Gatekeeper: A Reliable Reconfiguration Protocol for Real-Time Ethernet Systems

Brendan Luksik
NSF Center for Space, High-performance, and Resilient Computing (SHREC)
University of Pittsburgh
brendan.luksik@nsf-shrec.org

Andrew Loveless
NASA Johnson Space Center
University of Michigan
andrew.loveless@nasa.gov

Alan D. George
NSF Center for Space, High-performance, and Resilient Computing (SHREC)
University of Pittsburgh
alan.george@nsf-shrec.org

***Abstract*—Real-time Ethernet systems are becoming increasingly popular for avionics and embedded applications. By regulating network traffic according to predefined configurations, these protocols enable highly deterministic communication, while still conforming to the Ethernet standard. However, the strengths of a statically configured system become weaknesses when the system requirements are changed. In the worst case, the entire network may need to be reloaded with new configurations, resulting in significant downtime. As a result, there is significant growing interest in reconfiguring real-time Ethernet networks *online*, without restarting the network. Several recent works focus on minimizing frame loss and configuration conflicts during online reconfiguration. Unfortunately, in doing so, they also sacrifice the system's ability to tolerate faulty components.**

In this paper, we describe GATEKEEPER, the first reconfiguration protocol for real-time Ethernet systems that minimizes downtime while still tolerating faults. GATEKEEPER consists of two main sub-protocols: 1) a reliable distribution protocol that ensures consistent configurations are deployed on all non-faulty devices (i.e., switches and network cards), and 2) a dependable test-and-migrate reconfiguration protocol that allows the system to gain confidence that the configurations are correct as they are rolled out to an increasing number of devices. Our evaluation shows that GATEKEEPER has modest overheads, and causes up to $2.5\times$ shorter network disruptions than other solutions.

I. INTRODUCTION

Avionic, automotive, and industrial domains are increasingly adopting real-time Ethernet variants as their networks of choice. Some examples include Time-Triggered Ethernet (TTE) [1], Avionics Full-Duplex Switched Ethernet (AFDX) [2], and IEEE 802.1 Abv [3]. These protocols provide many advantages over standard Ethernet, such as support for different traffic criticalities, deterministic timing, ordering guarantees, and built-in redundancy schemes to avoid the need for message re-transmission. The behavior of each protocol is determined by a static configuration, which is developed offline and loaded onto the network. This configuration is implemented as a matching set of tables, each intended for use by one device in the system. The tables govern the timing of, and paths taken by, frames sent over the network.

Real-time Ethernet protocols are generally used in static applications with fixed requirements. As a result, the network configurations in these systems do not typically have to change once the system is deployed. For example, the cost of recertification often deters designers from making configuration changes in airplane avionics networks, unless a problem is identified with the previous configuration [4]. Moreover, even if network configuration updates *are required*, they can be made during well-defined periods of downtime, such as between flights, when the system is in a safe state.

In contrast, emerging systems, such as spacecraft for deep space exploration [5] and Industry 4.0 platforms [6], are envisioned to operate continuously *without downtime* and are required to evolve over time in response to changing mission requirements. One way to meet the needs of these systems is to start with *broad* configurations that can accommodate many different future traffic flows. However, this approach requires significant overprovisioning, and thus wasted network resources, for systems with long service lifetimes. Moreover, it is impossible to predict every future requirements change.

A more desirable approach for these emerging systems is to change the network configuration as needed when the requirements evolve. Unfortunately, online network reconfiguration has significant challenges. Most obviously, it must be done in a way that maximizes network availability for the end devices. Also, it must be done in a way that controls the interaction between devices with different configurations. Otherwise communication between devices with subtly incompatible configurations can result in incorrect system behavior. Finally, it must be done in a way that maintains the network's resilience to faulty components. Often times, fault tolerance guarantees that a network makes for a *fully configured* network no longer hold for a *partially* configured one.

There has been much work on minimizing reconfiguration disruptions in real-time Ethernet networks [7], [8]. In general, these techniques focus on maximizing network availability and maintaining consistency between configuration changes. However, they take for granted that the network hardware operates reliably during the reconfiguration process. In critical applications like spaceflight [5], [9], this is an unacceptable assumption. Instead, care must be taken to ensure the system ends up in a correct state, even if some faulty devices attempt to disrupt the reconfiguration protocol.

In this paper, we introduce GATEKEEPER, a new protocol for the reliable online reconfiguration of fault-tolerant, real-time Ethernet networks. GATEKEEPER combines two sub-protocols. The first, a reliable distribution protocol, uses

Byzantine consensus to deploy new configuration tables to the network devices. Using Byzantine consensus prevents faulty devices from blocking correct devices from accepting configuration tables, as well as ensures that different non-faulty devices cannot accept conflicting configurations. The second, a group-based test-then-migrate protocol, provides a mechanism for a select group of network devices to gain confidence that the new network configuration is correct before it is rolled out to the rest of the network. Moreover, it provides multiple opportunities to catch common configuration errors while they are still easily recoverable.

To evaluate GATEKEEPER, we implemented it in a real TTE testbed. Our results show that GATEKEEPER imposes modest communication and execution time overheads compared to non-fault-tolerant solutions. Importantly, however, GATEKEEPER significantly decreases the amount of time that the network may become unavailable. In the worst case, GATEKEEPER reconfigured end devices and allowed them to communicate successfully in only 9072 ms, which is only 2 ms slower than is optimal for our hardware.

In summary, we make the following contributions.

- GATEKEEPER: a novel online reconfiguration protocol for real-time Ethernet networks that can reliably and consistently deploy configurations in both the presence and absence of faults (§IV).
- A prototype of GATEKEEPER for TTE networks (§V).
- An experimental evaluation of GATEKEEPER combining both real benchmarks and offline analysis (§VI).

II. BACKGROUND

In this section we describe real-time Ethernet protocols and the processes they use for reconfiguration.

A. Real-Time Ethernet

Switched Ethernet is becoming increasingly common in embedded applications due to its many favorable characteristics. For example, Ethernet is compatible with a wide array of commercial-off-the-shelf devices and boasts a large and active community of developers. Additionally, the protocol is scalable, and new components can be easily introduced to expand existing setups. As a result, Ethernet networks are well suited to support modern industrial, avionic, and automotive embedded applications [10].

