiving is efficient.

Finally, each replica p_i executes the *reduction* stage, where it traverses the tree from level $x = R-1$ to $x = 1$ and updates the data item stored in each $EIG_i[x][y]$ to reflect the simple majority among all the valid children of this node. At the end of the reduction stage, each replica p_i's EIG tree is reduced to a two-level tree, whose second level represents its decision vector D_i. EIGByz, and by extension RT-EIGByz, guarantees that, if $N > 3f$ and $R = f+1$, where f denotes the number of faulty replicas, then interactive consistency is assured.

Networking. ⑤ As explained above, RT-EIGByz's design involves exchanging large chunks of the EIG tree among peers. UDP packet size is limited to only 65535 bytes, which may not be sufficient. We therefore use TCP as our transport layer protocol, which, unlike UDP, automatically fragments large payloads and ensures in-order delivery.

⑥ The flow control algorithm in TCP may interfere with our predictability goals, especially in the presence of faulty processes. For instance, if faulty receivers do not consume data sufficiently fast, TCP's flow control limits the rate at which senders may send data. The fault-induced rate limiting in turn affects the network latency incurred between another pair of correct proceses. We address this problem by setting a (configurable) timeout every time a message is being prepared for sending; after the timeout, any attempt to write the message to the TCP socket is aborted, allowing RT-EIGByz to progress despite slow or Byzantine receivers. Faulty senders are tolerated similarly using timeouts on the receiver side.

⑦ The networking layer is implemented using $boost.asio$'s [37] $async$ API, which allows us to send data via multiple TCP sockets in parallel without blocking or having to manage threads, thereby maximizing network utilization. To start a round of RT-EIGByz, each replica first adds a list of send and receive tasks to its $async$ task queue. The $async$ task queue executes tasks concurrently until all tasks are complete. Upon timeout, all incomplete tasks are cancelled, and the algorithm moves on to the next round.

In the next section, we describe how RT-EIGByz interfaces with our key-value store In-ConcReTeS, and how we scale up our implementation for synchronizing multiple data items.

IV. Key-Value Store

The example from MAFT [34] (Section II) consisted of nine tasks and their replicas provisioned over six independent nodes. Real-time applications today are at least an order of magnitude more complex, consisting of tens to hundreds of real-time tasks with lots of inter-task dependencies and a much larger state space, requiring multiple instances of RT-EIGByz or other such distributed agreement protocols, if interactive consistency is desired. The benchmark from Robert Bosch GmbH [41] that we use for evaluation is a good example.

Commodity platforms do not have a dedicated processor like the operations controller in MAFT attached to each node, which is hard-wired for implementing Byzantine fault tolerance. RT-EIGByz realizes this in software, but how can we scale it up to coordinate hundreds of data items? How do we interface it with real-time applications without putting the burden of initialization and redundancy management on application programmers? How do we ensure that RT-EIGByz finishes coordination in time for other dependent tasks?

A. Overview

To address the aforementioned challenges, we propose In-ConcReTeS, a middleware that provides a uniform and consistent global key-value store (KVS) abstraction to all application replicas and uses RT-EIGByz for interactive consistency.

As shown in Fig. 1 earlier, the real-time applications hosted on each node interact with a local instance of In-ConcReTeS, which consists of three components: a *frontend* that interfaces with applications, a *backend* that interfaces with In-ConcReTeS instances on other nodes, and a local *in-memory datastore*.

Real-time applications can be easily interfaced with In-ConcReTeS by replacing reads and writes of specific (global and critical) variables in their programs with the KVS read and write primitives exposed by In-ConcReTeS's frontend API. The data items written to the KVS can then be transparently synchronized by the backend, which may occur even at a later point in time based on the timing requirements of the data. As shown in Fig. 1, a real-time application need not be aware of the IC protocol, does not directly communicate with other replicas, and is not privy to the number of replicas that interface with In-ConcReTeS. As a result, the replication factor or the IC protocol can be transparently adjusted based on fault assumptions, without any changes to the application.

We discuss the key aspects of In-ConcReTeS's design and implementation details in the following sections.

B. Time-Aware Key-Value API

The utility of a data item in real-time applications typically decreases with time as the state of the physical world evolves. To simplify reasoning about maximum and minimum data ages, and to assist testing and validation efforts by reducing the space of possible executions, good design emphasizes *temporal determinism*. That is, "as fast as possible" or similar best-effort data propagation semantics can be severely detrimental in real-time systems because, under such semantics, implementation-level changes and random fluctuations in system state (*e.g.*, minor

changes in network traffic or CPU load) can affect the age of data consumed by controllers in unpredictable ways, which opens the door to race conditions, increases model uncertainty, and thus drives up the costs of rigorous validation.

In-ConcReTeS is hence designed to ensure temporal determinism, which also differentiates it from other contemporary distributed key-value stores. In-ConcReTeS requires programmers to provide with each read and write operation an *absolute* point in time that expresses data availability and freshness requirements, respectively. This requirement is embedded into In-ConcReTeS's frontend API, which consists of the usual read/write interface common to all KVS, but enhanced with an absolute *publishing time* parameter, as defined below.[2]

The $write(k, t_{pub}, v)$ interface takes key k, publishing time t_{pub}, and value v as parameters, and returns *success* if interactive consistency for key-value pair (k, v) can be assured by time t_{pub}. The key-value pair (k, v) written to the store becomes visible to applications only at its publishing time t_{pub}, that is, no read of k prior to time t_{pub} will return v. Hence, the value v is considered *published* at time t_{now} only if it is already globally synchronized across its replicas by then and if $t_{now} \geq t_{pub}$. The $read(k, t_{pub})$ interface takes key k and publishing time t_{pub} as parameters, and returns the latest key value that has been published no earlier than t_{pub}.

The write interface's time parameter decouples the time of data production from data availability in a predictable manner: if a data item is produced early, it will nonetheless not be consumed early, and if a data item is produced late, it will be rejected outright (rather than confusing consumers in difficult to anticipate ways). Conversely, the read interface's time parameter imposes a data freshness constraint, *i.e.*, it limits the publishing time of the oldest value that can be accepted by a successful read operation. The parameter, again, refers to an absolute point in time, which renders the result independent of the time when the operation is executed. In short, by expressing temporal constraints as absolute time instants, In-ConcReTeS both decouples the semantics of the provided KVS abstraction from implementation-level details and leverages the strong synchrony provided by the clock synchronization protocols already in common use in real-time systems today.

For performance reasons, both interfaces are designed to be non-blocking. Based on offline timing analysis or runtime admission control, the write interface assumes that the given value can be published in a timely manner in the absence of any faults. The write interface hence stores the given value and its publishing time and immediately returns *success*; coordination with other replicas occurs asynchronously. Similarly, the read interface immediately yields a matching published value or an error signaling the absence of one. Hence, both interfaces incur

[2]The publishing time parameter is inspired by the *logical execution time* paradigm [30, 36], which decouples the read and write time of global data used by a task from the actual execution time of the task, thereby enabling programmers to make definitive statements about when written data is available in the system and ready to be read by tasks on different nodes. The resulting data determinism eliminates execution-time dependent race conditions.

