TINYBFT: Byzantine Fault-Tolerant Replication for Highly Resource-Constrained Embedded Systems

Harald Böhm, Tobias Distler, Peter Wägemann
System Software Group
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—Byzantine fault-tolerant (BFT) state-machine replication offers resilience against a wide spectrum of faults including hardware crashes, software failures, and attacks. Unfortunately, having been mostly designed for use on large servers, existing implementations of such replication protocols consume vast amounts of memory and therefore are not available to embedded systems that consist of highly resource-constrained devices. In this paper we address this problem with TINYBFT, the first BFT state-machine replication library specifically developed to run on nodes comprising 1 MB of RAM or less. To achieve this, TINYBFT relies on a memory-efficient implementation of the PBFT protocol that allocates all of its memory statically and thus, in contrast to common state-of-the-art PBFT-based libraries, has a guaranteed worst-case memory consumption that is known at compile time. Experiments show that our library provides sufficiently low latency even on tiny ESP32-C3 microcontrollers.**

***Index Terms*—Byzantine Fault Tolerance, Embedded Systems, State-Machine Replication, Highly Resource-Constrained Devices**

I. INTRODUCTION

Byzantine fault-tolerant (BFT) state-machine replication [1] is a powerful technique to implement resilient distributed systems as it enables them to tolerate arbitrarily faulty ("Byzantine" [2]) behavior in a subset of nodes. While a large part of the research in this area is focused on server-grade hardware [3]–[5], especially in the context of blockchains [6]–[8], Byzantine fault tolerance in general (i.e., not necessarily in the form of state-machine replication) has also gained importance in embedded systems. One of the first major examples of use cases were robust system architectures in the avionics industry relying on custom hardware components [9], [10], however more recently there also have been proposals to provide Byzantine fault tolerance on small general-purpose platforms such as Raspberry Pi [11]–[13], for example to protect critical components in Internet-of-Things (IoT) systems.

Taking into account these efforts to push resilience mechanisms towards the edge of computing infrastructures, in this paper we go one step further than previous works by exploring how to perform BFT state-machine replication in embedded systems that are composed of tiny devices. In particular, our work differs from existing approaches in two main aspects: (1) Nodes in our target environments are highly resource-constrained devices that comprise only 1 MB or less of internal RAM (e.g., ESP32-C3: 160 MHz, 400 KB RAM [14]). In terms of memory size, this is more than three orders of magnitude smaller than the 4 GB of RAM available on the Raspberry Pis used in [12], [13]. (2) While other works in the embedded-systems domain usually concentrate on individual BFT mechanisms such as reliable broadcast [15]–[17], atomic broadcast [17], or other forms of consensus [13], our goal is to provide full-fledged state-machine replication. In addition to BFT agreement and the establishment of a total order on all operations, this for example also includes means for checkpointing and the periodic garbage collection of protocol state [18].

The main result of our research efforts is TINYBFT, the first library that enables BFT state-machine replication on tiny devices. TINYBFT relies on PBFT [3] for agreement, but in contrast to existing libraries [3], [4] implements the protocol in a way that is both memory efficient and (by means of static allocation) able to ensure an upper bound on memory consumption. Our experiments with a replicated key-value store confirm that TINYBFT runs on highly resource-constrained nodes such as ESP32-C3 microcontrollers [14]. Furthermore, the measurements show that with a response time of about 80 ms on this tiny platform the library clearly meets our latency demands (i.e., hundreds of milliseconds to a few seconds).

As pointed out by Roth and Haeberlen [19], when it comes to Byzantine fault tolerance in embedded systems, without a proper analysis of the requirements of the respective application and/or the properties of the underlying platform, there is the risk of employing protocols that are unnecessarily complex for the intended use cases. Hence, we do not advocate for the use of our library in scenarios in which less powerful mechanisms (e.g., reliable broadcasts without ordering guarantees) are sufficient. On the other hand, state-machine replication can be a valuable and effective technique even in embedded systems, as we illustrate in this paper based on several examples of use-case scenarios that are able to benefit from a library such as ours. Specifically, this applies to settings in which a group of tiny devices needs to replicate a common state (e.g., subsequently collected sensor data) in a consistent manner, and Byzantine faults are not only limited to non-malicious behavior (e.g., random memory corruption) but may also include an adversary's attempts to deliberately introduce inconsistencies between the participating nodes.

In particular, this paper makes the following contributions: (1) It analyses existing BFT state-machine replication protocol designs with respect to their suitability for embedded systems with tiny nodes. (2) It presents TINYBFT, the first BFT state-machine replication library for highly resource-constrained devices. (3) It experimentally evaluates the memory consumption and performance of TINYBFT on the actual target hardware.

2642-7346/24/$31.00 ©2024 IEEE
DOI 10.1109/RTAS61025.2024.00026

II. System Model and Background

In this section, we discuss details on the architecture and common characteristics of our target environments as well as potential use-case scenarios. Furthermore, we provide background information on BFT state-machine replication.

A. Target Environments

As illustrated in Figure 1, our target systems comprise two main parts: (1) a *device tier* consisting of tiny embedded devices that rely on a BFT state-machine replication protocol to keep crucial parts of their states consistent even in the presence of arbitrary faults, as well as (2) one or more *gateways* that enable the device tier to communicate with the outside world, for example to receive commands from an administrator or to externalize collected data. Compared with device-tier nodes, the gateways run on significantly more powerful machines for which limited resources are not an issue. Since there already are solutions to achieve Byzantine fault tolerance through replication for these kinds of gateways [12], [15], our work focuses primarily on the device tier. This goes as far as the gateways may become an optional system part if a use-case scenario, for example, does not require administrator intervention due to the group of device-tier nodes running autonomously. In the following, we discuss each of the two parts in detail.

Device Tier. The device tier is composed of highly resource-constrained nodes (e.g., microcontrollers with additional communication interfaces [20] or integrated, embedded system-on-chip platforms [14]) that each comprise only a very limited amount of RAM (i.e., 1 MB or less). If available, using inter-chip communication busses such as SPI [21], I^2C [22], or I^3C [23], it is possible to extend the available resources with non-volatile memory [24] or external RAM [25], however such measures only offer a small number of additional megabytes. Furthermore, with the supplemental memory not being directly available in the processor's address space, access times are significantly higher than for integrated RAM, which is a factor we take into account at design time (see Section III).

Apart from memory, a second scarce resource in device-tier nodes is energy. In this context, we do not make any specific assumptions on how the energy for operating a node is supplied, thereby supporting both battery-backed devices [26] as well as energy-neutral devices that harvest all the energy they consume from the environment [27]. Either way, a node may temporarily run out of energy and thus in the meantime cannot participate in the replication protocol. In our target systems, such scenarios are infrequent and limited to comparably short periods of time. For the rare case that a node is out of energy for a prolonged time we count this as a node failure.

In addition to energy faults, we assume that nodes in the device tier can show any form of arbitrary faulty behavior. Besides problems rooted in hardware or software errors, this also includes scenarios in which an adversary gained control of a node and actively uses the node in an effort to introduce inconsistencies into the device tier's replicated state. To prevent such efforts from being successful, device-tier nodes replicate their state using a protocol that is able to tolerate Byzantine faults.

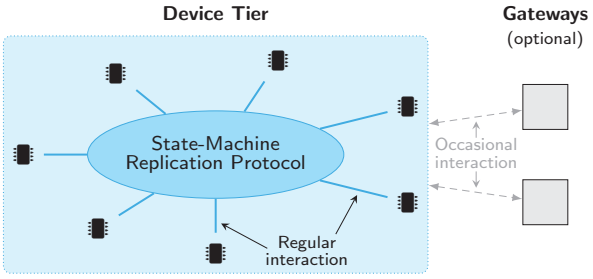


Figure 1. Basic system architecture

To run the protocol, each device-tier node is equipped with a transceiver that allows the node to communicate with others by sending and receiving messages over wireless links. Although unicast primitives are available, the wireless nature of the communication makes multicast the most efficient (and hence preferred) option for the interaction within the device tier. Possible technologies used for this purpose include WiFi, Bluetooth (LE) [20], and LoRa [28], the latter being a method specifically targeting low-power and long-range transmission. As a result of network communication being unreliable, (multiple) retransmissions of a message may become necessary for the message to eventually arrive at the intended receiving node(s). Consequently, we neither assume an upper bound on the delay with which messages are delivered to receivers, nor require the state-machine replication protocol to provide hard real-time guarantees. With regard to protocol latency, we target overall response times in the range of hundreds of milliseconds to a few seconds, which for example is sufficient for many IoT control applications [16].