However, switched Ethernet has an important downside. Frames in a switch cannot simultaneously access the same egress port, and thus must be serialized by the device. This results in frames being unpredictably delayed on busy devices, or even dropped if frame buffers are exhausted. While these communication bottlenecks are acceptable in consumer-grade applications, they are intolerable in real-time embedded environments, where the usefulness of data expires after set deadlines. For real-time distributed systems, communication characteristics need to be predictable in order to ensure that requirements are met. This means enforcing bounded latencies and preventing the need for frame retransmission.

To achieve more predictable timing, a variety of real-time Ethernet variants have been introduced, all of which coordinate access to network resources [2], [11], [12]. Each protocol uses either *a priori* calculations or set rules to reserve network hardware for privileged (critical) traffic. For example, the AFDX protocol defines a minimum gap between frames constituting a data flow and reserves enough buffer space within each switch to handle all traffic flows at runtime [2]. TTE, meanwhile, uses time-division multiplexing to define transmission windows across the network in which only pre-selected frames can be sent. Other protocols, like PROFINET IRT, segment the wire's bandwidth into best-effort and real-time segments and oscillate usage on a common clock between the two modes [13]. These mechanisms allow critical traffic to enjoy predictable timing characteristics — often while also co-existing with traditional Ethernet frames.

The rules governing the behavior of the network are stored in a system wide configuration, which typically does not change during normal system operation. This configuration typically takes the form of a set of tables, one corresponding to each device in the network. Matching tables are necessary to ensure that timing constraints are consistent and the appropriate amount of buffer space is reserved — allowing frames to have guaranteed transmission characteristics. Such guarantees are only possible if every piece of hardware in the transmission path applies a matched set of constraints about when, as well as down which paths, data is forwarded.

B. Multi-Plane Ethernet Architectures

Ethernet networks contain two types of devices: end systems and switches. End systems are computation devices with a physical interface to the network. Typically, end systems take the form of a host processor and a tightly-coupled network interface card (NIC) acting together to generate or receive data. Switches forward frames between the end systems.

To improve fault tolerance, real-time Ethernet systems can be configured in a multi-plane architecture, as shown in Figure 1. In this configuration, each switch and connection is replicated to create independent channels for data transmission, called *planes*. Real-time Ethernet end systems typically replicate outgoing frames and simultaneously transmit them onto each of these planes, and the copies travel independently to the destination device [14]. Each end system receiving data uses a redundancy management policy for handling the copies of each frame. This may be voting on received frames, passing the first valid copy, etc [15].

The multi-plane architecture for real-time Ethernet systems is commonly found in safety-critical applications like commercial aviation and spaceflight. For example, the Airbus A380 and 400M aircraft both use redundant AFDX networks for command and control [16]. Triplicated TTE networks are used in the European Space Agency's Ariane 6 launcher [17], NASA's Orion space exploration vehicle [18], as well as in NASA's upcoming Lunar Gateway space station [5].

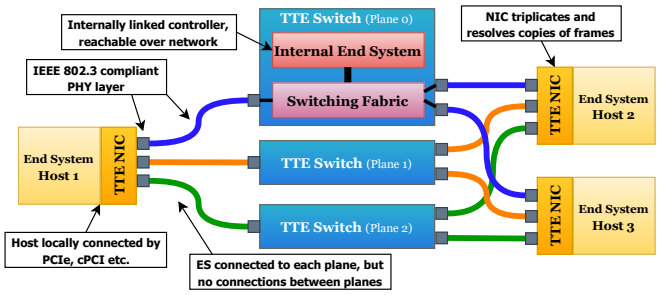

Fig. 1. An example of a 3-plane Time-Triggered Ethernet system

C. Network Reconfiguration

As the requirements for the system change, the network configuration must be updated to reflect the new requirements. For example, network configuration updates are typically necessary to change the sizes of messages, the route messages take through the network, and the timing at which messages are sent and expected to be received. In practice, changing the network configuration requires changing the table loaded onto each device impacted by the change.

To change a device's configuration table, a new table is pushed from a designated source node (e.g., a file server) to the target device and loaded into active memory. Tables destined for end systems are usually given to the co-located host processor and then loaded from the host onto the network controller hardware. Switches can accept configurations over the network through a programming interface. In most cases, large configuration changes require bringing the whole system offline and loading the devices one at a time.

In contrast, *online* reconfiguration, in which device tables are updated while the system is still operating, can lead to unexpected and incorrect behavior. For example, critical frames need a dedicated path while in flight, but reconfiguration can block available routes. Also, unless the whole network shifts configurations simultaneously, two partial configurations will co-exist. This means that a frame may cross between inconsistent configurations, leading the frame to be dropped. For example, while a device with one configuration may expect a message to be sent at a specific time, a device with a different configuration may send it at a different time. To prevent these types of problems, the order in which devices transition to new configurations must be carefully structured. While there is considerable work on minimizing disruptions during network reconfiguration, we are not aware of any that provides correctness guarantees in the presence of faulty devices.

D. The Case for Reliable Reconfiguration Techniques

The importance of fault-tolerant reconfiguration is clearly demonstrated by NASA's Lunar Gateway, an upcoming space station designed to enabled crewed missions to the moon. The first two modules of the station are planned to be launched in 2024, but new modules are expected to be added to the station over time. Gateway is also expected to host a variety of visiting vehicles, as well as house an ever-changing number of experimental payloads, much like its predecessor the International Space Station. All of these will require changes to the network configuration.

Reliable reconfiguration of Gateway's TTE network is critical to the station's overall mission. The avionics system is explicitly required to tolerate one faulty switch or end system in all phases of flight [5], including during this reconfiguration process. This requirement ensures that a faulty network component cannot cause network reconfiguration to result in inconsistent configurations, potentially causing critical messages to be dropped and threatening the safety of the crew. It also ensures that if no spares are available for a potentially faulty component, expansion and docking operations can still continue as planned.

III. Models

In this section, we describe the system and failure models used in GATEKEEPER. Our models are consistent with fault-tolerant real-time Ethernet systems used in practice [11], [14].

A. System Model

We consider a system of S switches and E end systems connected in a switched topology. Switches are arranged into m identical planes. Each end system consists of a host processor and a network interface card (NIC), which are connected to each other over a local data bus. Each NIC has m physical interfaces, one connected to each plane. Data is simultaneously transmitted from each end system over all m planes according to the rules of the protocol.

We assume the system is synchronous, meaning all computation and network operations, including the delivery of messages, is completed within an *a priori* known bounded time. We also assume all devices are synchronized within a bounded error. In practice, this timing synchronization is generally provided and maintained by the network [11].