Algorithm 1 Controller interfaced with In-ConcReTeS

```
 1: procedure KVSBackedInvertedPendulum
 2:     time ← LastActivationAt()                          ▷ Compute freshness constraint
 3:     globalTarget ← KVS.read("target", time)            ▷ Get globally consistent
 4:     globalIntegral ← KVS.read("integral", time)        ▷ ... values of key
 5:     globalError ← KVS.read("error", time)              ▷ ... parameters
 6:     current ← GetSensorData()
 7:     error ← globalTarget − current
 8:     integral ← globalIntegral + error
 9:     derivative ← error − globalError
10:     force ← kp ∗ error + ki ∗ integral + kd ∗ derivative
11:     time ← timeOfNextActivation()                      ▷ Compute publishing time
12:     KVS.write("error", error, time)                    ▷ Globally synchronize state with
13:     KVS.write("integral", integral, time)              ▷ ... other replicas
14:     actuate(force)
15: end procedure
```

only a small overhead (that of access to the local datastore) on the application's execution time.

It is possible for a key to have multiple values attached to it, each with a different publishing time. That is, the pair (k, t_{pub}) denotes the unique key internally for In-ConcReTeS.

Algorithm 1 illustrates an example control loop interfaced with In-ConcReTeS. The controller replicas synchronize the error and integral variables (which they use across iterations, *i.e.*, the control loop's global state) using In-ConcReTeS. For clarity, error handling has been omitted. In summary, In-ConcReTeS 's KVS API is a minimal, yet expressive foundation for writing temporally-aware, time-deterministic applications.

C. Implementation

In-ConcReTeS's implementation must address two main challenges. First, it must efficiently incorporate RT-EIGByz to achieve interactive consistency for all (k, t_{pub}, v) tuples that are written to the datastore. Second, since the primary objective of In-ConcReTeS is to cater to control applications, it must ensure that interactive consistency is achieved in time (specifically, before the publishing time of each data item), and without affecting other real-time applications' timeliness.

Managing multiple data items. The naive solution is to start a new instance of RT-EIGByz for each (k, t_{pub}, v) tuple that is written to In-ConcReTeS. This approach can be inefficient because it requires additional processing (multiplexing and use of multiple sockets) every time RT-EIGByz messages need to be sent over the network. Instead, multiple data items that are written by the same control application and which, typically, have the same publishing time can be batched together and synchronized using a single instance of RT-EIGByz, *e.g.*, the *error* and *integral* keys in Algorithm 1. In fact, since RT-EIGByz relies on TCP, its networking layer is not limited by packet sizes; we can thus extend this approach further to batch multiple data items that may be written to In-ConcReTeS by different control applications.

To accomplish batching, we modify RT-EIGByz to work for a predetermined number of batched keys, as well as predetermined key and value sizes. That is, since real-time workloads tend to be deterministic, we assume that upper bounds on the number of keys and their sizes can be estimated *a priori*, and

RT-EIGByz data structures (EIG tree nodes) are then allocated more space upfront proportional to these upper bounds.

We introduce two new local datastores (simple C++ maps), *published* and *unpublished*, which respectively act as a source and sink for RT-EIGByz. All write requests are first inserted into the unpublished datastore. During the initialization stage, each instance of RT-EIGByz batches all tuples in this datastore and copies the batch into the root of its EIG tree. During the following communication stages, RT-EIGByz exchanges batched values between nodes, which is efficient since large contiguous regions of memory are sent and received over the network. RT-EIGByz then reduces the large EIG tree into a decision vector, which contains the batches proposed by different nodes. Finally, the decision vector is un-batched before reducing multiple node-specific copies corresponding to each (k, t_{pub}) pair into a single copy using application-specific criteria, such as median or simple majority.

Managing time. Batching is limited by the publishing times of keys, some being more urgent than others. RT-EIGByz over batched data must therefore complete before the earliest publishing time. We ensure this by executing RT-EIGByz as a recurring task, whose time period is determined as a function of the application frequencies, their publishing times, their worst-case execution times (WCETs), and RT-EIGByz's WCET. Since this approach is workload specific, we discuss it in our evaluation for each benchmark separately.

Both application and RT-EIGByz tasks are realized as single-threaded real-time periodic tasks, and implemented in C++ using Linux's *clock_gettime* and *clock_nanosleep* APIs [14]. These tasks must be deployed on the same core, mimicking a uniprocessor scenario. Access to shared maps, *i.e.*, published and unpublished datastores, by the application tasks during reads and writes remains safe if the tasks do not interleave, which is ensured through appropriate priority and offset assignment.[3] To exploit the full capacity of a multicore platform, separate instances of In-ConcReTeS can be provisioned on each core, and application tasks may interface with their core-local instance; safety is ensured through partitioned scheduling.

Typically, a fault-tolerant N-modular redundant (NMR) datastore is realized by linking instances from N different nodes. For example, four dual-core platforms (nodes) can be abstracted as two NMR datastores with $N = 4$; the first datastore may consist of instances local to core 0 on each node, and the second may consist of instances local to core 1 on each node. Placing the N modules of an NMR datastore on different cores of the same node is not recommended.

V. EVALUATION

We answer three questions: **(i)** How does In-ConcReTeS compare against general-purpose fault-tolerant key-value stores like Redis and etcd, which are fast but not optimized for real-time predictability? **(ii)** Does In-ConcReTeS perform well even in the presence of faults or is there a performance penalty? **(iii)** How does In-ConcReTeS scale with the number of keys? We answer (i) and (ii) using an inverted pendulum simulation [52] and (iii) using emulated workloads based on a real-world automotive benchmark from Robert Bosch GmbH [41]. We first describe our evaluation platform and experimental setup, and the considered baseline configurations.

Setup. All experiments were performed on a cluster of four Raspberry Pi 4 Model B units [3], each equipped with a 1.5GHz Cortex-A72 quad-core processor and 4GB of memory. The four Pis were connected over IEEE 802.3ab Gigabit Ethernet using a 1Gbps Ethernet connection. The Pis were running Raspbian GNU/Linux 10. We use all four Pis, *i.e.*, N-modular redundancy (NMR) with $N = 4$, unless specified otherwise.

We installed $ptp4l$ on each Pi for clock synchronization using software timestamping, and for predictability configured it to run with the highest real-time priority and use a separate virtual network interface. We extracted the clock skew data during idle times and during experiments from $ptp4l.service$ logs using $journalctl$ [33]. When the Pis are idle, we observed a clock skew of around $20\mu s$. However, when the Pis are running a network-heavy workload, *e.g.*, In-ConcReTeS, we noticed more noise in clock synchronization and the skew occasionally reaching around $200\mu s$. While such transient disturbances are unavoidable in our current setup, we expect that hardware timestamping along with TSN-compatible Ethernet switches could significantly improve clock synchrony.

