An Intrusion Tolerant Architecture and Protocol for Substation Protection

Daniel Qian

A project report submitted to the Johns Hopkins University
in conformity with the requirements for the degree
of Master of Science in Engineering

Advisor: Yair Amir

August 2021

Acknowledgements

First of all, I would like to thank my advisor Yair Amir, who was a constant presence throughout my time at Johns Hopkins. He was my professor for Intermediate Programming, my very first computer science class at Hopkins, as well as Software For Resilient Communities, where I had my first experience working with intrusion tolerant systems. Later, I also took Distributed Systems and Advanced Distributed Systems with him and worked as a Course Assistant for some of his classes. Outside of Hopkins, I was fortunate enough to work with him one summer at his startup, LTN Global Communications Inc., on some other interesting problems involving real time networking protocols over cellular networks. However, I would especially like to thank Yair for his general mentorship; he inspired me to believe in my own ability to make an impact.

I would like to thank Sahiti Bommareddy, who I have worked with for the past year and a half and who made numerous contributions to this work. She is a model collaborator, always available to meet and sometimes even picking up tasks I could not get to. I would also like to thank her for taking care of and upgrading the lab equipment, which I know she spent many hours on.

I would like to thank the other members of the DSN Lab, both past and present, especially Amy Babay, Emily Wagner, John Schultz, Brian Wheatman, and Jerry Chen. The DSN Lab community has always been super welcoming, whether that meant mentoring me on projects, having productive discussions in lab meetings, or giving me advice based on their own experience.

Finally, I would like to thank my family. My parents, Jie and Ying, brought me into this world and raised me, and have always given me unconditional love and support. My older brother Justin has always been my role model, and my younger brothers Gavin and Oliver always bring me joy.

This work was supported in part by the Department of Energy / Pacific Northwest National Laboratory grant PNNL-SA-163885. Its contents are solely the responsibility of the authors and do not represent the official view of DoE or PNNL.

Contents

Abstract

While more recent works on intrusion tolerant replication for power grid SCADA systems have focused on wide area control centers, there has also been recent interest in applying similar techniques to individual electrical substations. However, the substation environment is fundamentally different from the wide area network, in both requirements and properties. For example, the IEC 61850 standard for substations requires a 4 millisecond latency for operations by high voltage protective relays. For BFT-based algorithms, meeting this requirement under all conditions, including in the presence of sophisticated attacks, is a steep challenge. More importantly, the state of the system depends entirely on the most recent operation, which means that the total ordering provided by such protocols is unnecessary. Therefore, we propose a new architecture and accompanying Byzantine protocol. The architecture consists of $2f + k + 1$ replicated relays, where f is the maximum number of simultaneous Byzantine faults allowed, and k is the maximum number of relays undergoing proactive recovery. This is less than the $3f + 2k + 1$ replicas that a traditional BFT protocol would need, which implies significant cost savings and makes our architecture much more feasible for deployment. Then, the protocol guarantees that if the relays behave similarly (because they observe the same physical reality), the system will issue events correctly and in a timely manner. Our implementation of this protocol was tested both with and without the presence of Byzantine failures, and in all cases was able to deliver messages faster than the 4 ms requirement.

Introduction

Critical infrastructure, such as power grids, is generally controlled by Supervisory Control And Data Acquisition (SCADA) systems, which allow operators to monitor and manipulate parts of the system remotely. However, as these systems move away from traditional, air-gapped networks and increasingly use existing IP infrastructure, they become more vulnerable to cyberattacks and intrusion from the outside world. Recent events have shown that this vulnerability can be exploited to cause disruptions and even physical damage to critical infrastructure [8].

One approach to solving these issues is presented in Spire [4]. Spire is designed for the wide area context, specifically regional power grids that have a control center and units in the field that need to communicate. Network attacks are addressed by using the Spines overlay messaging framework [3], which comes with an intrusion tolerant mode that provides authentication and enforces fairness [9]. For system level attacks, the Prime Byzantine replication engine [2] is used so that attackers would need to compromise multiple machines in order to affect the system.

One important note about replication is that in practice, the replicas should not be completely identical. Otherwise, an exploit that works on one replica could be repeated to compromise the other replicas in the same way, and attackers could easily gain access to a critical number of machines (i.e. more than f machines at once, where f is the number of assumed Byzantine machines). Spire proposes a number of ways to diversify replicas that are also applicable to our protocol, such as using compilers with built-in randomness.

Finally, the idea of proactive recovery, or periodically rebooting the replicas from a clean state with fresh randomness, is proposed to increase the long term resilience of the system. Essentially, even if an attacker is able to gain access to a replica, the replica would eventually be rebooted from read-only memory in order to flush out the attacker from that replica. Then, because fresh randomness is used to diversify the replica, the attacker cannot use the same exploit to access the machine. These proactive recoveries should be frequent enough that attackers should not be able to compromise more than f machines at once. We accommodate for this idea of proactive recovery in our protocol as well.

However, instead of entire power grids, our focus is on adding resilience to individual electrical substations. Substations are an essential part of power grids that help transmit and distribute electricity. As expected though, the same vulnerabilities that exist in wide area SCADA systems may also exist at the local level of SCADA for substations. For example, *Intelligent Electrical Devices* (IEDs), essentially electrical equipment with a general purpose processor, could be an attractive target for attackers. This is because they perform a variety of important functions, including gathering sensor data, issuing automated commands, and communicating with higher level control systems. Therefore, these IEDs would benefit from the same kind of techniques that Spire uses to address both network and system level attacks.

The substation environment is different than wide area SCADA environment though, and it comes with its own challenges. First of all, IEDs are not generic computers; they are expensive, specialized hardware, and an excessive number of replicated IEDs would make deployment infeasible. Furthermore, the *IEC 61850* standard [6, 7], which defines both the architecture and communications of substation systems, has a strict timing constraints for critical messages. For BFT-based algorithms, meeting these timing constraints under all conditions (including in the presence of resource consumption or other sophisticated attacks) is a steep challenge.

One operation with such a timing requirement is that of *high voltage protective relays*. They are a particular type of IED that run algorithms based on measurements of current, voltage, etc. to detect electrical faults. If a fault is detected, they protect the physical equipment of the substation by tripping a circuit breaker. Then, when the fault is cleared, they can automatically close the circuit breaker. It is clear why protecting these relays is important; unnecessary trips or failure to close the circuit breakers could result in loss of power for customers. Even worse, failure to trip and closing at the wrong time could result in physical damage to the equipment or injury to operators. These signals to trip and close would be carried by an IEC 61850 Generic Object Oriented Substation Events (GOOSE) message and are required to be delivered from the relay to the breaker within 4 ms[1][5, 6].

Our work is designed specifically for these protective relays, but also serves as a proof of concept for similar systems that may have less strict requirements. We use the aforementioned Spines for intrusion tolerant networking, and we accommodate for proactive recovery. In order to protect against intrusions of the relays, we also replicate them. However, instead of a traditional BFT (Byzantine replication) protocol, we use a novel protocol to coordinate them, which is the main contribution of the work.

At a high level, the protocol uses Threshold Cryptography [11] to ensure that compromised relays cannot unilaterally issue trip or close commands. Specifically, if f is the number of compromised/Byzantine relays, then the Threshold Cryptography scheme requires $f + 1$ signatures to

[1]This 4 ms requirement comes from the fact that a trip should occur within a quarter cycle of the power. Since in the US power runs at 60 Hz, a quarter cycle is $1/60/4 \approx 4.167$ ms.

ensure at least one correct relay agrees. If we account for k relays undergoing proactive recovery, then a total of $2f + k + 1$ relays are needed, since we need to ensure that $f + 1$ correct relays are available and actively participating. Note that any BFT protocol normally requires $3f + 2k + 1$ replicas under the same definitions of f and k. Since relays are expensive, this implies significant cost savings and makes the work much more feasible for deployment. In addition, the threshold cryptography requires less rounds of communication and therefore meets the timing requirements of IEC 61850.