In addition, we assume all end systems are assigned a predetermined role. There are three possible roles: bystander, witness, and initiator. A bystander is an end system that receives a new table of the deployed configuration but does not assist in its deployment. A witness is an end system that coordinates with other witnesses to control the reconfiguration process. Witnesses use Byzantine consensus [19] for coordination, and as such, we require at least $2f + 1$ witnesses. An initiator is an end system that is initially given the configuration to distribute to the rest of the network. There must be at least $f + 1$ initiators, which may be a subset of the witnesses, or a separate group.

Finally, we consider the placement of the end systems. GATEKEEPER requires end systems to communicate directly with switches. Since each switch exists in only one plane, this communication can not use the typical m redundant paths. Instead, we require that witnesses and initiators are placed such that, even if some switches are faulty, they cannot block the communication between these end systems and other non-faulty switches. The details are discussed in § IV-A.

B. Failure Model

GATEKEEPER is designed to tolerate Byzantine faults in end systems and asymmetric omissive faults in switches. This corresponds to the standard failure modes documented in the SAE AS6802 standard [11] for TTE ntworks. The total number of faults tolerated by GATEKEEPER is two less than the number of planes, or $m - 2$. The faults can be spread among an arbitrary group of switches and end systems.

By Byzantine, we mean that end systems are capable of incorrectly altering or omitting any data they touch. This can occur in an asymmetric way which manifests differently to different devices around the system. A broad failure mode is necessary because host devices may be any commodity processor, and guaranteeing these devices are restricted to narrower failure modes may not be feasible [18]. Further, host devices run foreign applications that can produce Byzantine behavior under faults, even if the network hardware itself is extremely reliable [20].

By asymmetric omissive, we mean that switches may drop messages, but will never undetectably alter or generate their own messages. Again, these omissions may happen in an inconsistent way — e.g., a message may be dropped for only some receivers. This downgrade from the broader Byzantine failure mode is realistic for our systems of interest. Since switches can be smaller and simpler than end systems, it is often possible to more thoroughly characterize their possible failure modes and reason about their probabilities. Further, coding techniques, like cyclic redundancy checks, can help minimize the probability of switch faults resulting in undetectable errors. In cases where higher integrity is needed, designers often employ self-checking switches with two independent switch processors [18]. For Byzantine switch faults to manifest, both processors would need to fail identically and simultaneously [21], which is often considered unlikely enough that it can be ignored [18].

IV. DESIGN

In this section we describe the design of GATEKEEPER. Overall, GATEKEEPER has three major goals.

1) **Consistency.** Non-faulty devices will never be configured with inconsistent or conflicting tables.
2) **Minimize downtime.** Network reconfiguration occurs without disrupting existing traffic flows.
3) **Fail-safe.** The configuration is either fully deployed, or deployment is prevented if errors occur.

GATEKEEPER proceeds in two phases. First, the Distribution Phase ensures that consistent configuration tables are delivered to all of the network devices. Next, the Conversion Phase coordinates the transition of devices from the old configuration to the new one. This design is shown in Figure 2.

First, GATEKEEPER gets the tables to distribute from a trusted authority, or root of trust. This root of trust can take many forms. For example, in spaceflight, it may be the mission control team that uploads the tables to the vehicle. The root of trust generates a one-way hash of each table, and then signs the set of hashes. These will be used by GATEKEEPER to ensure consistency between the tables, as well as to constrain the way in which faulty end systems can behave.

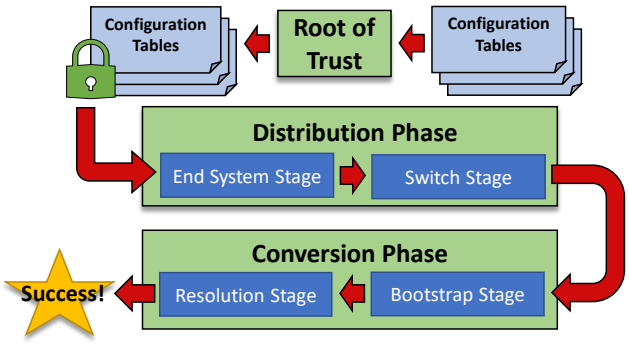

Fig. 2. The control flow of GATEKEEPER.

Next, the Distribution Phase uses a two-stage algorithm to copy the tables from the initiators to the network devices. In the first stage, or End System Stage, the initiators and witnesses agree on the configuration, and tables are sent to the end systems. In the second stage, or Switch Stage, witnesses coordinate with the initiators to provide tables to the switches. At the conclusion of this phase, consistent tables have been placed at every non-faulty device in the network.

Lastly, the Conversion Phase uses a two-stage algorithm to transition devices to the new configuration. In the first stage, or Bootstrap Stage, one plane is selected to load the configuration and verify that it operates correctly. If successful, then the second stage, or Resolution Stage, causes the rest of the devices to switch to the new configuration as well. At the conclusion of this phase, the network will either be operational under the new configuration, or all non-faulty devices will be aware the configuration did not succeed.

To make GATEKEEPER's design possible, we rely on two primitives, reliable broadcast and group test. Reliable broadcast ensures that consistent tables are distributed to all the end systems. Group test ensures faulty switches cannot interfere with migrating to the new configuration. In the following sections, we describe the GATEKEEPER protocol and these primitives in detail.

A. Distribution Phase

Before distributing new tables in the network, the tables must be preprocessed by a root of trust. First, the root of trust produces a set of hashes, one for each table. We call this the Hash Set. For simplicity, we assume each device maps to a unique index in the Hash Set, and that end system hashes come before switch hashes. Next, the root of trust signs the Hash Set to produce a digital signature, which it attaches to the set. The signed Hash Set uniquely represents the new network configuration. The initiators then use Hash Set to distribute tables to the end systems and switches.

End System Stage. The end system distribution protocol is shown in Algorithm 1. Let H be the signed Hash Set, R be

```
for r_i ∈ R do
    r_i ByzantineBroadcast(H);
    All Receive (H) from r_i;
    for w_j ∈ W do
        if Sign(H) = Sign(RootOfTrust) then
            w_j ByzantineBroadcast(accept);
        else
            w_j ByzantineBroadcast(reject);
        end
    end
    consensus ← 0;
    for w_j ∈ W do
        All Receive(bit) from w_j;
        if bit = accept then
            consensus++;
        end
    end
    if consensus > f then
        break;
    end
end
for r_i ∈ R do
    for j ← 0 ... (E − 1) do
        r_i Send(T[j]) to end system j;
    end
    All Receive(MyTable) from r_i;
    if Hash(MyTable) = H[MyIndex] then
        accept configuration table;
    end
end
```

Algorithm 1: Distribution Phase, End System Stage

the set of initiators, W be the set of witnesses, T be the set of tables, and $MyIndex$ be the index of the end system.

