Distributed Helios - Mitigating Denial of Service Attacks in Online Voting

Daniel Chung,[†] Matt Bishop,[†] and Sean Peisert[†*]
{dchung, mabishop, speisert}@ucdavis.edu

[†] Department of Computer Science, University of California, Davis

[*] Computational Research Division, Lawrence Berkeley National Laboratory

October 16, 2015

Abstract

One of many major issues that plagues Internet voting is the potential for a distributed denial of service attack on the voting servers. These denial of service attacks are harmful because they block voting during the downtime. In addition, most current online voting protocols are centralized with only one voting server, making such an attack likely to disenfranchise some voters. The question is how to combat these attacks. One solution is to distribute the servers in a parallel manner, so in case one server goes down, the others can still provide service to voters. Whereas many online voting systems assume the constant availability of the voting infrastructure, we focus on the event that a server becomes unavailable. We extend a previously established online voting protocol, Helios, by adding multiserver capability. These servers communicate using the Paxos protocol, an algorithm for fault tolerant distributed environments. An analysis of this solution concludes that a multi-server Helios network communicating through the Paxos protocol promises safety and robustness.

1 Introduction

As the Internet becomes increasingly accessible on an international scale, demand has increased for integrating aspects of public administration into this convenient infrastructure. Foremost in any democracy is the ability to vote, and the capability to vote online is a major point of interest. After all, the Internet allows anyone to purchase and sell goods, perform banking transactions, and conduct entire businesses, along with just about anything else imaginable. Why, then, have we not yet transitioned to Internet voting?

Internet voting has important merits that bear consideration. Access to physical polling sites is a significant issue for many demographics, including the disabled, elderly, people working or studying away from home, and overseas voters. The convenience and accessibility of voting from the comfort of home rather than waiting in line at a polling site can be appealing and could potentially engender better voter turnout [TM03]. Aside from saving time and transportation costs, the vast majority of people (especially in developed countries) have some sort of access to the Internet: according to the a 2012 FCC study, 94% of Americans have access to some sort of Internet [FCC12], while a 2015 study finds that 83% of Americans have advanced broadband [FCC15]. The omnipresence of the Internet has the potential to enable many voters to participate where doing so would otherwise be inconvenient or infeasible. It also could substantially reduce the cost of elections given the expenses of physical voting machines, election staff, ballots, and other election-related materials. And, regardless, there are times when desperate election officials have turned to the Internet on their own initiative, such as when Hurricane Sandy shut down the ability of many voters (and even pollworkers to be able to physically get to polling places, so some election officials resorted to allowing ballots returned by email—in at least one case, first to the official email address of voting officials, and then later, when the associated mailbox filled up, to the election official's personal AOL email address [Neu13].

However, voting over the Internet is not common in civic elections. Indeed, most countries disallow it outright for several reasons [SJ12, Eps13]. In general, Internet voting comes with drawbacks not found in traditional voting. One of the reasons why Internet voting has been slow to get accepted is that of trust. There is a significant lack of trust in both the client and the network. Voting systems may be attacked, and if they are, the attackers may be able to alter votes, block them, or forge them. They may also interfere with the proper tallying of the votes, or be able to associate ballots with individual voters, enabling voter coercion [VAL04] or vote selling [Ben13]. Some of these issues can be solved technically, but other problems require the development of policies and procedures to handle them. These issues come up at least in part due to the lack of humans to supervise an Internet voting process, as it all takes place within the "black boxes" of computer systemsthat may be remote and unattended by poll workers. This is in stark contrast to the public and heavily monitored physical polling sites. Additionally, incorporating the Internet into elections creates the possibility of remote, international sabotage — attacking election servers from different countries and on a wide scale suddenly becomes much more convenient.

