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

by

Brendan Kristopher Luksik

B.S. Computer Engineering, University of Pittsburgh, 2018

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2021

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Brendan Kristopher Luksik

It was defended on

July 16th 2021

and approved by

Mai Abdelhakim, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Jun Yang, Ph.D, Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Alan D. George, Ph.D., Mickle Chair Professor, Department of Electrical and Computer Engineering

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

Brendan Kristopher Luksik, M.S.

University of Pittsburgh, 2021

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 loss of connectivity. 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, Gatekeeper, the first reconfiguration protocol for real-time Ethernet systems that minimizes downtime in both the presence and absence of faults, is described. 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. We evaluated Gatekeeper's scalability to different system sizes, its timing and communication overheads, and it's correctness in the presence of different faulty configurations. Our results show that Gatekeeper can tolerate a faulty device with as little as 3% communication overhead, and while running faster than naive non-fault-tolerant solutions in large networks.

Table of Contents

List of Tables

List of Figures

Preface

This work was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

I would like to thank Andrew Loveless and the NASA Johnson Space Center Avionics Networks Team for their expertise, guidance, and feedback. I would also like to thank the students of SHREC for their in help navigating the research process. Finally, I would like to thank my family for their infinite patience and support throughout this process.

1.0 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) [11], Avionics Full Duplex (AFDX) [3], and IEEE 802.1 AVB [9]. These protocols provide many advantages over standard Ethernet, such as support for different traffic criticalities, time-synchronization, ordering guarantees, more deterministic timing, 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, airplane avionics networks and automotive networks rarely require any network configuration updates, unless a problem is identified with the previous configuration. 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 [1] and Industry 4.0 platforms [7], 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 minimizes network unavailability 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 in minimizing reconfiguration disruptions in real-time Ethernet networks [13] [15]. 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 [1], 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 thesis, Gatekeeper is introduced, 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 opportunities to catch common configuration errors while they are still easily recoverable.

Gatekeeper was then evaluated by creating a prototype controller and testing it on a TTE system. Cost scaling equations were then calculated for overhead in both communication and execution time, and the growth of the protocol to large systems was characterized. The results show that data overhead always approaches a consistent cost dictated by the configuration, which can be optimized as low as 3% of the cost to do an equivalent non-fault-tolerant deployment. Crucially, the results also show that the protocol minimizes the downtime of the system, regardless of fault placement or scale of the system.

In summary, this thesis make the following contributions.

- GATEKEEPER: an online reconfiguration protocol able to deploy configurations in the presence and absence of faults for a variety of real-time Ethernet protocols.
- A prototype of GATEKEEPER for TTEthernet systems
- An experimental evaluation of GATEKEEPER including observed benchmarks and calculated scaling characteristics.

2.0 Related Works

Gatekeeper joins a large body of research working towards seamless reconfiguration of real-time Ethernet systems. A popular reconfiguration paradigm in industrial networks is the reconfiguration agent [6] [5]. Such agents auto-detect and auto-configure new tables rather than simply deploying them, a key feature in the Industry 4.0 design principle [7]. Implementations either insert a scheduler into the link-layer path to manage dataflows and transitions [29], or provide a central controller which can communicate gate control lists to real-time devices [21]. While they do automate device and traffic detection, configuration agent protocols usually require hardware modifications along side the software or constitute their own standard, limiting the breadth of their applicability.

More generally acceptable solutions often involve K-phase protocols. K-phase reconfiguration uses version tags to tie each frame to *only* one configuration. These protocols aim for lossless, consistent network updates, using rules to guide each flow's transition. Using versions enables incremental updating and stability for co-existing configurations [10]. Using accurate time to coordinate the phase commits improves on temporal overhead [18] [17]. To this end, the Time4 algorithm introduces the idea of flow-swapping which can solve deadlocking scenarios in K-phased approaches [16]. While successful at maximizing network availability, these algorithms require tight coordinate of full end-to-end paths, which makes them vulnerable to fault manifestations.

Scheduled real-time Ethernet protocols can leverage their temporal nature to achieve lossless reconfiguration. Creation and analysis of dependence graphs is used to allocate points-in-time to reconfigure individual dataflows during windows where such flows are not in use [13] [15]. Originally requiring software-defined network (SDN) architectures, improved versions of zero-loss reconfiguration protocols can be applied to reliable, remote systems, such as satellites. The improved protocols work by both generating an update schedule and load-balancing traffic from the incoming configuration to limit contention with the outgoing configuration [30]. This allows for reset scheduling of networks become possible, where as prior presented methods could handle only minor additions or subtractions to the standing schedule. The shortcoming of this group of work is that zero-loss is only achievable if the scheduling data arrives at each node properly, which cannot always be taken for granted.

Lastly, it is worth touching on the idea of frame consistency, or the goal of having every packet processed properly and exactly once during a reconfiguration period. This idea underpins much of the K-phase and schedule-based reconfiguration techniques. SDNs have been a prime candidate for creating frame-consistency techniques because of their dedicated control structure. Consistent network updates can be abstracted [23] by making use of one-touch and unobservable updates [22]. Using such mechanisms raises the concern of schedulability, though this has recently been addressed for both online and offline scenarios by [19]. Even though these flow scheduling update paradigms continue to improve in cost-bounding and schedulability, they rely on computational analyses of the content of the schedule. This is an expensive task and can limit the ability to deploy new schedules if unresolved conflicts arise during scheduling.

The existing body of work addresses rigorous assurance that data is not lost during reconfiguration, but in all cases the functionality of all devices in the system is taken for granted. However, that cannot always be the case. GATEKEEPER brings the consideration of fault-tolerance to the body of research, and, as will be shown in its design, can work as a standalone protocol or act as a framework around these methods which removes the affects of faulty actors and simplifies constraints on schedulability.

3.0 Background

In this section, background into real-time Ethernet networks, the protocol chosen for evaluation, and information about real-time Ethernet reliability and reconfiguration techniques is given. A case for the proposed reliable reconfiguration techniques is also made.

3.1 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 [25].

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 [3] [24] [8]. 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 [3]. 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 [20]. These mechanisms allow critical traffic to enjoy predictable timing characteristics, often while 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 take 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.

3.2 Time-Triggered Ethernet

The main real-time Ethernet protocol of interest to this work is Time-Triggered Ethernet (TTE). TTE is a link-layer networking protocol which enables deterministic frame delivery through time-division multiplexing. The TTE (SAE AS6802) standard describes the time-triggered transmission protocol and a control protocol to keep network time aligned. This standard is often combined with the rate-constraining ARINC 664p7 protocol in commercial hardware. Both protocols are able to co-exist with traditional IEEE 802.3 best-effort Ethernet frames and hardware. This combined 3-class network is also referred to as a TTE network.

The time-triggered (TT) traffic protocol schedules windows of time for privileged frames to traverse dedicated paths with sub-microsecond jitter. For logically aligned flows of frames, called virtual links (VLs), time is reserved on each link in the transmission path at a scheduled frequency. During these windows, frames of a VL are the highest priority of any traffic and are forwarded exactly according to the set schedule. Dozens to hundreds of TT VLs can co-exist in a network, but at no point can any of their transmission windows collide over a physical resource (link, egress port, buffer, etc). This is prevented ahead of time by statically scheduling each VL. Producing viable schedules is discussed below. So long as TT frames exist within properly scheduled windows, they are guaranteed not to be dropped and take a highly deterministic time to transmit [24].

The rate-constrained (RC) traffic protcol (ARINC 664p7) reserves portions of network bandwidth to provide deterministic upper-bounds on transmission latency. The bandwidth reservation is enforced by requiring a period of time between each frame in a VL, called a bandwidth allocation gap (BAG). As long as the sender does not transmit frames faster than the BAG, the RC frames are prioritized over lesser traffic while in flight. RC frames are also guaranteed to not be dropped, as long the bandwidth for any link is not exceeded. This, too, is something scheduled ahead of time to ensure that network resources support the requested bandwidths. RC frames have much more temporal flexibility than TT frames, but have longer latencies and much higher jitters because of the protocol tolerates more bursty behavior than TT traffic [3].

To establish a tight notion of time, TTE networks employ protocol control frames (PCFs). The TTE standard describes a synchronization protocol in which timestamps from a group of end systems are gathered and averaged by a switch [24]. The switch then distributes corrections to every end system within the synchronization domain to establish a tight global time. This well-coordinated time means that the windows for TT frames can be made very small and that RC BAGs can be checked accurately across each hop. These synchronization communications use PCF VLs, which must be scheduled like TT or RC traffic. However, PCF VLs are independent of other data VLs, so multiple data schedules can exist on one synchronization domain if the PCF VLs between the two configurations match.

Creating schedules for TTE systems must be done statically offline. To generate accurate windows for TT and PCF frames, and to avoid over-reserving for RC frames, parameters such the physical topology, the hardware delays, the payload sizes, and transmission frequency, etc, are gathered into a formal description of the language [28]. This represents the full list of constraints a target network has to meet.