First, an initiator uses a Byzantine broadcast (BB) [19] protocol to send the signed Hash Set to all end systems. The BB protocol ensures that, even if the initiator is faulty, all non-faulty receivers agree on the Hash Set they receive. There are multiple ways to implement BB, including voting messages over the redundant planes, or relying on cryptographic methods [15]. Next, the witnesses BB a bit indicating whether the signature is valid. If $> f$ witnesses accept the broadcast, we proceed with the protocol. Otherwise, the Hash Set is rebroadcasted by the next initiator.

At the conclusion of this step, all non-faulty end systems agree on the correct Hash Set. The initiators then all send each configuration table to the corresponding end system. Each end system only accepts a table if it matches the corresponding entry in their accepted Hash Set.

LEMMA 1. All non-faulty end systems possess the correct Hash Set before the tables are distributed.

Proof. First, we prove that for each initiator r_i, either all non-faulty end systems accept the same Hash Set from r_i, or reject it. Since the Hash Set is broadcasted with a BB protocol, it is the same for all non-faulty end systems. Since the witnesses broadcast their `accept` bits using BB, it is also the same for all non-faulty end systems. Thus, all non-faulty end systems either choose to accept the same Hash Set, or reject it.

Next, we prove that, if an initiator r_i is faulty, it cannot cause the non-faulty end systems to accept an incorrect Hash Set. A non-faulty end system accepts the Hash Set if it is authenticated by $> f$ witnesses, of which ≥ 1 must be non-faulty. A non-faulty witness only authenticates a Hash Set if it is signed correctly by the root of trust. Thus, any Hash Set accepted by the non-faulty end systems must be correct.

Next, we prove that all non-faulty end systems will end up with the correct Hash Set. Since there are $\geq f + 1$ initiators, there is ≥ 1 non-faulty initiator. That means that, if all other initiators are faulty and fail to get the Hash Set accepted, ≥ 1 non-faulty initiator will broadcast the correct Hash Set to all end systems. In the worst case, $2f + 1 - f = f + 1$ witnesses accept the broadcast, which causes all non-faulty end systems to accept the correct Hash Set. □

LEMMA 2. At the conclusion of the algorithm, all non-faulty end systems possess their correct configuration table.

Proof. All initiators send each table to the corresponding end system. Since there are $\geq f + 1$ initiators, there is ≥ 1 non-faulty initiator that sends the correct table to each end system over the redundant planes. Since there are $\geq f + 2$ planes, at least one plane is non-faulty. Lemma 1 implies that all non-faulty end systems possess the correct Hash Set. Thus, all non-faulty end systems receive a correct table, check it against the correct Hash Set, and accept the table. □

Switch Stage. After tables are distributed to the end systems, initiators send them to the switches. The switch distribution protocol is shown in Algorithm 2. Again, H is the signed Hash Set. U is a set of indices of switches that have not yet been configured. Initially, U contains all switch indices. `REQUEST` is a command sent to a switch to request a hash of its table. `LOCK` is a command sent to a switch to tell it to stop accepting new tables. A switch only responds to `LOCK` if it receives $\geq f + 1$ `LOCK` commands.

First, an initiator sends each table to the corresponding switch. The witnesses then request a hash of the table from each switch[1], and broadcast a bit indicating whether the received hash matches the Hash Set. If enough witnesses accept the table, then the switch is considered configured and removed from U. The process is repeated for every initiator, with witnesses only requesting the tables of unconfigured switches. To reduce the communication overhead, the initiators could also listen to the witness broadcasts and maintain U in order to avoid sending tables to already configured switches.

Importantly, since frames transmitted to switches cannot be duplicated on independent channels, a faulty switch has the ability to block an end system from communicating with another switch. The severity of this problem depends on the topology. For simplicity, we assume all switches are *reachable*, meaning: (1) there are $\geq f + 1$ paths from each switch to initiators (some of which may be faulty), and (2) there are

[1]We note that the ability to broadcast the hash of a loaded table is a standard function of some real-time Ethernet switches [22].

```
for r_i ∈ R do
    for j ← E ... (E + S − 1) do
        r_i Send(T[j]) to switch j;
    end
    for w_j ∈ W do
        for u ∈ U do
            w_j Send(REQUEST) to switch u;
            h ← hash from switch u;
            if h = H[u] then
                Broadcast(accept);
            else
                Broadcast(reject);
            end
            consensus ← 0;
            for w_k ∈ W do
                w_j Receive(bit) from w_k;
                if bit = accept then
                    consensus++;
                end
            end
            if consensus > f then
                w_j Send(LOCK) to switch u;
                U.remove(u);
            end
        end
    end
end
```

Algorithm 2: Distribution Phase, Switch Stage

$\geq 2f + 1$ paths from each switch to witnesses. We note that, these conditions are always met in single-hop architectures (like Figure 1). For multi-hop architectures, they can be met with careful device placement, or by increasing the number of initiators or witnesses.

LEMMA 3. At the conclusion of the algorithm, all non-faulty switches possess their correct configuration table.

Proof. First, we prove that a non-faulty switch will never lock an incorrect table. Assume a non-faulty switch *did* lock an incorrect table. This means the switch received $\geq f+1$ `LOCK` commands from witnesses, which means ≥ 1 came from a non-faulty witness. A non-faulty witness only sends `LOCK` if it receives $> f$ `accept` broadcasts, which means that ≥ 1 came from a non-faulty witness. A non-faulty witness only broadcasts `accept` if the hash it received from the switch matches the corresponding entry of Hash Set. Thus, the switch table is correct, which is a contradiction.

Next, we prove that a switch will eventually lock the correct table. All initiators send each table to the corresponding switch. Since there are $\geq f + 1$ initiators, there is ≥ 1 non-faulty initiator. This means that every non-faulty switch receives > 1 correct table. Since there are $\geq 2f+1$ witnesses, there are $\geq f+1$ non-faulty witnesses. Each non-faulty witness will request the switch's hash, see it matches Hash Set, and broadcast `accept`. As a result, all non-faulty witnesses will receive $> f$ `accept` bits and send `LOCK` to the switch. Since there are $\geq f + 1$ non-faulty witnesses, the switch receives $\geq f + 1$ `LOCK` commands and locks in the table. □

B. Conversion Phase

Once the Distribution Phase is complete, all non-faulty devices possess their correct configuration tables. However, none of the devices have yet transitioned to *using* the new tables. The purpose of the Conversion Phase is to switch devices over to the new configuration, while minimizing interruption to the network. This is done by leveraging the redundancy of the network planes (see §III-A), and migrating the planes to the new configuration one at a time.