Gateways. Deployed on comparably large machines, gateways serve as juncture between the device tier and external entities. Specifically, they may be used to provide device-tier nodes with additional inputs, for example by administrators submitting reconfiguration commands in order to make changes to the system. Conversely, gateways can also be employed as relays for data that has been collected by the device tier and should be further processed, analyzed, or stored on a more powerful computing infrastructure (e.g., a cloud-based framework).

Due to the interaction with gateways requiring additional energy, device-tier nodes typically communicate with them at a much lower frequency than they exchange messages with each other. As a consequence, the device tier should be primarily viewed as a standalone system that needs to be able to remain operational on its own, a requirement that is fulfilled by the group of device-tier nodes keeping their states consistent using the BFT state-machine replication protocol.

B. Use-Case Scenarios

Having presented our target system architecture, in the following we outline several application examples for it.

Resilient Control Applications. As recently shown by Gujarati et al. [13], a key-value store managing the latest values recorded by sensors is a powerful abstraction for building

control applications. Replicating such a key-value store with a state-machine replication protocol enables device-tier nodes to update sensor data in a coordinated fashion, which in turn would allow a control application operating on this information to make decisions that are consistent across all nodes. More concretely, to add a new value obtained by a local sensor, a device-tier node in this use-case scenario issues a write request to the replicated key-value store, which is then totally ordered by the replication protocol with respect to all other write requests. (If necessary, administrators can submit their own write requests via gateways.) With each device-tier node executing the write requests in the order determined by the replication protocol, the key-value store copies on all nodes advance in the same deterministic way. By also synchronizing the points in time at which the control-application process on each node reaches decisions (e.g., on every 10th update), it is ensured that all nodes act in a consistent manner.

Blockchain-based Recording. Blockchains (or more general: distributed ledgers) provide means to store data in an indelible and unalterable fashion. Compared with permissionless blockchains such as the one used for Bitcoin [29], permissioned blockchains have the advantage of requiring significantly less resources due to relying on state-machine replication protocols for consensus [30]. For our target environments, this for example creates the opportunity to reliably record sensor data within the device tier, while at the same time protecting the data against undetected manipulation. With the size of a blockchain increasing with every block that is added to the ledger, device-tier nodes in such a use-case scenario occasionally upload older blocks to the gateways and solely keep the hash of the latest block in local memory, thereby making room for new blocks. This way, despite having only a small amount of memory available, the device tier is able to continuously record new data in the blockchain even if the communication with gateways is limited to infrequent interactions.

Energy-Neutral Room Booking. Energy-neutral room displays [31] offer an eco-friendly way to present up-to-date information (e.g., the meeting schedule for a conference room) without requiring a stationary power supply or the replacement of batteries; instead, all the energy they consume is gathered from the environment (e.g., by turning a crank that is attached to the display [31]). Interconnecting such devices via a state-machine replication protocol allows them to synchronize their states and provide enhanced functionality, for example in the form of a room-booking service. In this use-case scenario, the replicated state represents the occupancy plan for a large room with multiple entrances (each equipped with its own display) or the combined plans of a group of neighboring smaller rooms. New reservations for a room can be submitted at each of the associated displays, leading to state-modifying operations that are then consistently distributed and applied on all displays. In this context, the total order established by the state-machine replication protocol on all operations enables the system to handle cases in which conflicting reservations for the same room are concurrently submitted at different displays.

C. Byzantine Fault-Tolerant Replication

As illustrated in Figure 2, BFT state-machine replication provides fault tolerance by maintaining copies of the replicated state on multiple nodes, which are also referred to as *replicas*. Without further assumptions, at least $n = 3f + 1$ replicas are required to tolerate up to f faulty nodes [32]. As is common in this domain, we use the term "faulty" for nodes that at some point deviate from their specification, independent of whether the abnormal behavior is visible from the outside or not. Nodes that are temporarily unavailable (e.g., due to an intermediate lack of energy) are not considered to be faulty, provided that they keep their relevant protocol state when resuming operation. Non-faulty nodes are referred to as "correct".

The main tasks of a replication protocol are to keep the state copies of all correct replicas in sync and to enable coordinated reads and writes to the replicated state. For this purpose, the application logic operating on the replicated data implements a deterministic state machine [1] that retrieves/modifies the state by executing commands. To ensure consistency, the order in which commands are processed is identical across all correct replicas. Establishing this order is the responsibility of an agreement protocol that assigns each new command a unique, monotonically increasing sequence number.

As explained in Section IV, the inner workings and provided guarantees of the agreement stage depend on the specific protocol in use; in Section V-A, we discuss an example of such a protocol in detail. In general, there are two main guarantees that typically are of major interest for most use cases: *safety* and *liveness* [33]. Informally, safety refers to the fact that correct replicas are in the same state after having executed commands up to the same sequence number, thereby enabling the replica group as a whole to behave like a centralized service implementation. Liveness, on the other hand, means that new commands will eventually be ordered and executed by the group, and the system hence is able to make progress.

Traditionally, BFT replication protocols clearly distinguish between nodes hosting the replicated application ("replicas") and nodes issuing commands to the service ("clients") [18]. In our target systems, the nodes in the device tier commonly perform both roles, which is why in the remainder of this paper we use the term client only to emphasize the associated functionality. Notice that we do not impose any restrictions on when commands may be issued. That is, at any given moment each device-tier node in the system is allowed to create a new command (e.g., to record a recently collected sensor value) and submit it to the state-machine replication protocol.

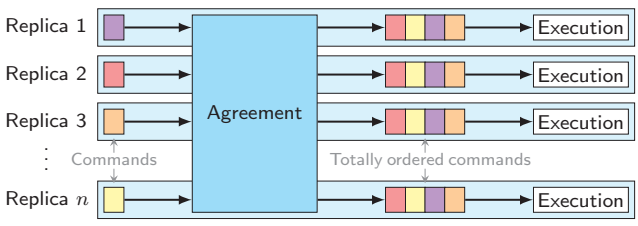


Figure 2. State-machine replication

III. Problem Statement

Operating a BFT state-machine replication protocol in the environment and under the circumstances described in Section II-A poses several challenges. In the following, we discuss three of them in detail and outline our respective contributions.

Challenge #1: Selecting the Right Protocol Design

Decades of research in the area of BFT state-machine replication have brought up a variety of protocol designs [18] that are tailored to different use cases and hence make it difficult to determine which design is best suited for our specific target environment. As a consequence, and due to the lack of a one-fits-all solution, it is crucial to precisely examine the particular assumptions each protocol design is based on (e.g., regarding the availability of participating nodes as well as the synchrony and reliability of the network connecting them), in order to be able to make a well-informed selection decision.

Our Contribution. In Section IV, we present an overview of existing BFT state-machine replication protocol designs and analyze their suitability with respect to our target environment.

Challenge #2: Reducing a Protocol's Memory Footprint

To our knowledge, we are the first to study BFT state-machine replication among nodes that each only have a tiny amount of resources available, especially when it comes to memory. Related works in the context of embedded systems and IoT systems typically either based their analyses on simulations [15]–[17] or performed the evaluation with Raspberry Pis comprising multiple gigabytes of RAM [12], [13], which in terms of memory is three orders of magnitude larger than the devices we are focusing on in this paper. Outside of the embedded-systems domain, BFT protocols are commonly run on server-grade hardware [4], [5], [8], [34], resulting in memory consumption to become an (almost) irrelevant factor in the design of these protocols. As a consequence, due to their huge memory footprint we are not able to directly use existing implementations in our target systems. This is especially true for state-of-the-art BFT replication libraries such as BFT-SMaRt which are written in Java [4], since hosting a Java runtime on our tiny device-tier nodes is out of the question.

Given these circumstances, with regard to memory, the challenge we face consists of two aspects: (1) We need to find a way to reduce the memory footprint of a BFT state-machine replication protocol (including the state of the replicated application) to an extent that it fits on our device-tier nodes. (2) To enable the replication protocol to uphold its safety and liveness guarantees, we must ensure that nodes do not run out of memory while executing the protocol. Ideally, this can be achieved by providing an upper bound on the size of a protocol's memory footprint, however attaining this goal is made complicated by the fact that existing protocol implementations [3], [4] usually do not guarantee such a limit.

Our Contribution. Using the seminal PBFT protocol as example, in Section V we present a detailed study on where and how much memory is typically spent in a BFT state-machine replication protocol. Leveraging these insights, in Section VI we then describe TinyBFT, the first BFT state-machine replication library for highly resource-constrained devices.