The network description is input into a scheduler. In this work a tool called TTE-plan is used, where the constraints are met using a general problem-solver algorithm [28]. The scheduler attempts to satisfy the rules and outputs device-wise schedules describing the activity each piece of hardware will experience. It is not guaranteed that an arbitrary description is schedulable however, as constraints may conflict and be impossible to satisfy.

Scheduled solutions are input into a configuration builder, called TTEbuild. This transforms the device schedules into the configuration table binaries used by network hardware. This step is where device characteristics like memory space and functionality support are checked. A schedule is only a high-level specification and so only satisfies logical constraints of the traffic needs. Configuration tables are built against hardware specifications for each target device so the schedule can be properly fit to the network resources.

3.3 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 frames between the end systems. Real-time Ethernet architectures and hardware are similar to and often compatible with standard Ethernet equipment. As such, standard Ethernet components can exist within these networks. Typically though, standard Ethernet end systems cannot attach to multiple planes simultaneously, even though they may communicate with real-time end systems. As such, these devices cannot exhibit the necessary fault tolerance and will be ignored in this research.

To improve fault tolerance, real-time Ethernet systems can be configured in a multi-plane architecture, as shown in Figure 1. In this network model, 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, where the copies travel independently to the destination device

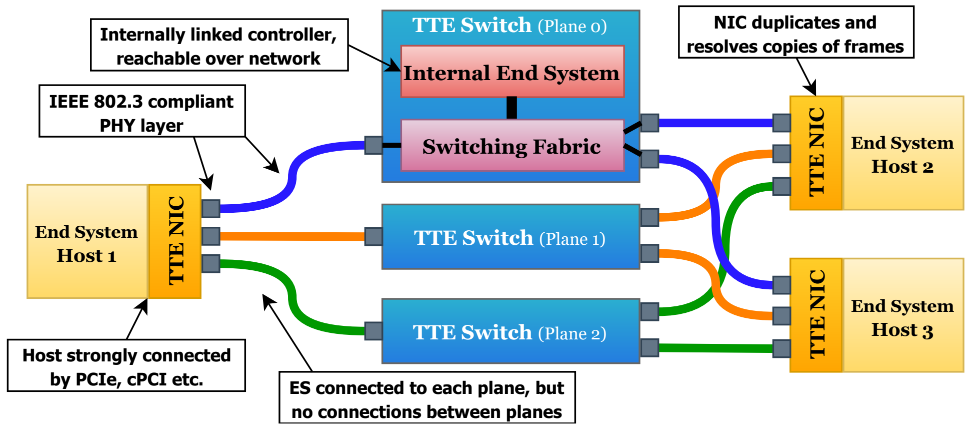

Internally linked controller, reachable over network
TTE Switch (Plane 0)
Internal End System
Switching Fabric
NIC duplicates and resolves copies of frames
IEEE 802.3 compliant PHY layer
End System Host 1
TTE NIC
TTE Switch (Plane 1)
TTE Switch (Plane 2)
End System Host 2
End System Host 3
Host strongly connected by PCIe, cPCI etc.
ES connected to each plane, but no connections between planes

Figure 1: An example of a three-plane Time-Triggered Ethernet system

[28]. 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 [14].

The multi-plane architecture is a common fault-tolerance technique found across many domains, especially avionics. These systems are highly reliable and mission-critical, and the approach is a straightforward way to create the desired level of reliability at the network level. The Airbus A380 and 400M aircraft use a duplicated AFDX avionics backbone for command and control [4]. The European Space Agency's Ariane 6 launcher [2] uses a 3-plane TTE network, as does the Sikorsky Skyraider S-97 helicopter. NASA's Orion space vehicle also uses a 3-plane TTE network, as will NASA's Lunar Gateway space station [1].

3.4 Network Reconfiguration

Network reconfiguration is necessary for adapting the statically configured hardware tables to evolving traffic requirements and timing. Table entries must account for frame payload sizes, bandwidth usage, and redundancy management for each traffic flow in the network. Since real-time table entries cannot be dynamically managed by the network hardware, a new table must be provided to a device each time shifting system requirements dictate the addition, subtraction, or change of parameters.

To change a device's configuration table, a new table is pushed from a source 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 pushed to the network interface hardware. Switches can accept configurations over the network through a programming interface. In most cases, reconfiguring a system entails bringing the system offline one device at a time as a technician controls the distribution of tables.

On the hardware evaluated for this research, switches can be configured over the network via trivial file transfer protocol (TFTP). Two tables must be pushed to each switch, for both the internal controller end system and the switch engine. Throughout the rest of this research, these two configuration files are abstracted as a single table. End systems require a single table, which can be loaded directly from a host process via the TTEthernet API.

Performing this process while the network remains online can lead to unexpected behavior. Critical frames need a dedicated path while in flight, but reconfiguration can block available routes. Further, unless the whole network shifts configurations simultaneously, two partial configurations will co-exist. During that period, frames may cross between configurations or be dropped depending how the traffic policing policies align. The order in which devices transition between configurations must be structured so dataflows swap without dropping in-flight frames. As was shown in Section 2, this has been an ongoing area of research.

3.5 The Case for Reliable Reconfiguration Techniques

The importance of fault-tolerant reconfiguration can be seen in a case study considering NASA's Gateway vehicle. The Gateway vehicle's mission within the Artemis program is to act as a partially crewed lunar-orbiting space station. The first two modules will be launched in 2024, but the station will grow to become more sophisticated over time. It is expected to host a wide variety of visiting vehicles and be able to adapt to new modules flown up over time, much like its predecessor, the ISS.

Reliable reconfiguration of its TTE avionics network is critical to Gateway's mission. The vehicle specification carries an explicit 1 fault tolerance requirement for the avionics [1]. As new modules are delivered and the station expands, the network will need to be periodically updated to accommodate the new data flows between modules. Further, visiting vehicles, a more regular occurrence, will need to access the Gateway network. The network will need to be updated to prepare for a new vehicle to visit. In either case, this may have to be done in the presence of faults. With the flight lifetime of many rad-tolerant devices limited, the vehicle is likely to outlive some of its components. Reconfiguration will inevitably be a regular network function, but any automated reconfiguration protocol will have to be able to overcome a faulty device, as replacement parts will not always be available.

4.0 Approach

In this section, the approach is outlined. First, the network model and fault assumptions are laid out. Second, the design of the protocol is presented and formally verified. Third, the experimental setup is described. Finally, the additional calculations necessary to evaluate Gatekeeper are defined and described.

4.1 Models

4.1.1 Network Model

Gatekeeper is designed for a network comprised of S switches and E end systems connected in a full-duplex Ethernet architecture. Switches are arranged into a multi-plane architecture of m planes. Each plane is generally expected to be either a daisy-chain or active star topology, though this is not an absolute requirement. End-system devices are comprised of a general-purpose host processor and a real-time Ethernet network interface card (NIC), which has $\geq m$ physical interfaces and transmits data according to protocol parameters. These components are tightly coupled over a host-NIC communication bridge, like Peripheral Component Interconnect Express (PCIe), Quad Serial Peripheral Interface (QSPI), SpaceWire, or similar.

The system is considered to be synchronous as every operation can be completed within a bounded amount of time. Real-time Ethernet communication latency always has an upper bound, and each computation required by Gatekeeper has a deterministic amount of work. As such, all devices can synchronize based on either reception of a message or by a timeout when a message is omitted.

End systems are assumed to have been assigned predetermined roles in Gatekeeper. There are three roles which can be taken: bystander, witness, and initiator. A bystander is an end system that receives a new table of the deployed configuration but does not assist in maintaining control of the deployment. A witness is an end system that coordinates with other witness end systems to maintain consensus and control during the reconfiguration period. For Byzantine consensus, the total group of witnesses should have at least *2f+1* members. An initiator is an end system that is initially given the configuration to distribute around the network. Initiators can be a subset of witnesses or a separate group, and there should be *f+1* members of this group.

Finally, the placement of role-playing end systems is considered. Network reliability is usually thought of in terms of end-to-end communication, but real-time Ethernet reconfiguration necessitates communication from end system to switch. Since each switch exists in only one plane, redundant paths are much harder to leverage for these messages. Instead, strategic placement of initiators and witnesses is used to ensure switches are reliably reached. The details for such placement are discussed in Section 4.2.1.2.

4.1.2 Failure Model

Gatekeeper is designed to tolerate Byzantine faults occurring in end systems and asymmetrically omissive faults in switches. This corresponds to the standard failure modes documented in the SAE AS6802 standard [24] used extensively in TTE and ARINC 664p7 [3] networks. The total number of faults tolerated by Gatekeeper is two less than the number of planes in the network, or *f=m-2*. The faults can be located among an arbitrary group of switches and end systems.