The details of the protocol, including why BFT is not needed and how the Threshold Cryptography works, are described in later sections.

2.1 IEC 61850 Substation Architecture

To give context for our intrusion tolerant architecture, we first examine in detail how a substation would be organized under IEC 61850. [6, 7].

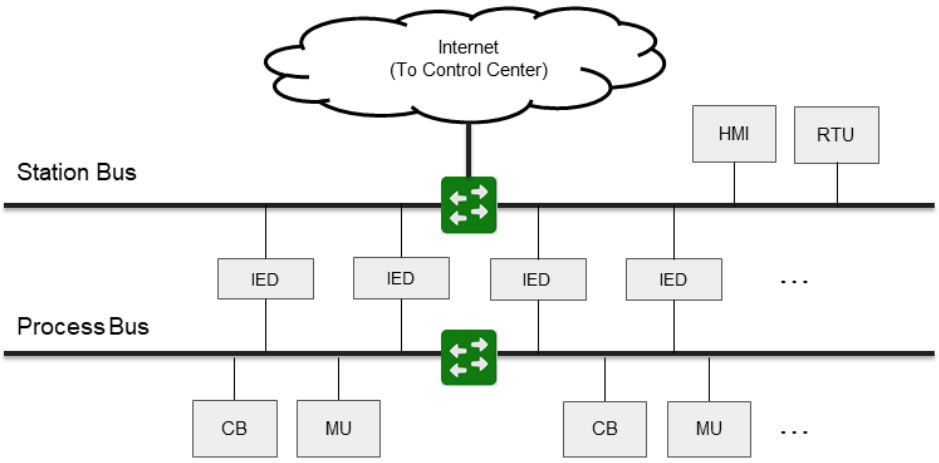

Figure 2.1: Architecture of a substation, with multiple IEDs

Following are details about the relevant components in Figure 2.1.

- **Networks**: The substation communication is split into two high speed Ethernet networks. The *Process Bus* connects the IEDs (protective relays in our case) with the various devices that interact with the physical equipment. The *Station Bus* is used to connect the IEDs to each other, and to higher level control.

- **Sensors/Merging Units**: Traditionally, sensors in the substation would directly connect to IEDs. However, the IEC 61850 standard instead uses *Merging Units* (MU). These MUs gather data from the sensors, convert the analog measurements to digital ones, and publish

them on the Process Bus using *Sampled Value* (SV) messages, which are also defined in the 61850 protocol.

- **Relays/IEDs**: The Relays are the decision makers. They take the SV messages from the sensors and apply algorithms over them to detect electrical faults. When such a fault occurs, they issue a GOOSE message over the Process Bus commanding the breaker to trip. Similarly, they also use the measurements from the SV messages to decide when to issue close commands through GOOSE.

- **Circuit Breakers**: Also connected to the Process Bus are the Circuit Breakers (CB) that would receive GOOSE messages from the relays and physically act on the system.

To summarize how a trip would occur, *Sampled Value* messages containing measurements are published on the Process Bus and received by a relay. If the relay detects a fault using these measurements, then a *GOOSE* message is issued over the Process Bus to a circuit breaker. The circuit breaker would then trip.

In addition to the components just described, the upper section of Figure 2.1 shows an HMI (Human Machine Interface) and an RTU (Remote Terminal Unit) connected to the Station Bus, along with the IEDs. These communicate with each other using another IEC 61850 protocol called *Manufacturing Message Specification* (MMS). The Station bus is also connected to the internet to facilitate communication between the substation and the control center.

Finally, one important detail not reflected in Figure 2.1 is that substations require some sort of time synchronization [10], such as NTP, IRIG-B timecodes or Precision Time Protocol (PTP). Because this synchronization is necessary for the substation anyways, our protocol also utilizes it. Note that while the time synchronization method may be a potential vector for an attacker to disrupt the substation, preventing such attacks is not within the scope of this work.

2.2 IEC 61850 Message Types

Following are additional details about the message types passed between parts of the substations as defined in IEC 61850.

- **Manufacturing Message Specification (MMS)**: These messages are used in the substation context to carry information from higher level control or between IEDs. They are based on TCP/IP. However, as stated before, our protocol does not interact with them.

- **Sampled Value (SV)**: SV Messages are sent by the MUs and carry measurement data, such as voltages and currents, from Merging Units to IEDs/Relays. These messages are published on a fixed interval, for example 80 times per power cycle, and are subscribed to by the IEDs.

These messages are sent using Ethernet multicast, and thus are restricted to Local Area Networks (i.e. the Process Bus).

- **Generic Object Oriented Substation Events (GOOSE)**: GOOSE is used to publish events in the substation, such as the relay deciding to trip or close. Similar to SV messages, GOOSE is also a Layer 2 protocol that uses Ethernet multicast. When new data needs to be published, a message is sent with this new data and a new "state number". Then, it is retransmitted multiple times with the same data and state number. The period of retransmission is short at first, but grows until it reaches a configured period, usually on the order of 1 second, and then continues to retransmit at that frequency. When a new event occurs, the state number is incremented, and the same retransmission sequence is started. Finally, as discussed before, the IEC 61850 standard also requires that GOOSE messages be delivered within 4 ms.

2.3 Intrusion Tolerant Architecture

The following section describes our intrusion tolerant architecture, which is shown in Figure 2.2. Note that while the previous figure (Figure 2.1) of a typical IEC 61850 architecture had multiple IEDs with different purposes, each labeled IED in this diagram is a replica. In other words, the replicas shown are logically equivalent to a single Relay in Figure 2.1. Furthermore, we only show one MU and one CB, as our single logical relay works with them.

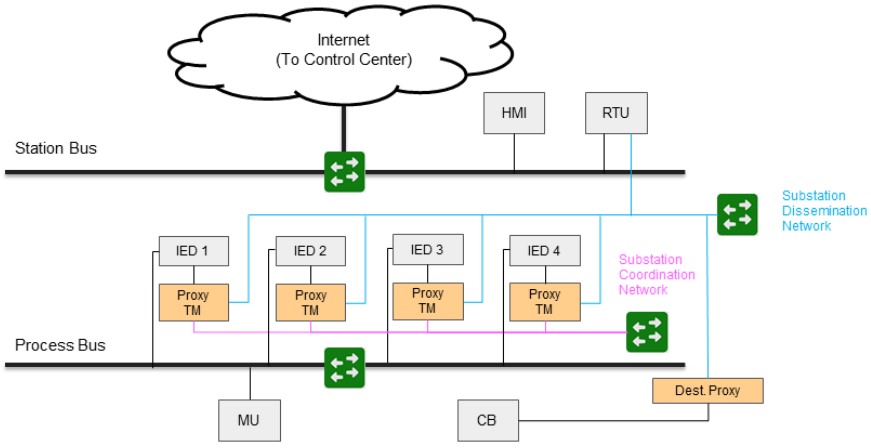

Figure 2.2: Intrusion Tolerant Architecture for a single IED with multiple replicas, $f = 1$ and $k = 1$

There are a number of new components. The outputs of each Relay, rather than being directly connected to the Process Bus, are instead connected to a separate machine. This machine contains two components, a *Relay Proxy* and a *Trip Master* (TM). Since the TM is the more important component, we will refer to this machine as the *TM Machine*. We will also refer to the TM Machine and Relay together as a *Replica*, since for the purpose of our Byzantine protocol, they are a single

unit. Similarly, the *Circuit Breaker* is no longer connected to the Process Bus, and is instead connected to a separate machine, the *Destination Proxy*. Finally, two Spines networks are used to connect these new machines. The purpose of each of these is described below.

