CrossTalk: Making Low-Latency Fault Tolerance Cheap by Exploiting Redundant Networks

ANDREW LOVELESS, NASA Johnson Space Center, USA
LINH THI XUAN PHAN, University of Pennsylvania & Roblox, USA
LISA ERICKSON, Georgia Institute of Technology, USA
RONALD DRESLINSKI, University of Michigan, USA
BARIS KASIKCI, University of Washington, USA

Real-time embedded systems perform many important functions in the modern world. A standard way to tolerate faults in these systems is with Byzantine fault-tolerant (BFT) state machine replication (SMR), in which multiple replicas execute the same software and their outputs are compared by the actuators. Unfortunately, traditional BFT SMR protocols are *slow*, requiring replicas to exchange sensor data back and forth over multiple rounds in order to reach agreement before each execution. The state of the art in reducing the latency of BFT SMR is *eager execution*, in which replicas execute on data from different sensors simultaneously on different processor cores. However, this technique results in 3–5× higher computation overheads compared to traditional BFT SMR systems, significantly limiting schedulability.

We present CrossTalk, a new BFT SMR protocol that leverages the prevalence of redundant switched networks in embedded systems to reduce latency without added computation. The key idea is to use specific algorithms to move messages between redundant network planes (which many systems already possess) as the messages travel from the sensors to the replicas. As a result, CrossTalk can ensure agreement *automatically* in the network, avoiding the need for any communication between replicas. Our evaluation shows that CrossTalk improves schedulability by 2.13–4.24× over the state of the art. Moreover, in a NASA simulation of a real spaceflight mission, CrossTalk tolerates more faults than the state of the art while using nearly 3× less processor time.

CCS Concepts: • **Computer systems organization** → **Fault-tolerant network topologies**; **Real-time system architecture**; **Embedded software**; • **Theory of computation** → **Distributed algorithms;**

Additional Key Words and Phrases: Byzantine fault tolerance, state machine replication, real-time systems, distributed systems

ACM Reference format:
Andrew Loveless, Linh Thi Xuan Phan, Lisa Erickson, Ronald Dreslinski, and Baris Kasikci. 2023. CrossTalk: Making Low-Latency Fault Tolerance Cheap by Exploiting Redundant Networks. *ACM Trans. Embedd. Comput. Syst.* 22, 5s, Article 147 (September 2023), 25 pages.
https://doi.org/10.1145/3609436

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2023.
This research was supported in part by the NSF Graduate Research Fellowship (award DGE 1256260) and NSF grants CNS-2111688, CNS-1750158, CNS-1955670 and CNS-1703936.
Authors' addresses: A. Loveless, NASA Johnson Space Center, USA; L. Thi Xuan Phan, University of Pennsylvania & Roblox, USA; L. Erickson, Georgia Institute of Technology, USA; R. Dreslinski, University of Michigan, USA; Baris Kasikci, University of Washington, USA.
Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1539-9087/2023/09-ART147 $15.00
https://doi.org/10.1145/3609436

1 INTRODUCTION

Real-time embedded systems are responsible for many critical functions in today's world, from steering vehicles [53, 61] to controlling energy generation systems [52, 72]. A common way that these systems tolerate faults, such as memory corruptions in nodes [73] and stuck bits in network cards [36], is with Byzantine fault-tolerant (BFT) state machine replication (SMR) [47, 53, 61, 65, 77]. In BFT SMR, the same function is performed simultaneously by multiple nodes (or *replicas*) and their outputs are compared. To ensure the non-faulty replicas have the same state (and can thus out-vote faulty replicas), it is necessary to ensure they all operate on identical inputs [53]. Traditionally, this is done by having the replicas run an *agreement protocol* on all sensor data they receive before each execution [61]. An example of a traditional BFT SMR protocol is shown in Figure 1(a).

One problem with traditional BFT SMR protocols is that reaching agreement on sensor data is *slow*. It is well known that replicas must exchange messages for at least $f + 1$ communication rounds to reach agreement if f replicas may be faulty [41]. If the replicas are not co-located, as is typical in order to tolerate scenarios like fires or attacks that could disable all replicas [48, 76], these messages must traverse several network hops in each round, resulting in significant added latency. Moreover, additional latency comes from the need for tasks on the replicas to read, process, and send messages between rounds [61]. Overall, the latency of ensuring agreement can be a significant portion of the time needed to execute the overall BFT SMR protocol (e.g., 50% or more), making it difficult or even impossible to meet certain hard deadlines or successfully schedule BFT tasks [61].

A recent RTAS paper, Igor [61], was designed to address this problem. Figure 1(b) shows an overview of Igor. Instead of reaching agreement on sensor data *before* executing, replicas in Igor reach agreement while execution on the sensor data is underway. Thus, in cases where agreement takes less time than execution, Igor's latency matches that of a non-BFT system (i.e., it is optimal). Unfortunately, while traditional protocols allow the replicas to down-select redundant sensor values to a *single* value before executing, Igor requires replicas to simultaneously execute on data from *every* redundant sensor. This means that: (1) Igor cannot be used on single-core processors (which are still common in practice [62]) and (2) Igor's computation overhead is typically at least 3× higher than traditional protocols. Moreover, Igor requires $3f + 1$ replicas to tolerate f faults [61], while traditional protocols often only need $2f + 1$ [54].[1]

We observe that the reason Igor is so computationally expensive (and traditional protocols are so slow) is that, in order to be general, they make no assumptions about the network topology — instead treating the replicas as if they are interconnected by point-to-point channels [61]. As a result, they cannot exploit the redundancy and connectivity that often *already exist* in the network in order to increase performance.

We present CrossTalk, a new BFT SMR protocol that minimizes latency (matching or beating Igor) *without* requiring multi-core processors, *without* requiring any additional processing or replicas compared to traditional BFT protocols, and while remaining generally applicable. The key insight is that embedded systems requiring BFT SMR are all moving towards or have already adopted the same general network architecture, which we refer to as a *redundant switched network* [11, 13, 63, 75]. In these networks, devices communicate through switches, and the whole network is replicated to form redundant *planes*. We observe that this design already provides enough redundancy and *nearly enough* connectivity to satisfy the "$f + 1$ round" [41] requirement for agreement, merely as a consequence of the paths messages take through the network.

[1] Cryptographic protocols (i.e., protocols with imperfect correctness) in the synchronous setting need only $2f + 1$ replicas [54], while Igor requires $3f + 1$ replicas even if perfect correctness is not required [61].

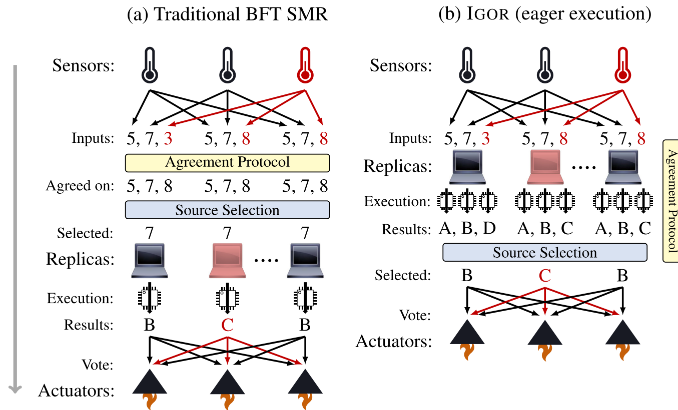

Fig. 1. Example executions of a traditional BFT SMR protocol (left) and the state-of-the-art Igor protocol (right). The network is not shown. Sensor 3 and replica 2 are faulty. Sensor 3 transmits conflicting values to different planes. Such behavior can have multiple causes in practice, including hardware faults [36] and radiation-induced upsets [61], and tolerating them is a major focus in industry [26, 58].

CrossTalk works by taking these redundant switched networks that many embedded systems *already use in practice*, and adding a small number of new connections, or *cross-links*, between the planes. We show in Section 3.2 that by exploiting state-of-the-art techniques to restrict the behavior of faulty switches, it is possible to add these cross-links safely. With these cross-links in place, and by moving messages between the planes according to a specific algorithm, CrossTalk can solve agreement *entirely in the network* as messages travel from the sensors to the replicas. In other words, agreement happens in a *single* round. Importantly, CrossTalk does not require any non-standard capabilities in the switches, and thus can be used with existing network hardware and protocols.

One unique challenge of CrossTalk's single-round design is tolerating timing faults, such as a sensor transmitting later than expected. In fact, we prove — to our knowledge, for the first time — that it is *impossible* to tolerate all timing faults with a single-round agreement protocol in synchronous systems. The main intuition is that, while the synchronous model assumes *bounds* on network latency, the *exact* latency is not known [38]. Thus, a message may arrive slightly before a deadline for some replicas (and thus be accepted) but after the deadline for others. In Section 4.4, we show how CrossTalk can circumvent the impossibility result using network timestamps.

We developed a prototype of CrossTalk for a NASA flight software framework [67] and evaluated it in a real avionics development lab. CrossTalk achieved comparable or lower latency than the state of the art at a fraction of the computation cost, resulting in 2.13–4.24× better schedulability (Section 6.2). By requiring fewer replicas, CrossTalk also reduced mass and cost (Section 6.3). We also executed simulated flight software from a real spaceflight mission and found that CrossTalk met all deadlines while tolerating more faults than the state of the art and using nearly 3× less CPU time.

In summary, we make the following contributions:

- CrossTalk: a novel BFT SMR protocol that exploits redundant switched network topologies that systems already possess to achieve low latencies without added computation costs (Section 4).
- A new proof on the impossibility of tolerating timing faults with single-round agreement protocols (Section 4.3) and a novel method for overcoming this result (Section 4.4).

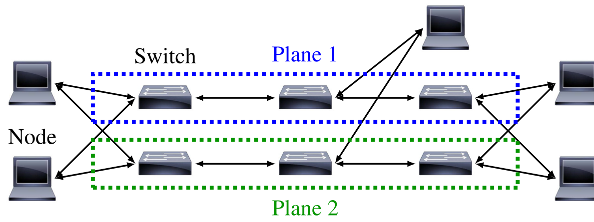

Fig. 2. Example redundant switched network with two planes. Nodes here may be replicas, sensors, or actuators. Note that the topology *inside* a plane can take many different forms, including a star or ring.

- An experimental evaluation of CrossTalk's performance and schedulability on a real avionics testbed (Section 6).
- A case study using CrossTalk in a real spaceflight abort scenario (Section 6.5).

2 BACKGROUND: REDUNDANT SWITCHED NETWORKS

Over the past decade, the need for higher bandwidths and increased scalability in embedded systems has driven designers to adopt switched networks. For example, aircraft have replaced ARINC 429 and 629 buses with Avionics Full-Duplex Switched Ethernet (AFDX) and Fibre Channel [43, 75]. Spacecraft have replaced MIL-STD-1553 buses with AFDX, Time-Triggered Ethernet (TTE), and SpaceFibre [43, 61, 63]. Trains and industrial control systems are replacing specialized fieldbuses with Time-Sensitive Networking (TSN) [11, 13].