In order to minimize downtime, end systems are reconfigured *after* the first plane is reconfigured, but *before* the other planes. This way, the interruption to the traffic flows between end systems is determined only by the time needed for the end systems to reconfigure. End systems can communicate with each other up until the moment they are commanded to reconfigure. Also, as soon as their reconfiguration is complete, a plane is already ready to direct their new traffic flows.

In order for this approach to be most effective, it is important to ensure that the first plane that is reconfigured is non-faulty. Otherwise, end systems will experience some interruption until the other planes are reconfigured as well. To help accomplish this goal, we split the Conversion Phase into two Stages. The Bootstrap Stage is used to identify the first plane to migrate, and to ensure it is non-faulty with high probability. The Resolution Stage is used to carefully migrate the end systems and remaining planes.

Bootstrap Stage. The Bootstrap Stage is shown in Algorithm 3. It is executed by all the witnesses. Let `CONVERT` be a command sent to a switch telling it to load a new configuration. A switch only responds to `CONVERT` if it receives $\geq f + 1$ `CONVERT` commands. Let `GroupTest` be a routine used by the witnesses to test a plane after it has been reconfigured.

First, the witnesses send a command to reconfigure the switches in a particular plane. Next, they execute the `GroupTest` to determine whether the plane behaves correctly with the new configuration. This `GroupTest` can be as simple as communicating a predetermined pattern between witnesses. If the test completes successfully, then the Bootstrap Stage is complete and the witnesses proceed with the rest of the Conversion Phase. Otherwise, the witnesses repeat the process with the next plane. Note that for the Bootstrap Stage to be successful, it is only necessary to test $f + 1$ planes.

The ability for GATEKEEPER to select a non-faulty plane in the Bootstrap Stage, and thus to minimize downtime, depends on the sophistication of the `GroupTest`. At a minimum, we assume that `GroupTest` has the following properties.

1) **Adequate evaluation:** The test demonstrates that the reconfigured plane has a high probability of operating successfully until the other planes are reconfigured.
2) **No false negatives:** If the reconfigured plane is non-faulty, faulty witnesses cannot cause the test to fail.
3) **Consistent results:** All non-faulty witnesses agree on whether the test succeeds or fails. This can be accomplished using a BB protocol, as in Algorithm 1.

```
// Bootstrap Stage
result ← Fail
for i ← 0 ... f do
    Send(CONVERT) to all switches in plane i;
    result ← GroupTest (i);
    if result = Pass then
        break;
    end
end
if result ≠ Pass then
    Exit, configuration is incorrect;
end
// Resolution Stage
Send(CONVERT) to all end systems;
for j ← (i + 1) ... (m − 1) do
    Send(CONVERT) to all switches in plane j;
end
```

Algorithm 3: Conversion Phase

LEMMA 4. At the conclusion of the Bootstrap Stage, ≥ 1 non-faulty plane will still be configured with the previous configuration.

Proof. The Bootstrap Stage reconfigures the planes one at a time, stopping as soon as the `GroupTest` succeeds. Per the assumptions above, a non-faulty plane must pass the `GroupTest`. Since there are $\geq f + 2$ planes, there are ≥ 2 non-faulty planes. Thus, when the `GroupTest` first succeeds, there must be ≥ 1 non-faulty plane that has not been reconfigured. □

We note that, besides selecting an (ideally) non-faulty plane to reconfigure, the Bootstrap Stage can also be used to detect errors in the configuration itself. The Bootstrap Stage runs `GroupTest` on $f+1$ planes in the worst case, of which one must be non-faulty. Per the assumptions above, `GroupTest` is guaranteed to succeed for any non-faulty plane. Thus, if no `GroupTest` has succeeded at the conclusion of the Bootstrap Stage, there must be a problem with the new configuration. For example, the constraints for sending and receiving messages may not be consistent. Per our earlier assumptions, all non-faulty witnesses agree on the results of each test. Thus, if `GroupTest` never succeeds, all non-faulty witnesses are aware and can work together to perform the appropriate recovery action. We note that, even if the new configuration is incorrect, ≥ 1 plane is guaranteed to still have the old configuration, so in most cases, the system can continue to operate normally while the recovery is performed.

Resolution Stage. At the conclusion of the Bootstrap Stage, one plane has been migrated to the new configuration and passed the `GroupTest`. In the Resolution Stage, the end systems are commanded to switch to this new configuration. Afterwards, the other planes are reconfigured as well. The protocol is shown in Algorithm 3. It is executed by all the witnesses. `CONVERT` is a command used to reconfigure the switches and end systems. Both require $\geq f + 1$ `CONVERT` commands in order to take action.

Whether or not frames are dropped during the Resolution Stage depends on how tightly coordinated the end systems are when switching to the new configuration. In a TTE architecture, where devices are tightly synchronized, and the times at which frames are sent and received are known, drops can be eliminated by having end systems reconfigure at specific times at which no frames are in transit. Many recent works have studied how to minimize drops in non-time-triggered networks, or networks in which frames are in transit [23], [24]. Any of these could be used in GATEKEEPER.

After the end systems are reconfigured, all that remains is to reconfigure the planes that were not reconfigured in the Bootstrap Stage. Once this is done, GATEKEEPER terminates.

THEOREM 1. At the conclusion of GATEKEEPER, all non-faulty devices have loaded their correct configuration table.

Proof. Lemma 2 implies that, at the conclusion of the Distribution Phase, all non-faulty end systems possess their correct table. In the Conversion Phase, each witness sends `CONVERT` to all end systems. Since there are $\geq 2f+1$ witnesses, $\geq f+1$ witnesses are non-faulty. Thus, all non-faulty end systems receive $\geq f + 1$ `CONVERT` commands and reconfigure.

Lemma 3 implies that, at the conclusion of the Distribution Phase, all non-faulty switches possess their correct table. In the Bootstrap Stage, each witness sends `CONVERT` to all switches in planes $0...i$. Since $\geq f + 1$ witnesses are non-faulty, all switches in planes $0...i$. receive $\geq f+1$ `CONVERT` commands and reconfigure. In the Resolution Stage, each witness sends `CONVERT` to all switches in the remaining planes. Again, since $\geq f + 1$ witnesses are non-faulty, all switches in those planes also reconfigure. □

V. IMPLEMENTATION

We evaluated GATEKEEPER by implementing a prototype for a fault-tolerant TTE network. Altogether, the prototype comprises around 4,100 lines of C code.