However, there are some situations in which Internet voting may be useful. In America, voters who fall under the Uniformed and Overseas Citizens Absentee Voting Act (UOCAVA) voters have to deal with a slew of problems such as only 75% of the votes cast being counted, and voters having to surrender their privacy when voting [Wag13]. As such, 49 out of the 50 states do allow UOCAVA to submit absentee ballots electronically. While in many cases, this definition of "electronically" includes faxes, one state does allow web uploads and several others allow ballots to be returned via e-mail. Only Alaska has allowed all eligible voters to vote online [Nat15]. Even this is possible only because UOCAVA voters returning their ballots electronically typically agree to waive their privacy in order to cast their vote. In this situation, Internet voting may be appropriate, as some have suggested [Wag13], and further development of techniques to improve the robustness of such systems, such as end-to-end cryptographic schemes, to avoid UOCAVA voters having to waive their privacy rights, would be useful.

To that end, significant efforts have been made to overcome some of the key vulnerabilities that arise in Internet voting. For example, several online voting systems give the voter the option to audit their ballot cryptographically to ensure that the vote they cast was correctly recorded and counted. Numerous usability issues have been identified in three of the most prominent such systems, Helios [Adi08], Prêt à Voter [RBH+09] and Scantegrity II [CCC+08] that have prevented voters from casting a ballot [AKBW15]. That said, one additional vulnerability that these systems have not yet addressed is the possibility of a denial of service attack on the election servers. While a distributed denial of service (DDoS) attack on any web server can be harmful, an attack during an election can be particularly dangerous. During a successful DDoS attack, a web server will lose its ability to provide its service to a client. This means that voters would be unable to cast their ballot during that time. Given the limited timetable of an election, a DDoS attack on an election server might mean that some ballots never get cast or counted. It is unreasonable to expect voters who were unable to connect to the server to find transportation to a physical voting site to cast their ballot.

This paper describes a means of combining Helios, one of the three noteworthy systems mentioned earlier, with a system that can defend against many denial of service attacks. We do this by implementing a crash-resistant, fault tolerant scheme with multiple election servers. Thus, if some election servers go down or become unavailable due to denial of service attacks, enough additional functioning servers would remain to handle the election. We distinguish this method from traditional Internet voting security schemes, most of which focus primarily on front-end user interaction and cryptographic protocols. In doing so, they often assume the constant accessibility of the Internet voting infrastructure — an unsafe assumption in this day and age. We demonstrate our approach by augmenting an existing Internet voting system, Helios [Adi08], with multiple servers that communicate through the Paxos fault tolerant protocol [Lam98]. By simulating denial of service attacks by shutting off servers at random times, we can provide evidence of safe progress using Paxos in the face of servers that have crashed or become unavailable.

In this paper, Section 2 identifies related work about Internet voting and Paxos. Section 3 describes the methodology behind setting up Distributed Helios. Section 4 covers the experiments and an analysis of the results. Finally, Section 5 concludes the paper with a look into the feasibility of Distributed Helios and ideas for future work.

2 Related Work

A variety of on-line public elections have been proposed or have been occurred. Additionally, a variety of research approaches have been suggested. In this section, we describe several examples where there were denial of service attacks present or a documented solution to denial of service attacks that could be improved.

2.1 Public Elections

The Secure Electronic Registration and Voting Experiment (SERVE) was a 2004 attempt by the U.S. Department of Defense Federal Voting Assistance Program (FVAP) to enable UOCAVA voters to vote over the Internet from anywhere in the world [JRSW04]. SERVE was never actually deployed [WP07], and indeed, there were several vulnerabilities in the proposed system, including a lack of voter verifiable audit trails. An important point regarding SERVE is that whereas physical voting locations are generally spread out enough that possible attacks would only affect individual polling stations, SERVE's centralized servers were vulnerable to denial of service attacks, which could cripple the voting service itself, thereby interfering with the voters' ability to cast votes, or the attack could have targeted the networks around the area of a particular demographic, skewing the votes. Although these attacks are detectable, there is no clear protocol on how to respond should they affect the results of the election.