Today, the most dominant strategy for tolerating switch faults in these systems is to replicate the entire network to form multiple *planes* (2–3 planes are typical) [59, 63, 75]. Each node is connected to all planes. To minimize latency, all planes are typically active simultaneously [2, 7, 13]. A node communicates by sending the same message to the destination over all planes. Figure 2 depicts this *redundant switched network* topology. This is the standard network topology in AFDX [2] and TTE [7]. It is also being used in standardization work on IEEE P802.1DP (TSN aerospace profile) [17]. Today, it can be found in a variety of systems, including commercial aircraft [31, 75], spacecraft (including all modern NASA crewed spacecraft) [12, 59], rockets [40], military helicopters [5, 27], and wind turbines [72].

Despite their redundancy, redundant switched networks are highly susceptible to Byzantine faults, where different replicas observe different data from the same transmitter (e.g., a sensor) [36]. Such fault manifestations are possible even if the switches are fault-free. We show one example in Figure 3(a). Consider a sensor that attempts to broadcast the same message to all replicas. On the physical layer, this is done by sending an identical message X out both of the ports on its network interface card towards both network planes [2, 7]. However, if the sensor experiences a fault — for example, due to cosmic radiation [61] — the sensor could incorrectly transmit a different message Y out one of its network ports. Such scenarios can be caused by, for example, the contents of the original message being altered to form a new valid message (e.g., with valid frame check sequence), or the transmitter incorrectly replaying an old message. Once in the network, the exact traversal time of each message depends on network loading and contention on each plane; hence, message X may arrive first at some replicas (top replica in Figure 3(a)), while Y may arrive first at others (bottom replicas in Figure 3(a)). The standard redundancy management scheme in most redundant switched networks, including AFDX, TTE, and TSN, is to accept the first valid message copy to arrive on any plane (discarding the redundant copies that arrive later) [2, 7, 9]. Thus, some replicas may accept X while others accept Y, resulting in disagreement.

The possibility of switch faults creates several new opportunities for Byzantine behavior. This is true even if switch faults are restricted to being benign (e.g., a faulty switch can only drop

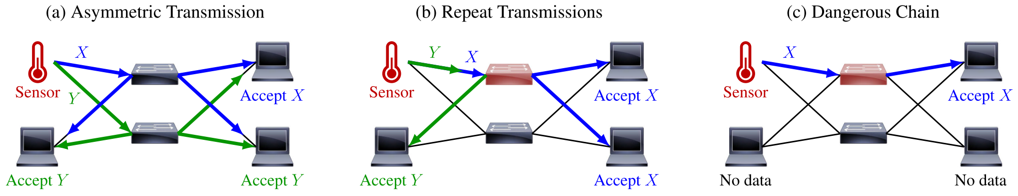

Fig. 3. Examples of Byzantine behavior in a redundant switched network. In (a), only the sensor is faulty. In (b) and (c), both the sensor and one switch are faulty. Note that all these examples are possible even if faulty switches are limited to dropping messages.

messages). One example is shown in Figure 3(b). Again, a sensor attempts to broadcast the same message to all replicas. However, a fault causes the sensor to transmit no message out one of its network ports but two messages, X and Y, out its other network port. Such behavior is typical of a babbling or slightly-off-specification sensor, where the next measurement (Y) is transmitted too early [36, 49]. If a switch in the plane that received the messages is faulty, the switch may drop X when forwarding it to some replicas and drop Y when forwarding it to some *other* replicas. As a result, some replicas only receive X and other replicas only receive Y.

Lastly, we note that Byzantine behavior is possible even *without* a sensor transmitting multiple values. An example is shown in Figure 3(c). This time, when the sensor broadcasts its message X, a fault results in X being sent out only one of its network ports. If the switch that receives X is faulty, it may forward X directly to one replica but drop the message on all other ports. Thus, only one replica, r_1, receives X, and all other replicas receive no message.

This scenario (in Figure 3(c)) is more difficult to handle than it appears. For example, if we *knew* r_1 was non-faulty, we could ensure agreement by having r_1 broadcast X to all replicas via all planes. However, if r_1 is faulty, it may send X to only the faulty plane, which again may drop the message for all but one replica. It has been shown that in the worst case, this scenario — a replica transmitting to only one device — can prevent agreement for up to $f + 1$ rounds of exchange [35, 41].[2]

As we will show, CrossTalk exploits the redundant switched network topology systems already use to make masking these Byzantine faults fast and efficient.

3 MODELS

3.1 System Model

Our system model is shown in Figure 4. The system consists of replicas, sensors, and actuators (collectively called *nodes*) connected to a redundant switched network, as described in Section 2. There are n network planes, and each plane has a unique identifier $\{1, \ldots, n\}$. We use b_{ik} to refer to the switch in plane i at position k. We assume the topologies of the planes are all identical, and that a given node connects to the planes at the same switch position on all planes (e.g., b_{1k} and b_{2k}). We make these assumptions to reflect systems in practice [2, 7, 12, 27, 31, 40, 59, 72, 75] and to simplify the presentation of the protocols in Section 4. However, CrossTalk could also be extended to work in networks where these assumptions do not hold.[3]

[2] We note that the possibility of Byzantine behavior is not unique to switched networks. It has been shown that even when the network consists only of a single broadcast bus, marginal signals transmitted by a sensor can be interpreted as different messages at different replicas [36].

[3] There is nothing about the CrossTalk approach that explicitly *requires* the network planes to be identical. Even if the planes differ, correctness can be assured as long as there are enough planes and cross-link positions (both $g + 1$ as

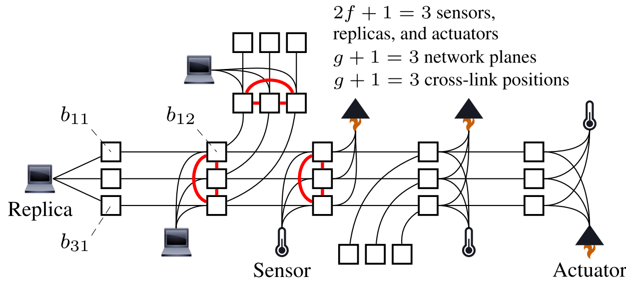

Fig. 4. CrossTalk's system model (when $f = 1, g = 2$). The squares are switches and the lines are links. The nodes can connect at any position on the network. The *only* thing we assume that systems in practice may not already contain are the cross-links (shown in red).

We assume the system is *synchronous*, meaning there are a priori known bounds on the time it takes nodes to execute tasks and for messages to traverse each switch. These assumptions are often satisfied using specialized real-time operating systems and deterministic switched networks [43, 59, 75]. As in past work [61, 73], we also assume nodes are synchronized within a bounded skew. A variety of fault-tolerant synchronization protocols could be used for this purpose [7, 28].

We assume messages take a pre-planned route from a sender, through the switches, and to one or more destination nodes. We refer to each such route as a *virtual channel*. Each message contains an identifier specifying its virtual channel. The switches are pre-programmed with a routing table indicating, for each message with a specific ID that arrives on a given ingress port, to which egress ports the message should be forwarded. If a message with ID x arrives on a switch port for which no route exists for ID x, the message is dropped. We use "sending on virtual channel x" to refer to transmitting a message containing the ID for virtual channel x, and "receiving on virtual channel x" to refer to receiving a message with the ID for virtual channel x.

Cross-links between planes. The only somewhat non-standard assumption we make, which enables the CrossTalk approach, is that switches at a subset C of switch positions in different planes are connected to one another. That is, for all $c \in C$, b_{ic} is connected to b_{jc} for any two planes i and j. We call these connections *cross-links* and the positions in C *cross-link positions*. We formalize the required number of cross-link positions in Section 3.2. The choice of C does not impact correctness, but does impact latency. We describe how to choose C to minimize latency in Section 4.

The required cross-links for CrossTalk can be added to systems with low size, weight, and power impacts. The reason is that, as shown in Figure 4, cross-links are only added between redundant versions of the *same* switch (i.e., switches at the same position in different planes). Often these switches are co-located to reduce harness mass. As we will show in Section 6.3, CrossTalk's cross-links add negligible mass and cost to the system. In fact, because CrossTalk requires fewer replicas than the state-of-the art (Igor [61]), CrossTalk actually *reduces* mass and cost overall (Section 6.3).[4]

explained in Section 3.2), and messages are routed through the cross-link positions on the way to the replicas (as described in Section 4).

[4]We acknowledge that, in cases where the redundant planes are physically separated (e.g., the fuselage of an aircraft), the cross-links may be long. However, we do not see this as a significant barrier to adopting CrossTalk— both because cross-links have very low mass (e.g., 0.1 kg/m [24]) and because, in such cases, there is typically ample space available to route cables between planes (e.g., along the wall [23]).

We note that CrossTalk is *not alone* in using cross-links between planes. In fact, some spacecraft systems already contain redundant switched networks with cross-links [60]. Emerging automotive and industrial control systems are also based on similar architectures [15, 20, 69].

3.2 Fault Model

We assume a *Byzantine* fault model for the nodes, where in a system with $2f + 1$ sensors, $2f + 1$ replicas, and $2f + 1$ actuators, up to f sensors, f replicas, and f actuators can deviate arbitrarily from the protocol ($\geq 2f + 1$ redundancy is needed to solve BFT SMR [54, 61]). This includes corrupting messages, sending different messages to each plane, and sending messages out of order [54, 61]. *Initially,* we assume faults do not occur in the time domain — i.e., messages from faulty nodes are generated in an a priori known bounded time [25]. This assumption is common in past work [73]. In Section 4.3, we relax this assumption to allow nodes to fail arbitrarily in the time domain.

The only restriction is that a babbling node cannot consume all the network bandwidth in order to prevent messages sent by other nodes from being delivered. This is almost always prevented in practice by using a bandwidth allocation system monitored and enforced by the switches [2, 71].

We consider a more restricted fault model for the switches, where in a system with $g+1$ network planes and at least $g + 1$ cross-link positions (i.e., $\geq (g + 1)^2$ switches have cross-links), up to g switches can be *omission* faulty [7]. This means faulty switches can fail to forward any messages they receive to any receivers, potentially resulting in some receivers getting a message and others not [7]. However, faulty switches cannot undetectably corrupt messages, create messages, or delay messages an unbounded amount of time [7, 25]. This is the standard switch fault model assumed in a variety of popular redundant switched networks, including AFDX [2, 28], TTE [7, 58], and SpaceFibre [71], and it is therefore used in a wide range of systems in practice. We surveyed engineers working on a variety of Byzantine fault-tolerant embedded systems and found that there are several reasons this is the case:

(1) Switch manufacturers use specialized hardware techniques to limit switches to benign faults. For example, switches use a "COM/MON" design in which all logic is hosted on two independent processors, each with separate power and clock domains [70] and potentially operating on independent memory. The switch cannot send a message unless both processors produce the same message at the same time [37, 58]. Such techniques are very expensive to develop and verify [46], but the high cost can be amortized across many programs using the same switch. In contrast, applying the same techniques to nodes (e.g., sensors), which often perform at least some amount of local computation [61], would be prohibitively expensive due to the need to design multiple point solutions for different types of devices.
(2) System designers use safety-layer protocols with features like layered CRCs, sequence number checking, and timestamp checking to minimize the probability of a faulty switch undetectably altering or replaying messages that it receives. [81, 82]. In contrast, safety-layer protocols are not an effective means of restricting the behavior of faulty nodes (e.g., sensors), which are the *originators* of messages, and thus may simply generate incorrect messages before correctly encapsulating them with the safety-layer protocol.
(3) Other critical parts of the system break if faulty switches can exhibit Byzantine behavior. For example, we are not aware of any BFT time synchronization protocol used in practice that tolerates more severe switch faults than omission [7, 28]. In contrast, systems are often designed to tolerate Byzantine faulty nodes [7, 30, 53, 63].

