CC-BY 4.0. This is the author's version of the work. The definitive version is published in the proceedings of the 2022 52th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2022).

ZUGCHAIN: Blockchain-Based Juridical Data Recording in Railway Systems

Signe Rüsch*, Kai Bleeke*, Ines Messadi*, Stefan Schmidt*,
Andreas Krampf‡, Katharina Olze‡, Susanne Stahnke†, Robert Schmid¶, Lukas Pirl¶,
Roland Kittel§, Andreas Polze¶, Marquart Franz†, Matthias Müller‡, Leander Jehl*‖, and Rüdiger Kapitza*

*TU Braunschweig, Germany, rueesch@ibr.cs.tu-bs.de, kapitza@ibr.cs.tu-bs.de
†Siemens AG, Germany ‡Siemens Mobility, Germany §DB Systel, Germany ¶HPI, University of Potsdam, Germany
‖University of Stavanger, Norway, leander.jehl@uis.no

***Abstract*—In modern trains, a juridical recording unit logs events that occur during operation. This data is used to reconstruct the exact chain of events in case of failures and crashes. To ensure data recovery after an accident, the recorder is hardened against physical damage and secured against tampering; however, it is a single proprietary device and by no means indestructible.**

This paper presents ZUGCHAIN, a distributed, blockchain-based juridical recording unit that opportunistically utilizes on-train hardware. ZUGCHAIN offers high reliability via replication and tamper-resistance due to the nature of blockchains. It implements a permissioned blockchain based on a Byzantine fault-tolerant agreement protocol suitable for diverse communication systems. To utilize the logged data for advanced services, e. g., predictive maintenance, ZUGCHAIN securely and continuously exports traces to private data centers. We demonstrate ZUGCHAIN's feasibility with an implementation running on real train hardware, where we show that ZUGCHAIN orders data within 14 ms using at maximum 15 % of the total available shared CPU resources, thus fulfilling requirements of juridical recorders.

***Index Terms*—railway, BFT, event recorder, blockchain**

I. INTRODUCTION

Trains, similar to planes and other safety-critical systems, have a "black box" that logs any activity in the form of distinct, configured events to facilitate investigations of accidents and incidents. The events that must be recorded include e. g., speed, brake activation, and door activity with their corresponding timestamps, as specified in IEC 62625 [1]. The data collected in such a black box, a so-called *juridical recording unit (JRU)*, can provide valuable insight into the operation of a train and is primarily used to detect malfunctions during inspections and for root cause analyses after accidents.

These JRUs are constructed according to strict requirements, which ensure that the components are able to withstand immense physical damage, e. g., in case of fire, impact, or pressure [1], [2]. However, it is still possible that they become damaged to the point of destruction, where logged information cannot be recovered [3]. The centralized JRU therefore is a single point of failure. There is also a question of trust in the recorded data; while a train contains components from multiple manufacturers, the proprietary JRU is constructed by only one of them. Thus, one company has authority over the juridical logging, instead of sharing this responsibility amongst these competing and distrustful companies. Lastly, the handling of recorded data is complicated, as the process of data extraction is designed to limit the possibility of physical tampering. The JRU can only be accessed by authorized personnel equipped with a physical key. Currently, the time-consuming extraction is done after longer intervals of operation, potentially up to several days, during a train's operational pauses. Despite the physical protection, manipulation or data losses still occur [3].

Increasing privatization of railway operations and fragmentation of companies lead to digitalization and interoperability efforts [4], [5], e. g., in the OCORA project [6]. This includes replacing centralized, expensive hardware such as the JRU, whose safety is highly regulated but which only has relatively weak security mechanisms. A theoretical analysis by Braband et al. [7] used mathematical modeling to show that replicating the JRU across multiple heterogeneous commodity hardware nodes along the train offers comparable availability and reliability as the centralized JRU in case of accidents. The probability of all replicated JRUs getting destroyed while using commodity train hardware is sufficiently low to ensure that at least one record remains. Replication can also facilitate shared logging authority between companies. Remote connections of replicas allow for more timely and effective uploading and analyzing of data. Continuously retrieving data during regular train operation could prevent gaps due to partial data destruction during accidents, as less data is stored only on the train.

Independent logging of multiple nodes is not sufficient: As a central JRU, despite being a hardened device [3], [7], is a single point of failure, the naïve approach is to use multiple independent JRUs distributed across the train. The trade-off for increased probability of them surviving a crash intact is having unsynchronized logs, which may lead to data loss. Manipulation of individual JRUs is still possible and hard to detect, and multiple JRUs are expensive. Next, one might consider a leader-follower architecture. Here, data can still be read from any surviving node in case of a crash, i. e., only one node is needed to get the recorded data. However, these data can still be compromised. If due to cost factors followers run on heterogeneous commodity train hardware, this also means losing the advantages of a hardened device.

The commodity follower nodes can be more easily compromised, and if multiple followers log different data, then in case of a crashed leader it cannot be distinguished between

Acknowledgements: We thank the anonymous reviewers and our shepherd Katinka Wolter for their valuable feedback. This research was supported by the German BMDV mFUND under grant no. 19F2093D.

correct and manipulated data. Additionally, events are received via a time-triggered bus system, where a message might get corrupted during transmission, leading to nodes missing data or receiving different input [8], [9]. Thus, multiple independent nodes need to receive the logging data and *agree* on their reception and order, which can be achieved via an agreement protocol. Furthermore, the replicated logging is co-located on the same machines as the observed system, which might behave arbitrarily and therefore cause software on these machines to fail unexpectedly. This, in turn, can influence the logging to provide erroneous messages to the replication group, meaning that any kind of errors from faulty hardware, interfering processes, or external manipulation can lead to a diverging or incomplete log. To ensure that such faulty nodes will not disrupt the correct recording of events, a Byzantine fault tolerant (BFT) protocol [10] should be used to tolerate arbitrary failures.

A traditional JRU protects data integrity mainly via physical hardening and access control. A JRU design with replicated nodes, however, must offer software-based data integrity. One solution is a blockchain [11]: a blockchain is an append-only, immutable record of transactions, where any modification of the content after it has been written is impossible without detection, and the data is stored on multiple independent, untrusted nodes. This allows external verification of data integrity, even in the case of only one remaining record. A blockchain-based solution therefore offers the required tamper-resistance. However, it is necessary to consider the safety-critical train communication which is performed via time-triggered bus systems such as ProfiNet, ProfiBus, or the Multifunction Vehicle Bus (MVB) [12]. The blockchain communication, in particular the BFT protocol, could lead to congestion.

This paper presents **ZugChain**, an efficient blockchain-based event recorder for railway systems with real-time capability. The traditional, specially tailored and hardened device of the JRU is superseded by a replicated solution of on-board commodity hardware while maintaining fault tolerance within the timing requirements. We design a permissioned blockchain platform with BFT consensus which is suitable for the train domain in order to achieve secure data logging according to the requirements of modern JRUs. We avoid changes to the safety-critical bus systems; instead, we utilize certified, non-intrusive solutions by opportunistically using existing, non-critical links for consensus communication. Storing the data on remotely accessible devices allows externalization of logged data to private data centers operated by railway companies, where data integrity can be verified due to the blockchain structure. This allows safe pruning of the blockchain on the devices, i. e., deleting exported blocks to reduce memory usage. Our contributions are as follows:

- To the best of our knowledge, ZugChain is the first blockchain system used in operational railway aspects. We show how the specialized device of the JRU can be replaced by software distributed on on-board devices, identify domain demands, and present a design tailored to the specific fault model and requirements, namely complete, tamper-proof, and timely recording.

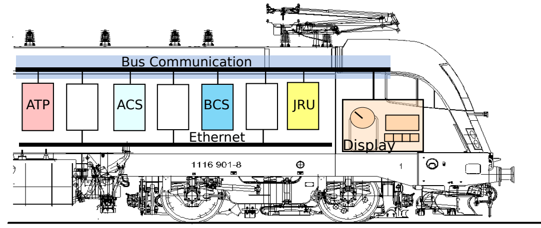


Fig. 1: Overview of data logging in trains today (ACS: acceleration control system; BCS: brake control system).