Baselines. We consider two baselines, Redis [4] and etcd [1]. Redis is an in-memory high-performance datastore with support for a variety of data formats. Redis is a single-threaded server written in C and is not designed to benefit from multiple CPU cores [5]. Instead, as with In-ConcReTeS, users are expected to deploy multiple Redis instances on several cores for scalability. Hence, although Redis does not offer Byzantine fault tolerance, it is a good baseline for evaluating In-ConcReTeS's performance. In contrast, etcd is a strongly consistent, distributed key-value store written in Go, which uses the Raft Consensus Algorithm [54] for fault tolerance. etcd is therefore closer to In-ConcReTeS in terms of its fault tolerance properties (even though it is leader-based).

We configured Redis as a collection of twelve standalone instances. Three instances were provisioned on every node, and each instance was pinned to a unique core. We left the fourth core on each Pi idle to reserve some bandwidth for kernel and networking utilities. Although different Redis instances can be tied together in a replicated setup, we left them as standalone instances since Redis does not ensure strong consistency. Instead, for fault tolerance, we introduced a *read* API that queries four separate Redis instances (on separate nodes) every time, reads a copy of data from each instance (which is analogous to constructing a "decision vector" in RT-EIGByz), and then using an application-specific function (*e.g.*, simple median for sensor values) fuses these into a single copy that is returned back to the application. In contrast, the

[3]The problem of interleaving does not arise if the application and coordination tasks are implemented in a single thread. Conceptually, such an implementation is feasible. Nonetheless, we use separate threads since we expect these tasks to be programmed separately. Our design also allows extension to more complex scheduling models in the future.

write API interacts with only the core-local Redis instance. We use the *sorted sets* data structure in Redis where every string value is stored along with a floating-point *score*, and clients can retrieve values using keys as well as range-based queries on scores. We store publishing times as scores.

Unlike Redis, etcd ensures strong consistency, therefore its configuration was relatively simpler. We provisioned etcd as a distributed cluster with one instance on every node. Given key k, publishing time t_{pub}, and value v, our write API writes the entire $(\textit{node ID}, k, t_{pub}, v)$ tuple to the local etcd instance. The write completes only after etcd has propagated the tuple to all nodes. During read, our API queries only the local instance but for multiple tuples of the form $(\textit{node ID}, k, t_{pub})$, each with a different node ID. The read API thus collects multiple copies, which are then fused into a single value, as in Redis.

We developed core-local proxies for etcd and Redis, both of which are extensions of our real-time periodic task implementation. The proxies are accessed by the application tasks for reads and writes. Other than that, in the case of etcd, once the proxies are initialized at the beginning, nothing needs to be done during periodic invocations. While etcd automatically garbage-collects old values, in the case of Redis, our core-local proxies periodically issue garbage collection queries.

A. Inverted Pendulum Simulation

In the first set of experiments, we compared In-ConcReTeS with Redis and etcd using an inverted pendulum C++ simulation by Morgado [52], which relies on numerical analysis to simulate the inverted pendulum physics and a PID controller (*e.g.*, see Algorithm 1) to control the simulated inverted pendulum. It consists of a periodic task, IvPSim, whose computation is split into three parts. The first part simulates reading a sensor by invoking the simulator to compute new values of parameters that uniquely determine the inverted pendulum's position and velocity in a two-dimensional space. The second part executes the PID controller, which computes the horizontal force that needs to be applied to the cart to ensure a target angular position of zero degrees. The third part "actuates" the cart, *i.e.*, updates the *current force* in the simulation.

Our objective is to run the simulation across multiple replicas in a synchronized fashion despite faults. Hence, periodic transitions in the simulation state must be identical across replicas (this is motivated from the fact that, in practice, a single physical control system may be controlled by replicated software controllers for reliability). Secondly, the PID controller outputs on each replica must be aggregated and reduced to a single correct output, which is then used for actuation (either by a designated primary or a dedicated hardware unit).

Using In-ConcReTeS, it is easy to ensure these requirements. In every iteration, the PID controller reads its global parameters *integral* and *error* from In-ConcReTeS at the start of the iteration, and writes their updated values back to the datastore in the end of the iteration. Similarly, the simulator reads (writes) all its state variables from (to) In-ConcReTeS at the start (in the end) of every iteration (respectively). Altogether, a single instance of IvPSim reads and writes 19 floating point values in every iteration, including parameters like *angular velocity*, *angular position*, *cart velocity*, and *cart position*, *etc.*

Data copies from different replicas are fused by computing the median to account for noise in the control system data.[4] When working with four replicas, we require for fault tolerance that at least three out of four copies are available for computing the median; otherwise, the fuse function returns an error. Recall from Section II that interactive consistency allows the use of such application-specific fuse functions.

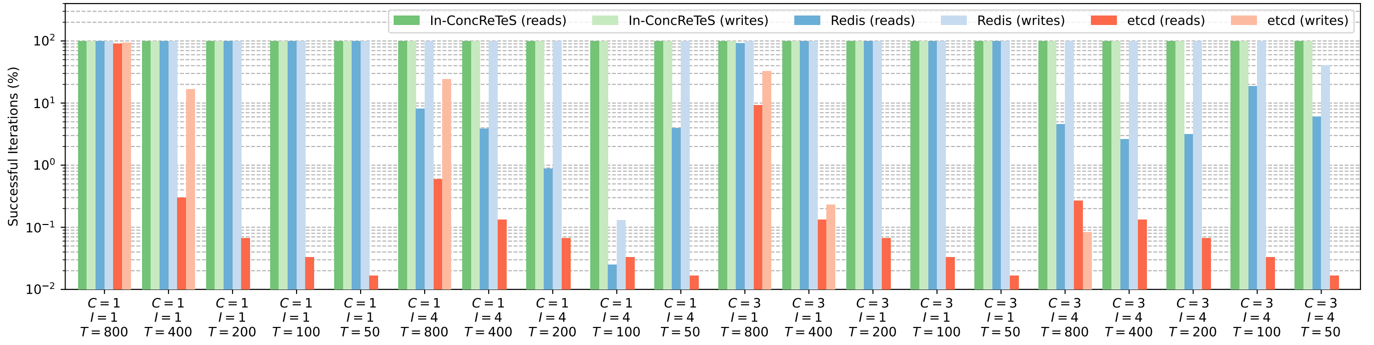
Experiment setup. Each experiment is configured using four parameters. N denotes the number of nodes, C denotes the number of cores used (in other words, the number of NMR setups we run in parallel), I denotes the number of distinct IvPSim tasks per core, and T denotes the time period of each IvPSim task. For example, suppose $N = 4$, $C = 3$, $I = 4$, and $T = 100ms$. In this configuration, cores 0, 1, and 2 on each Pi run four different IvPSim tasks per core, resulting in 48 IvPSim tasks altogether: $\{T_{ivp,n,c,i} \mid n \in [0,3], c \in [0,2], i \in [0,3]\}$. Each quadruple $\{T_{ivp,0,c,i},\ T_{ivp,1,c,i},\ T_{ivp,2,c,i},\ T_{ivp,3,c,i}\}$ consists of replicated tasks forming an NMR setup. Each core also hosts a local KVS instance, resulting in 12 more periodic tasks: $\{T_{kvs,n,c} \mid n \in [0,3], c \in [0,2]\}$. KVS tasks coordinate data written by core-local application tasks.