The omissive behavior of modern switches prevents CrossTalk's approach from compromising fault tolerance or complicating fault analysis. As long as the redundant network planes are powered independently and galvanic isolation is used on the links between planes,[5] a fault in one plane cannot propagate to another plane. Thus, the only way that one plane can negatively impact another is by passing messages over the cross-links. This can theoretically cause problems in only two ways. First, if the messages are incorrect or unsafe (e.g., generated by a faulty node) and the system acts on them. However, CrossTalk is specifically designed and proven to tolerate such faulty traffic (see Sections 4.1, 4.2). Second, babbling traffic from one plane could prevent messages from being delivered on another plane. However, as described at the start of this section, modern switches use bandwidth allocation or credit systems to prevent received messages from using up switch buffers and network bandwidth [2, 71]. For example, AFDX switches will drop any traffic that arrives faster than specified in a preloaded configuration table [2]. Since switches are limited to omission, this rate limiting cannot be violated even if the switches are faulty.

Finally, we note that unlike CrossTalk, most prior work on BFT SMR (including Igor [61]) does not consider switch faults [54, 73]. Thus, CrossTalk's fault model is strictly more general.

4 DESIGN

This section describes CrossTalk, a new low-cost BFT SMR protocol. CrossTalk has two main goals: (1) *Low latency* — CrossTalk aims to be fast in both the presence and absence of faults, and (2) *Low computation overhead* — CrossTalk aims to be usable on single-core processors and to have minimal processing overhead. In contrast, the state of the art in low-latency BFT protocols has high overheads and requires multi-core processors [61].

The key idea of CrossTalk, which allows it to achieve these goals, is to solve agreement *indirectly* as a consequence of routing messages between network planes. An overview is shown in Figure 5. ❶ First, each redundant sensor sends a message with its value to each plane. Rather than traveling directly to the replicas, the message is first routed through $g + 1$ switches with cross-links (i.e., switches in C). Note that two sensors at different positions in the network may send their messages through the cross-link switches in different orders to minimize latency. Moreover, if $|C| > g+1$, each sensor's messages may travel through a *different* set of $g+1$ cross-link switches (i.e., C can be different from each sensor's perspective). ❷ As a message travels through a cross-link switch on a given plane k, the switch forwards a copy of the message to the "next" plane $k+1$. Each copy continues to traverse the cross-link switches, which in turn create copies and forward them to the next plane (until no such plane exists). ❸ After traversing $g + 1$ cross-link positions, each message is routed to the replicas. The protocol ensures that any message that arrives at one replica (potentially from a faulty plane), must arrive at all replicas. ❹ The rest of the protocol continues as in a traditional BFT SMR system (Figure 1(a)), with replicas using source selection to pick a sensor value, executing on that sensor value, and sending results to the actuators.

CrossTalk reduces latency in two ways. First, the agreement process happens at *network speed*, with no need for any communication between replicas. That is, the time to reach agreement is bounded only by how fast the switches can forward messages to the replicas. Second, the cross-link positions can be chosen strategically. Latency is minimized when messages are not forced to go "out of their way" in order to traverse the cross-link switches before proceeding to the replicas. An example is shown in Figure 6. Designing an algorithm to find optimal cross-link positions is beyond the scope of this paper. In our evaluation (Section 6.1), we used a combination of depth-first and brute-force search to find cross-link positions that attempt to minimize latency.

[5] Ethernet-based networks are already transformer coupled by default, which provides the necessary isolation [10].

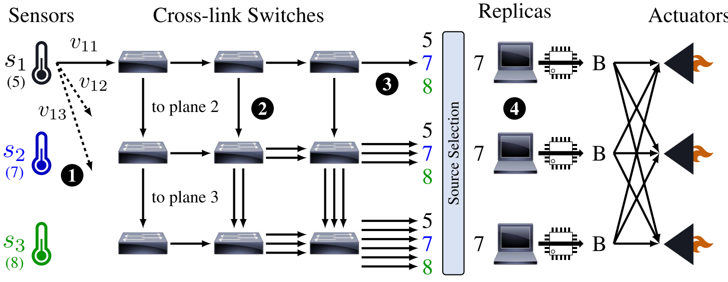

Fig. 5. Example execution of CrossTalk ($f = 1, g = 2$). We only show the flow of traffic sent by one sensor to one plane (v_{11}). However, the sensor also sends traffic to the other planes (v_{12} and v_{13}). Similar traffic flows exist for each sensor. For clarity we do not depict faults. Note that there may be any number of switches upstream or downstream of the cross-link switches, which are not shown.

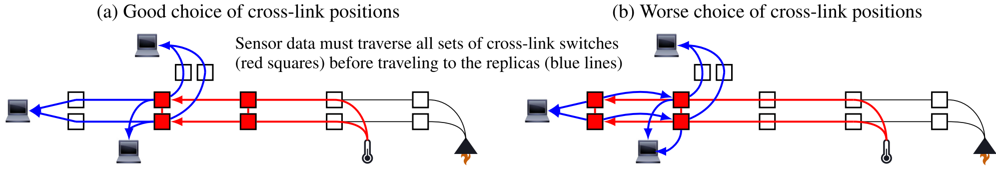

Fig. 6. Example of strategically choosing cross-link positions (red switches) to minimize latency in a system with 3 replicas and 2 planes. We only show one sensor for clarity. Note that in (a), traffic does not deviate from the shortest path on its way from the sensor to the replicas.

CrossTalk's approach also results in significantly lower computation overhead than the state of the art. Most importantly, since the replicas are not involved in agreement, the agreement process is *agnostic to the number of replicas*. Thus, unlike other systems with perfect correctness (i.e., not relying on cryptography [61]), CrossTalk requires only $2f + 1$ replicas (enough to mask output errors) instead of $3f + 1$ [54]. Moreover, unlike Igor, CrossTalk does not require replicas to perform any redundant computations on different cores.

Below we describe CrossTalk in more detail. We focus on the rules for routing sensor data to the replicas and the agreement process, since these aspects are unique to CrossTalk.

4.1 Routing Sensor Data to Replicas

This section describes the virtual channels CrossTalk uses to route sensor data to the replicas. Data from each sensor is routed using n different virtual channels — one corresponding to each plane. Let s_i be a sensor that sends data to the replicas. Let m_{ij} be a message s_i sends on plane j. Let v_{ij} be the virtual channel used to carry m_{ij}.

The routing rules for v_{ij} are shown in Protocol 1. Let P be a list of all the plane identifiers ($\{1, \ldots, n\}$), with ID j appearing first. The remaining order does not impact correctness but, as discussed next, can be adjusted to balance the traffic load across planes. We assume that C is sorted such that latency is minimized as m_{ij} travels through the switches in C to the replicas (as shown in Figure 6). Let $P[1]$ and $C[1]$ refer to the first items in P and C, respectively.

Figure 5 shows the results of routing some m_{ij} (m_{11} in the example) through v_{ij} (v_{11}) in a simple network. In a system tolerating $g = 1$ faulty switches, each transmission on some v_{ij} results in 3 message copies reaching the replicas (as a result of the switches duplicating the message). When $g = 2$, each transmission results in 10 messages copies. As shown in the figure, more network traffic is generated on planes at higher indexes in P. Thus, to balance the traffic load across the

Protocol 1: Routing Rules for Sensor Data

- m_{ij} is routed from s_i along plane j to switch b_{jc}, where $c \leftarrow C[1]$
- **for each** $x \leftarrow 1, \ldots, n$ **do:** *# for all planes*
  Let $p \leftarrow P[x]$
  for each $y \leftarrow 1, \ldots, g+1$ **do:** *# for all cross-link positions*
    Let $c \leftarrow C[y]$
    if $x \neq n$ **then:** *# route to next plane*
      b_{pc} routes m_{ij} to b_{qc}, where $q \leftarrow P[x+1]$
    if $y \neq g+1$ **then:** *# route to next cross-link position*
      b_{pc} routes m_{ij} to b_{pd}, where $d \leftarrow C[y+1]$
- **for each** $x \leftarrow 1, \ldots, n$ **do:** *# for all planes*
  Let $p \leftarrow P[x]$ and $c \leftarrow C[g+1]$ *# last cross-link position in plane*
  m_{ij} is routed from b_{pc} along plane p to all replicas

planes, P should be chosen so the same plane does not appear at the same index in P for multiple virtual channels originating at the same sensor. For example, P might contain {1, 2, 3} for v_{i1}, but {2, 3, 1} and {3, 1, 2} for v_{i2} and v_{i3}, respectively.

As a result of routing sensor data according to Protocol 1, we have the following guarantees.

Lemma 4.1. *If a non-faulty replica r receives a message m on v_{ij}, then all non-faulty replicas receive m on v_{ij} (agreement). In addition, if s_i and some plane j are non-faulty, and if s_i sends m, then all non-faulty replicas receive m on v_{ij} (validity).*

Proof. First, consider agreement. Since switches are limited to omission, a faulty switch cannot create or alter messages. Also, switches drop messages that arrive on ports where no routing rules exist. Thus, m must have traversed a legal route in v_{ij}. Since $|C| \geq g+1$, there must be at least one $c \in C$ for which b_{pc} is non-faulty for all $p \in \{1, \ldots, n\}$ (i.e., all cross-link switches at position c are non-faulty). According to Protocol 1, m's route to r must have entered such a b_{pc} from some plane $p = P[x]$. At that point, m is forwarded through all planes $P[x], \ldots, P[n]$. Also, to get to plane $P[x]$, m must have traversed planes $P[1], \ldots, P[x]$. Thus m traverses all planes $P[1], \ldots, P[n]$. Since there are $g+1$ planes, at least one plane k is non-faulty and forwards m through any remaining cross-over switches on plane k, and then to all replicas. Thus, all non-faulty replicas get m on v_{ij}.

Next, consider validity. Since s_i is non-faulty, it sends m to all planes (i.e., on $v_{i1}, \ldots, v_{in}$). Since plane j is non-faulty, m travels through each b_{jc}, where $c \in C$, and then to all replicas. Thus, all non-faulty replicas receive m on v_{ij}. □

We note that other routing rules could be used with CrossTalk besides those in Protocol 1. For example, a system could tolerate more scenarios with *more* than g faulty switches if each cross-link switch forwarded messages to *all* planes and not just the *next* plane. However, such approaches have much higher communication costs, and still cannot tolerate all $g+1$ faulty switch scenarios (which is impossible with only $g+1$ planes, since all planes could be blocked).