- We present the ZugChain BFT communication layer, which adapts the authenticated, individual clients of primary-based BFT protocols to input via a single, unauthenticated bus. It receives bus data and ensures completeness of juridical recording while reducing the overhead with filtering.
- We show how a blockchain can simplify data collection during train operations with our data export protocol. The lightweight export is decoupled from the agreement, and synchronizes between agreement and storage for secure, early, and consistent persistence of data.
- ZugChain runs on commodity hardware available on trains instead of a specialized JRU. We present a testbed and performance evaluation of ZugChain on this hardware, showing that we can match JRU requirements while utilizing at maximum 15 % of CPU resources, indicating the suitability of shared device usage.

The structure of the paper is as follows: §II gives relevant information; §III presents the system design; §IV gives implementation details; §V discusses evaluation results; §VI shows related work; and §VII concludes the paper.

II. Background

We give information on train communication networks and JRUs (§II-A), blockchains (§II-B), and BFT protocols (§II-C).

A. Communication and Data Logging in Trains

In today's trains, juridical recording units (JRUs) are the central logging unit of any juridically relevant train events and operational parameters. They allow data recovery after accidents, thus providing information related to these incidents. There is no legally binding global definition of the JRU requirements; instead, national railway companies issue their own based on existing standards [1], [2]. JRUs usually store data in a capacity-limited ring buffer in flash memory. Therefore, authorized personnel need to regularly access the recorder to extract the data if a continuous history is required. The JRU is located in a central point on the train, e. g., the locomotive as shown in Fig. 1, where it is connected to a bus to record all relevant signals. Which signals are to be recorded is configured during deployment or maintenance. Among others, these are the train's speed and location, safety-relevant commands from the train driver, or interventions by the automatic train protection (ATP) [1]. Devices of the ATP receive data from sensors, e. g., antennas or control elements, perform computations, and send the result via the bus to be displayed for the train driver and to be logged by the JRU. Although it is recording safety-critical data, the process of

recording itself is not safety-critical. A train continues its drive normally in case of a failed or malfunctioning JRU. The JRU is mainly secured against unauthorized access, physical damage, and tampering via integrity protection using checksums. It is not required to perform any verification of the received signals before logging. Yet, malicious attacks and bit manipulations are possible, and as the JRU is not indestructible and is a centralized component, data loss can still occur [3].

The standard infrastructure to reliably exchange information throughout multiple vehicles of trains is the train communication network. Commonly used systems include ProfiBus, ProfiNet, or CAN buses [12]. However, most widely used is the combination two types of buses: a wire train bus (WTB) to connect separate wagons and a Multifunction Vehicle Bus (MVB) to connect devices within one wagon [13]–[15]. This way, signals from the MVB of one wagon can be forwarded via the WTB to the JRU. The MVB is standardized by IEC 61375-3-1 [13], supported e. g., by Siemens and ABB [14], and transfers process, message, and supervisory data [15]. For the JRU, the focus is on periodically transmitted process data, which represents urgent data such as speed and emergency stops. The MVB is a synchronized bus system with a leader/follower communication scheme, where the bus master sets the cycle and polls the connected devices for data. Communication errors such as bit flips still occur despite its robust design [9].

Braband et al. [7] analyze the reliability of a JRU replicated across commodity hardware distributed across the train. For this, data must be neither partially nor completely lost or changed after a crash. They argue that this can happen when the JRU fails and is not logging data (note that train operation continues with a failed JRU), or when the JRU is compromised during a crash. Using data of crashes from recent years published by the International Union of Railways (UIC) and the European Union Agency for Railways (ERA), the authors evaluate different incident scenarios. They conclude that a JRU distributed across the train can reach the required reliability, thus allowing to relax the reliability requirements of the hardware itself. This enables the use of commodity hardware, with a mean time between failures of 20,000 h. However, the authors' work is purely theoretical and does not present a system or design of how such a distributed JRU could be realized. It builds the empirical basis for our fault assumptions that aim at avoiding any lost or changed data after malfunctions of any kind, including the categories of crashes considered by Braband et al.

ZUGCHAIN supersedes the data logging of a centralized JRU with distributed, tamper-resistant logging using a permissioned blockchain while maintaining the original requirements and protection of a JRU. ZUGCHAIN's blockchain runs on commodity on-train hardware. In our specific prototype, we assume components to be connected via an MVB, but our approach is independent of the underlying bus technology and can be extended to any bus, e. g., ProfiNet.

B. Selecting a Blockchain Technology for ZUGCHAIN

Blockchain-based distributed ledger technologies (DLTs) such as Bitcoin [16] or Hyperledger Fabric [11] enable a diverse set of use cases, from financial to industrial applications.

A *blockchain* is a replicated, distributed data structure that stores transactions occurring in a peer-to-peer network. This record of transactions is stored in the form of a timestamped list of blocks. Each block contains multiple transactions, is identified by its cryptographic hash, and includes a reference to the previous block's hash. Linking the blocks via their hashes results in a linear, chronological chain of blocks. Creating and agreeing on the next block of a chain across the network is not trivial; agreement protocols such as Proof-of-Work (PoW) [16], Proof-of-Stake (PoS) [17] or BFT protocols (see II-C) ensure integrity and consistency of the blockchain. They offer a sufficiently high difficulty for block creation, thus preventing a malicious actor from creating a chain of blocks containing manipulated transactions with correct hashes. This results in an immutable ledger, which allows the tamper-resistant storage of transactions or assets. Based on the permissions of the ledger, blockchains can be classified into *permissioned* or *permissionless*. Permissionless blockchains allow any new user to join and users are typically pseudonymous, whereas users in permissioned blockchains are known, authenticated entities.

Our system records train events which may contain sensitive information that should not be publicly accessible and runs on internal train components. We have a limited number of participating nodes reading events from the bus; these nodes are known and authenticated at startup and changes to this network are only expected during train maintenance or overhaul. The train requirements of ZUGCHAIN therefore match well with a privately accessible, permissioned blockchain; or, more precisely, a consortium blockchain, where nodes are typically provided by different companies. This allows using BFT for block creation: BFT protocols are more performant than PoW, though they generally are less scalable [18]. Due to these requirements and the demand to offer a lean and resource-efficient system, BFT is the fitting basis for ZUGCHAIN.

C. Byzantine Agreement for Juridical Event Logging

In the train, we have a closed system with a limited number of participants. Malicious behavior is therefore less of a concern compared to the multitude of hardware errors that can occur on the shared devices and which may lead to Byzantine behavior. As the JRU has to log reliably even in critical situations, we need to consider faults beyond simple device crashes. Especially after accidents, a wide spectrum of faulty behavior should be addressed, such as deleting events or changing their content or order [3]. Accordingly, we consider a Byzantine fault model [10]: in BFT protocols, a group of replicas aims to reach consensus on the order and result of client requests, despite a subset of these replicas potentially behaving arbitrarily faulty. Total order agreement is necessary for creating a blockchain since processes need to agree on the next block to be appended. BFT protocols have been used in blockchain platforms in the past, both traditional protocols for small, selected groups of participants [11], [18], [19], as well as for wide-area, high-scalability systems [20]–[22]. In the train, we do not require high scalability for large numbers of changing participants;

instead, traditional protocols such as Practical Byzantine Fault Tolerance (PBFT) [10], achieving high performance with a limited number of participants, are well suited.

PBFT implements a primary-based consensus and assumes a partially synchronous network for liveness. Clients are authenticated and issue requests to invoke operations on the replicated service. To agree on the order of requests, the replicas perform three phases: *preprepare*, *prepare*, and *commit*. In the *preprepare* phase, the primary proposes a message by broadcasting it to all replicas, during which a sequence number is assigned to this request. Then, replicas multicast a *prepare* message to confirm they have received the same number for this request. After receiving at least $2f$ *prepares* from different replicas, each replica multicasts a *commit* message to finalize the acceptance of the assigned order. Finally, after receiving $2f + 1$ *commit* messages, possibly including its own, each replica executes the request. The result is sent to the client, who waits for $f + 1$ identical replies to ensure correctness.

Any replica can become faulty or unavailable, provided that the number of faults is less than f. The primary can be affected as well, e. g., by not forwarding requests. Once backup replicas detect this via timeout, they trigger the *view change* to elect a new, correct primary. This subprotocol is crucial for resuming normal-case operation in the presence of faults. Here, replicas exchange *view change* messages informing about their state since the last stable checkpoint. Once $2f$ replicas have sent this to the new primary, which is selected in a round-robin way, the primary starts a new view.