End-system devices are able to exhibit Byzantine failures, meaning they are capable of altering or omitting any data they touch. This can occur in an asymmetric way which can present differently to devices around the the system. A broad failure model is necessary because host devices can be any kind of processor, and ensuring they are guaranteed to reduce faults to a narrower failure mode may not be feasible. Further, host devices run foreign applications which may contain their own faults, which are impossible to consider at a network level. Because host devices are expected to participate in Gatekeeper, Byzantine behavior must be tolerated, even if the network hardware is extremely reliable.

Faulty switches can exhibit asymmetrically omissive behavior and may arbitrarily drop data but cannot pass on altered data. This downgrade from a Byzantine failure model is a realistic expectation for systems of interest. Ethernet switches are independent, self-contained, and not affected by faults induced in other components. This makes it possible to more thoroughly analyze any potential failure states which could occur. Further, many techniques such as CRCs already exist for ensuring data integrity within switches. Additionally, high-reliability versions of real-time devices are built to reduce faults to this failure mode. For example, TTE switching hardware is available in high-reliability models, which use a command and monitor (COM-MON) architecture for this purpose [26].

4.2 Design

In this section the design of GATEKEEPER is described. Formal proof of the guarantee is laid out as the protocol is described. Overall, GATEKEEPER has three major goals.

1. **Consistency.** Non-faulty devices in the system are never configured with conflicts.
2. **Minimized downtime.** Network reconfiguration occurs without disrupting the traffic flows.
3. **Fail-safe.** GATEKEEPER is guaranteed to either fully deploy the configuration or prevent deployment if errors occur.

GATEKEEPER relies on two key ideas: maintaining Byzantine consensus while distributing configuration data and testing subsets of devices as they convert to a new configuration. The protocol uses two phases, termed Distribution and Conversion, to deploy new configurations. The Distribution Phase ensures that the correct portion of the configuration arrives at each network device. The Conversion Phase then coordinates device transitions from the existing configuration to the new one. This design is depicted in Figure 2.

GATEKEEPER first uses a root of trust to generate one-way hashes of each configuration table and sign them before handing the new configuration to the initiators. A root of trust can take many forms. From the Gateway example, mission control would play this role, testing

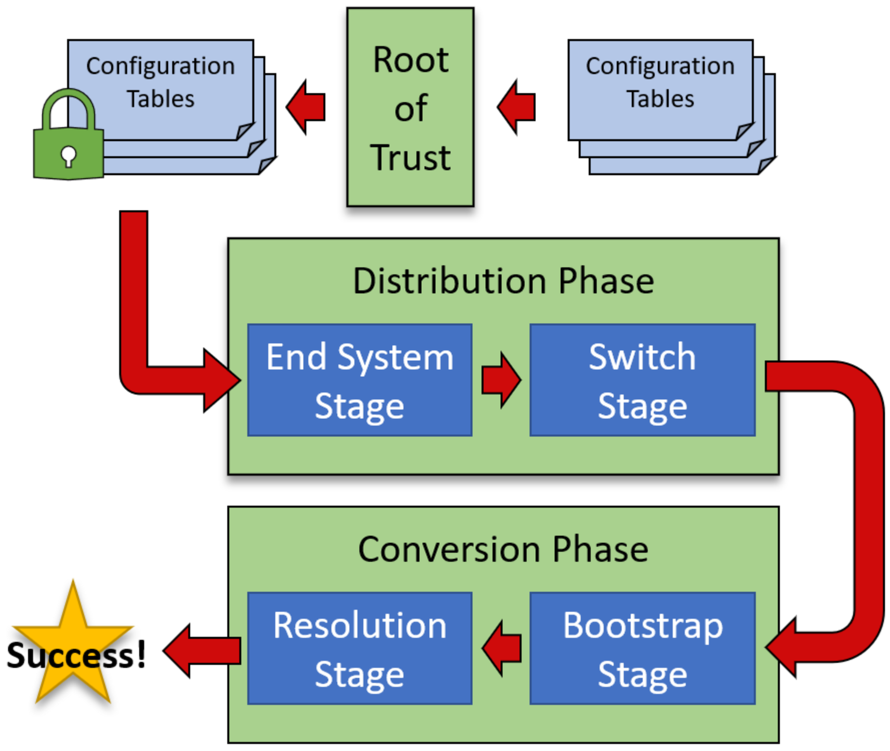


Figure 2: The control flow of GATEKEEPER

and vetting a new configuration on a ground test system, generating the hash metadata, and uploading it and the configuration to the vehicle. Generating this metadata is key to ensuring the protocol's consistency goal, allowing GATEKEEPER to reduce communication costs, and the number of ways an end system can fail.

The Distribution Phase uses a two-stage algorithm to copy the configuration tables to all devices in the network from a set of initiator devices. In the first stage, called the End System Stage, the initiators and the witnesses agree on the configuration, and all end systems are distributed their tables. In the second, termed the Switch Stage, witness end systems coordinate with the initiators to provide the switches with their configurations. At the conclusion of this phase, the appropriate configuration tables are available at every non-faulty device in the network, ready to be loaded in the conversion phase.

Next, the Conversion Phase uses a two-stage algorithm for transitioning devices to the new configuration. In the first stage, termed the Bootstrap Stage, a plane is selected to first be loaded with the new configuration and exercised to test how it handles real-time Ethernet traffic. The second stage, the Resolution Stage, completes the protocol by reconfiguring the end systems and any remaining planes in the network. At the conclusion of this phase, the network will be operational under the new configuration.

To make Gatekeeper's design possible, two primitives are necessary, reliable broadcast and group test. Reliable broadcast is required to ensure that consensus of transmitted data is achieved among valid end systems. Group test is necessary to guarantee that faulty switches cannot cause disruption. In the following sections, the protocol and these primitives are described in more detail.

4.2.1 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. This is called the Hash Set. For simplicity, it is assumed 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.

4.2.1.1 End-System Stage

The end system distribution protocol is shown in Algorithm 1. Let H be the signed Hash Set, R be 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 taking a given action.

First, an initiator uses a reliable broadcast [12] protocol to send the signed Hash Set to all end systems. At the conclusion of this step, all non-faulty end systems agree on the correct Hash Set. Reliable broadcast is a staple of Byzantine consensus algorithms and is fulfilled by three requirements:

- **Validity** - All correct receivers will eventually possess a message if a correct sender broadcasts it.
- **Agreement** - If one correct device decides on a message value, then *all* correct devices will decide the same.
- **Integrity** - A correct device decides on only one, correct value, ie. the one broadcast.

Reliable broadcasts can be realized through many mechanisms, such as voting on inputs from a multi-plane architecture, using application-level CRCs, or using other cryptographically-assured methods. In GATEKEEPER, the multi-plane architecture prevents data retransmission, which ensures the sender of the broadcast speaks only once. The reception ports perform a hybrid majority vote on the received message. This set up ensures integrity, agreement, and validity for any critical traffic transmission [14].

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 demonstrates that the system will only decide to deploy correct configurations, and Lemma 2 shows that configuration will be successfully distributed to end systems.

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 reliable broadcast, it is the same for all non-faulty end systems. Since the witnesses broadcast their `accept` bits also using reliable broadcast, 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.

```
for $r_i \in R$ do
    $r_i$ ByzantineBroadcast($H$);
    All Receive ($H$) from $r_i$;
    for $w_j \in W$ do
        if Sign($H$) = Sign($RootOfTrust$) then
            $w_j$ ByzantineBroadcast($accept$);
        else
            $w_j$ ByzantineBroadcast($reject$);
        end
    end
    $consensus \leftarrow 0$;
    for $w_j \in W$ do
        All Receive($bit$) from $w_j$;
        if $bit = accept$ then
            $consensus$++;
        end
    end
    if $consensus > f$ then
        break;
    end
end
for $r_i \in R$ do
    for $j \leftarrow 0 \ldots (E-1)$ 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

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. □

4.2.1.2 Switch Stage

After tables are distributed to the end systems, initiators send them to the switches. Because switches exist as part of only one plane, the Switch Stage requires a different distribution mechanism than the End System Stage. The first thing to consider is that any computing capability a switch can offer is limited, so it is advantageous for the end systems to handle decision-making. Second, frames transmitted to the switch cannot be duplicated on independent channels, so faulty switches can now block communication paths.

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.

```
for $r_i \in R$ do
    for $j \leftarrow E$ ... $(E + S - 1)$ do
        $r_i$ Send($T[j]$) to switch $j$;
    end
    for $w_j \in W$ do
        for $u \in U$ do
            $w_j$ Send(REQUEST) to switch $u$;
            $h \leftarrow$ hash from switch $u$;
            if $h = H[u]$ then
                Broadcast(accept);
            else
                Broadcast(reject);
            end
            $consensus \leftarrow 0$;
            for $w_k \in W$ 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

First, an initiator sends each table to the corresponding switch. The witnesses then request a hash of the table from each switch, and broadcast a bit indicating whether the received hash matches the Hash Set. The ability to broadcast the hash of a loaded table is a standard function of some real-time Ethernet switches [27]. If more witnesses accept the table than reject it, 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, it is assumed that 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 $\geq 2f + 1$ paths from each switch to witnesses. 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 where necessary.