4.2 Agreement and BFT SMR Protocols

With the virtual channels for routing sensor data to the replicas defined, we can now describe a simple protocol that ensures replicas agree on data from a potentially faulty sensor s_i. The protocol starts at a synchronized time t', at which s_i transmits and the replicas start gathering messages. It ends at a predetermined time $t = t' + \Delta$, which is chosen to accommodate the known worst-case latency from s_i to all replicas. In general, the protocol works by having the replicas select among the messages received on each virtual channel using a simple priority-based scheme.

The protocol is shown in Protocol 2. Let M_i be an n-dimension vector used by the replicas to store messages from sensor s_i (i.e., store what s_i sent to each plane). All elements are initialized to $\perp$ to indicate the messages are initially missing. Let m_i be the final message the replicas accept from s_i. It is initialized to a default value.

Protocol 2: Agreement Protocol

Sensor s_i (at time t'):

- Send message m on virtual channels $v_{i1}, \ldots, v_{in}$

Each replica (at time $t = t' + \Delta$):

- **for each** $j \leftarrow 1, \ldots, n$ **do:** *# for all planes*
  Let $W_{ij} \leftarrow$ the messages received on v_{ij}
  if W_{ij} contains only one unique message m **then:**
  Set $M_i[j] \leftarrow m$ *# set value received from s_i on v_{ij}*
- **for each** $j \leftarrow 1, \ldots, n$ **do:** *# for all planes*
  if $M_i[j] \neq \perp$ **then:** Set $m_i \leftarrow M_i[j]$ *# priority-based selection*

Theorem 4.2. *All non-faulty replicas agree on m_i (agreement). If s_i is non-faulty and sends m, then m_i is m (validity).*

Proof. First, consider agreement. All non-faulty replicas decide on m_i by applying the same priority-based selection to M_i. Thus, we simply need to prove all non-faulty replicas agree on M_i. We make the proof by contradiction. Say that at the end of the protocol, two non-faulty replicas r_1 and r_2 have different values $M_i[j]$ for some j. This means r_1 and r_2 have different sets W_{ij}. Thus, one replica (say r_1) received a message m on v_{ij} in time interval $[t', t' + \Delta]$ that the other (r_2) did not. Lemma 4.1 implies r_2 also receives m. Moreover, since Δ is chosen to accommodate the worst-case traversal time from the sensor to the replicas, sensor s_i is not subject to timing faults (we relax this assumption in Section 4.4), and faulty switches are omissive and thus cannot delay messages, m must arrive at r_2 in $[t', t' + \Delta]$. This is a contradiction. Thus, all non-faulty replicas agree on m_i.

Next, consider validity. Since at least one plane j is non-faulty, Lemma 4.1 implies all non-faulty replicas receive m on v_{ij}. Since switches are limited to omission, they cannot create or alter messages. Similarly, all switches drop messages that arrive on ports where no routing rules exist. Since s_i is non-faulty, it sent only m. Thus, non-faulty replicas can receive no message besides m on $v_{i1}, \ldots, v_{in}$. Thus, for all non-faulty replicas, $M_i[j] = m$ and M_i contains only m or $\perp$. Thus, all non-faulty replicas set $m_i \leftarrow m$. □

The rest of CrossTalk proceeds identically to a traditional BFT SMR system (see Figure 1(a)). Specifically, as a result of running Protocol 2 for each sensor, all non-faulty replicas possess an identical vector of sensor values (up to one per sensor). Next, the replicas use a source selection process to determine which of these values to use as the input to their execution. Source selection is application specific, but is often simply a mid-value selection [53]. It has been shown that, with $2f + 1$ sensors, the non-faulty replicas are guaranteed to select a value from a non-faulty sensor or a value bounded by values from non-faulty sensors [61].

Next, the replicas execute on the selected sensor value. If the same deterministic execution is performed on all replicas, all non-faulty replicas are guaranteed to produce the same output. Lastly, the replicas send their outputs to the actuators, with each output traveling from each replica to each actuator over each redundant plane. The actuators accept the first valid message that arrives

from each replica. The actuators perform a majority vote of the accepted messages to decide which operation to perform. Since there are $2f + 1$ replicas, the voted output is guaranteed to be correct.

4.3 The Trouble with Timing Faults

In the previous section, we made the simplifying assumption that the sensors were not subject to timing faults — i.e., if a faulty sensor generates a message, it is guaranteed to do so within an a priori known bounded time. Having a bound allows designers to calculate a time t at which, if a replica is going to receive a message from a sensor, that message is guaranteed to have arrived. When combined with the design of the virtual channels, it ensures that all non-faulty replicas agree on the data from all redundant sensors at time t (see Protocol 2).

Traditional agreement protocols tolerate timing faults as a consequence of requiring multiple communication rounds between the replicas. For example, a faulty sensor could delay sending its message m until right before the deadline for the first round, t_1, resulting in some replicas getting m before t_1, and others not. However, in the next round, the replicas share the messages they received with each other. As a result, any replica that failed to receive m by t_1 will receive it by the deadline for the second round, t_2.

Unfortunately, tolerating timing faults in a protocol with a single communication round (from sensors to replicas) is not so easy. Since the replicas cannot communicate with each other, they have no way to differentiate between a message that arrived before the deadline for *some* replicas, and one that arrived before the deadline for *all* replicas.

The challenge of ensuring agreement in the presence of timing faults is not specific to CrossTalk. In fact, it can be shown that timing faults make it impossible for *any* protocol with only one communication round to ensure agreement by some synchronized deadline t. The result holds even if the network is reliable and has the guarantees of a broadcast channel [44] (i.e., it is impossible for the sensor to send a message to some replicas and not to others). This detail is important, because it means some recent guidance on constructing BFT protocols using broadcast channels [73] is actually *incorrect* if timing faults can occur.

Proposition 4.3. *In a synchronous system subject to timing faults, it is impossible for a protocol that uses only a single communication round to guarantee both the agreement and validity properties of Theorem 4.2 by an a priori known time t.*

Proof. We construct the proof by contradiction. Assume that such a protocol exists and that all replicas are non-faulty. We will show that there exists a scenario in which agreement is violated. Consider three possible scenarios below.

Scenario 1: Sensor s_i is non-faulty and sends m on time. m arrives at all the replicas before time t. By the validity property of the protocol, all replicas must set $m_i \leftarrow m$.

Scenario 2: Sensor s_i is faulty and crashes. Thus, no message arrives at any replica before t. By the agreement property of the protocol, all replicas must set m_i to the same message (e.g., a predetermined default).

Scenario 3: Sensor s_i is faulty and sends m after some arbitrary delay. Now, say we split the replicas into two sets — Set 1 and Set 2. The network latency in a synchronous system is bounded, but *not exactly known* (the exact latency depends on network loading) [54]. Thus, even if Set 1 and Set 2 are perfectly synchronized, m may arrive at Set 1 and Set 2 at slightly different times — e.g., before t for Set 1 and after t for Set 2. Even if t is increased, a faulty sensor can always delay longer to cause the same scenario. Alternatively, it is well known that perfect synchronization in a distributed system is not achievable [64] — meaning local time t on Set 1 may occur slightly after local time t on Set 2 in wall-clock time. Thus, even if the latencies from s_i to all replicas were *exactly known and the same*, m may arrive before t for Set 1 and after t for Set 2.

From the perspective of the replicas in Set 1, this scenario is identical to Scenario 1 — thus, all replicas in Set 1 must decide on m. From the perspective of the replicas in Set 2, this scenario is identical to Scenario 2 — thus, all replicas in Set 2 must decide on a default value. m and the default value may not be the same. Hence, the agreement property of the protocol is violated. □

4.4 Overcoming the Impossibility Result

In order to circumvent the impossibility result in Proposition 4.3, we need a way to bound when sensors can send messages to some safe temporal window, such that no replica can receive a message too early or too late. A Byzantine faulty sensor may send messages at arbitrary times regardless of any rules we impose on it. Thus, enforcement of this window must be outside the control of the potentially faulty sensor.

One solution is to use a time-triggered network, like TTEthernet [7] or IEEE 802.1Qbv [6], in which the switches are tightly synchronized to the nodes, and the exact timing of all message transmissions is scheduled offline. This design allows the switches to enforce a window during which a message from a node is allowed to arrive. Unfortunately, there are downsides to such networks, including the need for expensive specialized switches and network cards, as well as proprietary network scheduling tools [4].

A simpler approach is to use network timestamps. Let b_{jk} be the switch on plane j connected to sensor s_i. Then, in Protocol 1, have b_{jk} place the synchronized time that each message m arrives from s_i in m before forwarding m on v_{ij}. Since switches are constrained to omission (Section 3.2), meaning b_{jk} cannot put different timestamps in different outgoing copies of m, this scenario is indistinguishable from s_i sending messages to b_{jk} that *already* contain timestamps and b_{jk} not altering any messages. This means the agreement and validity conditions of Lemma 4.1 still hold. Moreover, since b_{jk} is limited to omission, the timestamp in m accurately reflects when m arrived at b_{jk}.

With this timestamping function in place, CrossTalk can be altered to tolerate timing faults by slightly altering Protocol 2. Specifically, rather than letting $W_{ij} \leftarrow$ all messages received on v_{ij}, let W_{ij} only contain messages with timestamps that fall within a specific *acceptance window*.[6]

Figure 7 defines suitable bounds for the acceptance window (which are similar to [74]). In addition, it defines the window during which messages timestamped within the acceptance window may arrive at the replicas (i.e., the *receive window*). The duration of the agreement protocol (i.e., the revised Protocol 2) is equal to the width of the receive window, which is $4(SkewMax) + L_{S\to SW} + L_{SW\to R}$, where $SkewMax$ is the maximum skew between any two synchronized devices, and $L_{S\to SW}$ and $L_{SW\to R}$ are the worst-case latencies from the sensor to switch b_{jk} and from b_{jk} to the replicas, respectively. In modern networks, $SkewMax$ values as small as a microsecond are possible [4, 8], while worst-case message latencies (i.e., $L_{S\to SW} + L_{SW\to R}$) are typically several milliseconds [57, 75]. Thus, the extra latency needed to make CrossTalk tolerate timing faults is largely insignificant.

With these windows defined, we can prove the correctness of the revised CrossTalk protocol.

Lemma 4.4. *If a non-faulty replica r receives a message m on v_{ij}, then if m was timestamped within the acceptance window, m must have arrived within r's receive window.*

Proof. Assume the receive window shown in Figure 7 starts at time 0 for each device. Since the switches are limited to omission and they drop messages that arrive on ports where no routing rules exist, the timestamp in m must be the time m entered the network on plane j at some switch

[6] We acknowledge that this technique requires replicas to have a priori knowledge of when sensors are scheduled to send messages. However, this knowledge is already available in synchronous flight software frameworks that we are familiar with (e.g., NASA's cFS [67]).

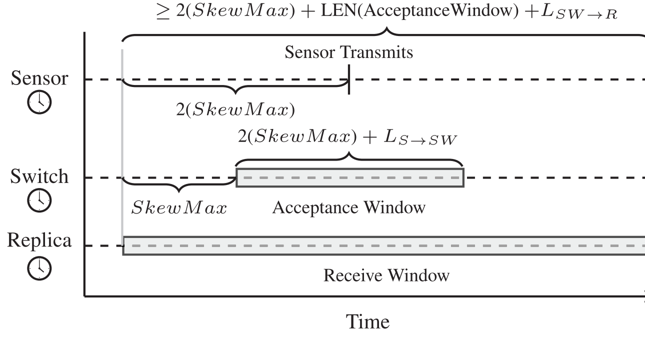

Fig. 7. Suitable time bounds for timestamping in CrossTalk (shown with no skew between devices).