In the train, logging input is received via unreliable bus communication and potentially diverging input from all replicas should be logged. By reading from the bus on all nodes, it is feasible to receive and process all available inputs. However, in the case of standard PBFT clients, this would result in all clients individually submitting their requests to the primary. If the received data is identical, this incurs high logging overhead as multiple requests containing these data are ordered. In ZUGCHAIN, we present a BFT communication layer suitable for the train domain by replacing traditional BFT client behavior with efficiently handling requests received over bus communication, thereby preventing payload duplication.

III. DESIGN

In this section, we give a complete system overview (§III-A), discuss system and fault models (§III-B), and present the BFT communication layer (§III-C) and the export protocol (§III-D). We discuss how our following requirements are fulfilled:

- $R1$ Replacing the JRU with a replicated system, ensuring high performance despite arbitrary failures;
- $R2$ Replacing an expensive, dedicated device with opportunistically utilizing on-train commodity hardware, without changes to safety-critical communication infrastructure;
- $R3$ Fulfilling the JRU requirements while preventing omission of data received over the bus as well as retroactive manipulation of logged data;
- $R4$ Securely and consistently export data while protecting integrity and garbage collect without omission of data.

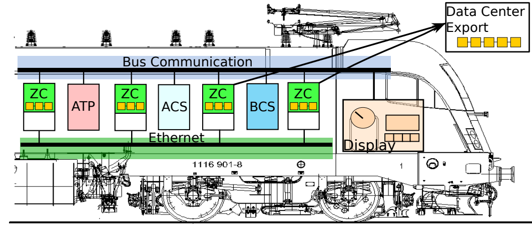


Fig. 2: ZUGCHAIN (ZC) runs on shared nodes, uses existing Ethernet links and exports the blockchain to private data centers.

A. ZUGCHAIN *in a Nutshell*

There are several parties in our system: the railway company, multiple companies producing trains or train components, and federal authorities that investigate responsibility in the case of incidents. No single party should be able to tamper with the logged data. As a train is a heterogeneous system, not only one company is producing installed hardware or software components. In the case of an incident, currently the previous behavior of each safety-relevant component is analyzed and reconstructed based on JRU data. As we replace a certified, tamper-proof device with software running on commodity hardware, the new setup allows for more options of involuntary and possibly malicious interference. Accordingly, we aim to avoid one single company having control over the logging and thus the ability to possibly implicate others for any misconduct, and to instead allow all companies partial responsibility.

Blockchain-Based Train Event Logging. Our design, which is shown in Fig. 2, replaces the JRU with a distributed blockchain-based solution. By using already deployed, on-board hardware as ZUGCHAIN nodes, we minimize changes to the train. These nodes, which have enough resources available to host the ZUGCHAIN system, can come from each of the companies, satisfying *R2*. Messages are received as signals via bus communication. Data loss on the bus, e. g., deletion of signals sent via the bus or blocking of data reception on nodes, is covered by bus specifications [13] and thus out of scope. Messages are read from the unreliable bus by all nodes independently. ZUGCHAIN records the same data as the JRU, i. e., we do not require changes of the signals after they have been sent, e. g., by the ATP. Data is recorded in the same bus frequency and quality as in the original JRU, meaning that existing requirements are still fulfilled. Some data is received by the JRU in encrypted form and logged as is, which ZUGCHAIN handles identically. This data may include sensitive information that is encrypted and authenticated at the data source before bus transmission and which should only be available to authorized parties after export.

From Signals to Blocks. Data is received via the bus in a raw format from which we derive the signals. This uses identical transformation steps as in the JRU, which have been verified and approved ensuring that they are correct and free of side-effects. Nodes receive, parse, and filter the data according to relevance and for higher efficiency as is common practice in JRUs, e. g., to log the speed only upon changes. After this, messages received from the bus are unique.

Once the message has been transformed, it is submitted to the BFT protocol. By using a combination of a BFT protocol for agreement and a blockchain for integrity, we safeguard both against faults during operation, e. g., crashes that destroy all blockchain nodes except one, as well as faults during ordering, ranging from arbitrary influences of any co-located processes on shared nodes to adversarial behavior. The BFT nodes agree on the content of the next block, thereby recording any input from the unreliable bus and maintaining a consistent log despite potential arbitrary behavior. The ordered messages are stored in a blockchain, making deletion impossible without detection or tampering of the blockchain copy on all nodes, thus satisfying *R3*. This is especially relevant after accidents: after salvaging the ZUGCHAIN nodes, at no point should it be possible to selectively delete, reorder, or otherwise modify the logged events without detection, which is achieved by the blockchain. Additionally, a checkpoint of the BFT protocol is created for each block, which is thus backed by $2f+1$ asymmetric replica signatures. Even one remaining ZUGCHAIN node prevents undetected modification of logged requests. If all copies are lost or in the hand of an attacker, then they can delete the head or even the complete chain; however, this is identical to the JRU [3], and the probability that multiple ZUGCHAIN nodes are destroyed is sufficiently low [7].

While the nodes read the relevant signals from the bus, the consensus messages are exchanged over a secondary communication link, e. g., Ethernet, which has become increasingly common in trains [14], [23], [24]. This prevents changes to the bus schedule, does not interfere in the existing, safety-critical communication, and older trains can be retrofitted more easily. The blockchain is permissioned, where the participants are the train devices and no unauthorized device can join the network. The blocks are created on the nodes after finishing BFT ordering, and stored until they can be securely exported to the railway companies' private data centers. Different from traditional BFT systems, where clients submit unique requests, the ZUGCHAIN nodes read mostly identical data from the bus, but omissions and reordering may occur. In §III-C, we describe a communication layer for BFT protocols that filters such duplicates to reduce system load while ensuring that no information is omitted from the log, thus satisfying *R1*.

Secure Export of Blockchain Data. Continuously exporting logged data traces during train operation simplifies the extraction process, enables mechanisms for predictive maintenance, and is desired due to the nodes' limited memory. The train is connected to the outside world via external gateways, e. g., via LTE or Wi-Fi, whose coverage has immensely improved over the last years. Additionally, as the competing railway companies mutually distrust each other, it is not desirable to have only one record of the blockchain; instead, multiple synchronized records are maintained in independent, private data centers each operated by a company. When the train is in range of a cell tower, the ZUGCHAIN nodes establish a connection to a data transfer endpoint, which can request the nodes to send any blocks created since the last export. Our protocol ensures that only correct blocks are accepted by data centers, and that recent blocks are actually exported and subsequently deleted. For this, data centers query the nodes independently, synchronize and verify the data, and confirm the export to allow deletion of blocks on nodes. The private data centers permanently store the blockchain data, and faster, less complicated access to the collected data for predictive maintenance is possible. The protocol for export and blockchain pruning satisfies *R4* and is described in more detail in §III-D.

B. System Model and Fault Assumptions

Ordering and Blockchain. For ZUGCHAIN nodes, we assume a Byzantine fault model [10]: we have a set N of $n \geq 3f+1$ nodes of which up to f can behave arbitrarily faulty. As consensus messages cannot be allowed to congest the safety-critical bus, an Ethernet network is used between replicas. For the consensus, we therefore have a partially synchronous network, i. e., communication is asynchronous and messages can be delayed for up to a constant, but unknown time δ, after which the network is temporarily synchronous. All nodes are equipped with a public-private key pair for signature generation and verification, and all messages exchanged by the ZUGCHAIN nodes are signed via asymmetric cryptography to ensure integrity. We assume that cryptographic techniques are secure and cannot be broken by the adversary.

As typical in BFT models, replicas may behave arbitrarily faulty, which includes delaying, duplicating, omitting, reordering, or corrupting protocol messages. A faulty replica may, for example, omit or send inconsistent protocol messages to other replicas, to get the system in an inconsistent or dysfunctional state. Additionally, faulty nodes may omit, duplicate, reorder, or even add new data not received from the bus. Reordering or addition of data may be done either to mislead analysis or to incur additional system load as a denial of service attack. The detection of such added data and the reestablishing of the correct order is done after export during lab analysis and out of scope for the juridical logger. Other DoS attacks, e. g., spamming of messages with false authenticators is out of scope.