In case of In-ConcReTeS, since reads and writes are non-blocking, we observed based on initial profiling that the WCETs of IvPSim tasks were always under $\omega_{ivp} = 0.5ms$. We therefore considered an upper bound of $\omega_{ivp}^{ub} = 1ms$ on their WCET, and provisioned each task $T_{ivp,n,c,i}$ with an offset of $\omega_{ivp}^{ub} \times i$ and each task $T_{kvs,n,c}$ with an offset of $\omega_{ivp}^{ub} \times I$. Furthermore, if multiple cores were used, we introduced different starting offsets for each core, so that core-specific network I/O could be spread out in time (this benefits all baselines).

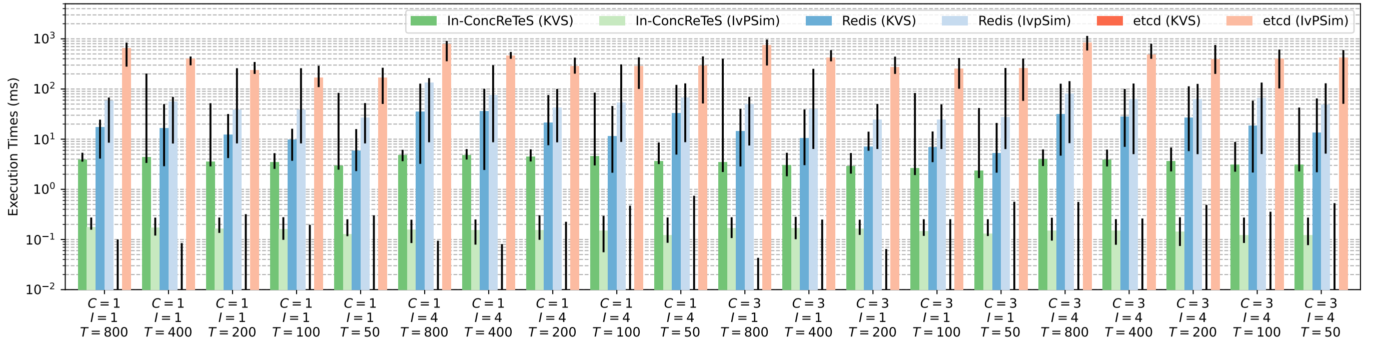
Experiment A1. We considered 20 configurations: a uniprocessor setup with $C = 1$ or a multiprocessor setup with $C = 3$ (one core is idle for background tasks), either $I = 1$ or $I = 4$ IvPSim tasks per core (the second scenario generates four times more workload for the KVS), and time period as $T \in \{800ms, 400ms, 200ms, 100ms, 50ms\}$. We let each configuration run for five minutes, and measured **(i)** the percentages of IvPSim iterations during which all reads succeeded and all writes succeeded (respectively); and **(ii)** the best-case, average-case, and worst-case execution times (BCETs, ACETs, and WCETs) across all IvPSim and KVS tasks (respectively). Results are illustrated in Figs. 3(a) and 3(b).

There are two key takeaways from this experiment. First, in terms of success rates, In-ConcReTeS scores a hundred in every configuration, and outperforms Redis and etcd, as shown in Fig. 3(a). Redis matches In-ConcReTeS when $I = 1$, but starts to underperform every time the workload per core is increased to $I = 4$, since the number of keys also increases proportionally (notice the darker blue bars for $I = 4$, which have less than 10% success rates). etcd, on the other hand,

[4] Simple majority does not apply as all values are floats and hence expected to be unique even in fault-free scenarios. We are yet to evaluate simple majority after rounding the floats based on a pre-configured accuracy threshold.



(a) Number of successful iterations with In-ConcReTeS, Redis, or etcd as the key-value store (KVS). The six bars in each group, from left to right, correpond to In-ConcReTeS (reads), In-ConcReTeS (writes), Redis (reads), Redis (writes), etcd (reads), and etcd (writes), respectively.



(b) Average execution times of different periodic tasks with In-ConcReTeS, Redis, or etcd as the key-value store (KVS). The six bars in each group, from left to right, correpond to In-ConcReTeS (KVS), In-ConcReTeS (IvPSim), Redis (KVS), Redis (IvPSim), etcd (KVS), and etcd (IvPSim), respectively.

Fig. 3. IvPSim Experiment A1 results. Both (a) and (b) consist of 120 bars each. These 120 bars are divided into 20 groups of six bars. The 20 groups correspond to the 20 different configurations that we considered in experiment A1; the configuration relating to each group is specified under its bars. Missing etcd bars in (a) denote a success rate of zero. Error bars in (b) depict the observed BCETs and WCETs.

almost always underperforms (notice the red bars). Since etcd ensures strong consistency at write time, its success rate for writes is often zero (notice the missing bars); this implies that in each IvPSim iteration, since each write blocks until etcd has synchronized it globally, some of the writes do not manage to even start before their publishing times. Second, we realize from the execution times shown in Fig. 3(b) that KVS and IvPSim tasks in In-ConcReTeS incur reasonably low overheads (under $10ms$) irrespective of the configuration, which highlights In-ConcReTeS's focus on real-time predictability.

We further investigated if In-ConcReTeS can support smaller periods. We evaluated with $T \in \{40ms, 30ms, 20ms, 10ms\}$, $C = 3$ and $I = 4$. For $T = 40ms$ and $T = 30ms$, the results remained the same. However, the success rates started dropping at $T = 20ms$, and reached zero for $T = 10ms$.

Overall, the experiment highlights that In-ConcReTeS's design choices allow it to ensure interactive consistency among periodic tasks at high frequencies, whereas commodity datastores like Redis and etcd are not suitable for such applications, because they are optimized for metrics like peak throughput and average response times.

Experiment A2. We injected omission and corruption faults and studied their effects on the execution times of IvPSim and KVS tasks. We considered a single configuration in this experiment, $N = 4$, $C = 1$, $I = 4$, and $T = 50ms$, and conducted five experiment runs of 100s (2000 task iterations) each. The first run executes fault-free. In the second run, the first node *fail-stops* (*i.e.*, it does not send or receive any messages). In the third run, both first and second nodes fail-stop. In the fourth run, no node fail-stops, but the first node *lies* by presenting false data. Finally, in the fifth run, both first and second nodes lie. The average execution times of all KVS and IvPSim task activations on nodes 3 and 4 (which remain fault-free throughout these five runs) are shown in Fig. 4. The grey background denotes unsuccessful experiment runs.

