A Byzantine Fault-Tolerant Key-Value Store for Safety-Critical Distributed Real-Time Systems

Malte Appel[†*], Arpan Gujarati[*], and Björn B. Brandenburg[*]
[*]Max Planck Institute for Software Systems (MPI-SWS), Germany
[†]Saarland University, Germany

I. Motivation

From modern cars to airplanes to industrial plants, many applications that must execute in a timely manner are deployed on distributed systems. In case of safety-critical applications, like the anti-lock braking system of a car, the underlying system must tolerate inadvertent environmentally-induced faults to guarantee user safety. Since such systems often operate at high frequencies, fault-induced failures have to be masked through active replication. Furthermore, before such a system is deployed, it typically has to be analyzed w.r.t. its runtime, safety guarantees, *etc.* This is required for common safety-certification standards such as the DO-178C standard for aviation or the ISO 26262 standard for automotive systems.

To ease the development of such systems, our goal is to design a fault-tolerant middleware on which real-time control applications can be effortlessly replicated, that respects real-time and low-latency requirements, and whose reliability can be analyzed *a priori* for the purpose of safety certification.

II. Model and Assumptions

We assume a distributed system consisting of multiple networked processing elements (PEs) that hosts one or more distributed real-time control applications. An application *fails* if the control loop output, *i.e.*, its final physical actuation, is incorrect due to failures in one of the intermediate stages of the control loop, as explained next.

We consider failures caused by transient soft errors and/or permanent errors due to environmental conditions (such as electromagnetic interference (EMI), thermal effects, *etc.*) and manufacturing defects. In particular, we assume that failures are environmentally induced and not malicious.

The aforementioned failure sources may result in program-visible *Byzantine PE failures*, *i.e.*, PEs may behave arbitrarily, resulting in the delivery of incorrect or inconsistent outputs to other PEs, or in no outputs at all. For example, a PE may end up sending differing messages during a broadcast to its neighbors, say, due to two PEs interpreting the same signal differently owing to a soft error [1], or due to inconsistencies in the underlying network protocol, as in CAN [2].

In contrast to PEs, the network connecting the distributed PEs is assumed to be both *synchronous* and *reliable*, *i.e.*, message delivery times are bounded and message deliveries are ordered. Any network failures are attributed to PE failures, *e.g.*, transient network partitions or delayed message transmissions are considered as PE omission failures.

We assume that the PEs are reliably synchronized using a high-precision clock synchronization protocol, such as [3].

III. Problem Statement

Byzantine failures include both *value failures*, *e.g.*, incorrect computation or inconsistent message deliveries, and *timing failures*, *e.g.*, crashes or message omissions. Value failures may lead to incorrect system behavior, *e.g.*, when wrong inputs are delivered to an actuator, it performs an incorrect action. Crashes of critical components may lead to immediate system failure. Omission failures may lead to a delay or complete lack of reaction. Thus, depending on the extent of value failures and the duration of timing failures, they can have catastrophic consequences in a safety-critical real-time application.

Existing Byzantine fault tolerance (BFT) protocols (see §IV) mitigate the effects of Byzantine failures, but focus on soundness while compromising timeliness. A majority of them were designed primarily for large-scale, predominantly throughput-oriented distributed systems, and thus these protocols (occasionally) exhibit unpredictable, long execution times unsuitable for high-frequency real-time control applications.

This work targets the problem of providing BFT in a predictable, preferably short, time suitable for applications with activation frequencies as high as 10 kHz. In particular, an ideal implementation of a BFT real-time control application and the underlying distributed system must guarantee the following correctness properties despite Byzantine failures.

- *Validity:* If a correct (non-faulty) task of the control application receives or reads a value, that value should have been sent or written by a correct task.
- *Freshness:* If a correct task of the control application receives or reads a value, that value should have been sent or written less than $X\ ms$ ago, where X is application-specific.
- *Agreement:* If a correct task has state S at the end of a control-loop iteration, then all correct replicas of the task have state S at the end of the control-loop iteration.
- *Timely termination:* During each control-loop iteration, the control loop should perform the intended action (the final plant actuation) before the end of that iteration.

Example domains that have such requirements include control systems in ships, avionics, air traffic control, *etc.* [4].

In addition to guaranteeing these correctness properties, any BFT mechanism added to an otherwise certified application should also be *analyzable*. That is, given the peak soft error rates in different system components, it should be possible

Table I: BFT protocols tolerating f failures

Name	Network model	Hosts	Type
BChain-3 [5]	partial synchrony	$3f+1$	chain
Zyzzyva [6]	partial synchrony	$3f+1$	broadcast
PBFT [7]	weak synchrony	$3f+1$	broadcast
Q/U [8]	asynchronous	$5f+1$	quorum
HQ [9]	asynchronous	$3f+1$	quorum
Aliph-Chain [10]	asynchronous	$3f+1$	quorum/chain/broadcast
Ben-Or *et al.* [11]	synchronous	$4f+1$	randomized; quorum
Mostéfaoui *et al.* [12]	asynchronous	$3f+1$	randomized; broadcast
AER [13]	asynchronous	$3f+1$	randomized; quorum
Patra *et al.* [14]	asynchronous	$3f+1$	randomized; broadcast
HoneyBadgerBFT [15]	asynchronous	$3f+1$	randomized; broadcast

to quantify the overall system reliability (*e.g.*, in terms of its mean time to failure) for safety-certification purposes.