Bus Communication System. The main challenge arising from bus communication is to filter duplicated input, while ensuring that all individual inputs are included in the blockchain. Inputs are received synchronously by the nodes via a time-triggered bus where the bus master regulates communication. All signals transmitted in a bus cycle are consolidated into one BFT request. Safety is ensured for the bus and the connected devices by fulfilling corresponding standards [25]. Data is not authenticated, which means that individual data sources are indistinguishable. Requests that a node reads from the bus are unique, but the same message is read by multiple nodes. Such duplicates should be filtered to avoid high load. However, messages from the bus can be dropped or reordered, i. e., a replica does not receive any signals in a cycle, and all signals from one bus cycle are received during a different one, respectively. It is also possible for nodes to read diverging input during the same bus cycle. Filtering must not prevent such individually received signals to be logged. As system load

Module	Call	Explanation
① down	PROPOSE(r)	proposes request to consensus group
① down	SUSPECT(id)	suspect node to be faulty, init. view change
① up	DECIDE(r, sn)	totally ordered request and seq.no.
① up	NEWPRIMARY	returns new primary after view change
② down	RECEIVE(req)	read parsed request from bus
② up	LOG(req, id, sn)	append request to totally ordered log

TABLE I: Interfaces of BFT (①) and ZUGCHAIN (②).

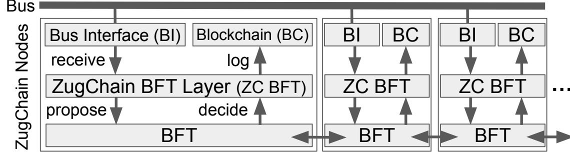


Fig. 3: Overview of the ZUGCHAIN node components.

can be accurately estimated during setup, message buffers for phases of asynchrony can be appropriately sized. All data sent over the bus is considered valid data to be logged, even if it is only received by one node. However, nodes can propose corrupted or fabricated data, which was not originally received via the bus. This data should be logged in combination with the node identifier, as it is especially relevant for failure analysis to have a complete log of the whole system behavior. Detection of such messages and analysis of data is out of scope. The blockchain, as an append-only data structure, prevents new data (fabricated or correct) to be mixed with older data. While also correct data can be delayed, e.g. due to network failures or omission by a faulty primary, out of order data that is included long after its proposed creation should be regarded sceptical during analysis. ZUGCHAIN employs a specific communication layer that ensures *no payload duplication*: No correct process logs the same payload more than once. Additionally, ZUGCHAIN ensures correct filtering and avoids discarding inputs despite primary changes as client-side bus retransmissions are unavailable, handles input from multiple sources such as several independent buses, and writes the ordered requests into the blockchain. We note that, in accordance with the JRU specification and requirements communicated by domain experts, ZUGCHAIN does not guarantee that the order in which data is recorded reflects the order in which it is sent on the bus.

C. *The* ZUGCHAIN *BFT Communication Layer*

ZUGCHAIN is an extension layer to primary-based BFT protocols which replaces traditional client interactions with optimal handling of input via bus communication. It ensures that all bus input received on a correct node is captured, while avoiding the overhead of recording identical input multiple times via filtering. In case of diverging input, ZUGCHAIN ensures that requests that are received on single nodes are also logged, and that each request is logged in conjunction with the *id* of a node that has actually received it. ZUGCHAIN uses an interface as typically provided by a primary-based BFT protocol, shown as ① in Tab. I. This interface explicitly exposes the BFT primary election and suspicion to the ZUGCHAIN layer, allowing implementation of filtering on the primary and suspecting primaries that are not filtering correctly.

Algorithm 1 ZUGCHAIN Communication Layer

```
 1: function INIT
 2:     R ← ∅                                          ▷ ZUGCHAIN node request queue
 3:     id ∈ ℕ                                         ▷ ZUGCHAIN node id
 4:     primary ∈ N                                    ▷ initial primary

 5: upon RECEIVE(req_i) do                             ▷ read parsed data from bus
 6:     insert(R, req_i)
 7:     if (id == primary) ∧ (!inLog(req_i)) then
 8:         r ← sign(req_i, id)                        ▷ authenticate and include (primary) node id
 9:         PROPOSE(r)                                 ▷ propose to BFT if co-located with primary
10:     else
11:         t[req_i] ← timer.start(SOFT_TIMEOUT)

12: upon DECIDE(r, sn) do                              ▷ receive ordered request returned from BFT layer
13:     if r.req ∈ R then
14:         delete(R, r.req)
15:     if ∃ t[r.req] then
16:         cancel t[r.req]                            ▷ cancel hard or soft timeout
17:     if inLog(r.req) then                           ▷ primary has submitted duplicate request
18:         SUSPECT(primary)                           ▷ initiate view change
19:     else
20:         LOG(r.req, r.id, sn)                       ▷ append to log, include id of origin node

21: upon SOFT_TIMEOUT t[req] expires do
22:     r ← sign(req, id)                              ▷ authenticate and include node id
23:     t[req] ← timer.start(HARD_TIMEOUT)
24:     BROADCAST(r)

25: upon BROADCAST(r) do
26:     if inLog(r.req) then
27:         return                                     ▷ ignore duplicates already in the log
28:     if (id == primary) ∧ (r.req ∉ R) then
29:         PROPOSE(r)                                 ▷ propose with id of broadcasting node
30:     else
31:         t[r.req] ← timer.start(HARD_TIMEOUT)
32:         forward r to primary                       ▷ ensure primary receives request

33: upon HARD_TIMEOUT t[r] expires do
34:     if !inLog(r) then
35:         SUSPECT(primary)                           ▷ initiate view change

36: upon NEWPRIMARY(pid) do                            ▷ after view change
37:     primary ← pid
38:     for ∀ req ∈ R do                               ▷ for all open requests
39:         if (id == primary) ∧ (!inLog(req)) then
40:             r ← sign(req, id)                      ▷ authenticate and include (primary) node id
41:             PROPOSE(r)                             ▷ propose on new primary
42:         else
43:             t[req] ← timer.start(SOFT_TIMEOUT)     ▷ reset timers
```

Tab. I also shows the interface of ZUGCHAIN nodes (②), which RECEIVE data from the bus and LOG requests ordered by the BFT module to the blockchain. The interactions of the ZUGCHAIN node components are shown in Fig. 3.

Agreeing on Requests. The ZUGCHAIN algorithm is shown in Alg. 1. The algorithm combines content- and primary-aware request filtering, where duplicate requests are filtered based on their payload and nodes other than the primary avoid submitting duplicates. This filtering is based on the log of previously decided requests as well as the request queue R, which contains any open and in-flight requests. If a faulty primary proposes duplicates, this is detected during log checks, leading to suspicion and change of the primary. This filtering lowers the ordering overhead while still ensuring a complete log. Note that the filtering is done for performance reasons, not for correctness of the log. Further, ZUGCHAIN employs timers to ensure swift logging and prevent lost or omitted requests, e. g., backups submit requests only when their received input has not been decided upon within a certain time.

The nodes receive the requests via the bus and maintain

their own request queues (ln. 5, 6). Backup nodes start a `SOFT_TIMEOUT` timer for each received request (ln. 11), while the ZUGCHAIN node co-located with the current BFT primary (in the following referred to as (ZUGCHAIN) primary) signs the first request in its queue and calls PROPOSE (ln. 9). Once the BFT module delivers an ordered request via the DECIDE call, ZUGCHAIN nodes check whether they have the corresponding request in their queue, in which case they remove it and cancel its `SOFT_TIMEOUT` (ln. 14, 16). Next, the log is checked for duplicates and the primary is suspected when duplicates are detected (ln. 17). When the `SOFT_TIMEOUT` on a ZUGCHAIN node expires, the corresponding request has not been proposed by the primary thus far, in which case the node proposes it by broadcasting the request to all replicas (ln. 21-24). The ZUGCHAIN nodes then start a `HARD_TIMEOUT` timer and forward the request to the primary (ln. 23, 32). If the request is decided within the `HARD_TIMEOUT`, then the backups cancel this timer (ln. 16). Otherwise, the backups suspect the current primary to be faulty, triggering a view change and, eventually, a new primary (ln. 35). After a new primary has been established (ln. 36), the co-located ZUGCHAIN node calls PROPOSE for all its open requests, i. e., requests without a corresponding DECIDE or running consensus instance (ln. 41), while the other nodes restart their `SOFT_TIMEOUT`s (ln. 43).