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 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. □

Though not desirable, isolated switches are tolerated by both the system and GATEKEEPER. Both protocol and network architecture operate on the worst-case assumption that a plane may be completely incapable of sending data. Even if a switch cannot be reconfigured over-wire, the resulting fault cannot propagate out of a plane, and the system degrades gracefully. Thus, in situations where path thresholds absolutely cannot be met, GATEKEEPER can still perform the reconfiguration properly with just those devices which are reachable.

4.2.2 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 Section 4.1.1), 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 accomplish this goal, the Conversion

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

Algorithm 3: Conversion Phase

Phase is split 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.

4.2.2.1 Bootstrap Stage

The Bootstrap Stage is shown in Algorithm 3, which 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. 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, the `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 reliable broadcast protocol, as in Algorithm 1.

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. □

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. An example of such a defect would be the constraints for sending and receiving messages may not be consistent, and thus misalignment of transmission windows occurs as the new configuration deploys. Per the 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.

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. The only case which cannot support recovery occurs when the reconfiguration order puts all non-faulty planes first, leaving only faulty planes after the bootstrap stage. It is possible to introduce another plane to guarantee recovery by adding another plane, but due to the associated hardware costs, this is not considered a good tradeoff.

4.2.2.2 Resolution Stage

At the conclusion of the Bootstrap Stage, one non-faulty 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, again 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 [19, 18]. Any of these could be used in Gatekeeper.

The end systems can also be treated as groups and migrated piecemeal, similar to the switches. In cases where network availability is less important than protecting critical end systems from stepping into defective configurations, a subset of less key end systems can be tested to ensure that end-to-end paths of the same configuration are function. This is only particularly useful when there is little confidence that the root of trust will only sign and upload trustworthy configurations.

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.

Finally, theorem 1 demonstrates that at this termination GATEKEEPER has successfully reconfigured all non-faulty devices.

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. $\square$

4.3 Evaluation on a Time-Triggered System

To evaluate GATEKEEPER, a prototype was implemented for a TTEthernet TTE system. The prototype takes the form of a controller, written in C code, running on each end system as a bystander, witness, initiator, or a combined witness-initiator. This is shown in Figure 3. The controller program uses several common Linux utilities:

- **sha256sum** for generating the SHA-256 hashes needed for consensus
- **OpenSSL** for using the RSA-2048 signature from the root of trust
- **TFTP** used by TTEthernet for pushing configuration tables to switches
- **SNMP** used by TTEthernet for polling for switch hashes & reconfiguration commands

The system is composed of four TTTech A664 Pegasus XMC end systems attached to quad-core Intel i3-450 host processors via PCIe, and three TTTech 24 port Space ASIC Lab switches. The network is configured with one switch per plane using 100 Mbps connections.

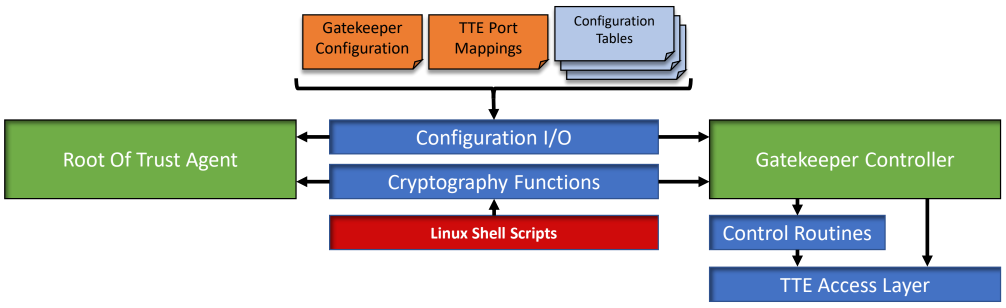


Figure 3: Software architecture for TTE prototype implementation of GATEKEEPER

Because the end system cards do not have a standard Ethernet interface, the TFTP and SNMP functions were run through the best effort ports on the end system hosts and a separate switch to the reach the TTE switches. Testing any faults arising from the specifics of traditional Ethernet traffic is not critical to characterizing the behavior of the protocol, saving significant time by not rewriting the utilities to conform to a new interface.

Although in a field-ready implementation, switch `LOCK` and `CONVERT` functions should be able to handle redundant commands to guard against a faulty "babbling-idiot" node, this functionality is not tested on the test system. Testing this fault mode is not critical to characterizing performance as the work around shown in Figure 4 achieves similar performance and implementing necessary changes would require rewrites to switch firmware. For this evaluation it assumed this specific fault never arises.

The group test for a TTE system is implemented by maintaining two RC VLs across the life of the system. To create a testable end-to-end path without interrupting the standing configuration, the end systems contain the definition for both VLs at all times, while the switches only ever contain one. On the test system, this can be done after the system is scheduled by striping the appropriate definition from only the switch schedules before building the configuration tables. The VLs held by the deployed configuration can be driven any time by the end systems *in the original configuration*, but can only reach their destination when a non-faulty plane is reconfigured to have the missing portion of the path.

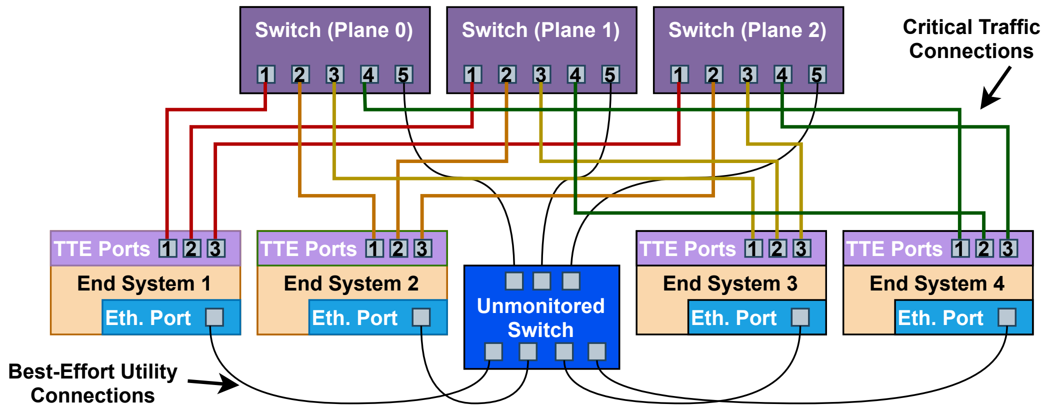


Figure 4: Evaluation testbed for GATEKEEPER

On this set up, three operating modes of GATEKEEPER were evaluated directly: Operating in the absence of faults, referred to as GK_c, operating in the presence of a worst-case end-system fault, GK_{es}, and operating in the presence of a worst-case switch fault, GK_{sw}. A worst-case end-system fault is a device acting as both a witness and initiator, which has also been assigned the *first* turn in order, acting as either a babbling idiot (worst communication cost), or fully omissive (worst time cost). A worst-case switch fault occurs when a switch in the plane first in line for bootstrapping becomes fully omissive.

As a ground truth for this system, a simple reconfiguration protocol called FastRec is used. FastRec uses a single initiator to distribute each configuration and then reconfigure them, first with switches, then end systems. Some variants of FastRec are also considered, which account for some naive optimizations and provide some additional performance context for GATEKEEPER. The FastRec group does not consider fault tolerance and so would break under the conditions of GK_{es} or GK_{sw}. They only act as the most efficient reconfiguration method for the testbed system.

4.4 Calculating Scalability

It is crucial to characterize performance of the system at larger scales. Faults are far more likely in networks composed of dozens of devices and the reconfiguration period for such systems is significant. However, limited hardware supply makes implementing larger systems infeasible for this thesis. Luckily, GATEKEEPER's algorithms are highly deterministic and were very consistent in operating cost for the configurations tested. This allows for the construction of equations that can provide accurate scaling characteristics. Cost takes the form of either communication overhead or execution time overhead, so equations were generated for both. Below are the equations used for every *GK* and FastRec variation presented in experimental results. Note these are specific to a single-fault-tolerant system.