We make three observations. First, as already noticed in the previous experiment, the first run demonstrates the real-time predictability of In-ConcReTeS. When there are no faults, the KVS task execution times are small as well as extremely predictable, largely due to our RT-EIGByz implementation, which is optimized for periodic real-time workloads. Second, since our fault-tolerant median requires at least three out of four copies to compute the median, the success rates remain at 100 percent during the second and fourth runs when only one node is faulty, but drop down to zero when more than one node is faulty (note that a different choice of fuse function could

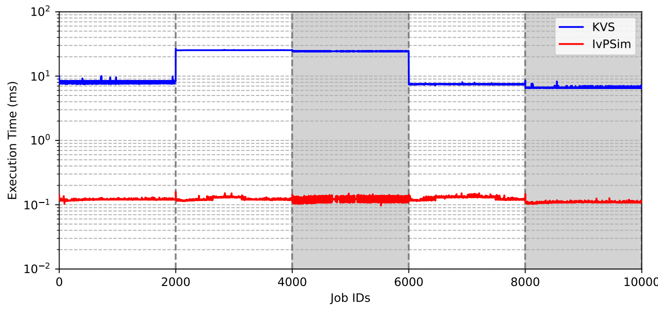


Fig. 4. IvPSim Experiment A2 results.

ensure liveness in case of two fail-stop failures). Third, the average execution time of each KVS task increases when faulty nodes are unavailable, since RT-EIGByz must timeout during every round before proceeding further. The timeout latency must be accounted for when computing the WCET of the KVS tasks. In contrast, when nodes lie, execution times are not affected. Overall, the results demonstrate that In-ConcReTeS remains predictable while being fault-tolerant.

B. Experiments with the Bosch Benchmark

In the second part, we evaluated In-ConcReTeS by emulating real-world automotive workloads based on the benchmark presented by Kramer et al. [41] from Robert Rosch GmbH. Such automotive workloads consist of a set of periodic functions, *runnables*, that communicate with each other and with the rest of the system using *messages* and *labels*. They are therefore quite apt to evaluate research on distributed real-time systems. We start by describing our workload generator.

Workload generator. To be free of any intellectual property concerns, the Bosch benchmark is anonymized. It does not specify the definitions of runnables, labels, and messages involved, nor does it specify their exact parameters, such as the execution time profile and the period of each runnable, or the size of each label. Instead, the benchmark reports aggregate statistics summarizing how different time periods are distributed among runnables, distributions of label sizes and communication patterns, *etc.* We developed a parameter generation tool that generates workload parameters adhering to the statistical distributions in the Bosch benchmark, and then emulated workloads based on these parameters.

The first step is to scale the number of runnables specified in the benchmark by a (configurable) scale factor, f_{scale}, generate as many runnable instances, and assign to each a period in $\{1, 2, 5, 10, 50, 100, 200, 1000\}$ (*ms*) based on the given distribution (we ignore angle-synchronous periods, which vary according to the rotation speed of the crankshaft).

The next step is to generate an execution time profile for each runnable, consisting of its best-case, worst-case, and average-case execution times (BCETs, WCETs, and ACETs). The benchmark contains multiple constraints in the form of **(a)** the minimum, maximum, and average ACET $\alpha^-_{p_i}$, $\alpha^+_{p_i}$, and $\tilde{\alpha}_{p_i}$ (respectively), **(b)** the minimum and maximum BCET/ACET ratio g^-_{β,p_i} and g^+_{β,p_i} (respectively), and **(c)** the minimum and maximum WCET/ACET ratio g^-_{ω,p_i} and g^+_{ω,p_i} (respectively) among all runnables with period p_i, which make this step challenging. To ensure that all constraints are satisfied, we first use a four parameter *beta* distribution for modeling the ACETs based on (a),[5] and then model the BCETs and WCETs based on the constraints in (b) and (c), respectively. Runnables with the same period are mapped to a single task.

As with runnables, the generator also scales the number of labels specified in the benchmark by f_{scale}. Each label is assigned a size between 1 and 16 bytes based on the distribution in the benchmark (more than 99% of labels in the benchmark are 16 bytes or shorter; we therefore ignore larger sizes). Each label can be either read-only, write-only, or read-write, and each read-write label can be either inter-task or intra-task.[6] We used the distributions provided in the benchmark to classify each label accordingly and map it to a reader and/or a writer.

Finally, since In-ConcReTeS cannot support all periods, as shown in the previous set of experiments, we prune the workload such that all runnables (and associated labels) with periods smaller than a given P_{min} are pruned.

Experiment setup. We followed the procedure explained above to generate hundreds of workloads, using a scale factor between $f_{scale} = 0.01$ and $f_{scale} = 0.32$ (we scale down significantly since In-ConcReTeS is expected to consume a large fraction of CPU). We emulated each workload using periodic tasks with appropriate parameters and interfaced each task with In-ConcReTeS's read and write APIs for label access. Based on initial profiling, we observed that when RT-EIGByz is initialized to support hundreds of labels (keys), its ACET frequently exceeds 10 ms. Therefore, we generated two sets of workloads, where we set a lower bound of $P_{min} = 50\,ms$ and $P_{min} = 100\,ms$ on the time periods, respectively.

Since the workload parameters were randomly generated, the number of labels varied significantly between 100 and 5000 across instances, and not every instance consisted of runnables with all possible time periods. One of our objectives is to evaluate In-ConcReTeS when application tasks have varying time periods. Hence, we shortlisted 8 workloads with $P_{min} = 50\,ms$ and 19 workloads with $P_{min} = 100\,ms$ for evaluation, each of which includes runnables with all possible time periods greater than or equal to P_{min}. The shortlisted workloads with $P_{min} = 50\,ms$ consist of up to 1000 labels, whereas workloads with $P_{min} = 100\,ms$ consist of up to 2000 labels.

[5]The standardized beta distribution has two shape parameters a and b and is defined over range $[0, 1]$. The *generalized beta distribution* has two additional parameters c and d and is defined over range $[c, d]$ [53, Section VII]. The latter is suitable because the ACET distribution for runnables with period p_i must be defined over a custom range $[\alpha^-_{p_i}, \alpha^+_{p_i}]$. We configure $a = 2$, which results in an asymmetric bell curve, and $b = a \times (\alpha^+_{p_i} - \tilde{\alpha}_{p_i})/(\tilde{\alpha}_{p_i} - \alpha^-_{p_i})$, which ensures that the mean value of the distribution is indeed $\tilde{\alpha}_{p_i}$. We do not configure $a = 1$ since it results in $\alpha^-_{p_i}$ being the most likely execution time (not a bell curve), and we also do not configure $a > 2$ since it results in narrower bell curves with extremely predictable execution times.

[6]Inter-task labels are written and read by runnables in different tasks, whereas intra-task labels are written and read by runnables in the same task.