Challenge #3: Exploiting Additional (Non-Volatile) Memory

As explained in Section II-A, some of the embedded platforms we consider for the device-tier nodes [14], [35] can be equipped with additional memory modules in order to increase the available memory by a few megabytes. However, with accesses to extension modules being significantly slower than accesses to internal RAM, making efficient use of the additional resources is not straightforward. Consequently, the design of a replication library needs to take the specific access times of the different memory regions into account and minimize the number of costly accesses by carefully deciding which part of the replica state to maintain in internal RAM, and which information to move to the additional memory modules.

Using non-volatile memory technologies, such as FRAM [24], for the extended memory not only has the benefit of increasing the amount of available resources, it also offers another important property: persistence. Being able to keep data in non-volatile storage is crucial in replicated systems, because it enables a node to resume operation after temporary outages (e.g., due to running out of energy before its battery is recharged) without becoming faulty, as discussed in Section II-C. For this purpose, it is essential that a node persists all protocol information that is required to avoid inconsistent statements. For example, if a node has acknowledged the reception of a command proposal p for a specific sequence number, it must be ensured that the node remembers this action even after having recovered from an outage in order to prevent the node from later accepting a different proposal $p' \neq p$ for the same sequence number.

Our Contribution. In Section VI, we provide details on TinyBFT's support for additional non-volatile memory modules, discussing both efficiency as well as resilience aspects.

IV. Analysis of BFT Protocol Designs

In this section, we analyze state-of-the-art BFT state-machine replication protocols based on several criteria that should be taken into account when selecting a protocol for the group of device-tier nodes in our target environment.

A. Synchrony Models

One of the most important factors influencing the design of a replication protocol is its synchrony model, that is the assumptions made about the clocks of correct nodes, the duration of local computations, and especially the time it takes messages to be transmitted between correct senders and receivers. In the following, we discuss the three main synchrony models used for BFT state-machine replication [36] in detail.

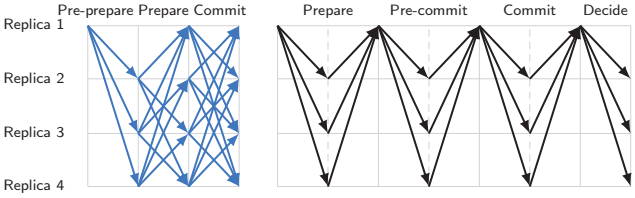
Synchrony. Protocols in this category commonly require the clocks of correct nodes to run at the same rate, and furthermore assume known upper bounds on processing times as well as the transmission delay of messages. As a major benefit, under such conditions it is possible to tolerate f Byzantine faults with

only $2f+1$ replicas [37]. Unfortunately, although these strong assumptions hold in many embedded systems, especially those providing hard real-time guarantees, they are not applicable to our target environments. In particular, as explained in Section II-A, the unreliable wireless network through which device-tier nodes communicate cannot guarantee timely delivery, thereby ruling out the use of the synchronous model.

Partial Synchrony. Protocols designed for this model [38] typically rely on $3f + 1$ nodes [3], [7] and in general do not require any timing assumptions for safety, meaning that the replicated state for example remains consistent even in the presence of arbitrarily long message transmissions. On the other hand, to circumvent the FLP impossibility [39], liveness (i.e., protocol progress) is only ensured if there are phases during which the network behaves synchronously. Specifically, during these phases a worst-case delay for the transmission of messages must exist, although the value of this upper bound does not necessarily have to be known to participating nodes. Protocols such as PBFT [3], which assume their nodes to not possess this knowledge, make up for this by dynamically increasing the lengths of timeouts so that during synchronous phases the message interactions that are needed to make progress (e.g., the election of a new leader) eventually complete before the timeouts protecting them expire.

As detailed in Section II-A, in our target environments the transmission of a message in some cases may take longer than usual as a result of retransmissions being necessary to mask network unreliability or temporary node outages. However, with such phases of asynchrony being limited to short periods of time compared with the overall lifetime of the system, correct nodes in our target environments are predominantly able to communicate with each other in a synchronous fashion, hence making partial synchrony a suitable model for our use cases.

Asynchrony. Unlike protocols designed for partial synchrony, completely asynchronous protocols usually do not depend on any timing assumptions at all, neither for safety nor for liveness. That is, for asynchronous protocols to make progress, it is sufficient if messages of correct senders are eventually delivered to correct receivers, potentially after multiple retransmissions. Unfortunately, offering such weak requirements comes at the cost of increased protocol complexity, and especially a large number of required rounds of node interactions, which normally results in comparably high latencies [40]–[42]. For our highly resource-constrained target systems, both of these aspects are prohibitively expensive



(a) PBFT [3] (b) HotStuff [7]

Figure 3. Comparison of communication patterns

which is why, despite the asynchronous model itself being generally suitable for our application scenarios, we decided not to select an asynchronous protocol as basis for our library. Notice that this should not categorically rule out the use of asynchronous protocols in embedded systems. In fact, their weak assumptions probably make them good candidates for systems in which nodes experience prolonged outages and/or comprise significantly more memory than our tiny devices.

B. Communication Patterns

To reach an agreement on the order in which to execute new operations, the nodes of a replica group repeatedly need to interact by exchanging messages with each other. As illustrated in Figure 3, BFT state-machine replication protocols employ different communication patterns for this purpose. PBFT-style protocols [3], [5], [43], [44], for example, rely on multiple phases in which senders broadcast their messages to all other nodes in the group (see Figure 3a). In contrast, phases in protocols with HotStuff-like designs [7], [45] consist of two separate steps (see Figure 3b): an initial broadcast performed by one node, followed by unicasts of the remaining nodes which all transmit their answers to the same receiver (i.e., typically, but not necessarily the original broadcaster).

Compared with PBFT's approach, HotStuff's communication pattern involves significantly more sequential steps, but on the other hand has the benefit of entailing a lower overhead with regard to the number of transmitted messages (i.e., $O(n)$ instead of $O(n^2)$, with n denoting the group size). Furthermore, HotStuff requires fewer send operations if broadcasts are implemented as a collection of individual unicasts to different receivers [4], as it is often the case in server-based systems in which nodes are linked via point-to-point TCP connections. However, the same does not apply to the environments we target with our work in this paper (see Section II-A). Here, nodes exchange messages using wireless communication that directly offers broadcasts by default, whereas unicasts have to be specifically implemented as broadcasts with only a single active receiver. Consequently, PBFT-style protocols are a better fit for our target systems, especially since they allow us to complete the agreement process in fewer communication steps and hence minimize overall protocol latency.

C. Resource Efficiency

Requiring a group size of at least $3f + 1$ to tolerate up to f faulty nodes, minimizing the resource consumption of BFT state-machine replication is also an important research goal in the context of systems with server-grade nodes [44], [46]. To achieve this, several works have shown that it is possible to reduce the minimum number of necessary nodes to $2f+1$ (or in other words: improve the resilience provided by n nodes from $\lfloor \frac{n-1}{3} \rfloor$ to $\lfloor \frac{n-1}{2} \rfloor$) by equipping each node with a trusted component that is assumed to only fail by crashing. Proposals for the implementation of such a module include a smartcard [47], an FPGA [48], a trusted platform module [49], a virtual machine [49], and a trusted execution environment [50]–[52] such as Intel SGX [53]. Unfortunately, these technologies

are not generally available on our highly resource-constrained target devices. As a result, our focus remains on protocols that do not make limiting trust assumptions but instead tolerate Byzantine faults in all parts of a node.

D. Summary

Our analysis of different designs yielded PBFT-style protocols as the best fit for our embedded target systems, which is why we select PBFT as foundation for our replication library. As a beneficial side effect, this also allows us to profit from a large number of optimizations that have been developed for these kind of protocols over the years [18], including the ordering of operation batches (instead of individual operations) as well as the agreement on hashes (instead of full operations), as further detailed in Section VI-E.

V. Memory-Consumption Analysis

With memory being a scarce resource on our target devices, we need an estimate of how much is actually required to run the PBFT replication protocol that we chose for TinyBFT. In addition, we analyze how close the actual memory consumption of Castro's PBFT implementation [3] is to the determined amount. Since both of these aspects concern specifics of PBFT, we start this section by presenting the protocol in detail.

A. PBFT Overview

Developed for non-embedded systems, PBFT separates the roles of clients and replicas into different entities. Using $3f + 1$ replicas, the protocol is able to tolerate up to f faulty replicas as well as an unlimited number of faulty clients. Clients and replicas in PBFT communicate by exchanging messages that are authenticated and thus enable receivers to detect and ignore messages that have been manipulated, or which do not originate from their alleged sender. In this context, PBFT assumes an adversary to be computationally bound and therefore unable to break cryptographic techniques.