Our prototype runs on a real TTE testbed consisting of 4 end systems and 3 switches. End systems are composed of TTTech A664 Lab end system cards attached to 4-core Intel i3-450 host processors over a Peripheral Component Interconnect Express (PCIe) bus. Switches are 24-port TTTech Space Lab switches. End system hosts run Ubuntu 14.04 LTS with kernel 3.13.0-36-generic x86_64. End systems use configuration tables that are approximately 1.93 kB in size ($\pm$ 18 bytes). Switches each require two tables totaling 4.50 kB.

The network is configured with 3 redundant planes. Links between TTE switches and end systems are 100 Mbps. Loading tables onto the switches is done via Trivial File Transfer Protocol (TFTP). Switches are commanded and publish telemetry via Simple Network Mapping Protocol (SNMP). Because our TTE end systems do not have access to the Linux network stack, protocol steps that require TFTP and SNMP were done via a standard Ethernet utility network connecting the switches to the hosts. In the future, these steps can instead be performed over the TTE network.

For comparison to GATEKEEPER, we also implemented FASTREC. FASTREC is a reconfiguration script that simply distributes new tables to the devices, then orders them to load

the tables. Switches reconfigure first, then the end systems. FASTREC represents the fastest possible method for reconfiguring a network, but does not provide any fault tolerance guarantees. As a result, a single end system or switch fault could result in an inoperable network.

VI. EVALUATION

To characterize GATEKEEPER's performance, we conducted a series of benchmarks on our prototype, as well as simulations to extrapolate the results to larger systems. We strived to answer three key questions: (1) how high is GATEKEEPER's communication overhead?, (2) how long does GATEKEEPER take to finish reconfiguration?, and (3) how effective is GATEKEEPER at minimizing downtime during reconfiguration?

For each question, we characterized GATEKEEPER in three fault settings. GK_c signifies trials of GATEKEEPER without any faults. GK_{es} signifies trials with one faulty end system assigned as both the first initiator and a witness. GK_{sw} signifies trials with a faulty switch in the first plane to be bootstrapped. GK_{es} and GK_{sw} use worst-case faults for their respective device types (see §III-A).

A. Communication Overhead

Experimental Setup. For this experiment, we measured the number of bytes transmitted from each end system port during reconfiguration of our testbed. Since frame duplication is handled by the switches, broadcasts and unicasts contribute equally to the communication overhead.

To determine the communication overhead for systems *larger* than our testbed, we used the following equations. Note that each time the number of end systems exceeds the available switch ports, a new switch is added.

$$\text{FASTREC} = 2(E+S)$$
$$\text{GK}_c = RE + S + (4S + E + 1)W + G + 2$$
$$\text{GK}_{es} = RE + 2S + (7S + E + 2)W + G + 4$$
$$\text{GK}_{sw} = R(E+1) + S + (4S + 3R + E)W + 2G + 1$$

The equations calculate the total number of messages transmitted. The total number of bytes is found by multiplying each constant by the corresponding message size. E, S, W, and R are the number of end systems, switches, witnesses, and initiators respectively. G is the number of messages transmitted during the group test.

Results. The results of our testbed measurements are shown in Figure 3. The main contributor to the communication overhead is the transfer of redundant tables durng the Distribution Stage. Faults increase the communication overhead by increasing the number of times tables are transferred. However, in the worst case, GATEKEEPER only communicated around 30 KB more than the non-fault-tolerant solution.

Figure 4 shows how GATEKEEPER's communication overhead scales to larger systems. We see that, like FASTREC, GATEKEEPER's communication overhead grows gracefully. The main reason for GATEKEEPER's scalability is the use of the Hash Set, which can represent the tables for a large number of devices with only a small increase in size. Furthermore,

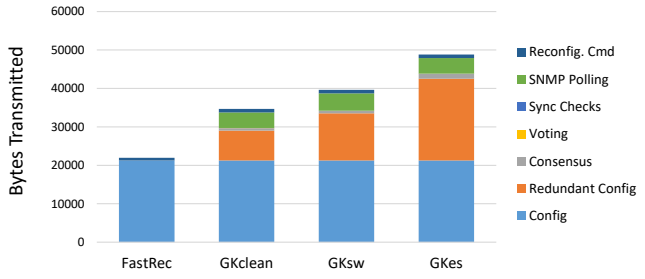

Fig. 3. Number of bytes communicated by different parts of the protocols in our testbed. In the worst case, GATEKEEPER communicated < 50 KBs.

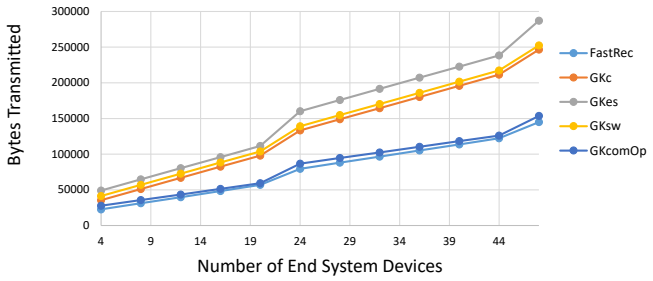

Fig. 4. Communication overhead for increasing network sizes.

the number of initiators and witnesses is determined by the number of faults, not the number of devices. This means that, even for large systems, much of the communication stays between a relatively small number of devices.

B. Execution Time

Experimental Setup. For this experiment, we measured the execution time of each part of GATEKEEPER in our testbed. To do this, we had to set timeouts for each GATEKEEPER operation. For timeouts used by initiators and witnesses, 2 s was used. For underlying utilities (e.g., SNMP), 5 s was used, which in most cases was the default.

Again, we used equations to predict the execution time of GATEKEEPER on systems larger than our testbed.

$$\text{FASTREC} = (D_{SW} + K_{SW})S + (D_{ES}E + K_{ES})E$$
$$\text{GK}_c = C + D_{ES}E + D_{SW}S + B + K_{ES} + 2K_{SW} + G$$
$$\text{GK}_{es} = C_{TO} + C + D_{ES_{TO}} + D_{SW_{TO}} + D_{SW} + B_{TO} + G + K_{ES} + 2K_{SW}$$
$$\text{GK}_{sw} = C + D_{ES}E + D_{SW}(S-1) + 2D_{SW_{TO}} + B_{TO} + B + 2G + K_{ES} + K_{SW}$$

The variables are the time taken for consensus (C), distribution to a device (D), bootstrapping (B), the group test (G), and reconfiguration of a device (K) in seconds. We used timeout values for variables denoted by TO. Since FASTREC does not wait for the result of reconfiguration, K_{ES*} denotes the time to send an SNMP `SET` command (0.293 s).