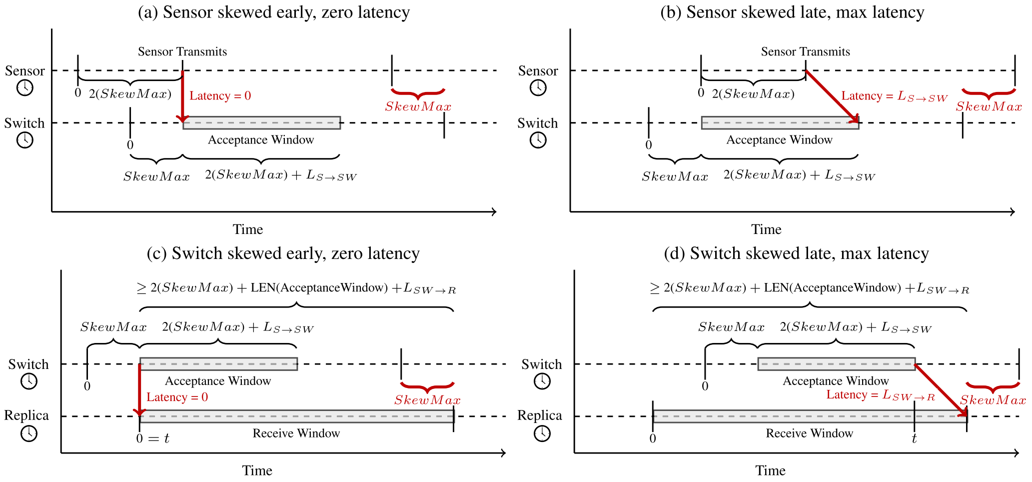

Fig. 8. Examples demonstrating the correctness of the time bounds in Figure 7.

b_{jk} (the switch directly connected to the sensor on plane j). Since switches are not subject to timing faults, the latest time at which r receives m (on r's clock) is the time b_{jk} timestamped m according to r's clock (call this time t) $+L_{SW \rightarrow R}$. Say b_{jk} is skewed by $SkewMax$ after r. In this case, t can be at most $4(SkewMax) + L_{S \rightarrow SW}$ at r — i.e., the end of the acceptance window, skewed right by $SkewMax$. This scenario is shown in Figure 8(d). Thus, m can arrive at r as late as $4(SkewMax) + L_{S \rightarrow SW} + L_{SW \rightarrow R}$ according to r's clock, which is at the end of r's receive window. Similarly, the earliest m can arrive at r is at t (if the network latency from b_{jk} to r is zero). Say b_{jk} is skewed by $SkewMax$ before r, in which case t can be as little as time 0 at r — i.e., the start of the acceptance window, skewed left by $SkewMax$. This scenario is shown in Figure 8(c). Thus, m can arrive at r as early as time 0 (per r's clock), which is in r's receive window. □

Lemma 4.5. *If s_i and some plane j are non-faulty, and s_i sends m, then all non-faulty replicas receive m on v_{ij} within their respective receive windows. Moreover, m is timestamped within the acceptance window.*

Proof. Lemma 4.1 implies all non-faulty replicas receive m on v_{ij}. Since s_i is non-faulty, it transmitted at the scheduled time ("Sensor Transmits" in Figure 7). Thus, m arrived at b_{jk} as early as time

$SkewMax$ on b_{jk}'s clock (shown in Figure 8(a)), and as late as $3(SkewMax) + L_{S \to SW}$ on b_{jk}'s clock (shown in Figure 8(b)). Thus m must contain a timestamp within the acceptance window. Thus, Lemma 4.4 implies m arrives at all non-faulty replicas within their respective receive windows. □

Theorem 4.6. *In the presence of timing faults, all non-faulty replicas agree on m_i (agreement). Also, if s_i is non-faulty and sends m, then m_i is m (validity).*

Proof. First, consider agreement. Like in the original Theorem 4.2, we simply need to prove all non-faulty replicas agree on M_i. We will prove by contradiction. Suppose that at the end of the protocol, two non-faulty replicas r_1 and r_2 have different values $M_i[j]$ for some j. This means r_1 and r_2 have different sets W_{ij}. Thus, one replica (say r_1) accepted a message m on v_{ij} that r_2 did not. This means m contained a timestamp within the acceptance window and arrived within r_1's receive window. Lemma 4.1 implies r_2 receives the same m on v_{ij} as r_1 (with same timestamp). Lemma 4.4 implies m arrives within r_2's receive window. Since r_1 is non-faulty and accepted m, r_2 also accepts m, which is a contradiction.

Next, consider validity. Since at least one plane j is non-faulty, Lemma 4.5 implies all non-faulty replicas receive the same m on v_{ij} within their respective receive windows, and that m's timestamp is within the acceptance window. Thus, all non-faulty replicas accept m. Since switches are limited to omission, they cannot create or alter messages. Similarly, all switches drop messages that arrive on ports where no routing rules exist. Since s_i is non-faulty, it sent only m. Thus, non-faulty replicas receive no message besides m on $v_{i1}, \ldots, v_{in}$. Thus, for all non-faulty replicas, $M_i[j] = m$ and M_i contains only m or $\perp$. Thus, all non-faulty replicas set $m_i \leftarrow m$. □

5 PROTOTYPE IMPLEMENTATION

To evaluate our approach, we implemented a prototype of CrossTalk in ~8800 lines of C and Python code (including supporting tooling). The replicas were realized on a cluster of 5 embedded AsRock J3160 computers with 1.6 GHz quad-core processors. We selected this platform because its performance is comparable to state-of-the-art single board computers used in avionic systems [62]. The replicas communicated with two HP Z2 workstations with 3.2 GHz Intel Core i7-8700 processors — one representing a set of 3 redundant sensors, and the other a set of 3 redundant actuators. The computers all ran CentOS 7.9 with kernel 3.10.0-1160.53.1 and the PREEMPT_RT patch.

The replicas, sensors, and actuators communicated using real AFDX network cards manufactured by TTTech. We used NASA's TTX library [57] for interfacing to the cards. The cards each connected to 2–3 network planes (depending on the experiment), each consisting of a variable number of AFDX switches. Due to limited hardware availability (some experiments required 10+ switches), we also used a Dell Precision 7920 server with dual Intel Xeon 6242R processors to emulate switches in some experiments. The server was directly wired to the real AFDX switches, and was configured to execute the AFDX protocol and to mimic the delays of the real switches.

The replicas executed NASA's Core Flight System (cFS) [67], an open flight software framework used in real spacecraft. cFS tasks run in fixed time slots according to a cyclic executive. We synchronized the replicas using periodic interrupts from an external timing circuit. We used a 500 Hz interrupt rate, which matches real systems and past work [61].

We implemented two state-of-the-art protocols for comparison. The first, SM, is a traditional BFT SMR protocol based on Lamport's signed messages [54], which solves agreement in the theoretical minimum number of rounds. Like CrossTalk, SM requires only $2f + 1$ replicas to tolerate f faults. We used a simple signature scheme in SM based on modular inverse [54], which takes only around 0.01 ms to sign and verify a 200-byte message on our replicas. The second, Igor [61], is a recent BFT SMR protocol that, to our knowledge, achieves lower worst-case latency than any existing protocol. Igor has an optional "Filtering Stage", which is designed to reduce latency when message

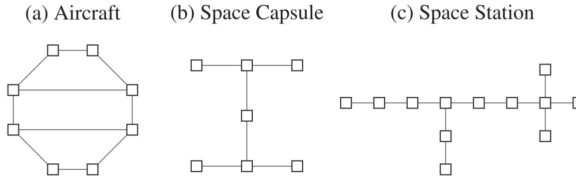

Fig. 9. Network topologies used in our evaluation. Each figure shows the switches in a single network plane.

fragmentation is slow (e.g., there is 1+ ms between sending fragments). However, NASA's TTX library and our AFDX cards allow fragments to be sent back-to-back with no delay. Thus, we report Igor's latency without the optional Filtering Stage, which minimizes Igor's latency on our testbed (it is 10–20% lower than with Filtering). We used SM as the agreement primitive in Igor. Unlike CrossTalk, Igor requires $3f + 1$ replicas to tolerate f faults.

Lastly, we compared CrossTalk to NoBFT, an SMR system with $2f + 1$ replicas, but no agreement protocol. Thus, NoBFT does not tolerate Byzantine faults, and represents the theoretical minimum latency that can be achieved.

6 EVALUATION

Our evaluation was designed to answer five key questions: (1) How does CrossTalk's latency compare to the state of the art? (2) How much does CrossTalk's low computation overhead improve schedulability? (3) What is the cost of CrossTalk's cross-links? (4) What is CrossTalk's communication overhead? and (5) How does CrossTalk perform in a NASA simulation of a real spaceflight mission?

6.1 Latency

Experimental setup. For this experiment, we considered three representative network topologies shown in Figure 9. The first is an aircraft network from a recent ONERA paper [29]. The second and third are a space capsule network and space station network from a recent NASA presentation [60].

For each topology, we considered both $f = 1$ and $f = 2$ faulty replicas and $g = 1$ and $g = 2$ faulty switches. These values were chosen to reflect the requirements of real systems in practice [31, 63]. For each combination of f and g, we executed three trials. In each trial, we randomly selected the positions of $2f + 1$ replicas, one redundant set of sensors, and one redundant set of actuators. As is typical in practice [48, 76], replicas were required to connect to different switches to reduce the chance of common-cause failures.

We developed tools based on Python's NetworkX [45] library to determine the rules for routing virtual channels between the nodes and switches, as well as to select the cross-link positions for CrossTalk. The tools start by considering a single-plane version of the network, and use a combination of depth-first and brute-force search to identify the shortest routes between all nodes that need to communicate, while meeting any protocol-specific constraints. For example, CrossTalk requires that all routes from a given sensor to the replicas traverse a common set of $g + 1$ switches in the same order (as shown in Figure 5). These common switches were then used as the cross-link positions for that sensor, and the final multi-plane routes were determined by replicating the original single-plane routes on all planes, then extending them across planes at the cross-link positions. For NoBFT, SM, and Igor, the final routes were determined by simply using the original single-plane routes on each plane.

In each trial, the worst-case execution time (WCET) of the execution on the replicas was 10 ms, and the size of the sensor and actuator data was 200 bytes. Both were chosen to reflect typical values in commercial aircraft [29, 55, 75].