- **Relay Proxy**: The Relay Proxy simply receives and parses all GOOSE messages sent by the relay it is connected to, and then it passes the needed information to the TM. The Proxy also reads the state numbers and ignores any retransmitted messages of the GOOSE protocol. The main advantage of using such a Proxy is that Relays do not need to be modified for the protocol, and they can run in the same way as in a normal substation.

- **Trip Master (TM)**: The Trip Master performs the Threshold Cryptography based protocol on the output of the Relay Proxy. The details of the protocol are described in Section 4. However, one important note is that using Threshold Cryptography requires that during configuration, a trusted party would have to distribute signing keys to each TM.

- **Destination Proxy**: The Destination Proxy serves a similar purpose to the Relay Proxy in that it handles the protocols used by the Circuit Breaker (i.e. translating the message sent over Spines from the TM back into GOOSE). However, it also has the additional responsibility of verifying the signature from the Threshold Cryptography scheme. Since the purpose of the machine is rather simple, it should theoretically be more secure than the other components.

- **Networks**: Finally, the last new components are the Spines networks. Instead of sending GOOSE messages directly on the Process Bus using Ethernet Multicast, the TMs and Destination Proxy are connected through one of the Spines overlays, which is called the *Spines Dissemination Network*. In our proposed architecture, this is a separate physical switch from the Process Bus, so the Process Bus would only carry the SV messages.

  There is also a second Spines overlay network: the *Spines Coordination Network*. This network connects the TMs and is used to exchange messages, for example those containing partial signatures, during the protocol. This would also be its own separate physical switch.

Finally, though it is not directly relevant to our protocol, at the top of Figure 2.2, the HMI and RTU are still connected to the station bus. However, we propose that instead of connecting to the IEDs through the station bus, the RTU would also use the Spines Dissemination Network. Then, the TM machine could forward the MMS messages to the IEDs. This would add the Intrusion Tolerant Properties of Spines to the communication between the RTU and the IEDs.

2.4 Spines Details

As discussed, the reason we use Spines is for its Intrusion Tolerant networking mode. To provide more context, there are two Spines nodes (in reality processes) on all of the TM machines. One is

for the Dissemination Network, and the other for the Coordination Network. In addition, there is a Spines node for just the Dissemination Network on the Destination Proxy. Each of these nodes is connected to all other nodes on the same network; in other words each network is a complete graph from the Spines perspective. Spines is designed as an Overlay Network and sending along each edge is reliable.

Intrusion Tolerant Networking mode also ensures that each edge is authenticated, which is the first important property that we use. On startup, the nodes use Diffie-Hellman Key Exchange with every other node to setup a Secret Key, which is then used for HMACs. This ensures that even if an attacker has access to other machines in the substation, they would need to compromise one of the TM machines and its keys to send messages that would be accepted by other Spines nodes. We can therefore trust Spines to inform us of the identity of the senders.

The second important property of Spines is the enforcement of fairness. Essentially, each Spines node allocates resources for receiving messages according to max-min fairness. Therefore, a compromised node cannot deplete another node's resources (at the Spines level) by sending messages excessively.

More specifically, Spines defines two types of messaging semantics: Priority and Reliable messaging, each with their own type of enforced fairness. In our implementation, we use Priority messaging. However, we claim that because our network graph is complete, these two semantics only have minor practical differences. In fact, the fairness enforced by either is equivalent. Priority messaging enforces fairness per source, while Reliable messaging enforces fairness per flow (see [9] for more details). But all possible flows for a destination in such a network graph is the same as all the sources. The main difference is how the two semantics behave when resources are limited. In Priority messaging, upon receiving a message that would fill the buffer for the message's source, the oldest, lowest priority message from that source would be dropped. Reliable messaging obviously should not drop messages, so it will instead refuse to accept new messages when the buffer for the message's flow is full. However, due to the nature of our protocol, we do not expect correct nodes to ever send enough messages to fill these buffers, so this is not a practical difference.

Note that intrusion tolerant mode of Spines also has other features such as multi-path routing, but those are more relevant to wide area networks, where Spines nodes are used to forward messages to each other as part of an Overlay Network over existing IP infrastructure.

System Model

The following section provides definitions and assumptions used for our protocol description and analysis.

3.1 Threshold Cryptography

As stated before, the TMs make use of a threshold signature scheme. At a high level, a (t, n)-threshold signature scheme creates a public key and a signing key, just like a normal signature scheme. However, the signing key is split among n parties, and t of these are needed to create a signature.

Most threshold signature schemes (including the one used in this protocol) use a two step process to create these signatures. First each party can create *partial signatures/shares* on a message. We will denote this as $\langle \ldots \rangle_{\sigma_i}$ where $\ldots$ is the content of the message and i is the index of the party. Then if any distinct set of at least t of these shares exist on the same message, then they can be *combined* to form the *threshold signature*, which we simply denote as $\langle ... \rangle_{\sigma}$. Note that this combination can fail if one of the shares is invalid.

Finally, this threshold signature can be *verified*. For a given the threshold signature, the public key, and the message itself, the verify function will return true if the signature is valid, and false otherwise. Note that this is no different than verifying a normal signature, and in fact in the scheme we use [11], the verification process is the same as a normal RSA signature verification.

It is assumed that the security of the Threshold Cryptography is sound, i.e. it is computationally hard to forge a valid signature over a valid message without a set of t shares.

3.2 Failure Model, Network Model and Other Assumptions

As stated before, the protocol tolerates up to f Byzantine and k unavailable replicas. This applies to both the Relay itself, and the TM Machine attached to it. By a Byzantine fault, we mean that the attacker can execute arbitrary actions on compromised replicas, including coordinating with

other compromised replicas. However, the one exception is that they cannot break the cryptography of any correct replicas, without access to their secret keys.

We assume that the time synchronization will not be compromised. This is justifiable because it could be protected by some other system. We also assume that the time synchronization will be accurate to at least 1 ms, that is the clocks of any two machines will not diverge by more than 1 ms. This is well within the accuracy of protocols such as PTP, which in fact synchronize clocks on the order of microseconds.

We also assume that the communication between the Merging Units and Relays over the Process Bus cannot be interfered with and that the relays therefore receive the same SV messages. This is a reasonable assumption to make, given that the Process Bus could be protected by configuring the switch to only allow incoming packets from the MUs. Note that this also implies that correct relays will see the same input.

Another assumption that we make about correct relays is that they function logically with respect to trips and closes. Specifically, we assume that a correct relay will only issue a trip if power is flowing, i.e. if the breaker is already closed. Likewise a correct relay will only issue a close if there is no power flowing, i.e. if the breaker has been tripped.

Finally, to clarify the network model, it is clear that no protocol can ensure the timeliness requirement of 4 ms in a fully asynchronous network setting. In fact, our protocol requires a stable network that delivers messages within 1 ms. However, we believe this is justifiable, given that the protocol runs over a Local Area Network.

3.3 Other Definitions

First, let us define n as the total number of replicas and enumerate them $1 \dots n$. $n = 2f + k + 1$ where f and k are defined as before. We use a $(f + 1, n)$-threshold signature scheme, with the n shares of the signing key distributed among the n replicas during setup.

We also need to define the two actions of the protective relays: TRIP and CLOSE. Our protocol uses these to describe the statuses of the protective Relays and the Circuit Breaker. Intuitively, when the relays change their status from one to another, our TM protocol needs to issue a Threshold Signed message to the Destination Proxy.

Protocol Description

So far, the TM coordination protocol has been described abstractly; it allows multiple relays to generate a final signature for a trip or close message. Now, we define the state machine which specifies exactly how TMs behaves when they receive a message.