In early 2007, the country of Estonia developed its own Internet voting project after an earlier small-scale success in 2005 [WP07]. The Estonian system improved upon SERVE in several ways, including the use of a secondary, follow-up phase after the initial voting, where the eligible voters could vote traditionally and could overwrite their Internet votes if they so wished. However, a recent study by Springall and Halderman reproduced the Estonian Internet voting system in a lab environment and found several glaring weaknesses [SFD+14]. Also, even though the Estonian Internet voting system is an improvement over SERVE, its solution to DDoS attacks is to shut down all Internet voting and revert back to traditional voting. Currently, the Estonian Internet voting infrastructure relies on four servers, only one of which is accessible to the public and therefore easy to lock down during an attack, which helps explain Estonia's solution to a DDoS attack. If the government wants Internet voting to become a mainstay of elections, it must come up with more effective countermeasures to deal with DDoS attacks and other avenues of attack.

2.2 Research Efforts in Online Voting

Several research projects have sought to develop new voting schemes for online voting from the bottom up. In order to do this, the developers of these voting systems have attempted to take a holistic view of elections, beginning with the key properties and requirements of elections. There are several key security properties of election systems that are especially relevant to Internet voting [RBH+09]:

- **Integrity**: Integrity in an election requires that each stage of the election be honest, meaning that a vote must be recorded as it was intended by the voter, properly recorded as a cast vote, and tallied as it was recorded. Integrity ensures that nothing was altered.
- **Privacy**: The privacy property assures secrecy in both directions of a vote: given a particular vote, one cannot figure out who cast the vote, and given a particular voter, one cannot figure out how he or she voted. Voters should not be able to prove how they voted even if they wanted to; so, the privacy property must hold even when the voter does not want it to hold.
- **Verifiability**: An election system that is verifiable is able to be individually or publicly audited. An individual should be able to confirm their ballot was cast, and the public should be able to confirm that the votes were recorded properly and counted (without sacrificing the other security properties)
- **Robustness/Availability**: Systems that are robust are able to function properly in spite of random faults or intentional attacks. That is, the service should still be available if a server crashes or is compromised. Some voting schemes acknowledge the possibility of such failures and list possible

solutions, but they have yet to be implemented and tested in practice. Our system aims to promote the availability of the voting service even if servers are under attack.

In particular, with regard to these properties, one specific notion of verifiability is that of end-to-end verifiability – all stages of the election, from ballot casting to vote tallying, should be verifiable without relying on trusting the staff, the machinery, or any intermediate step, while being resistant to coercion and vote selling. Recent research efforts have seen trends toward using cryptographic primitives to accomplish this. Systems such as Helios [Adi08], Prêt à Voter [RBH+09] and Scantegrity II [CCC+08] use cryptographic techniques to satisfy this property. However, as noted earlier, despite the desired properties of robustness and availability, these systems still rely primarily on a single server to run elections, and are therefore significantly vulnerable to denial of service attacks.

2.2.1 Helios

Helios is an open-source Internet voting system created and managed by Ben Adida that promises end-to-end voter-verifiability and secrecy [Adi08]. Helios is available in its entirety from GitHub. It allows anyone, even non-voters, to audit elections via cryptographic proofs on a web browser. The Helios protocol borrows heavily from Benaloh's Simple Verifiable Voting protocol [Ben06]. The key feature of this protocol is the separation of ballot preparation and ballot casting, which is key to auditability. It allows anyone, including non-voters, to audit and test the system. Only upon casting the ballot is the voter authenticated. In the Benaloh process, the voter chooses the election and makes his or her candidate selections. After confirming the selections, the ballot preparation system encrypts the vote and offers the voter a choice to either audit the ballot or seal it. Upon auditing the ballot, the preparation system will display the ciphertext and allow the voter to verify that the ballot was correctly encrypted. If the voter wants to then seal and cast the ballot, the system will re-encrypt it with a new ciphertext. Only after sealing the ballot will the voter authenticate himself or herself to the voting system and, if successful, have the ballot recorded. After a voter casts their encrypted ballot, the ballot is posted ona public bulletin board along with the voter's name, allowing anyone to find the voter's encrypted vote. The votes can then be tallied, and all the election data, including the shuffles, decryptions, and tallies, can be downloaded by an interested party.