The ZUGCHAIN algorithm ensures that all requests received by correct nodes are logged consistently and with according *ids*. Additionally, in case the primary is correct, all duplicates are filtered before submitting them to the BFT protocol, reducing ordering overhead for BFT input via bus communication. If the primary is faulty, the duplicates are detected and the primary is suspected and eventually changed (ln. 17). This is achieved by checking the log of previously decided requests. In practice, a check of the complete blockchain for every request is not feasible; instead, we check against the recent history. This is done efficiently with a hashmap over the requests of a sliding window of past checkpoints as well as open requests in R. The soft timeout is an optimization to avoid the additional load of duplicate logging on the system, allowing to filter requests received on multiple nodes. The hard timeout is used to detect censorship of a faulty primary, equivalent to timers used, e. g., by PBFT. As in other work [10], the hard timeout has to be adjusted to the actual network delay to avoid false suspicion of correct primaries. As an optimization, for an underlying BFT protocol such as PBFT, ZUGCHAIN nodes can already use a primary's *preprepare* as an indicator that this request will be ordered and cancel the corresponding soft timeout.

Faulty Nodes. According to our fault model (§III-B), a faulty ZUGCHAIN node that is not the primary may misbehave in five ways: (i) *Request duplication:* the faulty node broadcasts duplicate requests. If these are already in the log or are currently getting ordered, they will be filtered out by all replicas (via the *inLog* function, ln. 26). Other request duplicates are filtered by the primary (ln. 28); if these duplicates are not part of the log yet, corresponding hard timeouts (ln. 31) will be cancelled on reception of the original message's DECIDE (ln. 16). (ii) *Request omission:* requests received only at faulty nodes may be omitted. This is identical to a lost bus connection and in accordance with JRU behavior; however, in ZUGCHAIN, we still log the input of the correct nodes. Typically, requests are received on all or multiple nodes. (iii) *Denial of service:* a faulty node may broadcast a large number of requests to deteriorate performance. To avoid this, ZUGCHAIN limits the number of open requests a node can send in parallel and other correct nodes drop any further received requests. The limit is calculated based on the bus frequency. (iv) *Faulty broadcasts:* during broadcasting (ln. 24), a faulty ZUGCHAIN node might send the request only to a subset of replicas, e. g., in the worst case omitting the primary. To avoid a false suspicion of a correct primary, backups forward the request (ln 32). (v) *Faulty suspicion:* a faulty node may suspect any primary. This is not problematic, since BFT protocols only change the primary after it has been suspected by at least $f + 1$ nodes.

Faulty Primary. Additional to the above, a faulty ZUGCHAIN primary may omit or delay proposing requests or incorrectly filter duplicates. Message omission is detected and will trigger suspicion and a view change. Delay of proposing messages may cause soft timeouts to trigger leading to additional broadcasts and load on the system, but no incorrect behavior. On repeated occurrence, this could also be detected and may result in a view change. Finally, incorrect filtering and proposal of duplicates is detected by the check of the log upon DECIDE (ln. 17), leading to a view change. In practice, if a duplicate whose original request is not available in the sliding window of checkpoints is proposed, we record this duplicate. This can lead to temporary performance degradation; however, it does not violate the correctness of the log and this duplication can be detected in post-operational analysis.

Blockchain Application. The signals have been parsed at reception into a format compatible with JRU analysis tools. Once a certain threshold of ordered requests has been reached, the replicas deterministically bundle and hash them and store the created block on disk.

Checkpointing. BFT checkpointing is used to garbage collect consensus messages of ordered requests. Replicas exchange application snapshots in signed *checkpoint* messages, creating a stable checkpoint once $2f + 1$ messages have been received. As we want to prevent any modification of blocks, especially after crashes when only one replica remains intact, we generate frequent checkpoints. A block is created after sufficient requests have been ordered, and for every block a checkpoint including this block and all its requests is created. A replica's signed checkpoint message thus confirms that it created this block. We also leverage checkpoints in §III-D for secure data export.

Multiple Input Sources. While so far we have only discussed nodes to be connected to a single bus, they may also be connected to multiple input sources, which can potentially be (partially) synchronous. ZUGCHAIN supports this: nodes have one request queue per connected link from which requests are processed, thus logging all messages from all input sources.

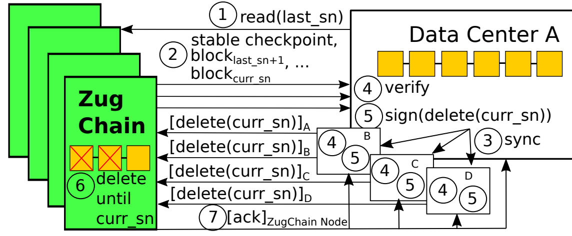


Fig. 4: Overview of the export protocol. Any data center can start the export. Signatures of id are denoted by $[\cdot]_{id}$.

D. Secure Data Center Export for Blockchain Data

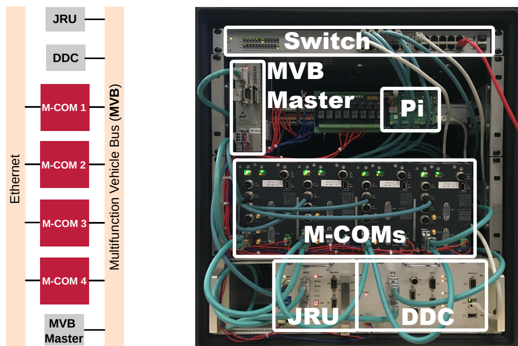
Newer JRU data is of higher interest, and older data can be discarded in case of memory shortage without losing current information. However, as we use a blockchain, we need the complete chain to verify its integrity. We therefore need to extract data before memory shortages or data losses occur. ZUGCHAIN can export blocks to one or more private data centers provided by the railway companies, who already store JRU traces. Any data center can request newly created blocks from the replicas, which are then synchronized between and verified by all data centers. This export allows us to collect the blockchain state and prevent it from growing indefinitely.

The train is connected to the data centers via LTE or Wi-Fi. As wireless connection bandwidth is limited, we minimize the transmitted data. Instead of requesting blocks from multiple nodes, we leverage the BFT *checkpointing*. Each block gets included in a checkpoint, and a *stable* checkpoint with its $2f+1$ replica signatures proves that the corresponding block is included in the blockchain. As stable checkpoints are not part of the active application state anymore, we can circumvent the consensus and directly query the replicas. Faulty nodes may manipulate or lie about their logged data; thus leveraging stable checkpoints ensures correctness while exporting from individual replicas. We query additional replicas for their checkpoints to ensure that recent blocks are actually exported, and resources can be freed. The export protocol guarantees that (i) only blocks logged by correct nodes are exported; (ii) all blocks up to the most recent stable checkpoint are exported; and (iii) exported blocks get deleted from the nodes to save resources. With this multi-data center export protocol we satisfy *R4*.

Message Flow. We have two operations: *read* to extract blocks from the on-board nodes, and *delete* to confirm a successful export so that replicas can safely delete the transmitted blocks. All ZUGCHAIN nodes are equipped with a public-private key pair, with which they sign ordering, checkpoint, and view change messages, allowing verification of checkpoints. Each data center also has a key pair, of which the public key is known to the ZUGCHAIN nodes and vice versa.

The communication steps, shown in Fig. 4, are as follows: ① The data center asks the BFT replicas for the latest block in a *read* broadcast which includes the index of the last successfully exported block (*last_sn*). ② Each replica sends its latest stable checkpoint, while one randomly determined replica also sends the full blocks (*last_sn* to *curr_sn*). As these *read* messages bypass the consensus and are therefore unordered, i. e., can be received at different times, replicas can send different checkpoints. This requires the data centers to determine the latest one with the highest checkpoint sequence number to maximize the number of exported blocks. ③ Once the checkpoints of $2f+1$ replicas and the full blocks from one replica have been received, they are synchronized with the data centers of the other companies. We wait for $2f+1$ answers to ensure reception of recent checkpoints: while in principle even one valid stable checkpoint would be sufficient, this checkpoint could be outdated and therefore leave more data on the train than necessary. With $2f+1$ replies, even if f slow nodes with outdated checkpoints as well as f faulty nodes are included, at least one node will reply with a recent checkpoint, for which we can issue the *delete*. ④ Using the replicas' public keys, all data centers validate the signatures of the latest checkpoint, and validate the received blocks up to the included block. If any blocks are missing between *last_sn* and the block included in the latest checkpoint, these can be queried directly from the replicas in a second round of communication. ⑤ After the blocks' reception and verification, the data centers each sign a *delete* message, which includes the index and hash of the block in the latest stable checkpoint. The *deletes* are broadcast to the replicas. ⑥ Replicas now verify that at least a certain, configurable number of data centers have sent a signed *delete* and remove the blocks up to this index, keeping the last exported block to serve as the first block for the pruned blockchain. ⑦ Finally, they send a signed acknowledgement to the data centers to confirm the *delete*.