The equations for communication overhead (number of exchanged messages) can be devised by summing every interaction in terms of its message size and the number of participants. This can be repeated again, weighing each term against the size of its messages. The total interactions can be known, given the number of end systems (E), switches (S), witnesses (W), and initiators (R) as well as the number of messages needed to drive a group test (G), yielding in the most condensed form:

$$FastRec = 2(E + S) \tag{4-1}$$

$$FastRecRep = 4(E + S) \tag{4-2}$$

$$GK_c = R * E + S + (4S + E + 1)W + G + 2 \tag{4-3}$$

$$GK_{es} = R * E + 2S + (7S + E + 2)W + G + 4 \tag{4-4}$$

$$GK_{sw} = R(E + 1) + S + (4S + 3R + E)W + 2G + 1 \tag{4-5}$$

$$GK_{comOp} = E + S + (4S + E + 1)W + G + 2 \tag{4-6}$$

The equations for execution time can be found by considering the major stages of the protocol and placing the divisions over each component of logic that may be cyclic. These are consensus (C), distribution (D) for both end systems and switches, bootstrapping (B), the group test (G), and reconfiguration (K). In the case of time, timeouts for silent actors must also be accounted for, denoted as (TO). This yields:

$$FastRecNaive = (D_{SW} + K_{SW})S + (D_{ES} + K_{ES})E \tag{4-7}$$

$$FastRecSmart = (D_{SW} + K_{SW})S + (D_{ES} + K_{ES}*)E \tag{4-8}$$

$$GK_c = C + D_{ES} * E + D_{SW} * S + B + K_{ES} + 2K_{SW} + G \tag{4-9}$$

$$GK_{es} = C_{TO} + C + D_{ES_{TO}} + D_{SW_{TO}} + D_{SW} + B_{TO} + G + K_{ES} + 2K_{SW} \tag{4-10}$$

$$GK_{sw} = C + D_{ES}E + D_{SW}(S-1) + 2D_{SW_{TO}} + B_{TO} + B + 2G + K_{ES} + K_{SW} \tag{4-11}$$

5.0 Experimental Results

In this section, Gatekeeper is evaluated to address three metrics 1) communication cost as the system scales, 2) runtime cost as the system scales, and 3) stability of network availability in the presence of faults. The primary varible manipulated is number of end systems. Though the effects of the number of switches are also noted, they will always tend to be the smaller group due to their high cost to SWaP-C considerations. Setups with large switch over end system ratios are rare. The scaling results were calculated from the equations presented in Section 4.4 and the cost values observed from the testbed (shown in Tables 1 and 2).

Table 1: Bytes of payload data used in each message of Gatekeeper

Size of Gatekeeper Messages	
Message/File	Size [Bytes]
End System Table	1938 (Avg.)
Switch Table	4508
One Vote	2
One Cmd	2
SHA-256 Hash	32
RSA-2048 Sig.	256
Signature File	350
Hash Vector File	715
SNMP Get	96+129 (Call+Resp.)
SNMP Set	98+101 (Call+Resp.)

In defining these cost values, the following considerations were made. In the prototype, votes and commands, like `CONVERT` and `LOCK`, are composed of a tag and value contained in two bytes. The end-system table sizes are tied to the number of VLs defined for each, so, for these calculations, the average size is used. The switches require two tables to operate, and the value here is the sum of both. SNMP interactions include a call and response frame conjoined into one operation.

Table 2: Time required to perform each operation of GATEKEEPER

Execution Time of GATEKEEPER Operations [s]		
Action	Time (Measured)	Timeout
Round of Consensus	0.492	2.0
ES Table Distribution	0.023 / Table	2.0
SW Table Distribution	0.483 / Table	5.0
SW Table Validation	0.068 / Table	5.0
Bootstrap Round	0.88	2.0
Group Test	1.010	2.0
ES Reconfiguration	9.070 / Device	–
SW Reconfiguration	0.293 / Device	5.0

To keep all end systems synchronized, every operation needs a timeout to be assigned. Operations of the GATEKEEPER controller were assigned a timeout of 2.0 s while timeouts of the Linux utilities are 5.0 s. The larger timeout is used because it is the default for most of the underlying utilities.

5.1 Communication Overhead

In investigating communication overhead, both the number of messages required and the total bytes transmitted were evaluated. The ground truths of this evaluation are FastRec and FastRecRep. FastRec shows the most efficient way a TTE configuration can be deployed, while FastRecRep shows the naive overhead of duplicating the whole process from a second initiator. The relative overhead is always considered with respect to FastRec.

First, considering the overhead in terms of number of messages generated, the test system implementation of GATEKEEPER required 114% more messages than FastRec under GK_c, 157% under GK_{sw} and 221% under GK_{es}. This is to be expected since FastRec generates only table distribution and reconfiguration messages. As the size of the system grows, the clear trend of overhead decays towards a limit determined by the number of repeat operations necessary to overcome a fault. This as shown in Figure 5.

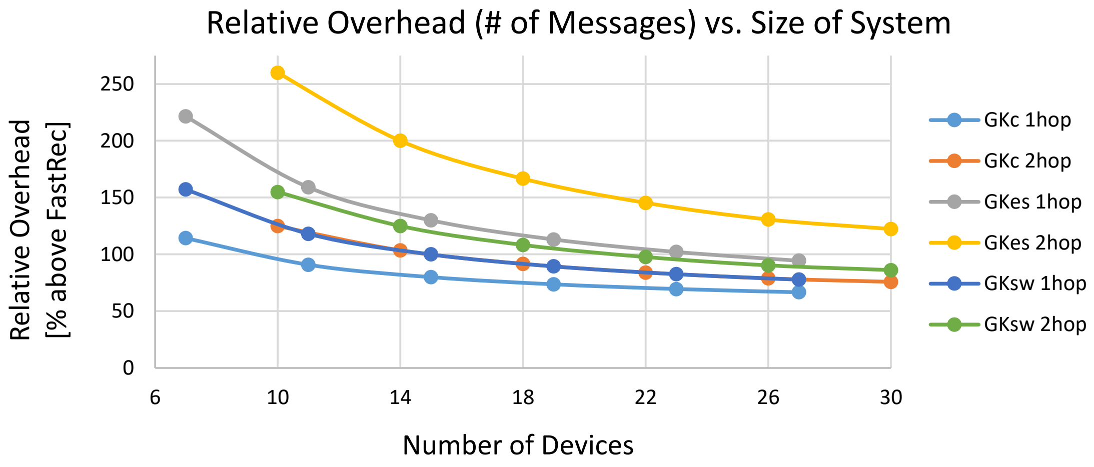


Figure 5: Relative message cost of GATEKEEPER compared to FastRec

Next, total bytes transmitted were evaluated. This properly weighs the impact of each message. The number of bytes for each message was measured from the egress port of each end system, which makes broadcast and unicast messages equivalently costly for these calculations. As shown in Figure 6, the overhead takes two major components. The first is control cost, composed of voting, commanding, synchronization, consensus-holding, and SNMP operations. The second is configuration cost, composed of the configuration and any redundant copies of tables used in distribution. The test system overhead for GK_c is 57%, while GK_{sw} generates a total of 80% overhead and GK_{es} generates 122%. All runs of prototype GK generate some redundancy cost, though only a worst-case end-system fault drives more overhead than FastRecRep.

Scaling the relative data cost by increasing the number of end systems is depicted in Figure 7. GK_{sw} mirrors the trend from Figure 5 and approaches a limit of 98%. Meanwhile, GK_{es} approaches 90% overhead and GK_c approaches 83%. In all cases, daisy-chaining a second switch in the network ('2hop') increased the overhead. Figure 8, shows the total bytes transmitted as the systems scale. As seen, all protocols scale roughly linearly with more end systems.