Helios's main interface and data storage revolve around its website and database, which are hosted on a single server.[1] One key aspect of the Helios website is that once the ballot is loaded on the voter's browser, no further network calls are made until the ballot is encrypted. Helios accomplishes this by using a single-page web application and relies on JavaScript to deal with pre-loading, rendering, and updating HTML pages without calling upon the server to load new pages. The current Helios back-end is a web application on a single server. All election data is stored in a PostgreSQL database. Server logic is implemented in Python and the web application makes liberal use of Django, a high-level Python web framework.

The administrator is authorized to create, edit, and review ballots. Upon final examination and approval of the ballot details, the election is frozen, and no further changes can be made to the ballot or voter list. An e-mail can then be sent to voters with their Helios user name, the SHA1 hash of the election, a link to the voting site, and a randomly generated 10-character password that the administrator does not know. Voters can then make their selections for the races on the ballot. Visiting the URL results in a single-page web application, where all the necessary parameters are preloaded. Page transitions and button clicks are handled by JavaScript code.

After finishing the selection process, the voter can then seal the ballot, which will cause the ballot to be encrypted. A SHA1 hash of the ciphertext is displayed. The voter can then either audit the ballot or cast the vote. Choosing to audit reveals the random key used to encrypt, and the voter can verify the encryption with the provided Python Ballot Verification program. After auditing, the ballot is "unsealed" and changes can be made, or the ballot can be resealed with a different random cryptographic key. Opting to cast the ballot will discard the plaintext and key. The voter must then log into the server, which is the authentication step — it is this step during which a network call is finally made. After successful authentication, Helios sends the voter a confirmation e-mail (the "receipt") with the encrypted vote and its SHA-1 hash. At any point

[1] Helios Web Site: `http://heliosvoting.org`

after successfully casting the ballot, the voter — or anyone else — can go to the election's public bulletin board and view the encrypted vote. If a voter opted to audit the ballot, the encrypted, audited ballot will be posted on a separate page for anyone to verify.

Finally, once the election is over, the election administrator can prevent further ballots from being cast and compute the encrypted tally. The encrypted votes will be combined into the tally, and the trustees of the election are responsible for decryption. Before the election, each trustee generates a keypair and submits the public key to Helios. During the decryption phase, each trustee will provide the corresponding private key. After decryption, the final tally is first reviewed by the administrator and then released to the public.

In Adida's original Helios paper, he notes that he made a conscious decision to "forgo complexity and opt for ... a single server" [Adi08]. While one server does lend itself to simplicity, the resulting system becomes vulnerable to attack. A distributed denial of service attack could easily bring down the server and cripple the entire system.

2.3 Recent Extensions

We note in passing that, while Greece has not yet embraced Internet voting, many Greek universities have used online voting for their elections [TPLT13]. The Greek university Internet voting system used is called Zeus, and is heavily based on Helios. Like Helios, Zeus is end-to-end verifiable and offers computationally secure privacy. Zeus was deployed by several universities in the face of opposition from political parties. The University of Thrace was forced to annul its first election due to a coding bug that opened (started) the election too early. A second attempt resulted in the server undergoing a slowloris attack [Ha.09], a denial of service attack where the attacker opens many connections to the target server and sends partial requests, causing the target to keep the connections open until the maximum concurrent connection pool is filled; this prevents additional connections from being made [ZJT13]. Fortunately, the attackers used static IP addresses and were swiftly blocked. The University of Patras also experienced a slowloris attack that was dealt with easily. However, some universities — including the University of the Aegean, the Agricultural University of Athens, and the University of Athens — had protesters stage sit-ins at the e-mail server rooms, during which they switched off the servers and prevented voters from receiving access links through their mail. Voters had to go to a physical location to receive their links. Overall, none of the attempted attacks on Zeus were particularly effective, but there were not any serious attacks during the elections. The voters in question were not particularly technology-savvy and thus were sometimes confused about steps in the process, such as the voting receipts, because they did not understand the purpose of the receipts. None of them chose to use the audit option, implicitly trusting that the Zeus system was working.