4.1 Key Concepts

Before getting into the formal specification, it is important to understand the concepts behind the design of the state machine.

4.1.1 Discretized Timestamps

Let us consider how combining shares into a Threshold Signature is actually done. Say a large enough set of correct relays that need to sign a message for the breaker exists. If the entire message just consists of a flag for TRIP or CLOSE, then in the future an adversary could replay such a message to change the system unilaterally.

A number of solutions to this were considered, including sequence numbers, nonces, etc. However, with all of these solutions, this extra field in the message would have to be agreed upon by all the TMs, requiring an extra round of communication before the partial signatures are exchanged.

This brings us to our solution of *discretized timestamps* (DTS), which is based on the time synchronization already present in the system. Essentially, the messages also contain a timestamp of when they are generated. However, instead of an exact timestamp, we "discretize" them, or round them to a regular interval. Then, when the TMs send partial signatures to each other, if the TMs are synchronized, they should also have generated a message with the same exact DTS. Therefore, only one round of communication between the machines is needed to sign the message. To make this process faster and more reliable, each TM initially sends 2 shares upon a status change, one with a rounded down DTS and one with the next DTS. Then if the threshold signature is not generated, the TMs continue to send the next DTS when that time is reached, until they can collect enough messages with the same DTS.

How do we choose the interval to round every DTS to? The minimum is the sum of the maximum network time synchronization error and network delay. This is because otherwise, a TM could send the two shares as described, but if they are maximally delayed until they reach another TM, and if that TM is maximally ahead, those shares would look too old. So under our assumptions, the DTS must be at least 2 ms, which is in fact the interval which is used in our implementation. On the other hand, a large DTS means that two operations that are close in time to each other may initially share the same DTS, so then the second operation would need to wait for the next DTS.

4.1.2 Role of the Destination Proxy

The Destination Proxy (connected to the circuit breaker) is a single point of failure, and so it is meant to be as simple as possible to prevent vulnerabilities. However, we must involve it minimally in the protocol to acknowledge that the threshold signed messages are received.

If we were to rely on the TMs to keep track of which threshold signed messages are sent (e.g. by having them multicast the signed message to the other TMs), then we run into the situation where a compromised TM could send the signed message only to the Destination Proxy and not the other TMs. Though this could be solved by ensuring that there is a quorum for each TRIP or CLOSE (like BFT), this would also mean an extra round of communication.

Note that the proxy is designed so it can undergo proactive recovery as well, i.e. shut down and restart from a clean state with minimal interruptions. Furthermore, multiple proxies can be used to ensure if that one is unavailable, the other could still be used.

4.1.3 Startup Protocol

In order to facilitate proactive recovery, we need TMs to have the ability to be shutdown and restart as a clean, newly diversified instance. This means that they would not have any knowledge of their prior state. Therefore, in order to join the protocol, we have them query for the status of the CB and wait for their relay to also issue a status. Because this procedure is independent of the other TM replicas, each TM can simply run the startup protocol separately to start the system from scratch.

4.2 States

The state machine defines a set of three *templates* for each status of TRIP and CLOSE. Note that if there was more than two possible statuses, these templates could be generalized.

Given x is either TRIP or CLOSE:

- **Done** x: The relay has issued an x status, which the TMs generated a threshold signature for and the Destination Proxy received.

- **Attempt** x: The relay has issued an x status, but the Destination Proxy is not up to date with it. Therefore the TMs are currently attempting to generate a threshold signed message.

- **Wait** x: The relay did not issue an x command, but the other TMs have generated a threshold signature that the Destination Proxy received. This indicates the Relay is either down, or behind in issuing the x command.

The flow between states in the normal case is straightforward. If the TM is currently at **Done** x, and receives a new status y from the relay, it will go to **Attempt** y. When a signed message is created or the proxy acknowledges a signed message from another TM, then the TM can go from **Attempt** y to **Done** y. If instead a TM receives an acknowledgement for an x status from the proxy, but its relay has not issued an x status yet, the TM goes to **Wait** x. If the relay then issues the x status late the TM can go to **Done** x.

More formally, each TM keeps track of the following information: the status of the relay and the (discretized) timestamp at which the status last changed, and the status of the breaker and another timestamp indicating the time of the last status change. Note that this second timestamp will update both when a TM sends a threshold signed message and when it receives an acknowledgement from the proxy.

We denote the relay status and timestamp as a object r. We define the breaker status and timestamp as an object b. Each of these has a *status* (TRIP or CLOSE) and *dts* field that we refer to using dot notation (e.g. $r.status$).

The state machine's states are then defined over the possible values of r, b, as follows:

1. **Done TRIP**: $(r.status, b.status) = (\text{TRIP}, \text{TRIP})$

2. **Done CLOSE**: $(r.status, b.status) = (\text{CLOSE}, \text{CLOSE})$

3. **Attempt TRIP**: $(r.status, b.status) = (\text{TRIP}, \text{CLOSE})$ and $r.dts \geq b.dts$.

4. **Attempt CLOSE**: $(r.status, b.status) = (\text{CLOSE}, \text{TRIP})$ and $r.dts \geq b.dts$

5. **Wait TRIP**: $(r.status, b.status) = (\text{CLOSE}, \text{TRIP})$ and $r.dts < b.dts$

6. **Wait CLOSE**: $(r.status, b.status) = (\text{TRIP}, \text{CLOSE})$ and $r.dts < b.dts$

These states cover all possible values of r, b.

4.3 Events

Following are all the possible events that can occur for a TM.

Event	Notes
Local Relay TRIP *Local Relay CLOSE*	From the relay, indicating a status change
TRIP Share *CLOSE Share*	From any TM, including self
TRIP Share Timeout *CLOSE Share Timeout*	Timeouts for resending shares
Signed TRIP Ack *Signed CLOSE Ack*	From Dest. Proxy, with signature and timestamp
Signed CLOSE Timeout *Signed TRIP Timeout*	Timeouts to resend Signed messages to Dest. Proxy

4.4 TM State Machine

The TM state machine, excluding the startup protocol, is shown in Figure 4.1.

For each event, the TM looks at how the event changes r or b, including the timestamp, and moves to the appropriate state. For example, a *Local Relay TRIP* means $r.status$ and $r.dts$ would change. If the TM was in **Done CLOSE** before the *Local Relay TRIP*, then $(r.status, b.status) = (\text{TRIP}, \text{CLOSE})$ and $r.dts \geq b.dts$, so the TM moves to **Attempt TRIP**.

However, one detail that the State Diagram does not capture is what the TMs send. Specifically, in the **Attempt** x states, the TM periodically sends x *Share* messages to try and gather enough shares for a threshold signed message. In the **Done** x states, the TM will send the final threshold signed message to the proxy until it gets an acknowledgment.

The dashed arrows (from **Done** x back to **Attempt** x) represent that these transitions are unlikely to be used in a real version of the protocol. However, they are necessary in order to keep the state of the TM consistent with the status and timestamp of the relay and breaker. The scenario in which the dashed arrows are taken requires a set of events to occur in a specific order and in rapid succession. Say TMs $1, 2, 3$ (with $f = 1, k = 1$) are currently in **Attempt CLOSE**. Then TMs 1 and 2 combine shares and move to **Done CLOSE**. If then a *Local Relay TRIP* occurs immediately on Relay 3, before the *Signed CLOSE Ack* from the proxy arrives, then TM 3 will first go back to **Done TRIP**. However, when the *Signed CLOSE Ack* from the proxy arrives, its timestamp will be before the *Local Relay TRIP*, and so would take the dashed arrow to **Attempt TRIP**.