TABLE I
WORKLOADS WITH A MINIMUM TIME PERIOD OF $P_{min} = 50\,ms$

Labels	Read success %				KVS latency (ms)	
	Pi 1	Pi 2	Pi 3	Pi 4	ACET	WCET
270	100.00	100.00	100.00	100.00	8.34	28.97
363	100.00	100.00	100.00	100.00	12.21	23.64
476	100.00	100.00	100.00	100.00	14.87	32.88
573	100.00	100.00	100.00	99.85	17.50	35.36
689	100.00	100.00	100.00	100.00	17.89	35.62
744	100.00	99.97	99.96	100.00	19.72	35.86
849	99.81	99.91	99.74	99.98	20.43	36.59
905	98.71	99.56	44.82	99.07	23.49	41.82

TABLE II
WORKLOADS WITH A MINIMUM TIME PERIOD OF $P_{min} = 100\,ms$

Labels	Read success %				KVS latency (ms)	
	Pi 1	Pi 2	Pi 3	Pi 4	ACET	WCET
190	99.65	100.00	100.00	100.00	11.77	49.89
231	99.92	100.00	99.71	100.00	28.10	51.77
395	97.17	100.00	99.98	100.00	28.86	55.11
482	100.00	100.00	99.96	100.00	30.90	57.70
572	99.88	100.00	100.00	100.00	34.30	60.53
663	99.83	100.00	100.00	100.00	36.83	62.79
774	100.00	100.00	100.00	100.00	34.92	64.47
835	100.00	100.00	100.00	100.00	28.39	65.24
979	99.83	100.00	99.25	99.83	38.23	67.17
1001	99.90	100.00	99.19	100.00	36.37	68.69
1157	100.00	100.00	100.00	100.00	37.72	68.31
1218	100.00	100.00	100.00	100.00	36.15	66.66
1320	100.00	99.81	99.92	100.00	42.53	69.61
1420	100.00	99.10	99.79	100.00	45.68	71.84
1531	100.00	97.67	99.77	100.00	47.66	73.94
1682	100.00	100.00	100.00	100.00	44.47	73.75
1712	100.00	100.00	100.00	100.00	49.64	78.64
1810	100.00	98.58	100.00	100.00	48.43	81.37
1986	100.00	100.00	100.00	99.94	47.13	81.21

For each workload, we pre-configured RT-EIGByz to work with an appropriate number of labels. Since all runnables with the same time period are mapped to the same periodic task, each configuration consisted of up to four application tasks with time periods 50 ms, 100 ms, 200 ms, and 1000 ms and one KVS task per node. The tasks were assigned offsets based on an inital profiling of their WCETs (as in the IvPSim experiments), and were provisioned on a single core on their respective nodes.

Experiment results. Tables I and II reports the number of labels, percentages of iterations during which all reads were successful (reported separately for each node) and the average ACET and the maximum WCET across all KVS tasks on all nodes, for workloads with $P_{min} = 50\,ms$ and $P_{min} = 100\,ms$ (respectively). The success rates for writes was always 100%, and hence not shown in the tables.

Unlike the IvPSim experiment, reads were not always successful. Some nodes occasionally failed to read a few key-value pairs from In-ConcReTeS. This happens when the local RT-EIGByz instance fails to publish a write, because it did not receive sufficient communication from other nodes (which is necessary for distributed agreement) before timeout. A read failure does not necessarily imply an application failure. In the case of a failed KVS read, the application may fall back on a fresh value albeit from the unpublished datastore (*i.e.*, which is equivalent to its local state), and continue. Most control systems can easily tolerate a few intermittent failed iterations, where they rely on stale sensor values [49, 55].

We attribute the read failures in this experiment to the occasionally high clock skews resulting from absence of hardware timestamping and from interference between PTP and In-ConcReTeS's network I/O. Unlike the IvPSim experiment, RT-EIGByz in this experiment is configured for larger labels and for many more keys, resulting in a network-heavy workload. Furthermore, in workloads with $P_{min} = 50\,ms$, as we approach 1000 labels, there is not enough time for the application and KVS tasks to complete within a 50 ms time period. These factors may cause timeouts during RT-EIGByz's execution, which limits In-ConcReTeS's ability to synchronize data across all nodes on time, during every single iteration.

C. Discussion and Limitations

In-ConcReTeS's design, specifically, mapping the BFT protocol to a set of real-time tasks, enables checking either statically or at admission time whether writes by application tasks can be propagated to all replicas in time, *i.e.*, whether each $write(k, t_{pub}, v)$ is published by time t_{pub}. Such an analysis would rely on the task parameters and an upper bound on the network latency; and the network latency in turn may depend on factors such as the number of tasks, their periods, the number and types of key-value pairs, and the publishing times used by them; there is hence an inter-dependency between the parameters. An end-to-end timing analysis may require multiple iterations, *e.g.*, as in the compositional performance analysis approach by Diemer et al. [21]. We leave such a formal timing analysis for future work. Currently, we take an engineering approach where we rely on profiling and set the WCET parameter of each task to two times its profiled WCET.

As mentioned before, in the absence of hardware timestamping, the clock skew in our setup typically remains around a few microseconds but occasionally spikes up to a couple of hundred microseconds. Since the task parameters as well as the publishing times are in the order of at least tens of milliseconds, the clock skew in general does not interfere with In-ConcReTeS's performance. Hence, we do not consider clock skews at runtime when interpreting publishing times and data validity constraints as part of our KVS API. The maximum clock skew can be considered instead during timing analysis or during admission control. The spikes in the clock skew however do affect the number of successful reads, but we simply treat them as transient timing faults.

Unlike most cloud systems, which favor agreement at all costs, In-ConcReTeS ensures by design that such transient timing faults do not cause future or cascading deadline misses by violating agreement in favor of timeliness. In other words, In-ConcReTeS's KVS task does not try to achieve consensus after its deadline, it does not "publish" values for which distributed

agreement was delayed, and applications may experience read failures when trying to read these values (as in the Bosch experiments). Application tasks may use appropriate fail-safe mechanisms to handle failed reads, such as relying on their most recent local state [49, 55].

Several optimizations are possible to improve In-ConcReTeS's performance. The KVS task currently spins during network I/O, which makes timing analysis easier, but increases CPU utilization. To reduce its CPU utilization, the KVS task can be split into round-specific tasks with relative offsets based on network I/O latency. To further reduce the CPU utilization, the KVS task frequency can be reduced further such that it does not depend on the application task frequencies anymore. In this case, writes by high frequency applications will be synchronized only every few iterations; during the remaining iterations, the KVS will simply return values from the local datastore, offering limited fault tolerance.

VI. Related Work

With the goal of safety certification, the avionics domain has been the first to acknowledge and tackle the problem of Byzantine fault tolerance in a systematic manner, while also taking into account real-time constraints. Two prominent fault-tolerant processor architectures developed as a result include the Fault-Tolerant Multiprocessor (FTMP) [31] and the Multiprocessor Architecture for Fault-Tolerance (MAFT) [34]. The *MeshKin* [64] and Reliable Optical Bus (ROBUS) [50] architectures had similar objectives, but realized Byzantine fault tolerance "inside" the communication bus instead. Other architectures like Advanced Information Processing System (AIPS) [42], Maintainable Real-Time System (MARS) [38], SAFEbus architecture [32], and Time-Triggered Protocol (TTP) [48] differ mainly in their placement and redundancy strategies [63]. However, all these designs, including the recent Ethernet standards for safety-critical applications, such as TTEthernet [6] and AFDX/ARINC 664 [51], require custom hardware, and hence are not readily applicable today.