Some clear lessons can be learned from these previous attempts at Internet voting, the most obvious of which is that Internet voting is not yet ready for large-scale deployment. Gaps in meeting the key requirements remain in the solutions that have been deployed thus far. It is difficult to find a system that is cryptographically secure yet simple enough for a layperson to understand and robust enough to withstand possible attacks.

3 Distributed Helios: Enhancing End-to-End Verifiable Online Voting with Robustness to Denial of Service Attacks

Our solution to mitigate the effects of distributed denial of service attacks on election servers is to replicate the servers that house the data using the Paxos protocol. This approach must be handled carefully, as it can lead to instances where replicated servers are not synchronized properly. Just having multiple servers is not enough, as their state can end up differing. If server A receives messages m_1 and then m_2, it is entirely possible that, due to latency or some other reason, server B receives m_2 before m_1, or worse, receives m_2 without receiving m_1 at all. If the messages both alter the system, servers A and B will end up diverging in state, possibly creating different outputs. Without some way to handle disagreements, the system cannot make any meaningful progress, because it does not know what course of action to take. Maintaining state across multiple servers can be handled by a fault tolerant protocol.

3.1 Paxos

Paxos is a fault-tolerant consensus protocol developed by Leslie Lamport [Lam98]. Paxos is meant to solve the problem of an unreliable network of unreliable parties trying to agree upon a result. In the network, members are able to communicate with each other through messages that take arbitrarily long to be sent and may even be dropped in transit. The members, which are the servers in our implementation, may also fail at any point. Under these conditions, these members must agree on certain proposed values. Under these assumptions, the Paxos protocol allows for the system to make progress even with a certain number of failures.

Though originally a research concept, today Paxos is used in many practical, production systems. For example, Chubby is a fault-tolerant system developed by Google that has been utilized in several of Google's distributed systems, such as Google File System and Bigtable [Bur06, CGR07]. At its heart, Chubby is a coarse-grained distributed locking mechanism meant to achieve the goals of reliability, availability to large numbers of clients, and simplicity. It provides a useful tool as a locking service, a name server, and for leader elections.

While there are many versions of Paxos, every version follows the same basic formula. There are three safety requirements for consensus: only a previously proposed value can be chosen, at most one value is chosen, and the server will not learn that a value was chosen unless it actually has been chosen. The network is made up of servers that take on different roles in the protocol:

- **Acceptor**: The acceptors are responsible for deciding whether to accept or reject a proposed value, generally by majority vote.
- **Proposer**: A proposer offers a value, usually submitted to it by a client, to a group of acceptors. In the event of multiple simultaneous proposers, a lead proposer is chosen to prevent deadlock.
- **Learner**: Once the acceptors agree on the proposer's value, the learners are responsible for taking action or executing the request.

Throughout the protocol, servers can take on multiple roles. Acceptors and proposers can end up as learners, and learners can act as proposers in subsequent Paxos rounds. In some cases, servers will take on all three roles. Paxos takes several rounds of communication [Lam01, MJ13]:

- **Prepare Phase**: In this phase, the proposer generates a prepare message, which includes a proposal number N. The proposal number acts as a sequence number, increasing as the proposer generates more proposals. Afterwards, the prepare message is sent to the acceptors, asking them to promise never to accept a proposal with a lower proposal number. Additionally, it asks the acceptors for the proposal with the highest number less than N that they have accepted, if any. The size of the set of acceptors that the proposer sends the prepare message to may vary, but the proposer must receive a response from a majority of acceptors to make meaningful progress.
- **Promise Phase**: If proposal number N received by the acceptor is greater than any of its previously received proposal numbers, the acceptor sends the proposer a promise message that assures the proposer that it will ignore all future messages with a lower proposal number. The acceptor stores the value of the highest received proposal number for future use. If the proposal number happens to be less than a previous proposal number, the acceptor will ignore the prepare message. At any point in the protocol, acceptors can choose to ignore any message from a proposer without jeopardizing correctness.
- **Accept Request Phase**: Once the proposer receives promise messages from a majority of the acceptors, it can then create an accept message, which includes the value (or client request) and the initial proposal number. This message is sent back to the acceptors.
- **Learn Phase**: If an acceptor receives an accept message, it will accept it as long as it has not already responded to a prepare message with a higher proposal number. Once the message is accepted, it sends a learn message to everyone else, including the learners.

3.2 Distributed Helios

We extend the Helios system by adding the capability to withstand distributed denial of service attacks. To accomplish this, we replicate the Helios server across multiple machines. We then have to ensure that the same state is maintained across the different servers. To do this, we use a crash tolerant protocol. Through combining Helios with the Paxos protocol, our goal is to create a robust online voting system that can continue functioning correctly in the face of a distributed denial of service attack. The Paxos protocol helps handle synchronization issues — any changes to the server must be approved by a majority of the network before being enacted.

Our implementation of Paxos makes no distinction between the acceptor set and learner set. All servers simultaneously act as acceptors and learners. If we want to expand the network to accommodate more servers, it may be beneficial to limit the subset of acceptors in the interest of latency.

3.2.1 Implementation

The Helios server code is publicly available on GitHub.[2] To replicate the servers and make testing easier, we used VMWare Workstation 10.0.3 to create new virtual Ubuntu 14.04.1 64-bit servers. We cloned the Helios server system from GitHub onto these clean virtual machines and modified the files thereto add the fault tolerance. The Django framework, used by Helios, allows for multiple databases in the network. These databases can be located on separate machines. Helios uses PostgreSQL for its database purposes.

Helios offers authentication using Google, Facebook, Twitter, and LinkedIn accounts. In our case, we enabled Google authentication for testing. Google recently revamped their authorization system API to use only the OAuth 2.0 protocol, causing Helios's implementation of OpenID to become obsolete as of 2015. Any new OpenID functionality needed to be migrated to Google's Google+ sign-in. Fortunately, Ben Adida was kind enough to update the Helios code on GitHub to accommodate the change in systems. As Google does not allow private IPs as redirect URIs, and the virtualizing servers used private IP addresses, we had to alter Helios's authentication code to route the redirect URI to an arbitrary domain name.

In Helios, updates to the PostgreSQL database are done using the Django `save()` function. `save()` by default will save to the primary database, but it also has the option to save to a particular database if there are multiple databases in the network. We overrode the `save()` function in each Python class to loop through all the servers available and save to each one (assuming that the Paxos protocol agreed upon the save):

```
def actualsave(classtype, obj):
      for currentserver in servers:
        super(classtype,obj).save(
                using=currentserver[0])
```

By default, Django objects have auto-incrementing primary keys that are saved as an attribute when the `save()` function is first called. Caution is needed when dealing with the primary keys, since multiple save requests that get rejected by Paxos might lead to primary key collisions.

3.2.2 Communication

To facilitate communication among servers, we used Twisted, an event-driven open-source networking engine for Python. Twisted is robust, flexible, and stable, but most importantly, it is asynchronous, meaning that waiting for messages will not block code from running like synchronous communication would. Twisted engines respond to events as they arrive onto the network.

Twisted instantiates protocols per connection to the network and configuration data is discarded when the connection is completed. Most of the code to handle incoming events is in a protocol instance. Persistent configuration is stored in a protocol factory, which is responsible for handling the creation of new protocol instances for each new connection.