Results. The results of our testbed measurements are shown in Figure 5. We see that, in the absence of faults, GATEKEEPER performs nearly as fast as the non fault-tolerant solution. This is expected, as GATEKEEPER performs only a couple

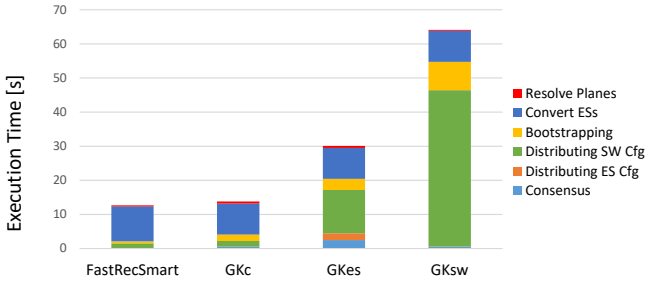

Fig. 5. Execution time of different parts of the protocols in our testbed. In the worst case, GATEKEEPER terminated in around 1 minute.

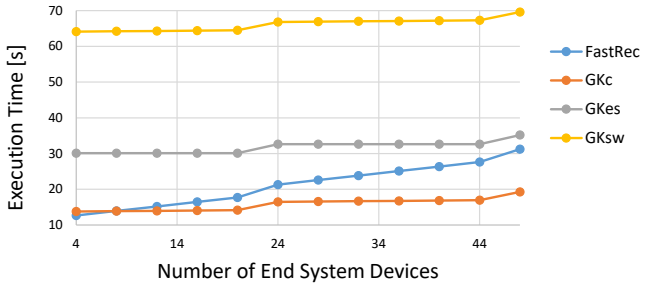

Fig. 6. Execution time for increasing network sizes.

additional steps — distributing the vector of hashes and confirming the switch tables.

Under faults, GATEKEEPER becomes relatively slower, as it may need to cycle through multiple initiators, or bootstrap multiple planes in order to deploy the new configuration successfully. However, we note that GATEKEEPER's worst case time overhead is modest compared to its substantial dependability improvements. In the worst case, reconfiguration took only around 64 s.

Figure 6 shows how GATEKEEPER's execution time scales to larger systems. We note that, in the absence of faults, GATEKEEPER actually performs *faster* than the non-fault-tolerant solution for moderate system sizes. The reason is that, as the system size increases, the overhead of FASTREC issuing reconfiguration commands in series begins to outweigh GATEKEEPER's fault tolerance overhead.

Moreover, we see that the execution time of GATEKEEPER under faults stays relatively constant. This means that GATEKEEPER can be used practically even in very large systems. For example, under worst case switch faults, GATEKEEPER still takes under 70 s to reconfigure a network with 50 end systems. This makes it only 120% slower than the non-fault-tolerant solution.

C. Network Availability

Experimental Setup. For this experiment, we measured the network downtime caused by GATEKEEPER in our testbed.

We defined network downtime as the time between when the network (with the previous configuration) becomes unavailable, and the network becomes available again with the new configuration. We considered the network as available if all non-faulty end systems are operational, and there is at least one non-faulty path between all non-faulty end systems.

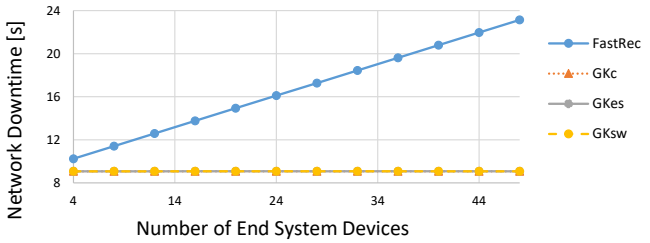

Fig. 7. Network downtime for our testbed. The smallest achievable downtime with our hardware is 9.07 s.

Results. As shown in Figure 7, GATEKEEPER achieves extremely small downtimes regardless of the size of the system, or whether there are faults. The reason is that all end systems are synchronized and reconfigure at the same time. Moreover, a network plane with the new configuration is available as soon as the end systems are ready to use it. The measured downtime was around 9.072 s, which is only 2 ms more than the time needed to reconfigure our end systems. That is, GATEKEEPER nearly minimizes system unavailability.

GATEKEEPER's small timing overhead stems from the synchronization method used in the prototype, which required asynchronous message passing. With an improved synchronization method, such as using the synchronized network to reconfigure at predetermined points in time, it could be possible to further reduce this overhead.

VII. RELATED WORKS

Network Configuration Agents. An emerging paradigm in industrial networks is the use of "configuration agents", which are hosted on each device and continuously optimize the device's traffic characteristics depending on the changing needs of the network [25]. Often, realizing these configuration agents requires a mix of both software and custom hardware, based on the specific network technology that is used [26]. In contrast, GATEKEEPER is a method for reliably deploying network configurations that are generated *offline*. It is compatible with existing hardware available today, and is broadly applicable to several different network technologies.

Frame Consistency. Many works strive for frame consistency, meaning that each frame is processed only once during a reconfiguration period [27]. This is usually done by analyzing traffic dependencies offline, then migrating traffic flows to a new configuration one at a time [23]. In contrast, GATEKEEPER leverages the redundancy of the network to enable traffic to flow simultaneously in both the old and new configurations. Thus, by synchronizing the times at which end devices reconfigure, e.g., to the TT clock, it is possible to achieve frame consistency without requiring offline analysis. Moreover, existing methods to ensure frame consistency could be integrated into GATEKEEPER's Resolution Stage (see §IV-B).

Zero-Frame-Loss Reconfiguration. Several techniques to ensure frame consistency have been improved to achieve zero-

frame-loss reconfiguration [9]. Like earlier approaches, these require traffic dependencies to be analyzed offline, and for traffic flows to be migrated to the new configuration at specific times in which the flows are not being used [8]. GATEKEEPER's use of redundant network planes could simplify the goal of achieving zero-frame-loss reconfiguration. Moreover, existing zero-frame-loss protocols all assume reliable distribution of the calculated configuration, and GATEKEEPER's Distribution Phase (§IV-A) can be used for this purpose.