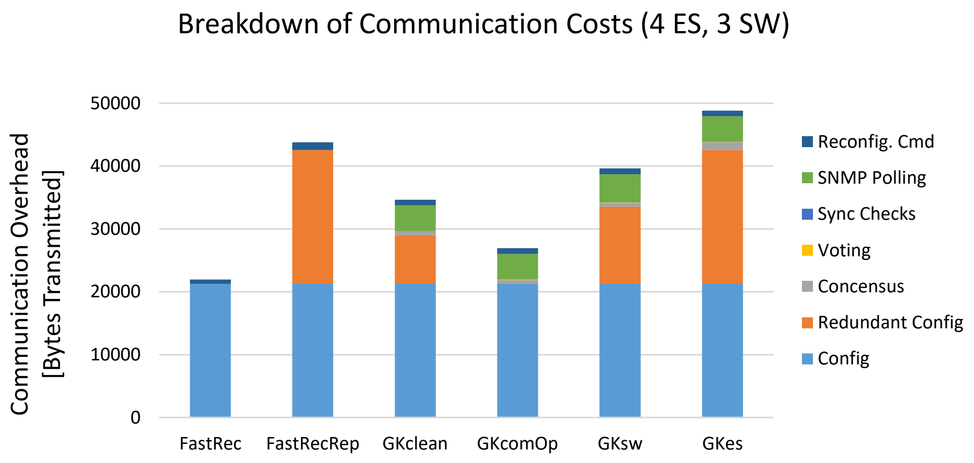


Figure 6: Relative data cost of each information component

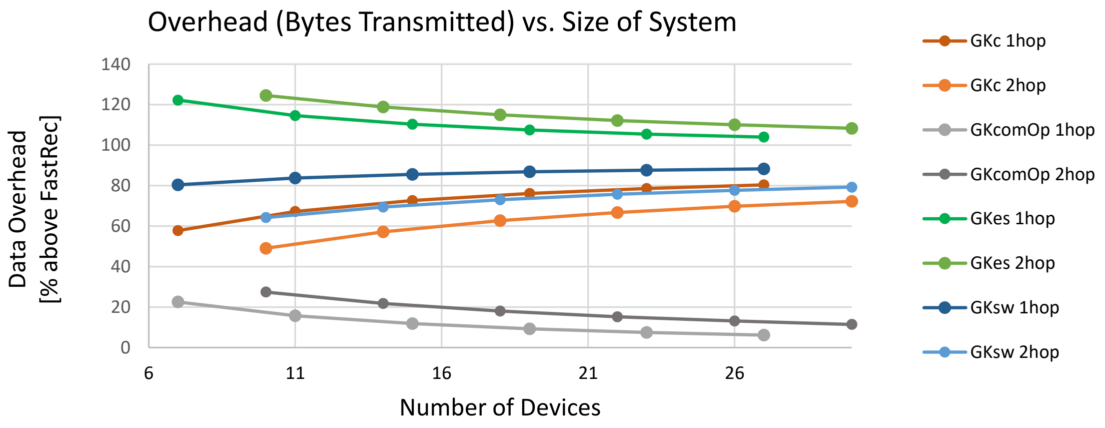


Figure 7: Aysmptotic behavior in terms of relative data overhead vs. increasing the number of end systems

In varying both the number of switches and end systems, as depicted in Figure 9, the convergence of each mode to a limit continues, but now with a "sawtooth" pattern caused by each increase in switches. For GK_c and GK_{sw}, additional switches drive down the overhead since the ratio of end systems to switches is decreased. These trends will converge together

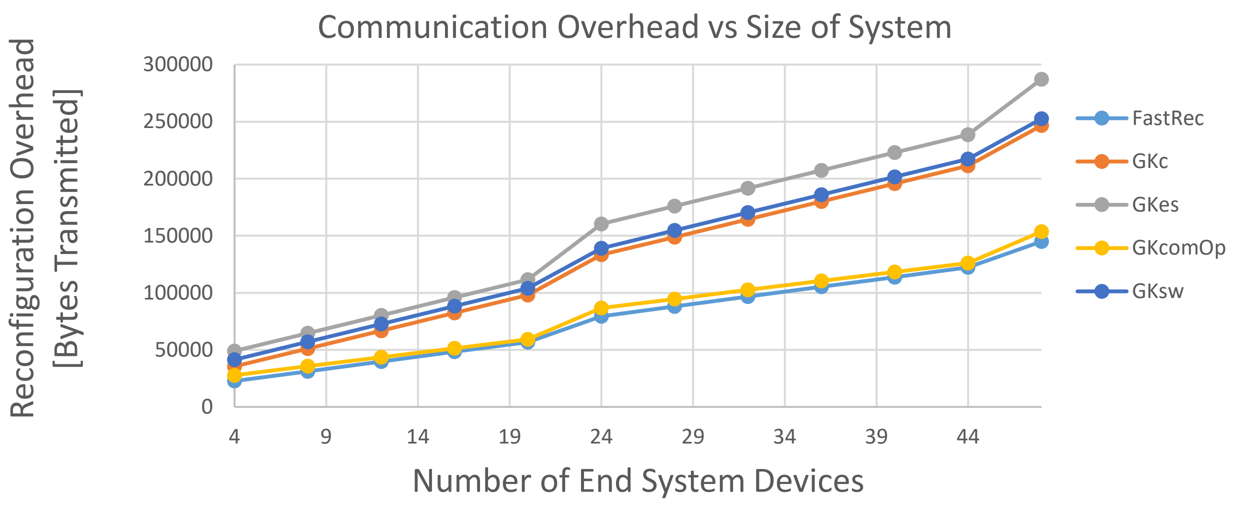


Figure 8: Communication overhead for the reconfiguration of systems containing 4 to 48 end systems

as the overhead of a faulty switch diminishes at scale. For GK_{es}, the overhead increases with each switch since the total data to transmit goes up.

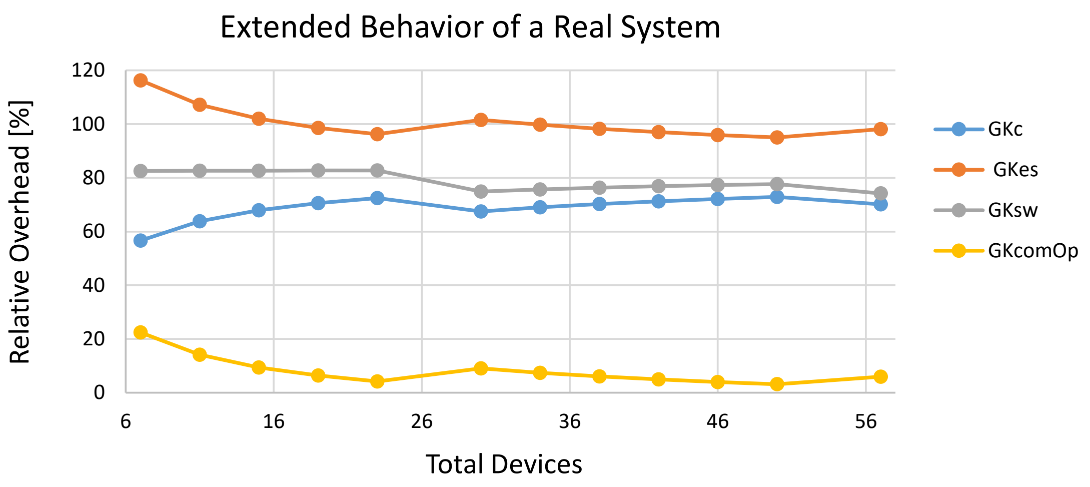


Figure 9: The converging sawtooth pattern of scaling both switches and end systems

The trends for GK_{comOp} are present throughout the communication analysis. This is a variation of Gatekeeper created by only using one end system at a time to distribute end-system tables. In the presence of faults, valid end systems can poll other initiators for more copies of tables. With no faults, this option leaves no configuration redundancy but forces costs of controller complexity and runtime. This tradeoff means GK_{comOp} requires 35% less overhead on the test system (Figure 6), scales *down* as the system grows (Figure 7), and can reach as low as 3% overhead in a large system (Figure 9).

5.2 Runtime Overhead

The runtime overhead was evaluated to ensure GATEKEEPER exhibits acceptable performance at scale. Measurements for this metric are in milliseconds, as that is the scale at which the execution time of GATEKEEPER become highly time-stable. The ground truths for this section are FastRecSmart and FastRecNaive. FastRecNaive distributes tables to and then reconfigures each device *one at a time*, much as a technician manually working on the system would. FastRecSmart does not wait for the result of the reconfiguration command, overlapping most of the time cost. FastRecSmart does have to pause between switch and end-system reconfiguration, because a switch cannot pass on data while it reconfigures itself.

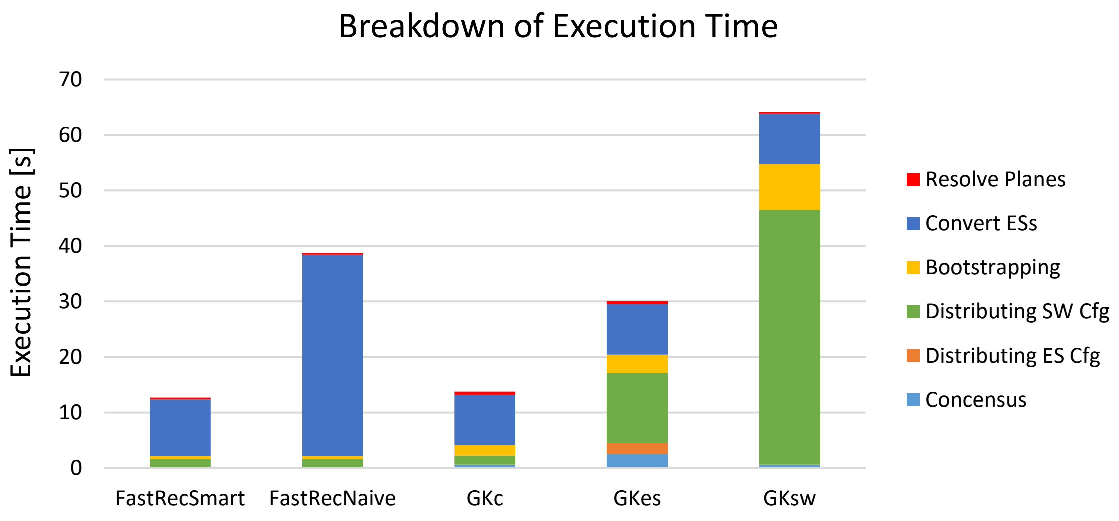


Figure 10: Runtime overhead of each operation in GATEKEEPER and FastRec

As seen in Figure 10, FastRecSmart is the fastest mechanism for the testbed at 12.662 s. GK_c lags behind at 13.775 s. Though GK_{es} experiences timeouts for every control action (totalling 30.106 s), it is still faster than FastRecNaive's approach (38.708 s). Because it triggers the most high-cost utility timeouts, GK_{sw} has the worst overhead at 64.116 s.