Clients. To access the replicated service, a client issues a request carrying the command to be invoked, and then waits for $f + 1$ matching replies from different replicas before accepting the corresponding result. In the presence of at most f faulty replicas, collecting $f + 1$ responses with matching contents ensures that at least one of those replies originates from a correct replica and hence contains the correct result.

Agreement. One of the replicas in the group serves as *leader* and has the responsibility to initiate the agreement process for new commands, in which the other replicas participate as *followers*. As shown in Figure 3a, PBFT's agreement stage consists of three phases, whose names are commonly also used for the messages sent within them. In the first phase, the leader assigns the lowest unused sequence number to a new command and proposes both in a *pre-prepare* message to its followers. Next, the followers distribute a *prepare* message containing a hash of the leader's proposal. This phase serves as a safeguard to detect cases in which a faulty leader has made conflicting proposals to different followers. The main goal of a replica in the prepare phase is to assemble a so-called *prepare certificate*, which is a set of $2f$ prepares from different followers and a matching pre-prepare; the existence of such a certificate proves that there cannot be other correct followers that continue the process with a different proposal. Having obtained a prepare certificate, a replica enters the final phase and broadcasts a *commit* message. The agreement process is complete once a replica collects a *commit certificate* (i.e., $2f + 1$ matching commits from different replicas), at which point the replica marks the associated command ready for execution.

Checkpoints. In order to be able to perform the agreement process for multiple sequence numbers concurrently, PBFT replicas maintain a window of active sequence numbers. For safety, it is essential that correct replicas keep all protocol state (e.g., certificates) for the sequence numbers inside this window. To move the window forward, and thereby garbage-collect old protocol data, replicas rely on checkpoints of the replicated state. Specifically, they periodically (e.g., every Kth sequence number, with K being a configurable number smaller than the window size) snapshot the replicated state and broadcast a hash of it in a *checkpoint* message. Once a replica assembles a *checkpoint certificate* for a sequence number s in the form of $2f+1$ matching checkpoint messages, it considers the checkpoint stable and increases its window start to $s + 1$.

Besides enabling garbage collection, checkpoints are also an important means to assist correct replicas that have fallen behind (e.g., due to a temporary outage). In such case, the affected replica initiates a *state transfer* during which it fetches the replicated state (stored in the latest checkpoint) from another replica, and afterwards resumes normal-case operation.

View Change. In PBFT, the term *view* refers to the time span a specific leader replica is in charge of initiating agreement. If followers suspect their leader to be faulty, they trigger a view change to assign the leader role to another replica. Similar to the other PBFT mechanisms discussed above, view changes also involve the collection of certificates, in this case to confirm a replica's vote for the view change. Once the new leader takes over, it informs the rest of the group by distributing a *new-view* message.

B. Protocol Analysis

In the following, we present a model for the memory consumption of PBFT replicas. The main purpose of this model is to get a basic estimate of where memory is spent in the protocol, and this way determine a baseline that enables us to assess the memory efficiency of implementations. Notice that the model is not intended to offer completeness and thus should not be used to calculate worst-case bounds, nevertheless the model captures all major sources of memory consumption in PBFT. In our model, we use M_x to denote the maximum size of a message x whose size depends on the specific replicated application and hence is typically configurable; for PBFT, this especially applies to client requests, which is why the sizes of all messages carrying requests are also application-dependent. In contrast, B_y indicates that the maximum size of a message y is fixed, which is the case for all remaining message types.

Client Handling. PBFT assumes a correct client to invoke operations sequentially, meaning that each client has at most one pending request at a time. Consequently, replicas only have to keep the most recent request they received from each client. In addition, to enable retransmissions, replicas also need to preserve their latest reply to each client. Assuming a maximum number of clients U, this results in a client-handling state of

$$S_{client-handling} = U \cdot (M_{request} + M_{reply}) \quad (1)$$

Agreement. As explained in Section V-A, PBFT's agreement process requires replicas to assemble certificates consisting of pre-prepare, prepare, and commit messages. For each sequence-number slot in its agreement window of size W, a replica must retain one prepare certificate (of maximum size $C_{prepare}$) and one commit certificate (of maximum size C_{commit}). Overall, this leads to the following memory consumption for the agreement stage:

$$C_{prepare} = M_{pre-prepare} + 2f \cdot B_{prepare} \quad (2)$$
$$C_{commit} = (2f+1) \cdot B_{commit} \quad (3)$$
$$S_{agreement} = W \cdot (C_{prepare} + C_{commit}) \quad (4)$$

Checkpoints. Checkpointing makes it necessary for a replica to keep multiple copies of the application state, which is of configurable maximum size A. Although techniques such as copy-on-write [3], [54] typically enable a reduction of the average snapshot size in practice, without additional assumptions (e.g., an upper bound on the number of state values that are modified between two checkpoints) they have no impact on the worst case. Consequently, for our conservative calculations we consider full state snapshots. Within the agreement window, there exist $\frac{W}{K}$ sequence numbers that trigger checkpoint creation for a checkpoint interval K. Furthermore, the latest stable checkpoint itself has to be kept as well:

$$S_{snapshots} = (\frac{W}{K} + 1) \cdot A \quad (5)$$

In addition, each time a replica creates a checkpoint, it seeks to obtain a checkpoint certificate. Combined with the main application instance, this results in the following memory consumption for the state of a replica's execution stage:

$$C_{checkpoint} = (2f+1) \cdot B_{checkpoint} \quad (6)$$
$$S_{execution} = S_{snapshots} + \frac{W}{K} \cdot C_{checkpoint} + A \quad (7)$$

View Change. A replica votes for a view change by distributing a message of size M_{vc}, which the receiver then attests by itself broadcasting an acknowledgment of size B_{vc-ack}. Together with the original message, $2f$ of these acknowledgments confirm the distribution of the vote. Once the view change is complete, the new leader starts its term by publishing a new-view message of size M_{nv}. Due to the fact that view-change efforts for multiple views may occur concurrently (e.g., as a result of network issues), a replica must be prepared to store the latest view-change-related messages of all $3f+1$ replicas in the system:

$$C_{vc-vote} = M_{vc} + 2f \cdot B_{vc-ack} \quad (8)$$
$$S_{view-change} = (3f+1) \cdot (C_{vc-vote} + M_{nv}) \quad (9)$$

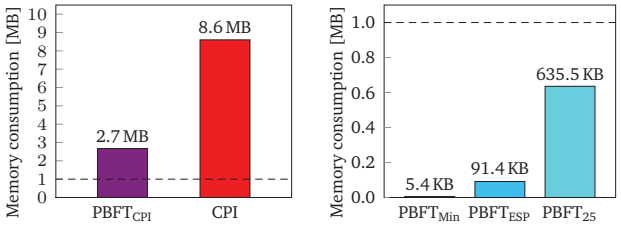
C. Implementation Analysis

Next, we complement our theoretical analysis with the study of a practical codebase, namely Castro's PBFT implementation [3], which in the following we refer to as CPI. We selected CPI for this purpose because despite its age it is still a rare example of an implementation that is open source, written in a language that fits the needs of our target environments (i.e., C/C++), and implements the PBFT protocol in its entirety; unfortunately, especially the later criterion ruled out many more recent research prototypes (e.g., Themis [55]). Below, we summarize the most important findings of our study, which then enable us to assess CPI's memory consumption based on our model presented in Section V-B.

Certificate Logs. To maintain prepare, commit, and checkpoint certificates, CPI relies on three instances of a log-based data structure that allocates memory for new messages on demand; view-change information is managed separately. In the worst case, the prepare and commit certificate logs contain one entry for each sequence-number slot in the agreement window, whereas for the checkpoint certificate log CPI allocates twice as many entries. The latter design decision is interesting for two reasons: (1) Although periodic checkpoints are only created for every Kth sequence number, CPI keeps (empty) certificates available also for other window slots. After a further analysis of the code, we believe that this is a by-product of the fact that CPI's state-transfer mechanism for reasons of efficiency involves the creation of an exceptional checkpoint, which may occur at an arbitrary sequence number. (2) CPI allocates checkpoint certificates for two full agreement windows instead of one. Presumably, this is due to a replica having to store the latest checkpoint for a sequence number s, which at this point is below the start of the replica's agreement window at $s+1$. In both cases, the design decisions made for the CPI implementation appear to have been mainly driven by convenience, but come at the cost of significant additional memory consumption.