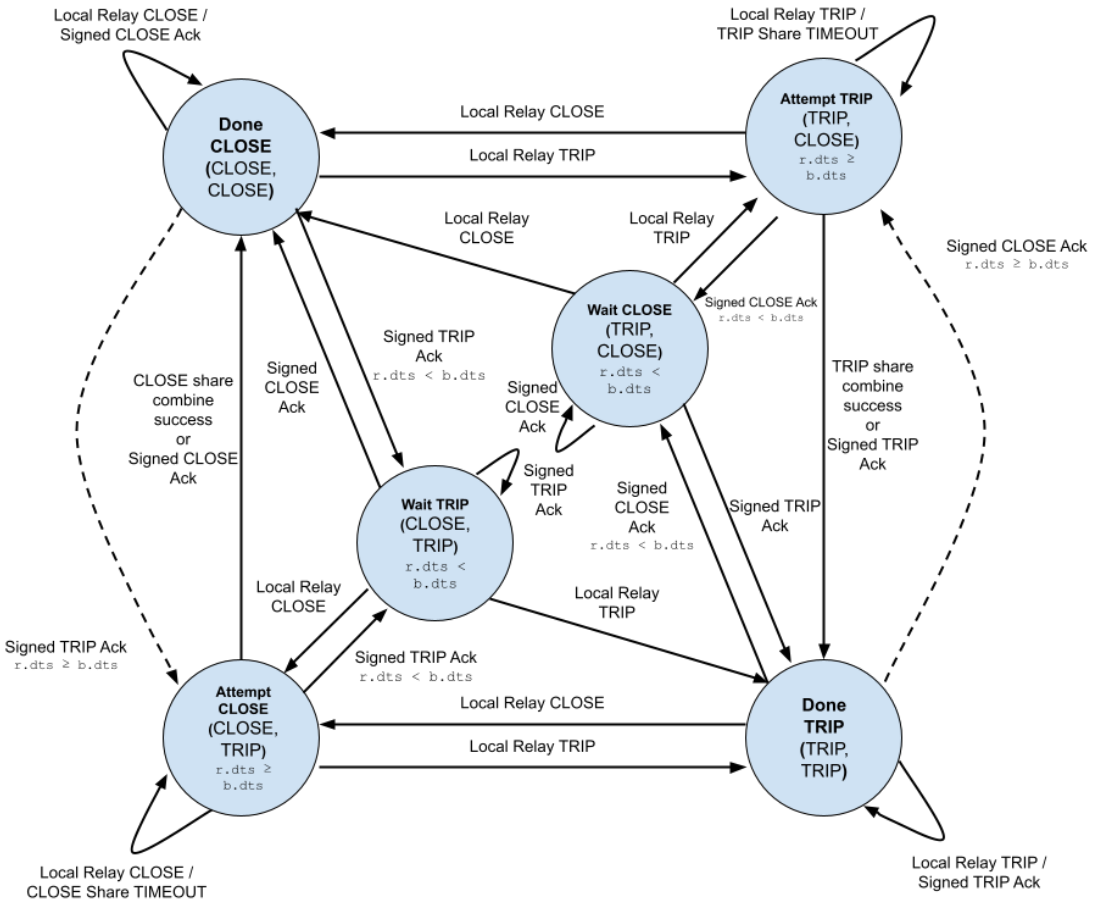

Figure 4.1: State Diagram Describing TM Protocol

4.5 Time Synchronization Issues

The reliance on comparing timestamps to decide on transitions can be an issue. For example, the timestamp on a *Signed CLOSE Ack* is from the Destination Proxy, whereas the timestamp on a *Local Relay TRIP* is from the TM machine. So, in the above scenario where the dashed transitions are taken, the timestamp of the *Signed CLOSE Ack* could be greater than the timestamp of the *Local Relay TRIP*, even though it really (i.e. with respect to a global time) happened before, as illustrated in Figure 4.2. Then, the TM erroneously would go to **Wait CLOSE** instead of **Attempt TRIP**.

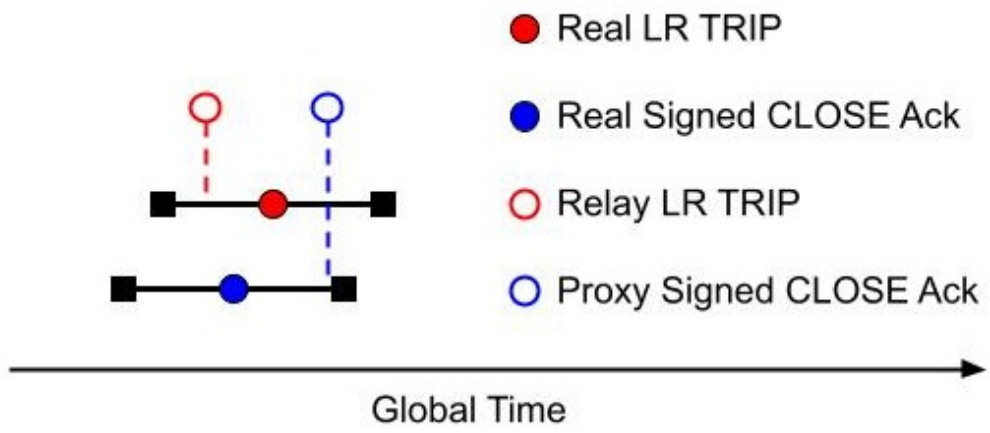

Figure 4.2: Timeline of events that appear out of order, due to time synchronization inaccuracies. The black bars represent how far off from a theoretical global time the clocks of the various machines could be.

The simplest solution to this would be to artificially wait an interval of time before any operation, equal to the maximum difference in clocks (1 ms in our case). This would ensure that each operation (e.g. the two shown in the Figure 4.2) would be sequential, no matter which machine's clock is used for the timestamp. However, it would also make the latency greater by 1 ms in all cases. Alternatively, we can make the assumption that TRIP and CLOSE operations cannot occur within 1 ms of each other and so there is no need for waiting. This is actually reasonable, given that the relays need to collect measurements to make decisions. However, the better compromise we have found is to assume that a CLOSE cannot occur within 1 ms of a TRIP, but that a TRIP can occur within 1 ms of a CLOSE. This is due to the physical nature of each operation: CLOSE operations

are manually triggered by operators or automatically done after a period of time. Then, we only need to wait to process CLOSE operations, which are less urgent. Specifically, when the Destination Proxy records the timestamp for a *Signed CLOSE Ack*, it then waits 1 ms before actually sending a command to the CB. Therefore, since the trip on the relay must happen after the breaker physically closes, then the timestamp of the *Local Relay TRIP* that any of the TMs receive will be after the timestamp of the *Signed CLOSE Ack*.

4.6 Protocol Specification

Following is the exact specification for the protocol, including the startup procedures and the Destination Proxy.

4.6.1 TM Specification

First, we define the verification of messages implicitly:

1. Each sender is authenticated by Spines, so a relay must present the correct ID corresponding with its Spines node or its message will be dropped.

2. *TRIP/CLOSE Share* and *Signed TRIP/CLOSE Ack* messages are discarded if their timestamp is older than $b.time$.

3. Similarly, *Local Relay TRIP/CLOSE* messages are discarded if their timestamp is older than $r.time$.

For combining shares, each TM keeps track of all shares it receives with DTS equal to the last DTS it sent, as well as the next DTS. Then, we define a helper function:

Algorithm 1 Helper Function to Combine Shares

```
1:  function TRY_COMBINE(x)
2:      Let dts_0 = the DTS of the last x Share sent by this TM
3:      for dts = dts_0 to (dts_0 + 1) do
4:          if there are ≥ f + 1 shares of type x for dts then
5:              Combine shares and create Signed x message msg = ⟨x, dts⟩_σ
6:              return True if successful, False Otherwise
7:          end if
8:      end for
9:      return False
10: end function
```

Then, let us define how the TM processes events in the normal case for each state. Note that we only specify **Done TRIP**, **Attempt TRIP**, and **Wait TRIP**. However, the behavior of the TM

in the other 3 states is analogous. Again, each event is verified using the implicit verification as described above. Also, we will use the term *multicast* to refer to sending a message to all TMs, including the sender itself, and i to refer to the index of the TM.