There exists a plethora of work on Byzantine fault tolerance in the cloud computing domain that focuses on general-purpose systems, *e.g.*, Rampart [59], SecureRing [35], Practical Byzantine Fault Tolerance (PBFT) [18], Zyzzyva [39], Spinning [67], Aardvark [19], Raft [54], Redundant Byzantine Fault Tolerance (RBFT) [9], On-Demand Replica Consistency (ORDC) [22], *etc.* These systems are designed with the objective of achieving high throughput (*e.g.*, by optimizing the fault-free scenario). Properties like low latency and predictability are not always achieved, which impedes their use for real-time systems.

Most of the recent work on Byzantine fault tolerance in the real-time systems community has focused on analysis and simulations. For example, Kozhaya et al. [40] and Gujarati et al. [29] propose extensions to existing BFT protocols while considering bounded delays, and evaluate failure probabilities using both analyses and simulations. In fact, Gujarati et al. [29] analyzed the same EIGByz algorithm that we use as a specification for our RT-EIGByz implementation, but did not evaluate any practical deployment issues, which is one of our objectives in this paper. Roth and Haeberlen [60] also present a probabilistic analysis, and argue when Byzantine fault tolerance is unnecessary for real-time applications.

Recently, Li et al. [45] proposed methods to reduce the latency of failure recovery in ZooKeeper [7]. However, they focus on time-sensitive edge computing applications, not real-time embedded applications (*e.g.*, their proposal RT-ZooKeeper is evaluated on edge hosts, each with eight CPUs, 64GB memory, and 1TB disk space). Loveless et al.'s work [47] on improving the latency of distributed agreement in BFT state machine replication through eager executions on multicores is closest to our work. In-ConcReTeS's high-level objectives are different though, since it also focuses on the programming API for control application developers, and prefers a leader-less BFT algorithm to avoid reliability bottlenecks.

Veríssimo et al. [65, 66] did not focus on Byzantine fault tolerance *per se*, but proposed a programming paradigm where a small fault-tolerant *timely computing base* can be used reliably by any component that requires synchrony. Casimiro et al. [15, 16] later used this approach to implement a distributed system with dependable quality-of-service guarantees, and discussed an implementation using Linux [17]. In principle, In-ConcReTeS can be redesigned based on their proposal. However, we conjecture it suffices to use clock synchronization protocols such as PTP as the timely computing base, as in this work. Of course, the clock synchronization protocol itself will need to be strengthened using fault tolerance primitives so that it becomes analogous to the timely computing base model assumed by Veríssimo and Casimiro (which is future work).

VII. Conclusion

We have presented a BFT key-value datastore In-ConcReTeS with support for periodic applications, real-time predictability, and low latency on commodity embedded platforms.

In-ConcReTeS's design is based on the key principle that if the application is known, a replica coordination service need not be designed for arbitrary workloads, but instead should make application-specific choices that lead to maximum efficiency. By making design choices around the needs of periodic real-time applications (*e.g.*, time-aware API, statically allocated data structures, predetermined batch sizes), In-ConcReTeS is able to outperform generic datastores like Redis and etcd, which offer plenty of configuration options for throughput-oriented applications, but hardly any relating to predictability or periodicity. Depending on the hardware available and the applications that need to be synchronized, In-ConcReTeS may be used in a soft or a hard real-time setting, and as a replica coordinator or a fault-tolerant key-value store.

Given that In-ConcReTeS is modeled off the classic real-time task model, it would be interesting to evaluate the possibility of deploying In-ConcReTeS on real-time operating systems, such as FreeRTOS or Zephyr, using their real-time task APIs. This would also make it more amenable to WCET and schedulability analysis, which is desired to make it certification-friendly. We leave these extensions to future work.

References

[1] "etcd: A distributed, reliable key-value store for the most critical data of a distributed system," 2022. [Online]. Available: https://etcd.io/

[2] "Apache Cassandra Documentation v4.0," 2018. [Online]. Available: https://cassandra.apache.org/doc/latest/

[3] "Raspberry Pi 4 Model B," 20122. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[4] "Redis: The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker." 2022. [Online]. Available: https://redis.io/

[5] "Redis benchmark," 2022. [Online]. Available: https://redis.io/docs/reference/optimization/benchmarks/

[6] "Time-Triggered Ethernet (SAE AS6802)," 2016. [Online]. Available: https://www.sae.org/standards/content/as6802/

[7] "Apache ZooKeeper," 2022. [Online]. Available: https://zookeeper.apache.org/

[8] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine Replication under Attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, 2011.

[9] P.-L. Aublin, S. B. Mokhtar, and V. Quema, "RBFT: Redundant Byzantine Fault Tolerance," in *33rd IEEE International Conference on Distributed Computing Systems*, 2013.

[10] M. Barborak, A. Dahbura, and M. Malek, "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, vol. 25, no. 2, pp. 171–220, 1993.

[11] A. Bessani, J. Sousa, and E. E. Alchieri, "State Machine Replication for the Masses with BFT-SMART," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[12] D. M. Blough and G. F. Sullivan, "A Comparison of Voting Strategies for Fault-Tolerant Distributed Systems," in *9th IEEE Symposium on Reliable Distributed Systems*, 1990.

[13] F. Borran and A. Schiper, "A Leader-Free Byzantine Consensus Algorithm," in *Distributed Computing and Networking*, 2010, pp. 67–78.

[14] B. B. Brandenburg, "Liu and Layland and Linux: A Blueprint for "Proper" Real-Time Tasks," 2020. [Online]. Available: https://sigbed.org/2020/09/05/liu-and-layland-and-linux-a-blueprint-for-proper-real-time-tasks/

[15] A. Casimiro and P. Verissimo, "Using the Timely Computing Base for Dependable QoS Adaptation," in *20th IEEE Symposium on Reliable Distributed Systems*, 2001.

[16] A. Casimiro and P. Veríssimo, "Generic Timing Fault Tolerance Using a Timely Computing Base," in *International Conference on Dependable Systems and Networks*, 2002.

[17] A. Casimiro, P. Martins, and P. Verissimo, "How to Build a Timely Computing Base using Real-Time Linux," in *IEEE International Workshop on Factory Communication Systems*, 2000.

[18] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[19] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults," in *6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[20] M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal, *Understanding and Using the Controller Area Network Communication Protocol*, 2012.

[21] J. Diemer, P. Axer, and R. Ernst, "Compositional Performance Analysis in Python with pyCPA," *3rd International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems*, 2012.

[22] T. Distler and R. Kapitza, "Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency," in *6th conference on Computer systems*, 2011.

[23] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine Fault Tolerance, from Theory to Reality," in *Computer Safety, Reliability, and Security*, 2003, vol. 2788, pp. 235–248.