Discussion. Several error scenarios can occur: (i) A *delete* arrives on a replica before the corresponding block has been created: The replica checks whether it has created the block included in the *delete*; if not, it delays the *delete* until block and checkpoint have both been created to ensure correctness. This could be avoided by ordering *delete* messages via the consensus protocol. However, export and agreement are intentionally decoupled as required for JRUs, where export strictly should not delay or influence agreement. (ii) Transferring a checkpoint to another replica: The replica receives the checkpoint and blocks between this and the last stable checkpoint. It then has to check the blockchain integrity and whether the most recent block and the open requests match the received checkpoint digest. As the blockchain on the replicas is pruned after an export and verification therefore cannot start at the genesis block, the transferred state must include the signed *deletes* that verify the base of the blockchain on the replicas. (iii) Not enough *deletes* received: a *delete* is marked as incorrect if the replica does not receive a sufficient number of matching signed *deletes* from the data centers. The replica then does not execute the operation. (iv) A data center is delayed and missing already exported blocks: In this case, blocks can either be exported again if they are still available on the replicas, or synchronized from other data centers. (v) A replica misses one or multiple *deletes* and does not free memory. Before any data is overwritten due to memory exhaustion, replicas can agree to remove the data of a certain number of blocks and



(a) Schematic overview. (b) Hardware setup.

Fig. 5: Overview of the ZUGCHAIN testbed.

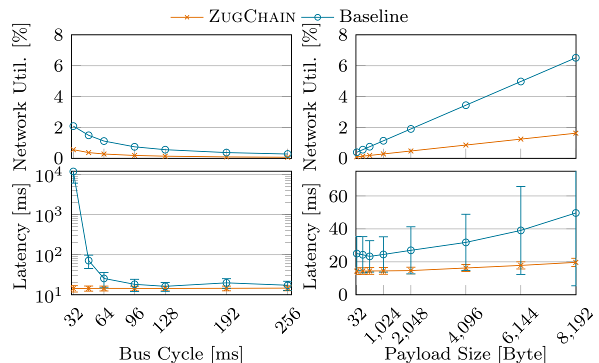


Fig. 6: Network utilization and latency for typical bus cycles (left) and payload sizes (right). Latency (left) in log scale.

only store their headers. The joint agreement is stored on the blockchain to signal that this was not due to faulty behavior. As the hashes are still available for verification, this allows the replica to preserve the blockchain integrity. However, the replicas' signed acknowledgement ((7)) allows early detection of this, allowing maintenance personnel to intervene in time. We can further assume nodes to have sufficient memory to store multiple days of logged data, similar to the JRU.

IV. IMPLEMENTATION

We implement ZUGCHAIN in Rust, combining high performance with memory safety and efficiency for resource-constraint devices. Rust prevents undefined behavior (e. g., out-of-bounds memory accesses or use-after-free bugs), thereby eliminating a whole class of possible BFT faults. ZUGCHAIN is written for Rust v1.44.0, and blockchain data is exchanged in Protobuf format. Our framework uses `ring`'s asymmetric cryptography for authenticity; specifically, we use Ed25519 signatures on all messages. It includes a full implementation of ZUGCHAIN and PBFT comprising the ordering, checkpointing, and view change subprotocols. While we use PBFT as a well-established protocol with thoroughly investigated correctness, ZUGCHAIN can support other primary-based BFT protocols as well. Our PBFT implementation exposes the SUSPECT and NEWPRIMARY interfaces. The export functionality is realized by extending the replicated application, and the data center side is also implemented in Rust.

In order to access the bus, our framework contains a connector to the underlying bus, in our prototype the MVB. It has read-only access to the MVB, using a specialized, proprietary C++ library to access the train communication network provided by Siemens Mobility (cf. §V-A). The data type and cycle time of the signals can be dynamically discovered from the bus configuration file, which allows for flexible and extensible handling of received bus signals. The code is available at `https://github.com/ibr-ds/zugchain`.

V. EVALUATION

A. Testbed Setup

We built a testbed with state-of-the-art train hardware, shown in Fig. 5. It contains four M-COM RT V1 QW, which are equipped with a Freescale Quad-Core i.MX 6 Cortex-A9 (ARMv7a) CPU@800 MHz, 2 GB of RAM, and three 100 Mbit/s Ethernet links plus an MVB link. They run a custom Yocto Linux with kernel v3.10.17. A signal generator for JRU test systems (DDC) generates ATP data. A Siemens mRec-s42 JRU is included in the setup as well. The components are connected to an MVB, with a SIBAS-KLIP AS318MVB as MVB master. Each component is equipped with a node supervisor database (NSDB) file, specifying which signals are written or read by it. The M-COMs are connected via Ethernet for the consensus communication. An LTE router connects to an AWS VM (`t2.xlarge`) for the data export.

The MVB is a well-established bus system [14], and ZUGCHAIN can easily be adapted to other bus systems, e. g., ProfiNet. We show that our approach is non-reactive, i. e., transmitted signals are logged as is without further computation. This testbed is thus comparable to a train setup with an MVB and M-COMs and allows us to closely simulate a real train environment. We have also deployed our system on an ICE TD train to verify ZUGCHAIN's feasibility.

Evaluation Setup. We compare ZUGCHAIN's communication layer with PBFT and traditional client handling ("baseline"), where each node runs a client and replica process and every client reads bus data and forwards it to the primary as a BFT request. Identical requests are thus ordered up to four times. We measure latency and network, CPU, and memory utilization on the primary from request reception to finalized *commit* stage in the testbed, and report averages over five runs each with a duration of 5 minutes. The block size is 10 requests. Varying bus cycles and payload size of the MVB requires changes to the proprietary NSDB, which is only available to railway companies. We instead simulate receiving messages over the bus, and verify our simulation with available MVB data.

B. Performance Evaluation

Network Utilization and Latency. Fig. 6 shows the network utilization and latency of ZUGCHAIN and the baseline for bus cycles from 32 ms, the MVB's minimum, up to 256 ms for a payload size of 1 kB, as well as for payload sizes from 32 Bytes up to 8 kB for a fixed bus cycle of 64 ms. The baseline's network utilization of the 100 Mbit/s links in the testbed is 4× that of ZUGCHAIN, as each request is ordered four times. The latency of the baseline is 1.1-4.9× that of ZUGCHAIN, as

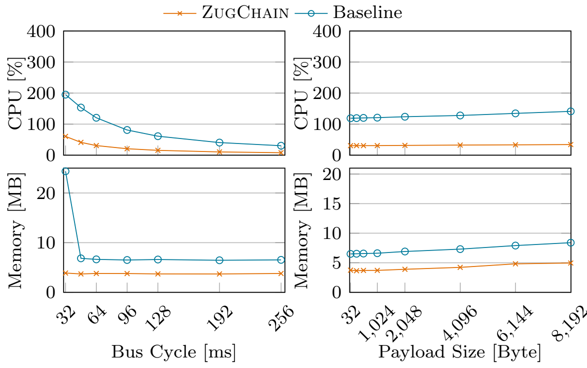


Fig. 7: CPU and memory usage of ZugChain and baseline for typical bus cycles (left) and payload sizes (right).