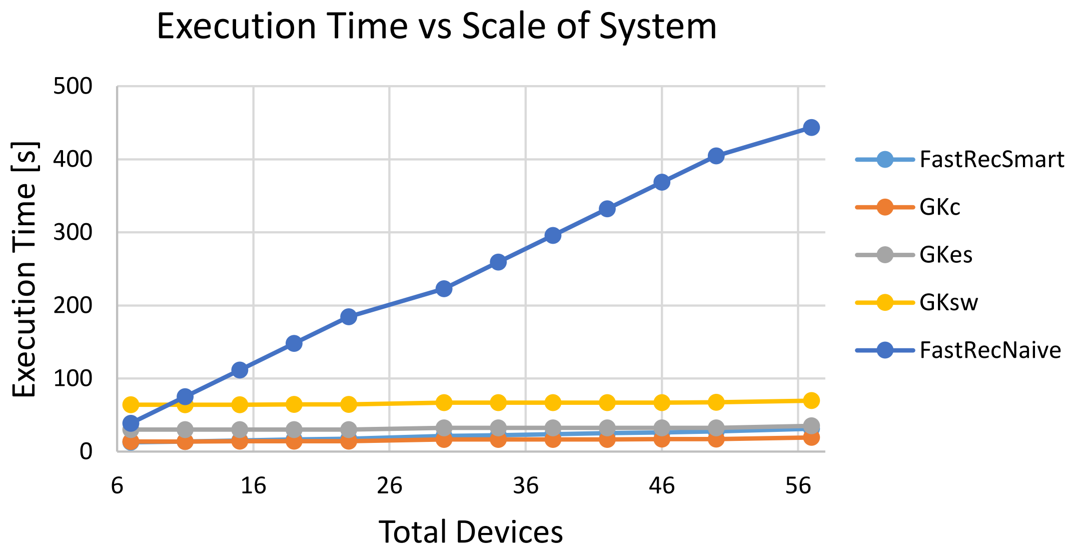


Figure 11: Runtime scaling for protocols vs size of system

As shown in Figure 11, FastRecNaive immediately falls behind the other protocols at scale, as reconfiguring each device in isolation is the least efficient way to perform the operation. Figure 12 more closely shows GK and FastRecSmart. GATEKEEPER takes on a tiered pattern as the system grows. While end systems require 0.092 s per new device, switches are introduced three at a time, requiring 2.5 s more for each new hop. FastRecSmart requires 0.283 s per new device giving it a more linear cost.

5.3 Network Availability

For this evaluation, an available network is defined as all end systems being operational with at least one valid path between all points in the system. Downtime exists between the last moment that is true for the standing configuration and the first moment it is true for

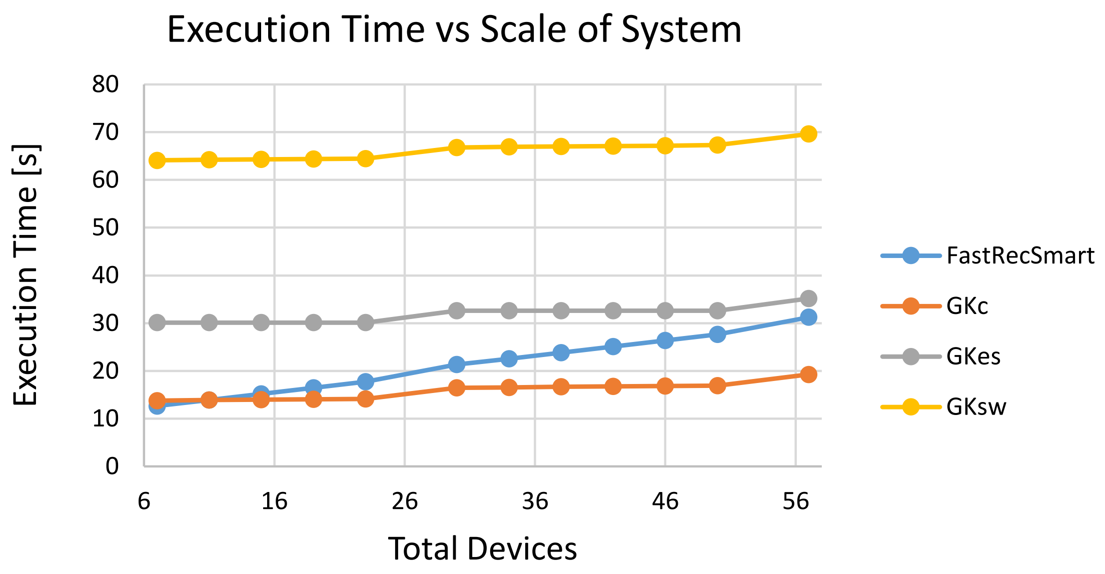
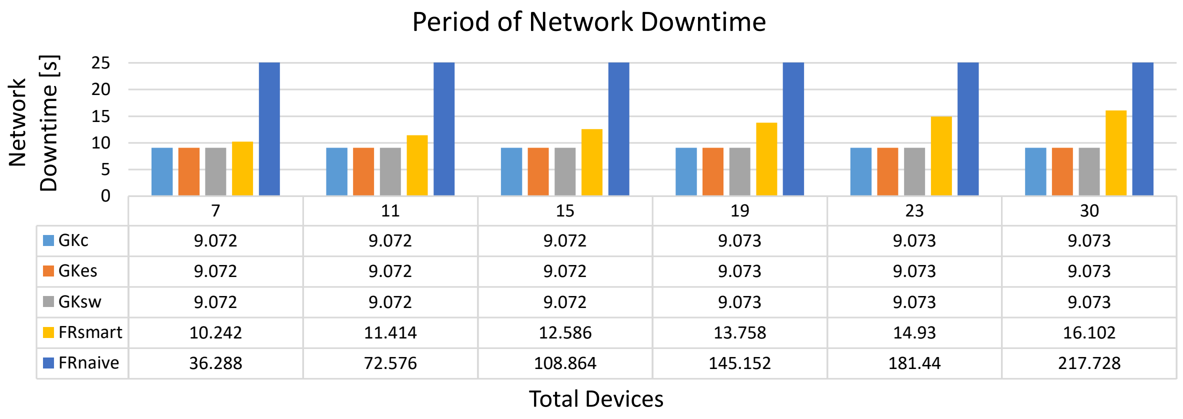


Figure 12: Runtime overhead focusing on only on the competitive portion of the data

deployed configuration. For GATEKEEPER, this is measured from the moment the system leaves the sync barrier preceding the end system reconfiguration logic until the last end system exits the logic. FastRecSmart and FastRecNaive are again ground truths, and their downtime is measured from the moment that the last switch is reconfigured, until the last end system completes reconfiguration.

For the testbed system, the minimum time an end system can spend in the reconfiguration period is 9.070 s. This is composed of three time segments: two 0.5 s pauses inserted to flush the network of in-flight frames, a vendor recommended 5.0 s delay to allow the system to stabilize, and the actual time the card takes to reconfigure, 3.07 s. Though this procedure could be optimized further, this method is applied uniformly between GATEKEEPER and FastRec.

Regardless of fault placement, GK consistently maximizes availability, requiring 9.072 s to reconfigure. The 0.002 s arises from a rudimentary synchronization mechanism and can be reduced. Other than expecting this jitter to increase with more end systems joining the synchronization, 9.072 s remains the downtime for the system at any scale. FastRecSmart



	7	11	15	19	23	30
■ GKc	9.072	9.072	9.072	9.073	9.073	9.073
■ GKes	9.072	9.072	9.072	9.073	9.073	9.073
■ GKsw	9.072	9.072	9.072	9.073	9.073	9.073
■ FRsmart	10.242	11.414	12.586	13.758	14.93	16.102
■ FRnaive	36.288	72.576	108.864	145.152	181.44	217.728

Figure 13: Network downtime for Gatekeeper and FastRec

increases its overhead by about 0.28 s per new end system, and FastRecNaive requires an additional 9.07 s per device.

Network availability is where Gatekeeper excels. This is expected as the witnesses coordinate a moment at which all end systems reconfigure together and the system has established a plane they can immediately begin synchronizing with. With a quality synchronization, the network downtime is always minimal, regardless of fault placement or number of devices. Even though FastRec is efficient at deploying the configuration, it carries with it some small linear availability overhead mainly from the cost to issue each command.