IV. Existing BFT Protocols

Since the Byzantine Generals problem was proposed by Lamport *et al.* [16], many BFT protocols have been proposed with the objective of ensuring some (if not all) of the correctness properties listed in §III. Representative protocols for different network models and different design types are summarized in Table I and discussed in brief below.

Broadcast-based BFT protocols always involve all replicas in the agreement process. The client broadcasts its proposal to all replicas [17] or to a designated primary replica that multicasts the proposal to the backups [6, 7]. In contrast, quorum-based BFT protocols require only a representative subset of replicas (the *quorum*) to form an agreement [8, 9]. Both broadcast- and quorum-based BFT protocols achieve relatively low end-to-end latency, but at the cost of large bandwidth consumption. It is thus challenging to incorporate them in distributed real-time systems that often use low-bandwidth networks for cost-efficiency and predictability.

Chain-based protocols arrange replicas in a chain. Clients send their value to the head of the chain and receive a reply only after the message has moved through either a part of [5] or the complete chain [10]. This provides higher throughput, but results in higher latency (compared to broadcast or quorum-based protocols). Depending on the latency requirements and the underlying network, such protocols can be prohibitive for real-time control applications.

The protocols discussed above are *deterministic*, which implies that the number of rounds required for agreement is lower-bounded by $f+1$ when tolerating up to f failures [18]. To improve upon this performance metric, *non-deterministic* or *randomized* BFT protocols were proposed [19, 20], which reduce the number of required rounds, but may violate one of the correctness properties listed in §III with low probability. For example, in the $(1-\epsilon)$-*terminating* protocol by Patra *et al.* [14], a correct task terminates with probability $(1-\epsilon)$, and protocols like AER [13] use *almost-everywhere* Byzantine agreement where agreement is guaranteed for all but $O(\log^{-1} n)$ correct tasks. Protocols such as the one proposed by Patra *et al.* [14] are favorable if ϵ is reasonably low and does not significantly affect the overall system reliability.

None of the protocols discussed above, however, guarantees freshness and timely termination (as stated in §III). For example, in a hard real-time application, if a value satisfying validity and agreement is delivered to a task after its deadline, it has nonetheless zero utility. It is thus better to receive a correct value (or maybe a value that is correct with high probability) on time, or to not receive it at all. To realize this, the BFT protocol must be aware of the timeliness requirements of all values that it handles. Similarly, to ensure freshness, it must be aware of the application-specific lifetime of each value.

V. Proposed Solution

We propose to build a BFT key-value store (KVS) that will act as a middleware for distributed real-time applications, that satisfies the correctness properties listed in §III, and that is analyzable. We first give an overview of the system design, and then explain the rationale behind our design.

Overview. The KVS provides a `write(k,v,t)` API for *publishing* a value `v` for key `k` at time `t` and a `read(k,t)` API for reading the latest published value `v` (that is published not earlier than `t`) for key `k` (see Listing 1 for an example). The time parameter `t` is application-specific and inspired by the logical execution time paradigm [21, 22]. For a write, it determines the absolute time at which the write should be published, *i.e.*, made visible to subsequent read requests for key `k`, and for a read, it determines the freshness requirement of the returned value, *i.e.*, values published earlier than time `t` are not returned. The middleware underlying the read and write APIs consists of one local data store per PE, which coordinate using a BFT protocol to tolerate Byzantine failures.

Freshness and timely termination. The time parameter `t` in the read and write APIs allows the programmer to convey application-specific freshness and timeliness requirements to the KVS. The agreement protocol disseminates any written value `v` by time `t` to enable timely termination of the control loop, where `t` must be sufficiently far in the future to allow the execution of the agreement protocol. For a read request, the value is served by the local data store. If no fresh value exists locally, a valid value is requested from other data stores with a consensus protocol. If still no fresh value exists (*i.e.*, there is no fresh value in the system), the read returns a default value. Thus, by using the time parameter `t`, the KVS guarantees both freshness and timely termination for the control application.

Validity. The agreement protocol guarantees that a valid value is stored in every local data store, and read correctly by the client, if the value or the read operation is not affected by failures on the client PE. Furthermore, to reduce the likelihood of invalid reads due to failures on the client side (say, when the published value in the local data store is corrupted just before being read), the local data store computes and stores a checksum for each published value. With this, the KVS has the option of invoking the consensus protocol to retrieve the value from other local data stores in case of a checksum mismatch.

Agreement. The agreement property requires that replicas have a uniform state at the end of every control-loop iteration.