Algorithm 2 Specification for Processing Events in Each State

```
1: function DONE_TRIP_PROCESS(event)
2:     switch event.type
3:         case Local Relay TRIP:
4:             Update r.dts := event.dts
5:             return
6:
7:         case Local Relay CLOSE:
8:             Set r.status := CLOSE and r.dts := event.dts
9:             Multicast Close Share(i, ⟨CLOSE, r.dts⟩_σi)
10:            Multicast Close Share(i, ⟨CLOSE, r.dts + 1⟩_σi)
11:            Cancel Signed TRIP Ack Timeout, if exists
12:            Start CLOSE Share Timeout for next DTS
13:            State change to Attempt CLOSE
14:            return
15:
16:        case TRIP Share:
17:        case CLOSE Share:
18:            return
19:
20:        case TRIP Share Timeout:
21:        case CLOSE Share Timeout:
22:            Not Possible
23:            return
24:
25:        case Signed TRIP Ack:
26:            Update b.dts := event.dts
27:            Cancel Signed TRIP Ack Timeout, if exists
28:            return
29:
30:        case Signed CLOSE Ack:
31:            Cancel Signed TRIP Ack Timeout, if exists
32:            Set b.status := CLOSE and b.dts := event.dts
33:            if r.dts < b.dts then
34:                State change to Wait CLOSE
35:            else
36:                State change to Attempt TRIP
37:            end if
38:            return
39:
40:        case Signed TRIP Ack Timeout:
41:            Resend msg, the threshold signed message from TRY_COMBINE
42:            Start Signed TRIP Ack Timeout
43:            return
44:
45:        case Signed CLOSE Ack Timeout:
46:            Not Possible
47: end function
```

```
48: function ATTEMPT_TRIP_PROCESS(event)
49:     switch event.type
50:         case Local Relay TRIP:
51:             Update r.dts := event.dts
52:             return
53:
54:         case Local Relay CLOSE:
55:             Set r.status := CLOSE and r.dts := event.dts
56:             Cancel TRIP Share Timeout
57:             State change to Done CLOSE
58:             return
59:
60:         case TRIP Share:
61:             Store share based on (event.sender, event.dts)
62:             if TRY_COMBINE(TRIP) then
63:                 Cancel TRIP Share Timeout
64:                 Start Signed TRIP Ack Timeout
65:                 Send msg to proxy (where msg is generated by TRY_COMBINE)
66:                 Set b.state := TRIP and b.dts := msg.dts
67:                 State Change to Done TRIP
68:             end if
69:             return
70:
71:         case CLOSE Share:
72:             return
73:
74:         case TRIP Share Timeout:
75:             Let dts be the DTS of the current timestamp.
76:             Multicast Close Share(i, ⟨CLOSE, dts⟩_σi)
77:             Start TRIP Share Timeout for next DTS
78:             return
79:
80:         case CLOSE Share Timeout:
81:             Not Possible
82:
83:         case Signed TRIP Ack:
84:             Cancel TRIP Share Timeout
85:             Set b.status := TRIP and b.dts := event.dts
86:             State change to TRIPPED
87:             return
88:
89:         case Signed CLOSE Ack:
90:             if r.dts < event.dts then
91:                 Cancel TRIP Share Timeout
92:                 Set b.status := CLOSE and b.dts := event.dts
93:                 State change to Wait CLOSE
94:             end if
95:             return
96:
97:         case Signed TRIP Ack Timeout:
98:         case Signed CLOSE Ack Timeout:
99:             Not Possible
100: end function
```

```
101: function WAIT_TRIP_PROCESS(event)
102:     switch event.type
103:         case Local Relay TRIP:
104:             Set r.state := TRIP and r.dts := event.dts
105:             State change to Done TRIP
106:             return
107:
108:         case Local Relay CLOSE:
109:             Set r.status := CLOSE and r.dts := event.dts
110:             Multicast Close Share(i, ⟨CLOSE, r.dts⟩_σi)
111:             Multicast Close Share(i, ⟨CLOSE, r.dts + 1⟩_σi)
112:             Start CLOSE Share Timeout for next DTS
113:             State change to Attempt CLOSE
114:             return
115:
116:         case TRIP Share:
117:         case CLOSE Share:
118:             return
119:
120:         case TRIP Share Timeout:
121:         case CLOSE Share Timeout:
122:             Not Possible
123:
124:         case Signed TRIP Ack:
125:             Update b.dts := event.dts
126:             return
127:
128:         case Signed CLOSE Ack:
129:             Set r.status := CLOSE and r.dts := event.dts
130:             State change to Done CLOSE
131:             return
132:
133:         case Signed TRIP Ack Timeout:
134:         case Signed CLOSE Ack Timeout:
135:             Not Possible
136: end function
```

4.6.2 Destination Proxy

Now, we specify the Destination Proxy. We similarly define an object b that keeps track of the status of the breaker, with fields $b.status \in \{\text{TRIP}, \text{CLOSE}\}$ and $b.dts$. On startup, the proxy queries the physical circuit breaker for its status, and sets $b.status$ accordingly (to TRIP or CLOSE), as well as $b.dts$ to the discretized version of its current time. The following pseudocode specifies the rest of the proxies behavior, when receiving messages from TMs.

Algorithm 3 Specification for Destination Proxy Behavior Upon Receiving Message

```
1: Let msg be the received message from Spines
2: if msg.type = Recovery Query then
3:     Send (b.status, b.dts) to msg.sender
4: else if msg.type = Signed CLOSE and b.status = TRIP then
5:     Set b.status = CLOSE and b.dts = dts_now()
6:     Sleep 1 ms (to prevent time synchronization issues)
7:     Issue CLOSE command to breaker and wait for response
8:     Multicast Signed CLOSE Ack to all TMs
9: else if msg.type = Signed TRIP and b.status = CLOSE then
10:    Set b.status = TRIP and b.dts = dts_now()
11:    Issue TRIP command to breaker and wait for response
12:    Multicast Signed TRIP Ack to all TMs
13: end if
```

4.7 TM Startup

Finally, we specify the behavior of a TM on startup. This is straightforward, so we simply provide the following informal description:

While the TM has not received updates from the Relay or the Destination Proxy

- Periodically query the Destination Proxy with *Recovery Query* messages.
- Upon receiving a *Local Relay TRIP* or *Local Relay CLOSE*, update r
- Upon receiving a response to the *Recovery Query*, or a *Signed TRIP Ack* or *Signed CLOSE Ack*, update b accordingly.
- Ignore other events

Then when the TM has received messages from both its Relay and the Destination Proxy, and so has values for r, b, it will go to the appropriate state based on the values of r and b. For example if $(r.status, b.status) = (\text{TRIP}, \text{CLOSE})$ and $r.dts \geq b.dts$, then the TM will go to **Attempt TRIP**.