State and Checkpoint Management. CPI partitions the state of the replicated application into equal-size blocks of 4 KiB and organizes these blocks in a data structure that resembles a Merkle tree [56]. As key advantages, this approach allows a replica to create efficient snapshots using copy-on-write techniques and to quickly compute snapshot hashes (as required for checkpoint messages, see Section V-B). Furthermore, a tree-based state management lays the foundation for an efficient state transfer by enabling replicas to identify newly modified blocks and fetch only those from other replicas [3].

CPI's partition tree has a fixed height of 4, and each non-leaf node a fixed number of 256 children. As a result, the partition tree provides the necessary metadata for managing an



(a) CPI default configuration (b) Memory-efficient configurations

Figure 4. Memory-consumption estimates for different configurations

application state of up to 64 GiB, which is an example of CPI's focus on server-grade hardware. In addition to the partition tree, CPI maintains a second tree of similar structure storing previously computed hashes of individual blocks and partitions. In this context, the fact that the structure of both trees is fixed (and hence requires a replica to always allocate all metadata objects needed for a 64 GiB state) leads to memory being wasted for replicated applications with small states.

When creating a new checkpoint, CPI stores the used parts of the current partition tree into a dedicated checkpoint record. By default, this record data structure initially comprises space for pointers to 256 blocks, even if the application state (for example) fits into a single block. With checkpoint records being organized in a similar log as checkpoint certificates, the record log suffers comparable problems of allocating memory for sequence numbers for which usually no checkpoints are created.

Comparison. Leveraging our model from Section V-B, we can now assess the memory efficiency of CPI's implementation of PBFT. For this purpose, we configure our model with CPI's default parameters to determine a baseline memory consumption PBFT$_{CPI}$ and compare this value to CPI's actual memory consumption (see Figure 4a). Based on the results we make two important observations: (1) Using CPI's default configuration, there is no possibility to keep memory consumption under the threshold we target in our work (i.e., 1 MB), which again is a consequence of CPI focusing on non-embedded systems. (2) Due to CPI's excessive allocation of memory, especially in the context of state and checkpoint management (see Section V-C), its actual memory consumption exceeds the baseline by more than a factor of 3, thereby making the implementation unsuitable for our target devices. Notice that this problem cannot be easily solved by applying a smaller configuration, because CPI's partition and hash trees alone consume more than 7.3 MB of memory, an amount that is not affected by configuration parameters.

D. Exploring Memory-Efficient Configurations

Apart from determining a baseline for the memory efficiency of implementations, our model can also be used as a tool to explore the vast space of possible configurations. Applying this approach, we searched for configurations that (in contrast to PBFT$_{CPI}$) can actually be deployed on highly resource-constrained devices. In this context, we identified three particular configurations of interest (see Figure 4b).

PBFT$_{Min}$. For the PBFT$_{Min}$ configuration, we set all parameters to the lowest values that still result in a functioning PBFT protocol execution, which allows us to study the smallest possible memory footprint. Our calculations show that this configuration consumes about 5.4 KB of memory. The fact that this value is well below our threshold objective (1 MB) indicates a significant degree of flexibility when its comes to developing custom configurations for our target systems.

PBFT$_{ESP}$. In a next step, our goal is to find a suitable configuration for the ESP32-C3 platform on which we conduct our experimental evaluation in Section VII. These devices are equipped with 400 KB RAM [14] of which the processor reserves 16 KB as cache, leaving 384 KB of memory for further allocations. The remaining amount is not exclusive to the replication library but also has to be used for fundamental components and services, including for example the operating system as well as the WiFi and IP stacks. In the end, this leaves at most 170 KB of memory for the replication library.

Using our model, we designed a configuration that meets this requirement while supporting applications with a comparably large state of up to 16 KiB, which can be accessed or modified with 1 KiB requests/replies. The configuration relies on an agreement window comprising $W = 4$ slots and a checkpoint interval of $K = 2$. Although significantly smaller than the windows typically used in server-based BFT systems (CPI's default window size is 256, for example), such small windows better fit our use-case scenarios. Specifically, with our replicas in general representing the only clients, the number of requests that are concurrently issued is usually low. Consequently, a large agreement window would solely occupy additional resources without offering major benefits.

PBFT$_{25}$. With our third configuration, we investigate the impact of the replica-group size on memory consumption by extending the number of participants from 4 to 25 replicas. Utilizing more than 600 KB of memory, the protocol state for such a large group size no longer fits into the internal RAM of our ESP32-C3 devices, but could still be supported by making use of additional memory modules (see Section VI-C).

VI. TinyBFT

This section presents TinyBFT, the first BFT state-machine replication library for highly resource-constrained devices. In particular, we provide details on our techniques to significantly reduce memory usage compared with CPI, and on TinyBFT's support for additional memory modules.

A. Static Memory Allocation

Unlike the vast majority of existing BFT replication libraries (e.g., [3], [4]), all memory consumed by TinyBFT is allocated statically, which follows the best practices in (safety-critical) embedded systems [57] and enables us to a priori determine an upper bound for TinyBFT's memory consumption. To achieve this, we designed a novel memory layout consisting of four regions with clearly defined purposes. In the following, we present each of these regions and elaborate on the type of information that TinyBFT manages in them.

Agreement Region. The agreement region hosts an array of the agreement window size W, in which each element reserves space for exactly one prepare certificate and one commit certificate. Treating the array as a ring buffer, when a replica shifts its window to a higher starting sequence number, TINYBFT reinitializes the slots that are now below the window and immediately assigns them to higher sequence numbers. As a key benefit, this approach allows TINYBFT to efficiently move the window without performing memory (de)allocations.

Checkpoint Region. Exploiting the insight that a replica has to keep at most $\frac{W}{K}+1$ checkpoints at the same time (see Equation 5), TINYBFT's checkpoint region stores exactly this number of checkpoint certificates. Apart from the certificates, this region also reserves space for one additional checkpoint message from each other replica in the system. Retaining the latest checkpoint message received from each peer allows a replica to collect information about checkpoints for sequence numbers above its current agreement window, and thereby detect scenarios in which the replica has fallen behind and hence should request a state transfer.

Event Region. In contrast to the agreement and checkpoint region, the event region is dedicated to messages whose lifetime is not directly related to the agreement window. Examples include newly submitted requests, as well as messages related to TINYBFT's view-change and state-transfer mechanisms.

Scratch Region. While the three memory regions discussed so far store messages until they are no longer needed, TINYBFT's scratch region is used for holding messages temporarily. Initially, each message created or received by a replica is placed here. Once the replica determines that a message has to be retained (e.g., as part of a particular certificate), it copies the message to one of the other three regions, meaning that at this point the corresponding part of the scratch region can be reused for the next incoming or outgoing message. Structurally, the scratch region manages a fixed number of slots that each provide space for up to the maximum message size. Hence, as in the other three statically allocated regions, memory fragmentation does not become a problem.

Although the involvement of the scratch region comes at the cost of an additional copy operation if a message is kept, the region provides more flexibility regarding the storage location of messages. Specifically, as further discussed in Section VI-C, it enables TINYBFT to transparently support non-volatile memory modules by moving the other three regions there. With the additional memory typically not being directly accessible in the microprocessor's address space, a copy operation in this scenario is required anyway.

B. State Management

Similar to CPI, TINYBFT also relies on a partition tree to manage application-state blocks, because this approach allows us to reuse CPI's optimized state-transfer mechanism (which only transmits blocks whose contents have changed) without further modification. However, TINYBFT's partition tree differs from its CPI counterpart in two important aspects: (1) With our target devices having just small amounts of memory available, we reduce the height of TINYBFT's partition tree to the minimum required to manage an application's state. For our example configuration PBFT_{ESP} a height of 2 is sufficient, meaning that all blocks are directly connected to the root node. As a main benefit, compared with CPI's four-level tree, this enables us to significantly minimize metadata overhead for managing application states of highly resource-constrained systems. (2) TINYBFT's partition tree only reserves space for blocks that are actually needed to handle an application state of the statically configured size, which further reduces TINYBFT's memory footprint compared with CPI.

C. Support for Additional Non-Volatile Memory

As discussed in Section III, our target devices offer the possibility to add memory modules providing persistent storage, and hence enable a replica to resume operation after an outage. TINYBFT facilitates this process by allowing its agreement, checkpoint, and event memory regions to be moved to such modules. In contrast to the scratch region, these three regions contain state that needs to be kept available across outages in order for a correct replica to remain correct.