K-Phase Reconfiguration. K-phase reconfiguration protocols use version tagging of frames to tie each frame to a specific configuration [28]. In this way, K-phase protocols can achieve high levels of frame consistency with a lower reliance on offline analysis than other protocols, and with less control over the times at which traffic flows move to the new configuration [29]. In contrast to these works, GATEKEEPER does not require frames to be tagged based on a given configuration. Moreover, GATEKEEPER is compatible with existing network technologies that do not support tagging.

VIII. Conclusion

This paper presented GATEKEEPER, a new fault-tolerant reconfiguration protocol for real-time Ethernet systems. GATEKEEPER allows designers to update their networks without bringing the system offline, and without sacrificing fault tolerance. Our evaluation shows that GATEKEEPER adds only modest communication and execution time overheads compared to a non-fault-tolerant solution. It also achieves downtimes that are within single-digit milliseconds of being optimal on our testbed. These characteristics make GATEKEEPER a great candidate for emerging applications like long-duration spaceflight and industrial control, where systems continue to evolve long after they are deployed.

IX. Acknowledgements

This research was supported by the NSF Center for Space, High-performance, and Resilient Computing (SHREC) industry and agency members, and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783. We also thank TTTech for their generous loans and discounts on equipment used to evaluate our research. A final thanks to NASA Johnson Space Center's Artemis Network Validation and Integration Lab (ANVIL) for their efforts in preliminary validation of our proposed techniques.

References

[1] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer, "The Time-Triggered Ethernet (TTE) Design," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, 2005, pp. 22–33.

[2] *Aircraft Data Network Part 7: Avionics Full-Duplex Switched Ethernet Network-ARINC Specification 664 P7-1*, ARINC, 2009.

[3] "IEEE Standard for Local and Metropolitan Area Networks — Audio Video Bridging (AVB) Systems," Institute of Electrical and Electronics Engineers (IEEE) Audio Video Bridging Task Group, Standard IEEE 802.1BA:2011, 2011.

[4] C. R. Spitzer, *The Avionics Handbook*. CRC Press, 2001.

[5] "International Avionics System Interoperability Standards (IASIS): Avionics Standard," NASA Johnson Space Center, Tech. Rep., 2019.

[6] M. Hermann, T. Pentek, and B. Otto, "Design Principles for Industrie 4.0 Scenarios," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, 2016, pp. 3928–3937.

[7] Z. Li, H. Wan, Z. Pang, Q. Chen, Y. Deng, X. Zhao, Y. Gao, X. Song, and M. Gu, "An Enhanced Reconfiguration for Deterministic Transmission in Time-Triggered Networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1124–1137, 2019.

[8] J. Lu, H. Xiong, F. He, Z. Zheng, and H. Li, "A Mixed-Critical Consistent Update Algorithm in Software Defined Time-Triggered Ethernet Using Time Window," *IEEE Access*, vol. PP, pp. 1–1, 04 2020.

[9] Q. Zhao, B. Liu, Y. Peng, Q. Liu, Q. Xu, and X. Li, "Dynamic Configuration Method of Satellite Time-Triggered Ethernet," in *2020 Chinese Automation Congress (CAC)*, 2020, pp. 4746–4753.

[10] J. Sommer, S. Gunreben, F. Feller, M. Kohn, A. Mifdaoui, D. Sass, and J. Scharf, "Ethernet – A Survey on its Fields of Application," *IEEE Communications Surveys Tutorials*, vol. 12, no. 2, pp. 263–284, 2010.

[11] "Time-Triggered Ethernet," SAE International, Tech. Rep. SAE AS6802, Nov. 2016.

[12] "TSN Profile for Industrial Automation," Institute of Electrical and Electronics Engineers (IEEE) Time-Sensitive Networking Task Group, Standard IEC/IEEE 60802, Jul 2020.

[13] "Industrial Communication Networks - Fieldbus Specification," Standard IEC 61158.

[14] *TTE-Plan User Manual*, TTTech Computertechnik AG, 2020, version 5.4.6000.

[15] A. Loveless, "On Time-Triggered Ethernet in NASA's Lunar Gateway," https://ntrs.nasa.gov/citations/20205005104, July 2020.

[16] E. Blasch, P. Kostek, P. Pačes, and K. Kramer, "Summary of Avionics Technologies," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 9, pp. 6–11, 2015.

[17] "Hi-Rel Solutions for Space Launch Vehicles," https://www.tttech.com/wp-content/uploads/TTTech-Launcher-Use-Case.pdf, TTTech Computertechnik AG, Tech. Rep.

[18] R. Zurawski, *Industrial Communications Technology Handbook, 2nd Edition*. CRC Press, 2017.

[19] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982.

[20] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, Berkeley, CA, USA, 1999.

[21] W. Steiner, "TTEthernet: Time-Triggered Services for Ethernet Networks," in *2009 IEEE/AIAA 28th Digital Avionics Systems Conference (DASC)*, 2009, pp. 1.B.4–1–1.B.4–1.

[22] *TTE Switch Lab Space User Manual*, TTTech Computertechnik AG, 2019, rev 1.0.0.

[23] Z. Pang, X. Huang, Z. Li, S. Zhang, Y. Xu, H. Wan, and X. Zhao, "Flow Scheduling for Conflict-Free Network Updates in Time-Sensitive Software-Defined Networks," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 1668–1678, 2021.

[24] T. Mizrahi, E. Saat, and Y. Moses, "Timed Consistent Network Updates," in *SIGCOMM Symposium on Software Defined Networking Research*. New York, NY, USA: Association for Computing Machinery, 2015.

[25] M. Gutiérrez, W. Steiner, R. Dobrin, and S. Punnekkat, "A Configuration Agent Based on the Time-Triggered Paradigm for Real-Time Networks," in *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, 2015, pp. 1–4.

[26] L. Wisniewski, S. Chahar, and J. Jasperneite, "Seamless Reconfiguration of Time Triggered Ethernet Based Protocols," in *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, 2015, pp. 1–4.

[27] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent Updates for Software-Defined Networks: Change You Can Believe In!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets. New York, NY, USA: Association for Computing Machinery, 2011.

[28] N. P. Katta, J. Rexford, and D. Walker, "Incremental Consistent Updates," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 49–54.

[29] T. Mizrahi, E. Saat, and Y. Moses, "Timed Consistent Network Updates in Software-Defined Networks," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3412–3425, 2016.