Listing 1 Example PID controller programmed over KVS

```
1: procedure PERIODICTASKACTIVATION
2:     freshness ← timeOfLastActivation()
3:     currentPos ← KVS.read("sensorDataKey", freshness)
4:     error ← KVS.read("targetPosKey", freshness) − currentPos
5:     integral ← KVS.read("integralKey", freshness) + error
6:     derivate ← error − KVS.read("errorKey", freshness)
7:     newPos ← (KVS.read("kpKey", freshness) ∗ error) +
                (KVS.read("kiKey", freshness) ∗ integral) +
                (KVS.read("kdKey", freshness) ∗ derivative)
8:     time ← timeOfNextActivation()
9:     KVS.write("errorKey", error, time)
10:    KVS.write("integralKey", integral, time)
11:    KVS.write("controlValueKey", newPos, time)
```

The KVS guarantees this by requiring that any stateful values used by the application are written to and read from the KVS (as illustrated in Listing 1). Multiple writes for the same key that should be published at the same time are resolved transparently by the KVS middleware. Applications can specify a key-level policy at configuration time, such as majority voting, averaging, median, *etc.*, that decides how the KVS processes differing values (say, noisy, but correct, values published by replicated sensor tasks). As a key benefit, this approach makes replication effortless for the application developer, since it suffices to instantiate the application (*e.g.*, the PID controller code in Listing 1) on an arbitrary number of hosts for replication, without any changes to the code. Furthermore, since all application state is persisted in the KVS, crashed applications or PEs can be trivially restarted.

Analyzability. The proposed design reduces the application failure domain to the KVS, *i.e.*, failures are attributed to the KVS implementation and not to the application code. It abstracts away any BFT mechanisms from the programmer and decouples it from the application logic, which makes it easier to reason about and formally model the KVS. In particular, a layered design consisting of a separate application layer, KVS layer, clock synchronization layer, networking layer, *etc.*, enables independent analysis of the worst-case reliability bounds for each layer while assuming that other layers are reliable, and then composition of these bounds to yield an overall system reliability bound.

Coordination protocol. The process of choosing and evaluating an appropriate BFT protocol for the coordination of data store replicas is still in progress. Since we focus on control applications, we concentrate on protocol latencies rather than their throughput. For fail-operational systems, protocols that completely degrade in performance as soon as failures occur are unacceptable. We plan to avoid using protocols such as Zyzzyva [6] that improve performance through speculative execution at the cost of unpredictable revert actions. However, a predictable version of such protocols, with manageable latencies, might be interesting. Clement *et al.*'s [23] work on *robust BFT*, which favors an equal performance in both failure and non-failure cases over optimizations benefiting only the failure-free scenario, is particularly interesting in this regard. Some of the non-deterministic protocols achieve much lower latencies and thus seem appealing, but they introduce a small risk of violating the agreement property. If this probability is reasonably low, randomization might be the favorable solution, but for the moment, we leave the possible incorporation of non-deterministic protocols as future work.

Next steps. Once the KVS is designed and implemented, we will conduct rigorous fault-injection experiments through the injection of bit flips in arbitrary memory locations (including the OS kernel), since environmental EMI sources are not restricted to the specific parts of the memory used by the KVS process. Finally, we aim to analyze the reliability of the BFT KVS to derive a safe bound on the mean time to failure of applications hosted on this platform, given bounds on the peak rates of soft and permanent errors in all PEs.

REFERENCES

[1] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *SafeComp*, 2003.

[2] G. M. Lima and A. Burns, "A consensus protocol for CAN-based systems," in *RTSS*, 2003.

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

[4] D. Locke, *Applications and System Characteristics*. Boston, MA: Springer US, 2002, pp. 17–26.

[5] S. Duan, H. Meling, S. Peisert, and H. Zhang, "BChain: Byzantine replication with high throughput and embedded reconfiguration." in *OPODIS*, 2014.

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *SOSP*, 2007.

[7] M. Castro, B. Liskov *et al.*, "Practical Byzantine fault tolerance," in *OSDI*, 1999.

[8] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *SOSP*, 2005.

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance," in *OSDI*, 2006.

[10] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *EuroSys*, 2010.

[11] M. Ben-Or, E. Pavlov, and V. Vaikuntanathan, "Byzantine agreement in the full-information model in $O(\log n)$ rounds," in *STOC*, 2006.

[12] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous binary Byzantine consensus with $t < n/3, O(n^2)$ messages, and $O(1)$ expected time," *JACM*, vol. 62, no. 4, p. 31, 2015.

[13] N. Braud-Santoni, R. Guerraoui, and F. Huc, "Fast Byzantine agreement," in *PODC*, 2013.

[14] A. Patra, A. Choudhury, and C. P. Rangan, "Asynchronous Byzantine agreement with optimal resilience," *Distributed Computing*, vol. 27, no. 2, pp. 111–146, 2014.

[15] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *CCS*, 2016.

[16] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM TOPLAS*, vol. 4, no. 3, pp. 382–401, 1982.

[17] G. Bracha, "Asynchronous Byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[18] M. J. Fischer and N. A. Lynch, "A lower bound for the time to assure interactive consistency," *Information processing letters*, vol. 14, no. 4, pp. 183–186, 1982.

[19] M. O. Rabin, "Randomized Byzantine generals," in *FOCS*, 1983.

[20] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *PODC*, 1983.

[21] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded control systems development with Giotto," in *LCTES*, 2001.

[22] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[23] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults." in *NSDI*, 2009.