Entailing higher latency than accesses to internal RAM, we designed TINYBFT to minimize the number of accesses to persistent memory modules. To achieve this, a replica initially places all incoming messages in the scratch region (i.e., in internal RAM). In a next step, the replica then performs several checks to (1) validate that the message is both well-formed as well as properly authenticated, and (2) determine whether the message contains relevant information that enables the replica to make progress. The latter especially aims at messages that arrive at a point in time at which a corresponding decision has already been made. For example, with $2f+1$ matching commits being sufficient to assemble a commit certificate (see Section V-A), in the presence of $3f+1$ replicas there are usually up to f commits that do not contribute to the completion of the certificate. Only if a message passes all checks a replica moves the message to its destination in one of the other three regions. Otherwise, a replica discards it without the message ever touching persistent memory.

D. Memory-Efficient Configuration

As shown by our analysis in Section V-B, PBFT's memory consumption to a significant extent depends on the values of a set of configurable variables, including for example the number of replica faults to tolerate, the size of the agreement window, and the checkpoint interval; TINYBFT adds further variables such as a maximum size for individual messages as well as the size of application-state blocks. Freely tuning these parameters makes it possible to flexibly trade off space-efficiency for performance, thereby providing an effective means to tailor TINYBFT to the particular characteristics and requirements of the target platform and application.

To protect TINYBFT against invalid configurations, we added a mechanism to its build process that automatically checks dependencies between parameters and in case of

conflicts triggers compiler errors. Our solution relies on CMake [58], a tool used in many embedded development platforms and real-time operating systems [59], [60]. If, for example, a user tries to change the maximum size of individual messages to 1 KiB (by adjusting the corresponding CMake variable) while leaving the size of state blocks at its default of 4 KiB, this would constitute a conflict, because as part of state transfer a replica must be able to transmit blocks over the network. Consequently, static assertions in our code in such case result in a compile error indicating that the block size is too large for the maximum message size, thereby preventing TinyBFT from being deployed with an invalid configuration.

E. Implementation

For our TinyBFT implementation, we reuse parts of CPI's codebase such as the agreement protocol logic and the mechanisms for checkpointing, view change, and state transfer. In addition to the design changes described in Sections VI-A through VI-D, we also integrate a new crypto library (MbedTLS [61]) to meet the requirements of modern systems. MbedTLS is part of the trusted firmware project [62], especially targets embedded systems, and therefore already supports a large variety of embedded platforms.

To account for the heterogeneity of embedded platforms, we designed the TinyBFT implementation as a layered architecture in which the actual library resides on top of a small hardware abstraction layer. As its main responsibility, this layer provides transparent access to platform-dependent hardware components such as a system's cycle counter and real-time clock, which are for example used to monitor view-change timeouts. With the hardware abstraction layer providing a stable function call interface to the library, only this small layer has to be implemented for each target platform, thereby making it significantly easier to port TinyBFT to new devices.

TinyBFT offers all common PBFT optimizations [3], which among others include the following: (1) To minimize consensus overhead, the leader may combine multiple requests into a batch, which from this point on is treated as one (large) request and thus assigned only a single sequence number. Once the batch is ready for execution, a replica processes the corresponding commands in the order in which they appear in the batch, thereby guaranteeing consistency among replicas. (2) To take load off the leader, TinyBFT enables replicas to apply hash-based ordering. Using this optimization, a replica directly broadcasts its own requests to all other replicas in the system, which in turn allows the leader to only include request hashes (instead of the full requests) into its pre-prepare messages. As a consequence, pre-prepare messages become smaller and the load associated with the transmission of full requests is distributed more equally across replicas.

VII. Evaluation

In this section, we evaluate the performance and memory consumption of TinyBFT using a replicated key-value store, which (as discussed in Section II-B) can serve as basis for resilient control applications in our target environments.

A. Performance Comparison with CPI

In the first experiment, we study the performance impact of our design decisions presented in Section VI, using CPI as baseline. As explained in Section V-C, due to its high memory consumption CPI does not fit on our target devices, which is why for this experiment we resort to server-grade hardware (Intel Xeon E3-1275v5, 3.6 GHz, 4 cores, 16 GB RAM, 1 Gbit Ethernet). To enable a fair comparison, we run CPI with the same configuration (i.e., PBFT_{ESP}, see Section V-D) and crypto library (i.e., MbedTLS, see Section VI-E) as TinyBFT.

The workloads in this experiment consist of reads and writes of different sizes. Since the four replicas themselves are not sufficient to issue the necessary amount of concurrent requests for this purpose, we rely on a set of 250 external clients. Each replica is hosted on a dedicated server, while the clients are co-located on an additional machine. All reported numbers represent the average of three runs.

Figure 5 presents the measurement results obtained from this experiment in the form of the relationship between throughput (on the horizontal axis) and provided latency (on the vertical axis). Along the plotted lines, we gradually increase the workload by instructing clients to send requests at higher frequency. For small and medium workloads, the latency achieved by both TinyBFT and CPI remains low. After a system reaches saturation, latency continues to increase while throughput stagnates or even decreases; as shown in the bottom graphs of Figure 5, this effect can result in multiple latency values being reported for the same throughput. Independent of the workload evaluated as part of this experiment, TinyBFT achieves a similar performance as CPI. For this reason, we conclude that our newly designed memory layout and partition tree have no measurable performance overhead.

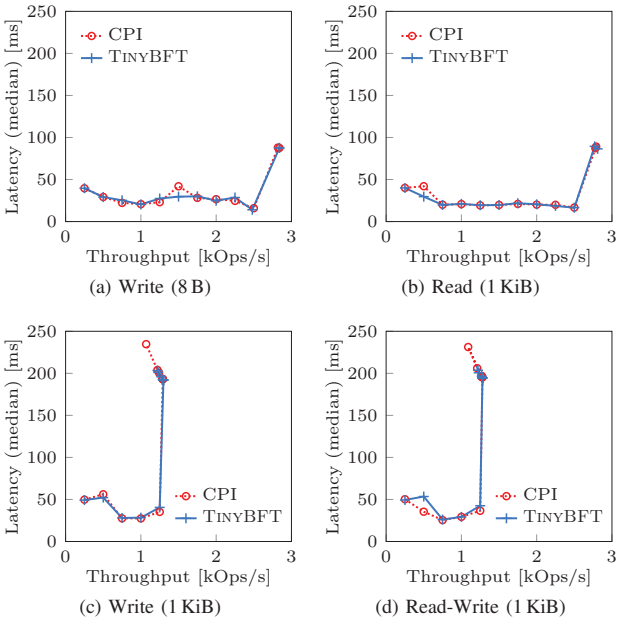


Figure 5. Performance comparison between CPI and TinyBFT

B. Performance on Tiny Devices

For our next set of experiments, we deploy each TINYBFT replica on an actual node of our target platform (ESP32-C3, 160 MHz CPU, 2.4 GHz WiFi 802.11 b/g/n). Once again, we measure throughput and latency for different operations modifying and/or retrieving entries managed by the replicated key-value store application. In addition to micro benchmarks, which primarily focus on individual operations, we also study more complex access patterns using YCSB [63], a benchmark suite designed to evaluate typical real-world use cases.

Micro Benchmarks. Offering a group of devices the opportunity to perform state-machine replication without the help of the outside world, in TINYBFT's target use cases the replicas themselves are usually the only nodes issuing commands to the replicated service (see Section II-B). To account for this characteristic, in this experiment we set the maximum number of concurrent requests to the number of replicas. Furthermore, we analyze different workloads by configuring individual targets for the throughput demanded by the service.

As shown by the measurement results presented in Figure 6, despite the small amounts of resources available, TINYBFT is able to process workloads with continuously low latency of about 50 ms for small commands and about 80 ms for large commands. This even applies to high-utilization scenarios in which a new request is immediately issued after the previous one completed. In this context, we observed throughputs of up to 31 operations per second for writes and up to 43 operations per second for reads. Overall, this experiment reveals that TINYBFT is able to provide good performance on highly resource-constrained embedded platforms such as ESP32-C3.