Analysis

We claim that our protocol satisfies the following properties:

Property 1 (Safety)**.** *The set of Byzantine replicas cannot unilaterally change the system. More formally, at least one correct relay needs to have changed its status to* TRIP *since the last time the CB changed to* CLOSE *in order for the Destination Proxy to issue a* TRIP *to the circuit breaker (or vice versa).*

Proof. We prove this in two steps. First, it follows directly from our assumptions about the threshold cryptography that a single TM cannot forge a threshold signature without a shares from at least $f + 1$ TMs. Since there are at most f Byzantine replicas, we can conclude that a share from a correct replica is needed to create a *Signed* x message that causes the circuit breaker to

Then, we know that correct TMs only send shares in the **Attempt** x state. Without loss of generality, let us examine **Attempt TRIP**. It follows from our state machine design that a correct TM will only enter into **Attempt TRIP** if the Relay issues a TRIP command after the circuit breaker has closed. Therefore our protocol will always provide **Safety**. □

Property 2 (Timeliness)**.** *Under stable network conditions and assumptions about the speed of computation, a Signed* x *will be delivered to the CB within 4 ms of when the last correct relay changes its status to* x*.*

First, to clarify what this definition means, let us assume without loss of generality that the system needs to trip, and therefore all the correct relays will issue a TRIP. However, when these trips occur could potentially vary. We measure the difference between when the CB receives a TRIP from the Destination Proxy and the latest time of all the correct relay trips. This definition is justifiable, because if it was just the last correct relay in a normal substation, the requirement would be the same.

The reason we use the latest correct relay is that this covers cases where Byzantine relays participate

in the protocol. Note that this means that a TRIP could be delivered even before the last correct relay issues a TRIP (and likewise for CLOSE).

Proof. Let us assume that the computation time for the TMs (i.e. processing and signing messages) is consistent across machines for the same tasks, and in total takes at most 2 ms. We also use the assumption that the network is synchronous and will deliver messages within 1 ms. When the last correct TM receives a *Local Relay TRIP* or *Local Relay CLOSE*, it sends its shares to the other TMs. Because it is the last correct TM, we can assume that all the other TMs also sent their shares. So within 1 ms of the last correct relay sending shares, at least one correct relay will have received $f + 1$ shares. Then it should take at most another 1 ms to send to the proxy. Therefore, the total latency time should be less than 4 ms. □

Performance Evaluation

6.1 Setup

Our implementation is written in C and developed and run on CentOS 8 machines. The test bed is shown in Figure 6.1 and consists of four TM machines ($f = 1, k = 1$). These machines are setup as described in Section 2.3 with the Relay Proxy, TM process, and Spines coordination (shown in purple) and dissemination (shown in blue) networks.

However, instead of an actual substation, we introduce some components that emulate the behavior of the MU and Relay. Specifically, a process that can send GOOSE messages according the standard runs on each TM machine alongside the Relay Proxy and TM. This process, which we call the Publisher (Pub), uses the loopback interface to send to the Relay Proxy on the same machine. Then, we use a separate machine that doubles as a sample value emulator and the Destination Proxy. The sample value emulator sends a normal IP multicast message to the Publisher on each TM machine, which tells the Publisher to send a TRIP or CLOSE. The Destination Proxy, on the other hand, is mostly the same as specified. The one modification is that the two processes communicate in order to measure the end-to-end latency, i.e. the time it takes for a command to leave the SV emulator until a corresponding command reaches the Destination Proxy. In order to test for delays or discrepancies in the relays, the SV emulator can also command a Publisher to wait a certain amount of time before issuing a trip. Then, to test for Byzantine failures, we modified a copy of the implementation to implement various attacks.

The Publisher is written using libiec61850 [12], an open source implementation of the various IEC 61850 communication protocols. This library is also used by the Relay Proxy to parse GOOSE messages from the Publisher.

6.2 Benchmark Scenario

Following are the various scenarios that we measured end-to-end latency for.

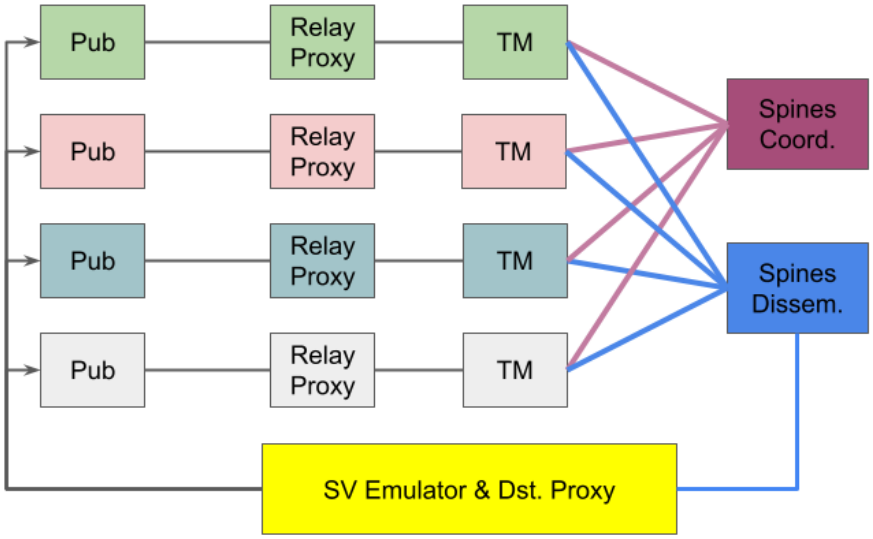

Figure 6.1: Diagram showing the lab test bed

Case 1: Normal Operation

In the first scenario, all the four replicas are working correctly. The SV emulator alternates sending TRIP and CLOSE commands as fast as possible, sending the next command immediately after the previous one is resolved. This was run for 1 million operations, and achieved a minimum latency of 1.71 ms, average latency of 2.14 ms and maximum of 3.32 ms. The histogram of the measured latencies is in Figure 6.2.

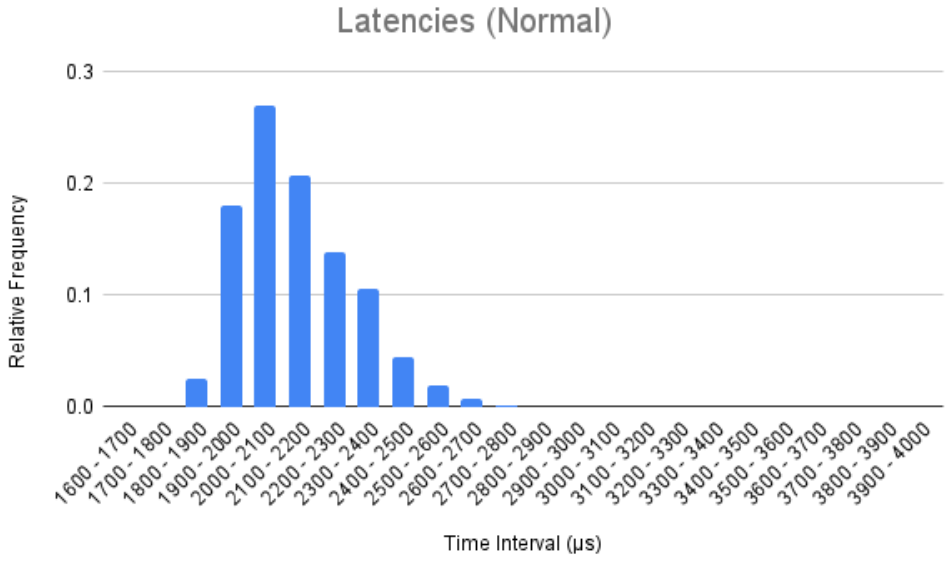

Figure 6.2: Latency histogram for one million operations, with normal operation

Case 2: One Proactive Recovery

In this scenario one replica is down for proactive recovery, so there are only three participating relays, all of which are correct. The SV emulator sends the same alternating operations as Case 1, but for 10,000 operations instead. The minimum/average/maximum latencies are 1.85/2.22/3.20 ms respectively. The histogram of the measured latencies is Figure 6.3.

As expected, the distribution is slightly shifted to the right, but without an increase to the maximum. Since there are less relays, there is a decreased probability of process scheduling and network timings lining up to provide a smaller latency, but the worst case is about the same.