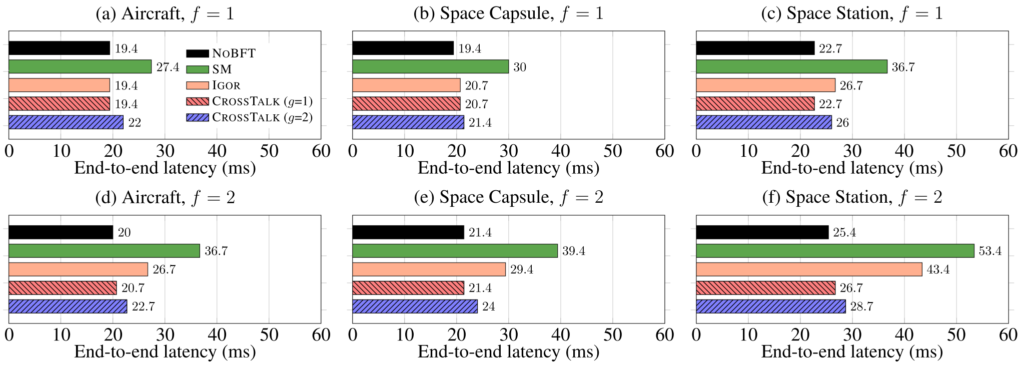

Fig. 10. Average worst-case latency of CrossTalk compared to state-of-the-art BFT protocols and NoBFT.

We instrumented the network so that each switch delayed each message approximately 1 ms when forwarding the message to a device on the same plane, which reflects typical delays in avionics networks due to network congestion [29, 75]. We configured the delay over cross-links to be 0.5 ms to reflect the fact that these links are significantly less congested (many systems do not currently use cross-links) [31].

We constructed the cFS task schedule by executing each segment of each protocol (e.g., a round of agreement) for 600 iterations and estimating each segment's WCET by adding 10% margin to the longest measurement. We then arranged the protocol segments in sequence, with each segment allocated enough time slots to accommodate the WCET. Lastly, we executed the resulting cFS schedule on our testbed, verified all deadlines were met, and recorded the end-to-end latencies of each protocol from sensors to actuators.

We note that the latencies of all the protocols are the same in both the presence and absence of faults. Thus, we did not explicitly inject faults when taking measurements. Moreover, the latencies for SM, Igor, and NoBFT are the same regardless of the number of planes, so we do not specify g in the results for those protocols.

Results. Our results are shown in Figure 10. In all cases, CrossTalk's latency was comparable to or better than the best existing protocol (Igor), meaning it can meet tighter deadlines. Importantly, CrossTalk achieves this latency *without* requiring computations on multiple cores (which, as we will see next, results in much higher schedulability).

Moreover, CrossTalk's latency savings increase as the number of faulty replicas, as well as the size of the network, increases. For example, in the space capsule, CrossTalk's latency is 1.2–1.4× lower than the best existing protocol when tolerating $f = 2$ faults. In the space station network, CrossTalk's savings increase to 1.5–1.6×. The reason is that, as f increases, the time needed for other BFT protocols to reach agreement increases due to the need for more communication rounds. Similarly, as the size of the network increases, the duration of each round increases.

We acknowledge that as the number of faulty switches (g) increases, the need for more cross-plane communication causes CrossTalk's improvement over Igor to decrease. However, unlike CrossTalk, Igor does not consider switch faults. Moreover, CrossTalk's latency is still 8.7% lower than Igor's on average when $g = 2$. Finally, we note that we are not aware of any system required to tolerate more than 2 switch faults in practice.

6.2 Schedulability

Unlike Igor, CrossTalk can be realized on a single processor core. In this experiment, we evaluated how much this design improves schedulability.

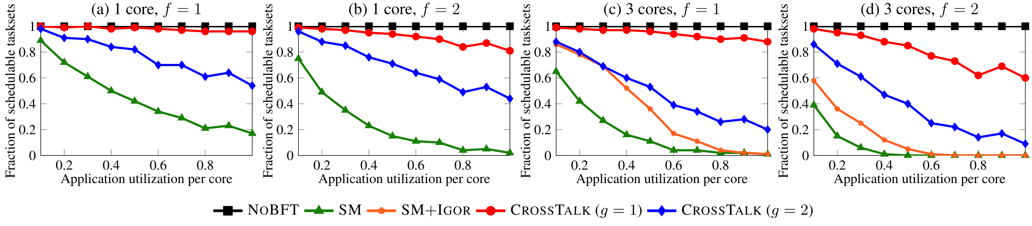

Fig. 11. Schedulability improvement with CrossTalk. Note that Igor cannot be used on single-core processors. Since task deadlines were all at least as large as NoBFT's latency (the minimum latency that can be achieved), all tasks were schedulable with NoBFT. Recall that NoBFT cannot tolerate Byzantine faults.

Experimental setup. We considered workloads consisting of independent constrained-deadline BFT tasks executed on either a single-core, or distributed over three cores. In each case, we varied the application utilization per core from 0.1 to 1, and randomly generated 1000 tasksets per utilization. The task WCETs and periods were randomly chosen from {5, 10, 15, 20} ms and {25, 50, 100, 200} ms respectively, which are common values in practice and match prior work [61].

For each task, we randomly selected a topology and set of replicas, sensors, and actuators from Section 6.1. We used the data from Section 6.1 to specify the WCTT of messages and WCET of each protocol segment (e.g., a round of agreement). Each task was randomly assigned an end-to-end deadline within which data was required to leave the sensors, be processed on the replicas, and arrive at the actuators. The deadlines were uniformly distributed between the worst-case latency for NoBFT (the best latency that can be achieved) and the task period.

For each utilization, we report the fraction of schedulable tasksets with each protocol. We scheduled tasks on the processor cores using the same heuristic as in Igor [61], where tasks with smaller periods were scheduled first.

As in past work, we did not use Igor for *every* BFT task when evaluating Igor. Otherwise, Igor would have very low schedulability due to requiring execution on all three cores. Instead, we used the strategy from the Igor paper [61] in which we first attempted to use SM for all tasks. If any tasks were not schedulable with SM, we attempted to schedule only those tasks with Igor. Thus, the system only incurs the overhead of Igor when needed to meet deadlines.

Results. Our results are shown in Figure 11. As shown, we were able to schedule significantly more tasksets with CrossTalk than with the best existing protocols. For example, in the single-core case, 2.37–3.12× more tasksets were schedulable with CrossTalk than with SM. In the multi-core case, 2.13–4.24× more tasksets were schedulable with CrossTalk than with SM+Igor. Importantly, CrossTalk also achieves very high schedulability at high utilizations. For example, with 3 cores in the $f = 1, g = 1$ case, we could schedule 88% of tasksets with CrossTalk at a core utilization of 1.0, while we could only schedule 1% of tasksets with SM+Igor.

6.3 Cost Trade-off of Cross-Links

In the previous sections, we used our network scheduling tools to select $g + 1$ cross-link positions for each sensor, such that the length of the routes from the sensor to the replicas was minimized. Thus, since there may be *many* sets of sensors in a system, all for different purposes and at different positions, the system may contain *more* than $g + 1$ total cross-link positions. In this section, we evaluate the cost of this approach (e.g., in mass), as well as determine how much reducing the number of cross-links positions — which lowers costs but also forces some sensors to use "worse" cross-link positions — still allows us to retain CrossTalk's latency benefits.

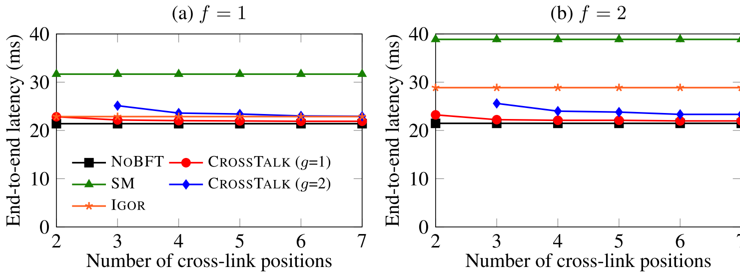

Fig. 12. Impact of the total number of cross-link positions (the number of switches with cross-links per plane) on CrossTalk's system-wide worst-case latency. We only show up to 7 cross-link positions, since more cross-link positions result in no additional latency savings for CrossTalk. Note that when $g = 2$, CrossTalk requires $\geq g + 1 = 3$ cross-link positions.

Experimental setup. We randomly generated 15 system configurations, each containing a topology from Section 6.1 and the positions of $2f + 1$ replicas, one set of redundant actuators, and 8 sets of redundant sensors (representing the many sensors in a real system). We varied the total number of cross-link positions allowed in the network from 2 to 7 (the number of switches in the smallest topology in Figure 9). For each BFT SMR protocol, system configuration, number of allowed cross-link positions, and combination of $f = \{1, 2\}$ and $g = \{1, 2\}$, we used our network tools (as described in Section 6.1) to find the exact cross-link positions and message routes that minimized the length of the longest route from the sensors to the actuators. Then, in each case, we calculated the protocol's worst-case latency using the switch delays and WCETs from 6.1. We report, for each number of cross-link positions, the average worst-case latency for each protocol across all sensors and configurations.

In addition, we calculated the mass of the replicas and cross-links needed for CrossTalk compared to SM and Igor for each combination of f, g, and the total number of cross-link positions in the system. We assumed each replica is 4.5 kg (based on a typical single board computer chassis [3]), each cross-link is 0.1 kg/m (a standard approximation in avionic systems [24]), and that each cross-link is 2 m long — which is more than the required network plane separation in safety-critical applications [1]. We then converted our mass results to monetary cost in a typical spaceflight application, assuming each replica costs $200K (the cost of a typical spaceflight computer [51]), each cross-link costs $150/m (the cost of space-qualified cables [19]), and that it costs $15K to launch 1 kg of mass to low Earth orbit (which is typical [16]).

Results. Figure 12 shows how the total number of cross-link positions in the system impacts CrossTalk's latency (and thus ability to meet deadlines). As shown, CrossTalk's latency is minimized with around 6 cross-link positions, with more cross-link positions yielding no additional benefit. This is because, in general, 6 cross-link positions is enough to allow each sensor's traffic to traverse the required $g + 1$ cross-link positions while still taking the shortest path to the replicas (as shown in Figure 6). Conversely, restricting the number of cross-link positions increases CrossTalk's latency — but only gradually. For example, with only 3 cross-link positions, CrossTalk's latency is still lower than Igor's in most cases. Moreover, we emphasize that when $f = 2$, CrossTalk's latency is always strictly smaller than Igor's for all numbers of cross-link positions.

Tables 1 and 2 show how CrossTalk's mass and cost compare to Igor when restricted to different numbers of cross-link positions. As shown, even with 7 cross-link positions (which is more than needed to minimize CrossTalk's latency), CrossTalk is *always* lighter and cheaper than

Table 1. Mass of the Replicas and Cross-links for Each Protocol when Configured to Tolerate Different Numbers of Faults

Faults f	g	SM	Igor	CrossTalk (# of cross-link positions) 2	3	4	5	6	7
1	1	13.5	18	13.9	14.1	14.3	14.5	14.7	14.9
1	2	13.5	18	N/A	15.3	15.9	16.5	17.1	17.7
2	1	22.5	31.5	22.9	23.1	23.3	23.5	23.7	23.9
2	2	22.5	31.5	N/A	24.3	24.9	25.5	26.1	26.7

Results are in kilograms.

Table 2. Cost of the Replicas and Cross-links for Each Protocol in a Typical Spaceflight Application

Faults f	g	SM	Igor	CrossTalk (# of cross-link positions) 2	3	4	5	6	7
1	1	8	10.7	8.1	8.1	8.2	8.2	8.2	8.3
1	2	8	10.7	N/A	8.3	8.4	8.5	8.6	8.7
2	1	13.4	18.7	13.4	13.5	13.5	13.5	13.6	13.6
2	2	13.4	18.7	N/A	13.7	13.8	13.9	14	14.1

Results are in hundreds of thousands of US dollars.