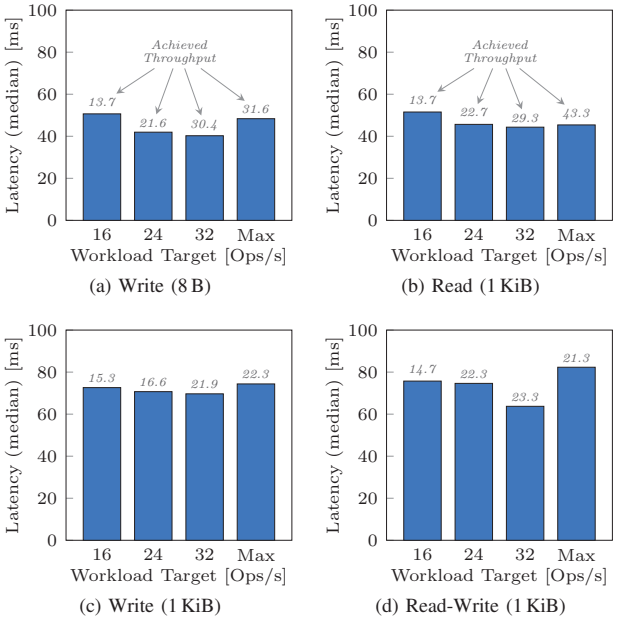


Figure 6. TINYBFT performance on ESP32-C3

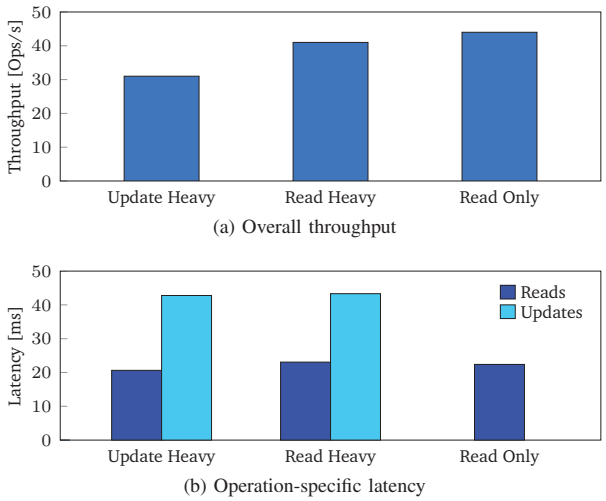


Figure 7. YCSB benchmark results on ESP32-C3

YCSB. In the next experiment, we rely on the YCSB [63] benchmark suite to evaluate TINYBFT with a variety of access patterns. For this purpose, we implement an adapter that enables the YCSB client-side implementation to interact with the replicated key-value store provided by our group of tiny devices. As values, YCSB uses hierarchical 256 B records that each comprise multiple fields. For retrieving and modifying one or more fields of a record, the adapter offers two dedicated methods: `read()` and `update()`.

As part of our study we examine the following scenarios: YCSB's *update-heavy* benchmark issues equal shares of reads and updates (i.e., 50 % each). In the *read-heavy* benchmark, the vast majority of operations are reads (95 %), mixed up with some occasional updates (5 %). Finally, the *read-only* benchmark performs no state modifications at all, but solely fetches record fields from the key-value store.

The results of this experiment in Figure 7 indicate that updates take about twice as long as reads to complete, which is a consequence of the fact that at the adapter level each `update()` call involves two commands to the TINYBFT library: one to fetch the current version of the record, and a subsequent one to write back the modified version. Therefore, the throughput achieved by a benchmark to a significant degree depends on its ratio of reads to updates. With regard to absolute numbers, the measurements confirm TINYBFT's high efficiency, both in terms of throughput as well as latency.

C. Memory Consumption

In our fourth experiment, we measure the memory consumptions of the evaluated systems immediately after the initialization process of each implementation is complete. Notice that this method favors CPI because due to its dynamic allocation of messages, its memory consumption typically increases once the system actually processes requests. In contrast, TINYBFT allocates all of its memory statically, meaning that the reported numbers remain constant over the entire system lifetime, independent of the workload that needs to be handled.

Table I
MEMORY CONSUMPTION (IN BYTES) AFTER INITIALIZATION

Protocol Part	CPI (Intel)	TINYBFT (Intel)	TINYBFT (ESP32-C3)
Client handling	5,952	12,428	11,460
Agreement	1,320	11,684	11,908
Execution	7,395,588	110,448	115,780
View	1,992	7,688	7,920
Other	554,984	40,320	18,447
Total	**7,959,872**	**181,604**	**165,515**

Based on the results shown in Table I, we make the following observations: (1) On ESP32-C3, TINYBFT consumes significantly less memory compared with the Intel setting, which can be mainly attributed to smaller `libc` buffers for file parsing. (2) The largest memory increase in comparison to our baseline model pertains to TINYBFT's execution stage, which (as discussed in Section VI-B) was a deliberate choice in order to optimize state transfer. (3) Despite both TINYBFT and CPI using the same configuration, TINYBFT's overall memory footprint is 97.7 % smaller than the memory footprint of CPI, confirming the effectiveness of our memory-efficient design.

D. Energy-Consumption Considerations

Although the fact that the ESP32-C3 platform uses WiFi is not an ideal fit for low-power communication in highly energy-constrained settings (compared to systems such as Bluetooth LE or LoRa), it allows us to give an approximation of a node's power demand when running TINYBFT. To approximate the minimum lifetime of a battery-operated TINYBFT system, we use the maximum reported peak-current estimates for the embedded platform [64]. For this scenario, we assume the transmission mode to be continuously active for the agreement-process duration of about 80 ms (see Section VII-B). Here, the specific transmission mode is IEEE 802.11n (HT20, MCS7 with transmission power of 18.5 dBm) and the peak current drawn is 276 mA. During the rest of the time, the system remains in light-sleep mode (i.e., 130µA) and wakes up once data is received with the support of 802.11's delivery traffic indication messages. Under the assumption of a single standard 18650 lithium-ion battery with 3,500 mAh, we approximate the lifetime of each node, with the workload of one agreement per hour, to be around 1,000 days. This approximation shows that TINYBFT is practical for resource-constrained settings. Moreover, using communication technologies designed specifically for (long-range) low-power communication would help TINYBFT to further increase the battery lifetime.

VIII. RELATED WORK

For many years, the avionics industry has been a major driving force when it comes to applying Byzantine fault tolerance in embedded and real-time systems, oftentimes solving the associated problems by relying on additional hardware components [9], [10]. More recently, these approaches were complemented by software-based solutions. Gujarati et al. [13], for example, presented a highly reliable key-value store that can be hosted on commodity embedded platforms and serve as basis for real-time control applications. Fröhlich et al. [15] and Kozhaya et al. [16], [17] developed reliable broadcast protocols for cyber-physical systems. Bhat et al. [65] proposed a BFT atomic broadcast protocol that is designed with a focus on energy efficiency and, in contrast to our work, requires a synchronous network with upper bounds on message delivery. Loveless et al. [11] leveraged speculation to improve the latency of BFT state-machine replication in embedded systems with multi-core processors. With TINYBFT, we join the efforts to further improve the resilience of embedded systems by providing a library that enables BFT state-machine replication on tiny devices.

TINYBFT benefits from the heterogeneity of memory-storage technologies, which are available on modern embedded platforms. In this context, Jayakumar et at. [66] present memory-mapping strategies for embedded systems that operate with both internal volatile and non-volatile RAM. The focus of their work is to reduce the energy demand of the targeted systems. Their setup is based on TI's MSP430FR57 microcontroller [67], a widely used platform in the domain of intermittent computing [68]–[70], where power outages are considered to be frequent. Unfortunately, the size of the internal FRAM of this platform is very limited with 16 KiB. When reconsidering TINYBFT's possible configurations from Section V-C, only the PBFT_{Min} configuration could run entirely from the available FRAM. Addressing hardware setups that support all variants of internal/external and volatile/non-volatile RAM, we are more flexible with choosing suitable embedded hardware platforms for TINYBFT. In any case, TINYBFT profits from the increasing availability of non-volatile RAM platforms in order to further support fine-grained checkpointing with the goal of energy-demand reduction.

IX. CONCLUSION

TINYBFT is the first library that enables highly resource-constrained embedded devices to provide resilience against arbitrary faults by performing BFT state-machine replication. Taking the specific characteristics of our target environments into account, we selected PBFT as replication protocol for our library and developed a memory-efficient implementation which, thanks to static allocation, is able to guarantee a worst-case memory consumption that is known at compile time. Based on our evaluation results, we conclude that (1) our design choices allow TINYBFT to reduce its memory footprint without having a measurable impact on performance and (2) TINYBFT offers comparably low response times of less than 100 ms when executed on tiny microcontrollers.

The source code of TINYBFT *is available at:*
`https://gitos.rrze.fau.de/tinybft`

Acknowledgments: We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) under grant number 20E2122B (LuFo VI-2, BALu) and by the German Research Foundation (DFG) under project number 502947440 (Watwa).

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[2] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[3] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, 1999, pp. 173–186.

[4] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*, 2014, pp. 355–362.

[5] J. Behl, T. Distler, and R. Kapitza, "Consensus-oriented parallelization: How to earn your first million," in *Proceedings of the 16th Middleware Conference (Middleware '15)*, 2015, pp. 173–184.

[6] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[7] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 38th Symposium on Principles of Distributed Computing (PODC '19)*, 2019, pp. 347–356.