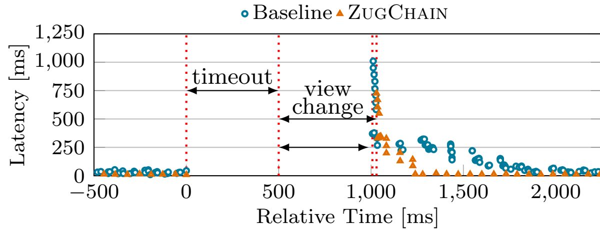


Fig. 8: Request latency during view changes.

more consensus messages have to be transmitted. Especially for a short bus cycle of 32 ms, we report up to 828× higher latencies. Here the baseline cannot keep up with the number of messages and requests are dropped. For evaluating different payload sizes, we keep the bus cycle fixed to the commonly used value of 64 ms. ZugChain's latency increases by 37 %, while the baseline's latency is 1.6-2.5× that of ZugChain. We therefore require less bandwidth and achieve lower, more stable latencies than the baseline. Our testbed can send messages of up to 1 kB over an MVB with a bus cycle of 128 ms. The results are consistent with the simulation.

Resource Usage. Fig. 7 shows memory and CPU usage, where 400 % CPU utilization means all four cores are working at capacity. ZugChain's CPU usage is 25-31 % of the baseline's for different bus cycles and 24-26 % for increasing payload sizes. The baseline's memory usage is 1.7-1.8× that of ZugChain for different bus cycles and 1.6-1.7× for increasing payloads. At short bus cycles, the baseline requires up to 6.3× more memory than ZugChain. With a maximum usage of 15 % of all available CPU resources, ZugChain is thus better suited for shared, resource-constraint commodity hardware.

View Change. Fig. 8 shows the request latency after a view change due to a faulty primary, occurring at relative time 0. The baseline's view change timeout is 500 ms, for ZugChain the hard and soft timeouts are each 250 ms for a total of 500 ms. The bus cycle is set to 64 ms. The replica starts the timer(s) once it discovers the fault; after it has expired, the view change is performed, which in the case of ZugChain takes 530 ms and for the baseline 507 ms. We have a checkpoint size of 10 requests, and consider the worst case with a maximum number of requests not yet included in a block (9 requests). After the view change, within 210 ms (ZugChain) and 824 ms (baseline), the latency has reached previous levels of 14 ms (ZugChain) and 25 ms (baseline), respectively. We see that while the baseline's view change finishes slightly faster, ZugChain stabilizes more quickly as it has less messages to process. The view change timeout in ZugChain can be shortened further depending on the requirements of the JRU. We do not optimize for throughput as we have a fixed number of messages per second, and ZugChain can order the most message-intensive setting available in our testbed without performance degradation (cf. Fig. 6). Instead, we aim for low latency to ensure efficient recording of logging messages. With our quickly stabilizing view change, we can use more aggressive timeouts and accept more frequent view changes compared to the baseline, to ensure fast recovery.

#blocks	500	1.000	2.000	4.000	8.000	16.000
read	4.9 s	9.8 s	19.7 s	42 s	81 s	162.2 s
delete	0.14 s	0.39 s	4.7 s	9.5 s	12.4 s	15.3 s
verify	0.02 s	0.04 s	0.07 s	0.15 s	0.29 s	0.58 s

TABLE II: Latency of *read*, *delete*, and *verify* during export.

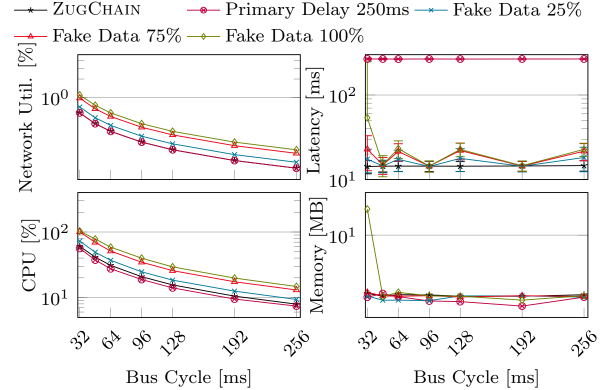


Fig. 9: Effects of Byzantine behavior for bus cycles (log scale).

Data Center Export. The export consists of reading checkpoints from $2f + 1$ replicas as well as the complete blocks from one replica, their verification by the data center, and their deletion. Tab. II shows the average latencies over 5 runs of these steps for the export of 500 to 16,000 blocks to the AWS VM. At a bus cycle of 64 ms, this corresponds to the data collected during operation of 5 minutes to 3 hours. The connection is via LTE at approx. 8.5 Mbit/s. The majority of the latency (80-96 %) is spent waiting for $2f + 1$ replies, especially the full blocks from one replica; here, the network communication until all replies have been received is the bottleneck. Verification takes 0.2-0.3 % of the total time, deletion 3-19 %. With a duration of approx. 3 minutes after 3 hours of operation, exporting data continuously or during train stops is feasible [26].

Byzantine Behavior. ZugChain nodes may deviate against (i) the BFT protocol, (ii) the export, or (iii) the communication layer. (i) Faults against the BFT protocol are similar to other BFT systems and, if done by the primary, will typically result in a view change (cf. Fig. 8). (ii) The export is done only periodically and allows little room for misbehavior since multiple nodes are queried. At worst, a faulty node denying to respond can delay the export until another node is queried.

(iii) The worst case attacks against the communication layer are the addition of corrupted messages and the delay of preprepares by the primary to deteriorate performance; these attacks are evaluated below to show their influence on the system. Other attacks such as omitting or corrupting ordering messages by the primary leads to forwarding and potential view changes, duplicate ordering messages are easily filtered on all nodes, and reordering is not critical. Bus messages omitted by the primary are forwarded, and duplicates are again filtered, as in normal case operation. Both forwarding and filtering is included in the handling of fabricated requests and shown in the experiment.

Fig. 9 shows the results of these Byzantine behaviors: a faulty backup node adds a fabricated request for 25 %, 75 %, and 100 % of all bus cycles. The injection of fabricated requests leads to an increase in CPU load by 20 %/68 %/92 %, memory usage by 0.7 %/1.6 %/294 %, and latency by 22 %/60 %/277 %, compared to normal operation. However, due to the rate limiting on open requests per replica (cf. §III-C), we can efficiently limit congestion. Requests are still ordered within performance bounds of the JRU, while allowing benign replicas e. g., to propose delayed or uniquely received messages.

Fig. 9 also shows a faulty primary delaying preprepares by 250 ms, triggering soft but not hard timeouts, but proposing it before a view change is triggered. This can stall ordering progress until other nodes get a soft timeout and forward the request. Accordingly, the latency increases with this delay, while network utilization decreases. Forwarding and soft timeouts are negligible regarding network and CPU utilization. This shows the importance of the soft timeout, which allows to limit the effect of such misbehavior to stay within system requirements. Due to space limitations, experiments for payload sizes are omitted, though their effect is identical to that in Fig. 6 and Fig. 7.

Comparison to JRU Requirements. A data recorder has to prevent data from being *deleted, changed, or overwritten*, and has to ensure *data integrity*. We fulfill this, as we detect deletions or modifications and correct logs are available on multiple nodes. It also has to offer *data extraction*: to prevent data loss, we frequently export blocks; and to ensure data integrity after e. g., power loss, we persist the blockchain on disk. Data is required to be stored within *500 ms after arrival* considering 10 events per second. With a bus cycle of 64 ms, we process 15.6 events per second, where we incur a latency of approximately 14 ms. Writing blocks to disk in our testbed is an additional 5.03 ms for payload sizes of 8 kB and thus well below the threshold. We therefore fulfill the JRU requirements and show that ZUGCHAIN satisfies *R1* and *R2*.

Train Deployment. We deployed ZUGCHAIN on an ICE TD. Due to regulatory constraints, the setup differs from our testbed, e. g., regarding hardware, MVB connection, and number of signals. We were able to verify during several test drives that ZUGCHAIN's agreement and export work robustly and well within timing requirements. We plan to publish the data of the test drives at `https://railchain.berlin/`.

VI. Related Work

Comparing ZUGCHAIN and PBFT. ZUGCHAIN presents a communication layer on top of a primary-based BFT protocol. It is tailored to the bus: multiple nodes receive data, and overhead is reduced by ordering identical data only once. In PBFT, clients either send their requests to the primary or broadcast them to all replicas [27]. Duplication is avoided only on complete requests including client *ids* and sequence numbers, not on payloads. Using traditional PBFT in combination with a bus results in high overhead. We provide an export of distributed replica storage to external infrastructure, which PBFT misses.