[24] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[25] J. C. Eidson, *Measurement, Control, and Communication Using IEEE 1588*, ser. Advances in Industrial Control, 2006.

[26] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[27] D. Fontanelli, D. Macii, P. Wolfrum, D. Obradovic, and G. Steindl, "A Clock State Estimator for PTP Time Synchronization in Harsh Environmental Conditions," in *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2011.

[28] A. P. Frantz, L. Carro, É. Cota, and F. L. Kastensmidt, "Evaluating SEU and Crosstalk Effects in Network-on-Chip Routers," in *12th IEEE International On-Line Testing Symposium*, 2006.

[29] A. Gujarati, S. Bozhko, and B. B. Brandenburg, "Real-Time Replica Consistency over Ethernet with Reliability Bounds," in *26th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 376–389.

[30] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: a Time-Triggered Language for Embedded Programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[31] A. Hopkins, T. Smith, and J. Lala, "FTMP—A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, 1978.

[32] K. Hoyme and K. Driscoll, "SAFEbus (for avionics)," *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, no. 3, pp. 34–39, 1993.

[33] V. Jordan, "Fun projects at Inatech: Sync up Your Clocks! Better PTP Settings on Raspberry Pi," 2020. [Online]. Available: https://medium.com/inatech

[34] R. Keichafer, C. Walter, A. Finn, and P. Thambidurai, "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 398–404, 1988.

[35] K. Kihlstrom, L. Moser, and P. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," in *31st Hawaii International Conference on System Sciences*, 1998.

[36] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm," in *Advances in Real-Time Systems*, 2012, pp. 103–120.

[37] C. Kohlhoff, "Boost.Asio," 2022. [Online]. Available: https://www.boost.org/doc/libs/1_79_0/doc/html/boost_asio.html

[38] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating Transient Faults in MARS," in *20th International Symposiuium on Fault-Tolerant Computing*, 1990.

[39] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin, "Zyzzyva: Speculative Byzantine Fault Tolerance," *Communications of the ACM*, vol. 51, no. 11, p. 86, 2008.

[40] D. Kozhaya, J. Decouchant, and P. Esteves-Verissimo, "RT-ByzCast: Byzantine-Resilient Real-Time Reliable Broadcast," *IEEE Transactions on Computers*, vol. 68, no. 3, pp. 440–454, 2018.

[41] S. Kramer, D. Ziegenbein, and A. Hamann, "Real World Automotive Benchmarks for Free," in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

[42] J. Lala, R. Harper, K. Jaskowiak, G. Rosch, L. Alger, and A. Schor, "Advanced Information Processing System (AIPS)-Based Fault Tolerant Avionics Architecture for Launch Vehicles," in *9th IEEE/AIAA/NASA Conference on Digital Avionics Systems*, 1990.

[43] D. Lavo, T. Larrabee, and B. Chess, "Beyond the Byzantine Generals: Unexpected Behavior and Bridging Fault Diagnosis," in *IEEE International Test Conference. Test and Design Validity*, 1996.

[44] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, ser. Lecture notes in computer science, 2001, no. 2050.

[45] H. Li, C. Lu, and C. D. Gill, "RT-ZooKeeper: Taming the Recovery Latency of a Coordination Service," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5s, pp. 1–22, 2021.

[46] J. Loeser and H. Haertig, "Low-Latency Hard Real-Time Communication Over Switched Ethernet," in *16th Euromicro Conference on Real-Time Systems*, 2004.

[47] A. Loveless, R. Dreslinski, B. Kasikci, and L. T. X. Phan, "IGOR: Accelerating Byzantine Fault Tolerance for Real-Time Systems with Eager Execution," in *27th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2021.

[48] R. Maier, G. Bauer, G. Stoger, and S. Poledna, "Time-triggered architecture: A consistent computing platform," *IEEE Micro*, vol. 22, no. 4, pp. 36–45, 2002.

[49] R. Majumdar, I. Saha, and M. Zamani, "Synthesis of Minimal-error Control Software," in *10th ACM International Conference on Embedded Software*, 2012.

[50] P. Miner, M. Malekpour, and W. Torres, "A Conceptual Design for a Reliable Optical Bus (ROBUS)," in *21st Digital Avionics Systems Conference*, 2002.

[51] J.-P. Moreaux, "Data Transmission System for Aircraft," 2005, US Patent 6,925,088.

[52] G. Morgado, "Inverted Pendulum," 2011. [Online]. Available: https://gmagno.users.sourceforge.net/InvertedPendulum.htm

[53] S. Nadarajah and A. Gupta, "Generalizations and Related Univariate Distributions," *Statistics Textbooks and Monographs*, vol. 174, pp. 97–164, 2004.

[54] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference*, 2014, pp. 305–319.

[55] P. Pazzaglia, L. Pannocchi, A. Biondi, and M. D. Natale, "Beyond the Weakly Hard Model: Measuring the Performance Cost of Deadline Misses," in *30th Euromicro Conference on Real-Time Systems*, 2018.

[56] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[57] S. Poledna, "Replica Determinism and Non-Determinism," *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, pp. 31–60, 1996.

[58] S. Poledna, A. Burns, A. Wellings, and P. Barrett, "Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems," *IEEE transactions on Computers*, vol. 49, no. 2, pp. 100–111, 2000.

[59] M. K. Reiter, "The Rampart toolkit for building high-integrity services," in *Theory and Practice in Distributed Systems*, 1995, vol. 938, pp. 99–110.

[60] E. Roth and A. Haeberlen, "Do Not Overpay for Fault Tolerance!" in *27th Real-Time and Embedded Technology and Applications Symposium*, 2021.

[61] J. Rufino, P. Verissimo, G. Arroz, C. Almeida, and L. Rodrigues, "Fault-Tolerant Broadcasts in CAN," in *28th Annual International Symposium on Fault-Tolerant Computing*, 1998.

[62] S. Schuster, P. Ulbrich, I. Stilkerich, C. Dietrich, and W. Schröder-Preikschat, "Demystifying Soft-Error Mitigation by Control-Flow Checking – A New Perspective on its Effectiveness," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 1–19, 2017.

[63] T. Smith and J. Yelverton, "Processor architectures for fault tolerant avionic systems," in *IEEE/AIAA 10th Digital Avionics Systems Conference*, 1991.

[64] A. K. Somani and M. Bagha, "Meshkin: A Fault Tolerant Computer Architecture with Distributed Fault Detection and Reconfiguration," in *Fehlertolerierende Rechensysteme / Fault-tolerant Computing Systems*, 1989, vol. 214, pp. 197–208.

[65] P. Veríssimo and A. Casimiro, "The timely computing base model and architecture," *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 916–930, 2002.

[66] P. Veríssimo, A. Casimiro, and C. Fetzer, "The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness," in *International Conference on Dependable Systems and Networks*, 2000.

[67] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary," in *28th IEEE International Symposium on Reliable Distributed Systems*, 2009.

[68] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostak, and C. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, 1978.