[8] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From Byzantine replication to blockchain: Consensus is only the beginning," in *Proceedings of the 50th International Conference on Dependable Systems and Networks (DSN '20)*, 2020, pp. 424–436.

[9] A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP—A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, 1978.

[10] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai, "The MAFT architecture for distributed fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 398–404, 1988.

[11] A. Loveless, R. Dreslinski, B. Kasikci, and L. T. X. Phan, "IGOR: Accelerating Byzantine fault tolerance for real-time systems with eager execution," in *Proceedings of the 27th Real-Time and Embedded Technology and Applications Symposium (RTAS '21)*, 2021, pp. 360–373.

[12] C. Berger, H. P. Reiser, F. J. Hauck, F. Held, and J. Domaschka, "Automatic integration of BFT state-machine replication into IoT systems," in *Proceedings of the 18th European Dependable Computing Conference (EDCC '22)*, 2022, pp. 1–8.

[13] A. Gujarati, N. Yang, and B. B. Brandenburg, "In-ConcReTeS: Interactive consistency meets distributed real-time systems, again!" in *Proceedings of the 43rd Real-Time Systems Symposium (RTSS '22)*, 2022, pp. 211–224.

[14] Espressif Systems, *ESP32-C3 Series Datasheet*, https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf, 2023.

[15] A. A. Fröhlich, R. M. Scheffel, D. Kozhaya, and P. E. Veríssimo, "Byzantine resilient protocol for the IoT," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2506–2517, 2018.

[16] D. Kozhaya, J. Decouchant, and P. Esteves-Veríssimo, "RT-ByzCast: Byzantine-resilient real-time reliable broadcast," *IEEE Transactions on Computers*, vol. 68, no. 3, pp. 440–454, 2019.

[17] D. Kozhaya, J. Decouchant, V. Rahli, and P. Esteves-Veríssimo, "Pistis: An event-triggered real-time Byzantine-resilient protocol suite," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2277–2290, 2021.

[18] T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective," *ACM Computing Surveys*, vol. 54, no. 1, 2021.

[19] E. Roth and A. Haeberlen, "Do not overpay for fault tolerance!" in *Proceedings of the 27th Real-Time and Embedded Technology and Applications Symposium (RTAS '21)*, 2021, pp. 374–386.

[20] Infineon Technologies AG, "CYW43439 – 1x1 single-band Wi-Fi 4 (802.11n) + Bluetooth 5.2 combo," 2023.

[21] M. Heene, S. Hill, and J. Jelemensky, "Queued serial peripheral interface for use in a data processing system," 1989, US Patent 4,816,996.

[22] NXP Semiconductors, "I^{2}C-bus specification and user manual," https://web.archive.org/web/20210813122132/https://www.nxp.com/docs/en/user-guide/UM10204.pdf, 2014.

[23] MIPI Alliance, Inc., "MIPI I3C & MIPI I3C Basic," 2018.

[24] Fujitsu Semiconductor, *FRAM MB85RC256V*, https://www.fujitsu.com/uk/Images/MB85RC256V-20171207.pdf, 2013.

[25] Espressif Systems, "ESP-IDF programming guide – Support for external ram," https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/external-ram.html, 2023.

[26] P. Perazzo, F. Righetti, M. La Manna, and C. Vallati, "Performance evaluation of attribute-based encryption on constrained IoT devices," *Computer Communications*, vol. 170, pp. 151–163, 2021.

[27] P. Wägemann, T. Distler, H. Janker, P. Raffeck, V. Sieh, and W. Schröder-Preikschat, "Operating energy-neutral real-time systems," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 1, pp. 11:1–11:25, 2017.

[28] Semtech, "LoRa – The ultimate long-range solutions," https://www.semtech.com/uploads/design-support/SG-SEMTECH-WSP.pdf, 2018.

[29] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, 2008.

[30] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," *CoRR*, vol. abs/1707.01873, 2017.

[31] P. Wägemann, F. Harbecke, B. Heinloth, H. Hofmeier, and W. Schröder-Preikschat, "An energy-neutral, WiFi-connected room display with hand-crank–based energy harvesting," 2019.

[32] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM*, vol. 32, no. 4, pp. 824—840, 1985.

[33] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, no. 2, pp. 125–143, 1977.

[34] T. Distler, M. Eischer, and L. Lawniczak, "Micro replication," in *Proceedings of the 53rd International Conference on Dependable Systems and Networks (DSN '23)*, 2023, pp. 123–137.

[35] Espressif Systems, *ESP32 Series Datasheet*, https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf, 2023.

[36] C. Berger, S. Schwarz-Rüsch, A. Vogel, K. Bleeke, L. Jehl, H. P. Reiser, and R. Kapitza, "SoK: Scalability techniques for BFT consensus," in *Proceedings of the 5th International Conference on Blockchain and Cryptocurrency (ICBC '23)*, 2023, pp. 1–18.

[37] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync HotStuff: Simple and practical synchronous state machine replication," in *Proceedings of the 41st Symposium on Security and Privacy (SP '20)*, 2020, pp. 106–118.

[38] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, p. 288–323, 1988.

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

[40] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proceedings of the 23rd Conference on Computer and Communications Security (CCS '16)*, 2016, pp. 31–42.

[41] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous BFT protocols," in *Proceedings of the 27th Conference on Computer and Communications Security (CCS '20)*, 2020, pp. 803–818.

[42] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding Dumbo: Pushing asynchronous BFT closer to practice," in *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS '22)*, 2022.

[43] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? Byzantine fault tolerance with a spinning primary," in *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS '09)*, 2009, pp. 135–144.

[44] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2807–2819, 2016.

[45] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: A scalable and decentralized trust infrastructure," in *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN '19)*, 2019, pp. 568–580.

[46] M. Correia, N. F. Neves, and P. Veríssimo, "BFT-TO: Intrusion tolerance with less replicas," *The Computer Journal*, vol. 56, no. 6, pp. 693–715, 2013.

[47] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems," in *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (OSDI '09)*, 2009, pp. 1–14.

[48] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient

Byzantine fault tolerance," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, 2012, pp. 295–308.
[49] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient Byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2013.
[50] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: SGX-based high performance BFT," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*, 2017, pp. 222–237.
[51] W. Xu and R. Kapitza, "RATCHETA: Memory-bounded hybrid Byzantine consensus for cooperative embedded systems," in *Proceedings of the 37th Symposium on Reliable Distributed Systems (SRDS '18)*, 2018, pp. 103–112.
[52] W. Xu, S. Rüsch, B. Li, and R. Kapitza, "Hybrid fault-tolerant consensus in asynchronous and wireless embedded systems," in *Proceedings of the 22nd International Conference on Principles of Distributed Systems (OPODIS '18)*, 2018.
[53] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *Proceedings of the 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*, 2013.
[54] M. Eischer, M. Büttner, and T. Distler, "Deterministic fuzzy checkpoints," in *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*, 2019, pp. 153–162.
[55] S. Rüsch, K. Bleeke, and R. Kapitza, "Themis: An efficient and memory-safe BFT framework in Rust," in *Proceedings of the 3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL '19)*, 2019, pp. 9–10.
[56] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology (CRYPTO '87)*, 1988, pp. 369–378.
[57] MISRA Consortium, *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)*, 10 2004.
[58] CMake, https://cmake.org/.
[59] Espressif ESP-IDF, https://github.com/espressif/esp-idf.
[60] Zephyr, https://zephyrproject.org/.
[61] MbedTLS, https://github.com/Mbed-TLS/mbedtls.
[62] Trusted Firmware, https://www.trustedfirmware.org/.
[63] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*, 2010, pp. 143–154.
[64] Espressif Systems, *ESP32-C3 Wireless Adventure: A Comprehensive Guide to IoT*, 2023.
[65] A. Bhat, A. Bandarupalli, M. Nagaraj, S. Bagchi, A. Kate, and M. K. Reiter, "EESMR: Energy efficient BFT-SMR for the masses," in *Proceedings of the 24th International Middleware Conference (Middleware '23)*, 2023.
[66] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan, "Energy-aware memory mapping for hybrid FRAM-SRAM MCUs in intermittently-powered IoT devices," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 3, pp. 1–23, 2017.
[67] Texas Instruments Inc., *MSP430FR57xx Family – User's Guide*, 2018.
[68] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL '17)*, vol. 71, 2017, pp. 8:1–8:14.
[69] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI '18)*, 2018, pp. 129–144.
[70] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *Proceedings of the 28th Real-Time and Embedded Technology and Applications Symposium (RTAS '22)*, 2022, pp. 40–54.