6.0 Discussion

As shown in the results, Gatekeeper achieves its goal of maximizing network availability. While using simple deployment scripts like FastRec may have faster overall performance, especially at small scale, they always carry some overhead from either causing a configuration mismatch between switches and end systems, serializing end-system reconfiguration, or both. The minimal possible downtime is the period of time it takes one end system to reconfigure and a proper implementation of Gatekeeper will always set up this minimum window, regardless of potential fault.

The cost to achieve availability in a fault-tolerant way is acceptable under the Gatekeeper scheme. Because all relative trendlines approached some asymptotic limit in the evaluation, Gatekeeper's communication overhead will always remain near a predictable overhead, usually around 96% more thant the most efficient solution, Further, the overall cost is linear with the size of the system. Additionally, the vast majority of the overhead comes from redundant copies of end-system tables, which can be optimized away if overall runtime is not the priority. The exact overhead of a network using Gatekeeper is dictated by the quantity of redundant configuration data plus the control data over the size of the original configuration.

Switches also have a notable impact on the communication overhead, tending to be the more expensive device. In terms of communication, changing the number of switches shift the *relative* overhead curve, always opposite the asymptote approached at scale. In real systems, designers will minimize the number of switches to save on size, weight, power, and cost. This minimization may cause the sawtooth pattern to emerge, with switches driving the relative overhead in one direction, usually down, and end systems driving it the other. However, at large scale, adding or removing single devices is less impactful, smoothing the curve.

Runtime overhead also tends to be linear and strongly tied to the number of switches rather than end systems, which makes sense as a group of end systems think for the switch. While slower than FastRec scripts at small scale, the ability to perfectly overlap the end-system overhead enables GATEKEEPER to improve over naive solutions at scale. Given the large reconfiguration time cost for the evaluation test system end systems, this was dramatic in these results. Further, most runtime cost in the presence of faults is owed to timeouts, a parameter which is highly tunable and can improve further than the options selected in this work.

GATEKEEPER can achieve such performance characteristics because its control overhead is tied to the number of faults, not the quantity of devices. This allows a small group of controllers to lead a potentially large group of devices. The use of the Hash Set allows the whole system to reduce overheads to sizes that are quickly reduced to noise compared to the size of the actual configuration. This makes emerging patterns of overhead more expensive for small systems, like the testbed, but helps quickly stabilize to reasonable costs in larger systems.

7.0 Conclusions

While many solutions that seek to improve the reconfiguration process of real-time systems exist, none are able to withstand the interference of faulty hardware. This leaves an important gap in the design of systems which are meant to grow and evolve beyond what can be contained in a single network configuration. Gatekeeper helps address this gap by creating a deployment mechanism which can withstand faults interrupting the distribution of data. Further, it shows how a test-and-migrate reconfiguration strategy is effective at navigating around faults, maximizing availability during the reconfiguration process.

To our knowledge, Gatekeeper represents the first approach to real-time reconfiguration featuring a fault tolerance angle. The protocol is able to both distribute data and perform the reconfiguration in the presence of faults. More powerfully, Gatekeeper is a as much a generally applicable strategy, as a strictly defined protocol, and can be coupled with other state-of-the-art techniques for dataflow and frame consistency and network availability to further boost its capabilities.

The results demonstrate that the techniques used in Gatekeeper scale linearly in both execution time and communication cost. Prototype evaluations showed that fault tolerance can be achieved with just 57% communication overhead, which is superior to naive duplication solutions. Even in the presence of faults, Gatekeeper successfully reconfigured the test network with only 120% more communication than the most efficient solution. In all cases, these overheads were calculated to scale up towards a constant relative overhead, which is dictated by the ratio of total end system data to switch data in the configuration. Gatekeeper can be optimized even further to only require about 3% more communication than communication optimal solutions in systems larger than 24 end systems. This optimal communication solution, and other tweaks like it, provides exciting design tradeoffs which can optimize Gatekeeper's performance at scale.

Gatekeeper is also capable at consistently minimizing network downtime, with or without faults present. This makes it competitive with similar state-of-the-art methods. The prototype was able to consistently come within 2 ms of maximum availability, despite having a simple synchronization method. Two milliseconds is 0.0001% of the end system reconfiguration overhead for the test system evaluated, and this tiny quantity of extra downtime is not expected to appreciably grow in scaled system. Not only does this lead to an interrupt 1000x less than simple solutions, which carried 1170 ms of unnecessary downtime on the evaluated system, but scaling calculations suggest that such tight coordination of all end system reconfiguration can enable GATEKEEPER's total runtime to be faster than even simple solution in large systems.

A useful avenue for expanding on this research is an in depth analysis of efficient designs for the group test. Satisfying the three properties is simple for small systems but naive solutions, like all-to-all broadcasting can exponentially grow reconfiguration costs. Scalable methods which still deliver highly certain outcomes will be key to applying GATEKEEPER to larger systems.

Additionally, more research into performance in the presence of multiple faults would improve upon this work. Hardware design limitations generally support only three planes, but investigating simultaneous faulty end systems would offer insight into best practices for device arrangements and role assignments. Specifically, multiple faulty devices cause more constraints for properly applying path thresholds during the switch stage. Efficiently addressing this design space will be key for large systems to implement GATEKEEPER.

Finally, this research could further be built upon by tying GATEKEEPER together with other reconfiguration techniques. As discussed, GATEKEEPER's conversion method ensures the co-existence of network paths for both old and new configurations. This could simplify considerations for complex, complementary techniques, like those in Section 2, which could be used to significantly reduce the minimum downtime achievable by the hardware. Matching GATEKEEPER to a zero frame-loss reconfiguration method has the potential to create fault-tolerant techniques which are completely invisible to the applications using the network.

Bibliography

[1] International avionics system interoperability standards (iasis). Technical report, Partnership of International Space Station Agencies, 2019.

[2] European Space Agency. Ariane 6. `https://www.esa.int/Enabling_Support/Space_Transportation/Launch_vehicles/Ariane_6`. Accessed: 2021-06-30.

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

[4] E. Blasch, P. Kostek, P. Pačes, and K. Kramer. Summary of avionics technologies. *IEEE Aerospace and Electronic Systems Magazine*, 30(9):6–11, 2015.

[5] M. Gutiérrez, A. Ademaj, W. Steiner, R. Dobrin, and S. Punnekkat. Self-configuration of ieee 802.1 tsn networks. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2017.

[6] 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)*, pages 1–4, 2015.

[7] M. Hermann, T. Pentek, and B. Otto. Design principles for industrie 4.0 scenarios. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 3928–3937, 2016.

[8] Tsn profile for industrial automation. Standard, Time-Sensitive Networking task group, Jul 2020.

[9] Ieee standard for local and metropolitan area networks—audio video bridging (avb) systems. Standard, Institute of Electrical and Electronics Engineers (IEEE) Audio Video Bridging task group, 2011.

[10] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 49–54, New York, NY, USA, 2013. Association for Computing Machinery.

[11] 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)*, pages 22–33, 2005.

[12] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[13] 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*, 27(3):1124–1137, 2019.

[14] Andrew Loveless. On time-triggered ethernet in nasa's lunar gateway. https://ntrs.nasa.gov/api/citations/20205005104/downloads/2020-07-26-AA-CoP.pdf, 2020.

[15] Jun Lu, Huagang Xiong, Feng He, Zhong Zheng, and Haoruo Li. A mixed-critical consistent update algorithm in software defined time-triggered ethernet using time window. *IEEE Access*, PP:1–1, 04 2020.

[16] T. Mizrahi and Y. Moses. Time4: Time for sdn. *IEEE Transactions on Network and Service Management*, 13(3):433–446, 2016.

[17] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates in software-defined networks. *IEEE/ACM Transactions on Networking*, 24(6):3412–3425, 2016.

[18] Tal Mizrahi, Efi Saat, and Yoram Moses. Timed consistent network updates. SOSR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[19] 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*, 17(3):1668–1678, 2021.

[20] PROFIBUS & PROFINET International. *PROFINET*.

[21] M. L. Raagaard, P. Pop, M. Gutiérrez, and W. Steiner. Runtime reconfiguration of time-sensitive networking (tsn) schedules for fog computing. In *2017 IEEE Fog World Congress (FWC)*, pages 1–6, 2017.

[22] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. SIGCOMM '12, page 323–334, New York, NY, USA, 2012. Association for Computing Machinery.

[23] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, New York, NY, USA, 2011. Association for Computing Machinery.

[24] Time-triggered ethernet. Standard, SAE Aerospace, Aug 2011.

[25] 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*, 12(2):263–284, 2010.

[26] W. Steiner. Ttethernet: Time-triggered services for ethernet networks. In *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, pages 1.B.4–1–1.B.4–1, 2009.

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

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

[29] 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)*, pages 1–4, 2015.

[30] 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)*, pages 4746–4753, 2020.