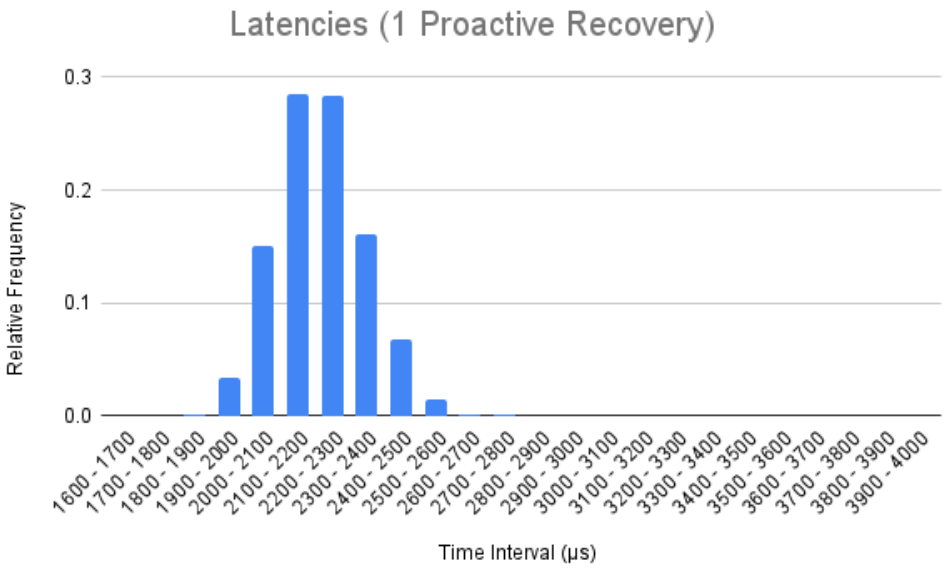

Figure 6.3: Latency histogram for 10,000 operations with one replica down for proactive recovery

Case 3: One Byzantine Fault

Our final test case is for a Byzantine fault. We designate one faulty replica, which sends incorrect shares, for example sending a TRIP instead of a CLOSE when a CLOSE is issued. It also attempts to consume the resources of other replicas with a Denial of Service attack by rapidly sending them messages that need to be processed, such as invalid shares or messages.

Under these conditions, we do the same test as the the Proactive Recovery case, with ten thousand alternating operations. The minimum/average/maximum latencies are 1.88/2.25/3.22 ms respectively. The histogram of the latencies are in Figure 6.4.

Despite the actions of the faulty replica, the 4 ms requirement for the GOOSE messages is not broken. However, the distribution does show a larger tail, indicating that the attacks by the

Byzantine replica had some effect. The fairness enforced by Spines likely prevented any stronger effects.

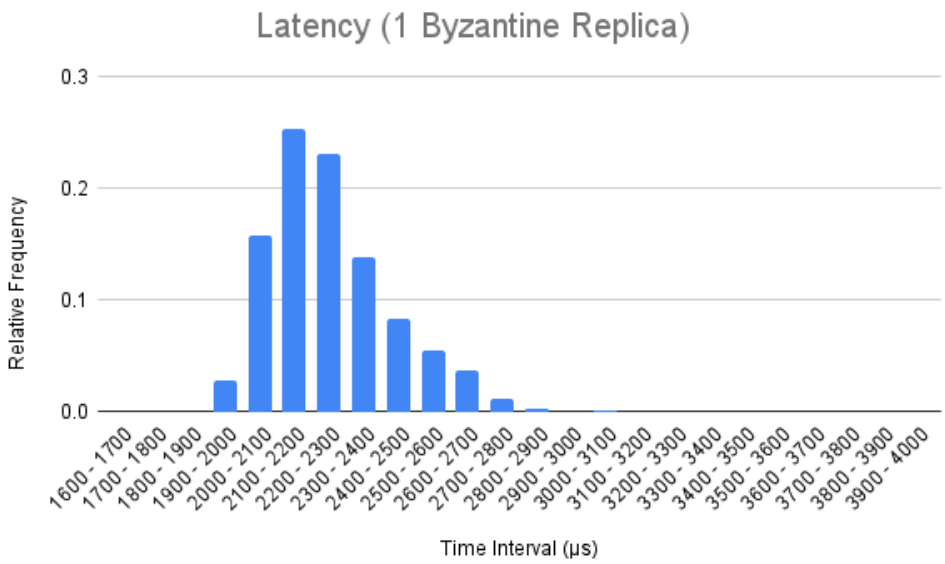

Figure 6.4: Histogram showing action times with one Byzantine replica

Conclusion

We have developed a Byzantine architecture and protocol for adding resiliency to high voltage protection schemes within electrical substations. Our design uses ideas from previous works on intrusion tolerant systems, such as proactive recovery, and additionally is adapted to the unique environment and strict constraints of the substation. Specifically, the protocol forgoes traditional Byzantine replication and instead uses threshold cryptography and discretized timestamps to coordinate replicas. This method not only uses less replicas than a traditional BFT, but also requires just one round of communication.

Our implementation of this protocol was successful in tests; maintaining timeliness (i.e. latency within the 4 ms requirement of IEC 61850) even in the presence of an attacker. An implementation of our architecture and protocol is part of a forthcoming release of the open source Spire system [1].

Bibliography

[1] Yair Amir, Trevor Aron, Tom Tantillo, and Amy Babay. *Spire: Intrusion-Tolerant SCADA for the Power Grid*. URL: `http://www.dsn.jhu.edu/spire/` (visited on 08/01/2021).

[2] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. "Prime: Byzantine Replication under Attack". In: *IEEE Transactions on Dependable and Secure Computing* 8.4 (2011), pp. 564–577. DOI: 10.1109/TDSC.2010.70.

[3] Yair Amir, Claudiu Danilov, John Schultz, Daniel Obenshain, Thomas Tantillo, and Amy Babay. *The Spines Overlay Messaging System*. URL: `http://www.spines.org/` (visited on 08/02/2021).

[4] Amy Babay, Thomas Tantillo, Trevor Aron, Marco Platania, and Yair Amir. "Network-Attack-Resilient Intrusion-Tolerant SCADA for the Power Grid". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2018, pp. 255–266. DOI: 10.1109/DSN.2018.00036.

[5] Pacific Northwest National Laboratory. *One Quarter Cycle Trip Requirement*. Personal Communication. 2020.

[6] Yingyi Liang and Roy Campbell. "Understanding and Simulating the IEC 61850 Standard". In: (May 2008).

[7] R.E. Mackiewicz. "Overview of IEC 61850 and Benefits". In: *2006 IEEE PES Power Systems Conference and Exposition*. 2006, pp. 623–630. DOI: 10.1109/PSCE.2006.296392.

[8] A. Nicholson, S. Webber, S. Dyer, T. Patel, and H. Janicke. "SCADA security in the light of Cyber-Warfare". In: *Computers and Security* 31.4 (2012), pp. 418–436. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2012.02.009. URL: https://www.sciencedirect.com/science/article/pii/S0167404812000429.

[9] Daniel Obenshain, Thomas Tantillo, Amy Babay, John Schultz, Andrew Newell, Md. Edadul Hoque, Yair Amir, and Cristina Nita-Rotaru. "Practical Intrusion-Tolerant Networks". In:

2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). 2016, pp. 45–56. DOI: 10.1109/ICDCS.2016.99.

[10] C.R. Ozansoy, Aladin Zayegh, and Akhtar Kalam. "Time synchronisation in a IEC 61850 based substation automation system". In: Jan. 2009.

[11] Victor Shoup. "Practical Threshold Signatures". In: *Advances in Cryptology — EUROCRYPT 2000*. Ed. by Bart Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 207–220. ISBN: 978-3-540-45539-4.

[12] Michael Zillgith. *libIEC61850*. Mar. 2020. URL: https://libiec61850.com/libiec61850/about/.