Igor. For example, when $f = 2$ and $g = 1$, CrossTalk is 7.6 kg lighter and >$500K cheaper. The reason is that CrossTalk requires f fewer replicas than Igor to tolerate f faults, and these replicas are much heavier and more costly than CrossTalk's cross-links. Note than since Igor is based on overlapping quorums of replicas, there is no way to reduce the number of replicas it requires.

We note that, while we assumed single and multi-core replicas have the same monetary cost, multi-core replicas may be more expensive in practice — in which case Igor's costs would be even higher (i.e., CrossTalk would have larger cost savings). Moreover, while we did not measure energy consumption, it is expected that CrossTalk's need for fewer replicas, as well as only single-core replicas, would result in significant power savings over Igor.

We also note that while CrossTalk's mass and cost were slightly higher than SM, SM has much higher latency (up to 2× higher, see Figures 10, 12) and worse schedulability (Figure 11).

Lastly, while not the focus of this paper, we note that CrossTalk could also be used in systems that *do not* currently employ redundant network planes. For example, an industrial control system that currently has a single non-redundant TSN network could gain the ability to tolerate both omissive switch faults *and* Byzantine node faults by adding a second network plane. The average industrial TSN switch costs around $1.7K and weighs 1.6 kg [21]. Cat 6 Ethernet cable costs around $0.7/m and weighs 0.06 kg/m [18]. Thus, in a system with a ring of five switches, designers could add a second plane at a cost of only ∼$8.5K and ∼10 kg (assuming 5 m links). In a system with a ring of ten switches, a second plane would cost only ∼$17K and ∼19 kg. Assuming that each switch in the original plane connects to ten nodes and that each node costs $2K and weighs 3.6 kg (reflecting a typical industrial PC [14]), then in both examples (rings of five and ten switches), the second plane only costs ∼7.3% and weighs ∼4.8% of the entire system. We think these costs are reasonable given the improved fault tolerance compared to non-BFT systems and the lower latency (Figure 10) and improved schedulability (Figure 11) compared to other BFT approaches.

6.4 Communication Overhead

Experimental setup. One common concern regarding BFT SMR protocols is that they produce large amounts of network traffic. Since the switches in CrossTalk create copies of messages in the network, one cannot evaluate CrossTalk's communication overhead by measuring the amount of traffic *transmitted* by a node. Instead, we placed a network tap between one replica and one network plane, and used the tap to capture all traffic received by the replica. We captured traffic for 60 executions of each protocol in a representative space station network (repeating the experiment from Section 6.1). We report the amount of traffic received by the replica per sensor.

We note that, as described in Section 4.1, the virtual channels in CrossTalk are balanced so that replicas receive the same amount of traffic from each plane. Similarly, the traffic in NoBFT, SM, and Igor is identical across all planes. Thus, the choice of plane to tap does not matter. Moreover, since we log traffic as it enters the replica and not elsewhere in the network, the choice of network topology and positions of the nodes (e.g., replicas) does not impact the results.

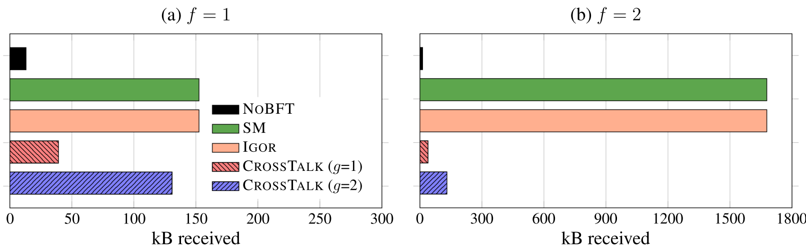

Fig. 13. Number of kB received by each replica per plane per sensor in CrossTalk compared to other protocols.

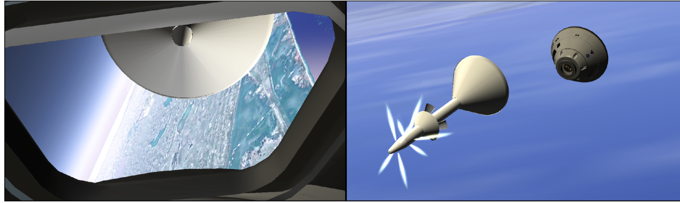

Fig. 14. View from inside and outside the simulated capsule after being jettisoned from the rocket.

Results. Our results are shown in Figure 13. In general, CrossTalk's communication overhead is lower than other state-of-the-art protocols. Additionally, its relative efficiency increases with f. For example, when $f = 2$, CrossTalk's communication overhead is 12.83× lower than Igor and SM. The reason is that CrossTalk does not require any rounds of communication between replicas, while other protocols need at least $f + 1$ inter-replica rounds (with increasing message sizes). We acknowledge that using Igor's optional Filtering Stage would decrease its overhead when $f = 2$, but at the expense of increased latency (as described in Section 5) due to needing an additional round. Moreover, we note that while Igor requires 7 replicas to tolerate $f = 2$ faults, there are only 5 replicas in our testbed. Running Igor with 7 replicas would result in higher communication overheads for Igor and more relative savings for CrossTalk.

6.5 Case Study: Spaceflight Abort Test

Lastly, we wanted to determine how CrossTalk performs in a concrete system. For this purpose, we used simulated flight software from a NASA abort test, which we obtained from the authors of Igor [61]. The simulation, shown in Figure 14, is based on a real mission conducted in 2019. During the mission, a rocket carried an empty crew capsule to an altitude of 9.5 km, at which an abort was intentionally triggered and the capsule was safely ejected from the rocket. We executed the provided flight software on replicas in our cluster. The flight software communicated through our AFDX switches with a provided NASA simulation, which modeled the vehicle dynamics, sensors, and actuators. The network was instrumented to induce delays comparable to AFDX avionics networks in practice [29, 75], as in Section 6.1.

The main flight software control loop executed at 40 Hz — in each execution reading data from the sensors, performing computations, and sending results to the actuators. The end-to-end latency requirement from the sensors to the actuators was 25 ms. If this deadline was not met, the rocket deviated from its correct flight path and the abort failed.

We scheduled the flight software using the same approach as in Section 6.1 and executed both CrossTalk and Igor on our testbed. Both protocols met all deadlines when $f = 1$, resulting in correct behavior of the rocket and capsule. This includes CrossTalk in the $g = 2$ (two faulty

switches) case, which executed with approximately 4 ms of margin. However, while IGOR failed to meet deadlines when $f = 2$ by several milliseconds (resulting in the capsule ejecting at an incorrect orientation that compromised safety), CROSSTALK continued to meet deadlines when $f = 2$, even with $g = 2$ faulty switches. Thus, CROSSTALK's ability to reduce latency compared to IGOR directly resulted in improved fault resilience — while both protocols could be used, CROSSTALK was able to tolerate an additional faulty replica. Moreover, CROSSTALK did so while using nearly 3× less total CPU time.

7 RELATED WORK

Making agreement fast. Several protocols allow replicas to stop an agreement protocol early if replicas initially have agreement or no faults occur [22, 34, 79]. However, these protocols provide no latency savings under faults. Others use speculation, allowing replicas to execute on inputs optimistically before agreement is reached [50, 66, 78]. However, if faults occur, the system is rolled back and the execution is repeated, resulting in high worst-case latencies. In contrast, CROSSTALK achieves low latency in both the presence and absence of faults. IGOR [61] has similar goals to CROSSTALK. However, unlike IGOR, CROSSTALK can run on a single processor core (resulting in better schedulability, Section 6.2) and only needs $2f + 1$ replicas to tolerate f faults.

Leveraging the network topology. Several protocols exploit network redundancy in their designs. One class, fault-tolerant routing protocols [39, 68], are concerned with creating a reliable communication channel between a sender and receiver. However, importantly, these protocols cannot tolerate Byzantine end nodes. Another class of protocols aims to solve consensus in directed graphs [80]. However, they are aimed at maximizing resilience in *arbitrary* network topologies. As a result, CROSSTALK is significantly more efficient when used in realistic embedded networks.

Other solutions leverage unique aspects of embedded networks to increase performance. One patent [26] inserts skew between redundant time-triggered messages to ensure agreement among receivers. Other protocols vote on message copies received on redundant planes to mask transmission faults [58, 63]. Importantly, unlike CROSSTALK, both approaches can only tolerate a single faulty device (switch or end node) at a time.

Agreement in the network. Several protocols modify the switches or use software-defined networks (SDNs) to execute an agreement protocol in the network or otherwise guarantee message ordering [32, 33, 56]. However, these protocols are all limited to tolerating crash faults of both the switches and end nodes. In contrast, CROSSTALK can tolerate omissive switches and Byzantine faulty end nodes, and can be used without requiring SDNs or switch modifications.

Impossibility results. Fischer and Lynch proved $f + 1$ communication rounds are needed for agreement in synchronous systems with point-to-point channels [41]. Despite its "single round" design, CROSSTALK does not violate this result. Rather, the required rounds ($g + 1$ for CROSSTALK) take place in the network as messages are forwarded between planes. The FLP result [42] showed agreement is impossible in asynchronous systems subject to crash faults. We use similar intuition in our impossibility proof. However, the proof is distinct. Specifically, while synchronous systems can tolerate timing faults with $f + 1$ rounds of exchange between replicas, as well as Byzantine faults in the *value* domain without requiring any rounds between replicas (given sufficient rounds occur in the network), we prove it is not possible for synchronous systems to tolerate timing faults without rounds between replicas (regardless of rounds in the network).

8 CONCLUSION

BFT SMR is essential for tolerating faults in critical embedded systems. However, existing BFT SMR protocols force designers to choose between high latencies and substantial computation overheads

due to redundant computations. We presented CrossTalk, a new BFT SMR protocol that leverages redundancy that already exists in embedded networks to minimize latency without requiring extra computation. Our evaluation showed that CrossTalk improves system schedulability by 2.13–4.24×, and can increase the resilience and control performance of real systems. In the future, we believe more opportunities will exist for protocols like CrossTalk to exploit emerging networking trends to increase the performance of real-time embedded systems.