Communication Models in BFT. The Wormhole model [28] proposes a hybrid system model combining synchronous and asynchronous communication. In ZUGCHAIN, replicas have partially synchronous links while input to the nodes is received synchronously. Echtle et al. [29] show a BFT protocol running over multiple bus systems that can handle non-cooperating Byzantine faults. Our system does not perform agreement via the bus and instead reads the logging input from it.

Causality in BFT. Causality guarantees that requests' order of execution is the same as that of their submission [30]. Current approaches have high computational overhead [31] or use specialized hardware [30], which is not feasible on shared train hardware. For ZUGCHAIN and JRUs, causality of logged data is established during analysis after export.

Blockchain in Railway Systems. Kuperberg et al. [32], [33] evaluate smart contracts to decentralize railway operations. Blockchains in railways have been investigated to reduce train delay [34], for WSN node identity authentication [35], and for improving digital ticketing [36]. Surveys [37], [38] focus on digital ticketing, supply chain management, and data distribution. Our system is the first to our knowledge to use blockchains in operational railway aspects.

Event Logging. Braband et al. [7] present a mathematical analysis showing the reliability of distributed JRUs. This theoretical work presents no design and builds the empirical basis for our fault assumptions. Hartong et al. [3] secure JRU data using multi-party secret sharing. Blockchain event recorders have been proposed for autonomous vehicles [39]–[43], robots [44], and IoT [45]. They either use BFT amongst multiple vehicles [39], [41], [42], develop new consensus approaches [40], or rely on complex blockchains [43], [45]. ZUGCHAIN considers agreement of juridical data contained in one train and leverages a lightweight blockchain infrastructure. Regarding restricted storage, Wang et al. [45] present a hierarchical blockchain including an export utilizing a round of BFT. Our export is independent of agreement to reduce overhead and avoid interfering with the logging.

VII. Conclusion

We presented ZUGCHAIN, a blockchain-based train event logger. It contains a BFT communication layer tailored for the bus, and offers secure block export. It utilizes on-board hardware, superseding the JRU's special-purpose device, thus furthering digitalization in railway systems.

References

[1] IEC, "IEC 62625-1:2013," 2013.
[2] "IEEE Standard for Rail Transit Vehicle Event Recorders - Redline," *IEEE Std 1482.1-2013 - Redline*, 2014.
[3] M. Hartong, R. Goel, and D. Wijesekera, "Protection and Recovery of Railroad Event Recorder Data," in *IFIP Digital Forensics*, 2008.
[4] Damiano Scordamaglia, "European Parliamentary Research Service – Digitalization in Railway Transport," 2019.
[5] A. Berrios Villalba, "How to Speed Up Digitization in the Railway [Viewpoint]," *IEEE Electrification Magazine*, 2020.
[6] DB, NS, ÖBB, SNCF, SBB. (2020) OCORA-40-001-Beta - System Architecture - Beta Release. [Online]. Available: https://github.com/OCORA-Public/Publication
[7] J. Braband and H. Schäbe, "The Reliability of Distributed Juridical Recording," *Signalling & Datacommunication*, 2021.
[8] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona, "The Real Byzantine Generals," in *IEEE Digital Avionics Systems Conference*, 2004.
[9] H. Chen and C. Qian, "Research on Functional Safety of Multifunction Vehicle Bus in Rail Transit," in *IEEE ICAIIS*, 2020.
[10] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *OSDI'99*.
[11] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger Fabric: a Distributed Operating System for Permissioned Blockchains," in *EuroSys'18*.
[12] D. Ludicke and A. Lehner, "Train Communication Networks and Prospects," *IEEE Communications Magazine*, 2019.
[13] IEC, "IEC 61375-3-1:2012," 2012.
[14] H. Kirrmann and P. A. Zuber, "The IEC/IEEE Train Communication Network," *IEEE Micro*, 2001.
[15] G. A. zur Bonsen, "The Multifunction Vehicle Bus," in *WFCS'95*.
[16] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.
[17] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "SoK: Consensus in the Age of Blockchains," in *AFT'19*.
[18] M. Vukolić, "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication," in *iNetSec'15*.
[19] J. Sousa, A. Bessani, and M. Vukolić, "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform," in *DSN'18*.
[20] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The Honey Badger of BFT Protocols," in *CCS'16*.
[21] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies," in *SOSP'17*.
[22] P. Li, G. Wang, X. Chen, and W. Xu, "Gosig: Scalable Byzantine Consensus on Adversarial Wide Area Network for Blockchains," in *SoCC'20*.
[23] M. Wahl, "Survey of Railway Embedded Network Solutions: Towards the Use of Industrial Ethernet Technologies," *Synthèse INRETS*, 2010.
[24] IEC, "IEC 61375-2-5:2014," 2014.
[25] ——, "IEC 62279:2015," 2015.
[26] D. Li, W. Daamen, and R. M. Goverde, "Estimation of Train Dwell Time at Short Stops Based on Track Occupation Event Data: A Study at a Dutch Railway Station," *Journal of Advanced Transportation*, 2016.
[27] M. Castro and B. Liskov, "Proactive Recovery in a Byzantine Fault-Tolerant System," in *OSDI'00*.
[28] P. E. Veríssimo, "Travelling through Wormholes: A New Look at Distributed Systems Models," *SIGACT News*, 2006.
[29] K. Echtle and A. Masum, "A Multiple Bus Broadcast Protocol Resilient to Non-Cooperative Byzantine Faults," in *FTCS'96*.
[30] C. Stathakopoulou, S. Rüsch, M. Brandenburger, and M. Vukolic, "Adding Fairness to Order: Preventing Front-Running Attacks in BFT Protocols using TEEs," in *SRDS*, 2021.
[31] S. Duan, M. K. Reiter, and H. Zhang, "Secure Causal Atomic Broadcast, Revisited," in *DSN*, 2017.
[32] M. Kuperberg, D. Kindler, and S. Jeschke, "Are Smart Contracts and Blockchains Suitable for Decentralized Railway Control?" *arXiv*, 2019.
[33] M. Kuperberg, "Scaling a Blockchain-based Railway Control System Prototype for Mainline Railways: a Progress Report," *arXiv*, 2021.
[34] G. Muniandi, "Blockchain-Enabled Virtual Coupling of Automatic Train Operation Fitted Mainline Trains for Railway Traffic Conflict Control," *IET Intelligent Transport Systems*, 2020.
[35] L. Zhang, Y. Huang, and T. Jiang, "High-Speed Railway Environmental Monitoring Data Identity Authentication Scheme Based on Consortium Blockchain," in *ICBTA'19*.
[36] J. D. Preece and J. M. Easton, "Blockchain Technology as a Mechanism for Digital Railway Ticketing," in *Big Data'19*.
[37] F. Naser, "Review: The Potential Use of Blockchain Technology in Railway Applications – An Introduction of a Mobility and Speech Recognition Prototype," in *Big Data'18*.
[38] J. Preece and J. Easton, "A Review of Prospective Applications of Blockchain Technology in the Railway Industry," *Preprint submitted to Int. J. Railw. Technol.*, 2018.
[39] C. Oham, S. S. Kanhere, R. Jurdak, and S. Jha, "A Blockchain Based Liability Attribution Framework for Autonomous Vehicles," *arXiv*, 2018.
[40] H. Guo, E. Meamari, and C. Shen, "Blockchain-inspired Event Recording System for Autonomous Vehicles," in *HotICN'18*.
[41] M. Cebe, E. Erdin, K. Akkaya, H. Aksu, and S. Uluagac, "Block4Forensic: An Integrated Lightweight Blockchain Framework for Forensics Applications of Connected Vehicles," *IEEE Communications Magazine*, 2018.
[42] R. W. van der Heijden, F. Engelmann, D. Mödinger, F. Schönig, and F. Kargl, "Blackchain: Scalability for Resource-Constrained Accountable Vehicle-to-X Communication," in *SERIAL'17*.
[43] C.-S. Shih, W.-Y. Hsieh, and C.-L. Kao, "Traceability for Vehicular Network Real-Time Messaging Based on Blockchain Technology," *JoWUA*, 2019.
[44] R. White, G. Caiazza, A. Cortesi, Y. I. Cho, and H. I. Christensen, "Black Block Recorder: Immutable Black Box Logging for Robots via Blockchain," *IEEE RA-L*, 2019.
[45] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "ChainSplitter: Towards Blockchain-based Industrial IoT Architecture for Supporting Hierarchical Storage," *Blockchain'19*.