REFERENCES

[1] 2008. *Survivability of Systems.* Technical Report AC 25.795-7. Federal Aviation Administration.
[2] 2009. ARINC 664 P7: Aircraft data network part 7 avionics full-duplex switched ethernet network. ARINC.
[3] 2010. Aitech's New Customizable 3U CPCI Enclosure Combines Flexible Electronic Configurations with Rugged, Reliable Operation. https://picmg.mil-embedded.com/news/aitechs-configurations-rugged-reliable-operation/
[4] 2015. TTEthernet Product Overview. http://konaka.com.tr/pdf/AS6802_TTEthernet.pdf
[5] 2015. TTTech to Provide ARINC 664 p7 Products for Mission System on UK AW101 Merlin Mk4/4a Helicopters. https://www.tttech.com/press/tttech-to-provide-arinc-664-p7-products-for-mission-system-on-uk-aw101-merlin-mk4-4a-helicopters/
[6] 2016. IEEE 802.1Qbv-2015: IEEE standard for local and metropolitan area networks – bridges and bridged networks – amendment 25: Enhancements for scheduled traffic. IEEE.
[7] 2016. SAE AS6802: Time-triggered ethernet. SAE International.
[8] 2017. *IEEE 1588 Precise Time Protocol: The New Standard in Time Synchronization.* Technical Report. Microsemi.
[9] 2017. IEEE 802.1CB-2017: IEEE standard for local and metropolitan area networks – frame replication and elimination for reliability. Institute of Electrical and Electronics Engineers.
[10] 2017. IEEE 802.3-2018: IEEE standard for ethernet. Institute of Electrical and Electronics Engineers.
[11] 2019. Safe4RAIL-2 Newsletter. https://safe4rail.eu/downloads/Safe4RAIL-2-Newsletter-Issue-1-April-2019.pdf
[12] 2022. Orion reference guide. https://www.nasa.gov/sites/default/files/atoms/files/orion_reference_guide_090622.pdf. NASA Johnson Space Center.
[13] 2022. TSN Is Set to Become a Must for Industry. https://iebmedia.com/technology/tsn/tsn-is-set-to-become-a-must-for-industry/
[14] 2023. ADLINK Technology MVP-5001. https://www.mouser.com/ProductDetail/ADLINK-Technology/MVP-5001?qs=pCidNA4Lr1nu0o3cwPnhIw%3D%3D
[15] 2023. Auto/TSN for In-Vehicle Networking. https://www.missinglinkelectronics.com/www/index.php/menu-solutions/menu-autotsn
[16] 2023. Cost of Space Launches to LEO. https://ourworldindata.org/grapher/cost-space-launches-low-earth-orbit
[17] 2023. IEEE P802.1DP: TSN for aerospace onboard ethernet communications. IEEE / SAE International (joint standard).
[18] 2023. Monoprice Cat6 1000ft Blue CMR Bulk Cable Shielded. https://www.monoprice.com/product?p_id=18608
[19] 2023. SpaceWire Cable GNSSW10028MS. https://www.wiremasters.com/suppliers/W-L-Gore-And-Associates/catalog/products/wire-and-cable/w-l-gore-and-associates-inc-/
[20] 2023. Time Sensitive Networking (TSN). https://us.profinet.com/digital/tsn/
[21] 2023. TSN-6325-8T4S4X Industrial L3 8-Port Switch. https://planetechusa.com/product/tsn-6325-8t4s4x-industrial-l3-8-port-10-100-1000t-4-port-1g-2-5g-sfp-4-port-10gbase-x-sfp-managed-tsn-ethernet-switch/
[22] Ittai Abraham et al. 2021. Good-case latency of byzantine broadcast: A complete categorization. In *Proc. PODC.*
[23] Vedant Agarwal. 2016. Photo Essay: Inside the airbus A380 test aircraft F-WWDD MSN4. https://www.bangaloreaviation.com/2016/10/photo-essay-inside-airbus-a380-test-aircraft-f-wwdd-msn4.html. (2016).
[24] Bjoern Annighoefer. 2014. Network topology optimization for distributed integrated modular avionics. In *Proc. DASC.*
[25] Günther Bauer et al. 2001. Assumption coverage under different failure modes in the time-triggered architecture. In *Proc. ETFA.*
[26] Günther Bauer et al. 2020. Method and Computer System for Establishing an Interactive Consistency Property.
[27] Woodrow Bellamy. 2015. TTEthernet avionics backbone a technology breakthrough for S-97 raider. https://www.aviationtoday.com/2015/07/20/ttethernet-avionics-backbone-a-technology-breakthrough-for-s-97-raider/. *Aviation Today* (2015).
[28] Frédéric Boulanger et al. 2018. A time synchronization protocol for A664 P7. In *Proc. DASC.*
[29] Marc Boyer et al. 2016. Performance impact of the interactions between time-triggered and rate-constrained transmissions in TTEthernet. In *Proc. ERTS.*
[30] Ricky Butler. 2008. *A Primer on Architectural Level Fault Tolerance.* Technical Report NASA/TM-2008-215108.
[31] Henning Butz. 2007. The airbus approach to open integrated modular avionics (IMA): Technology, methods, processes, and future roadmap. In *Proc. AST.*

[32] Huynh Tu Dang et al. 2015. NetPaxos: Consensus at network speed. In *Proc. SOSR.*
[33] Huynh Tu Dang et al. 2020. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.* 28, 4 (2020).
[34] Dan. Dobre and Neeraj Suri. 2006. One-step consensus with zero-degradation. In *Proc. DSN.*
[35] Danny Dolev. 1982. The byzantine generals strike again. *Journal of Algorithms* 3, 1 (1982).
[36] Kevin Driscoll et al. 2003. Byzantine fault tolerance, from theory to reality. In *Proc. SAFECOMP.*
[37] Kevin Driscoll et al. 2013. *Application Agreement and Integration Services.* Technical Report NASA/CR–2013-217963.
[38] Cynthia Dwork et al. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988).
[39] Abdol-Hossein Esfahanian and Seifollah Hakimi. 1985. Fault-tolerant routing in DeBruijn communication networks. *IEEE Trans. Compu.* C-34, 9 (1985).
[40] Christian Fidi et al. 2018. Radiation-tolerant system-on-chip (SOC) with deterministic ethernet switching for scalable modular launcher avionics. In *Proc. ERTS.*
[41] Michael Fischer and Nancy Lynch. 1982. A lower bound for the time to assure interactive consistency. *Inf.Process.Lett.* 14, 4 (1982).
[42] Michael Fischer et al. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (1985).
[43] David Gwaltney and J.M. Briscoe. 2006. *Comparison of Communication Architectures for Spacecraft Modular Avionics Systems.* Technical Report NASA/TM-2006-214431.
[44] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems.* Technical Report.
[45] Aric Hagberg et al. 2008. Exploring network structure, dynamics, and function using networkx. In *Proc. SciPy.*
[46] Brendan Hall et al. 2005. Ringing out fault tolerance. A new ring network for superior low-cost dependability. In *Proc. DSN.*
[47] John Hanaway and Robert Moorehead. 1989. *Space Shuttle Avionics System.* Technical Report NASA SP-504.
[48] John Knight et al. 2006. Dependability in avionics systems. In *Digital Avionics: A Computing Perspective.*
[49] Hermann Kopetz. 2011. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer US.
[50] Ramakrishna Kotla et al. 2007. Zyzzyva: Speculative byzantine fault tolerance. In *Proc. SOSP.*
[51] Jacek Krywko. 2019. Space-grade CPUs: How Do You Send More Computing Power Into Space? https://arstechnica.com/science/2019/11/space-grade-cpus-how-do-you-send-more-computing-power-into-space/
[52] Jaynarayan Lala. 1986. A byzantine resilient fault tolerant computer for nuclear power plant applications. In *Proc. FTCS.*
[53] Jaynarayan Lala and Richard Harper. 1994. Architectural principles for safety-critical real-time applications. *Proc. IEEE* 82, 1 (1994).
[54] Leslie Lamport et al. 1982. The byzantine generals problem. *TOPLAS* 4, 3 (1982).
[55] Michaël Lauer et al. 2014. End-to-end latency and temporal consistency analysis in networked real-time systems. *Int. J. Crit. Comput.-Based Syst.* 5, 3/4 (2014).
[56] Jialin Li et al. 2016. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proc. OSDI.*
[57] Andrew Loveless. 2015. On TTEthernet for integrated fault-tolerant spacecraft networks. In *Proc. AIAA.*
[58] Andrew Loveless. 2016. Notional 1FT Voting Architecture with Time-Triggered Ethernet. https://ntrs.nasa.gov/citations/20170001652
[59] Andrew Loveless. 2020. On Time-Triggered Ethernet in NASA's Lunar Gateway. https://ntrs.nasa.gov/citations/20205005104
[60] Andrew Loveless. 2022. Impact of Switch Plane Redundancy on Network Availability. https://ntrs.nasa.gov/citations/20220003523
[61] Andrew Loveless et al. 2021. IGOR: Accelerating byzantine fault tolerance for real-time systems with eager execution. In *Proc. RTAS.*
[62] Tyler Lovelly and Alan George. 2017. Comparative analysis of present and future space-grade processors with device metrics. *J. Aerosp. Inf. Syst.* 14, 3 (2017).
[63] Brendan Luksik et al. 2021. Gatekeeper: A reliable reconfiguration protocol for real-time ethernet systems. In *Proc. DASC.*
[64] Jennifer Lundelius and Nancy Lynch. 1984. An upper and lower bound for clock synchronization. *Inf. Control.* 62, 2 (1984).
[65] Christopher Marchant. 2009. Ares I avionics introduction. In *Proc. NASA/ARMY Software and Systems Forum.*
[66] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast byzantine consensus. *TDSC* 3, 3 (2006).
[67] David McComas. 2012. NASA/GSFC's Flight Software Core Flight System. https://ntrs.nasa.gov/citations/20130013412
[68] F.J. Meyer. 1988. Flip-trees: Fault-tolerant graphs with wide containers. *IEEE Trans. Compu.* 37, 4 (1988).
[69] Klaus Neuenhueskes. 2022. The role of ethernet in zonal architectures and automotive telematics. https://www.redeweb.com/en/Articles/el-papel-de-ethernet-en-las-arquitecturas-zonales-y-la-telematica-del-automovil/. (2022).
[70] Roman Obermaisser. 2011. *Time-Triggered Communication.*

[71] Steve Parkes et al. 2016. SpaceFibre networks: SpaceFibre, long paper. In *Proc. International SpaceWire Conference*.
[72] Markus Plankensteiner. 2011. TTTech Company Overview. https://www.slideshare.net/TTTech/tttech-2011companyoverview
[73] Edo Roth and Andreas Haeberlen. 2021. Do not overpay for fault tolerance!. In *Proc. RTAS*.
[74] John Rushby. 2001. *Formal Verification of Transmission Window Timing for the Time-Triggered Architecture*. Technical Report. SRI International.
[75] Jean-Luc Scharbarg and Christian Fraboul. 2014. Dimensioning of civilian avionics networks. In *Industrial Communication Technology Handbook*.
[76] Daniel Siewiorek and Priya Narasimhan. 2005. Fault-Tolerant Architectures for Space and Avionics Applications.
[77] J. T. Sims. 1997. Redundancy management software services for seawolf ship control system. In *Proc. FTCS*.
[78] Atul Singh et al. 2009. Zeno: Eventually consistent byzantine-fault tolerance. In *Proc. NSDI*.
[79] Yee Jiun Song and Robbert Renesse. 2008. Bosco: One-step byzantine asynchronous consensus. In *Proc. DISC*.
[80] Lewis Tseng and Nitin Vaidya. 2015. Fault-tolerant consensus in directed graphs. In *Proc. PODC*.
[81] Anis Youssef et al. 2006. Communication integrity in networks for critical control systems. In *Proc. EDCC*.
[82] Amira Zammali et al. 2015. Communication integrity for future helicopter flight control systems. In *Proc. DASC*.

Received 23 March 2023; revised 2 June 2023; accepted 13 July 2023