[2] `https://github.com/benadida/helios-server`

Twisted is useful for creating both clients and servers. We repurposed the Helios `manage.py` file to make Helios simultaneously act as both a Twisted server and a Twisted client. Both the client and server use a modified LineReceiver Twisted protocol; this simple protocol handles receiving and sending lines across a connection. The lines that we send are JSON objects that contain:

- **Message type**: The message is either a prepare (`PREP`), promise (`PROM`), accept request (`ACC`), or learn (`LRN`) message
- **Class name**: The name of the Python class that Helios is saving to the database
- **Highest sequence number**: The highest sequence number that the current server has seen
- **Current round number**: The round number of the server that sent the first message

In our implementation, the client protocol sends `PREP` and `ACC` messages to the server protocol and waits for the server to reply with the respective `PROM` and `LRN` messages. If it receives enough `LRN` messages, it will proceed to use the overridden `save()` function to save the update to all the servers in the database.

The client factory, which produces the client protocol instances, inherits from the ReconnectingClient-Factory subclass. A ReconnectingClientFactory factory will automatically attempt reconnection to a server if the connection is lost or broken. The number of reconnection attempts depends on a timer that grows with an easily modifiable (usually exponential) factor, and a successful reconnect attempt will reset the delay. We set the initial reconnect delay at 3 seconds with a 1.3 factor multiplying the delay on each subsequent attempt.

One significant aspect of Twisted is that neither protocols nor factories have any knowledge of the network. This is intentional so that neither of them require blocking on code to wait for network activity. The Twisted reactor class is the class that actually listens on a given port and network interface, whether it be TCP or UDP or some other protocol. The Twisted `reactor.run()` method, which is used to start listening for connections, does block on code, so we created a separate thread for the reactor to run on, allowing the rest of the code to continue running.

It should be noted that these connections are unencrypted to reduce latency. No personally identifiable information regarding the election itself — the voter, the ballot, the list of trustees, and so on — is ever sent over the connection. Also, as our threat model deals only with denial of service attacks, and does not include corruption attacks.

4 Results

4.1 Experiment Details

The two most important considerations with Distributed Helios are protocol correctness and running time. To test both, we created several different test case environments, with Helios running on a single server without Paxos and Helios running on 3, 5, and 7 servers with Paxos, respectively.

The first environment, where a single Helios server runs without Paxos, serves as the control environment. An odd number of servers for most of the other environments was chosen for reasons of clarity. It is more intuitive to demonstrate a majority of acceptors on an odd number of servers than an even amount. It should be noted that most real-life applications of distributed fault tolerance use between three and seven replica servers, thereby tolerating between 1–3 failures, given that Paxos-like protocols are able to tolerate f crash failures for every $2f + 1$ replicas [CGR07, RJ08].

4.1.1 Correctness Testing

Adding the Paxos layer on top of Helios introduced a level of complexity that increased the probability of error. Therefore, we needed to verify that the Helios servers operate correctly under the Paxos protocol.

A control test was done on the default non-Paxos Helios, in which an entire election process was recreated to verify normal operation. The entire election process consists of:

1. Creating an election
2. Updating or modifying the eligible voters
3. Adding or modifying trustees
4. Adding questions and candidates to the election
5. Freezing the ballot and allowing voting
6. Voting
7. Auditing the ballot
8. Posting the audited ballot to the public bulletin board
9. Casting the completed vote
10. Viewing audited ballots on the public bulletin board
11. Freezing the election, preventing any more votes
12. Tallying the votes
13. Releasing the election results

The same process was repeated with the Paxos-Helios environments. It was soon discovered that some pages would load more quickly than the Paxos protocol could execute, causing the pages to behave as if the objects were not being saved until they were refreshed. To counter this, we imposed artificial wait times on certain page loads. The wait times had to be increased as the number of servers increased. Even worse, sometimes sequential code fragments were executed before earlier code had finished saving objects, creating "list index out of bounds" errors because the code was trying to access the as-yet-unsaved objects. Special attention had to be paid to these areas of Helios, resulting in even longer loading times for the sake of consistency.

Before we started to introduce failures into the servers, we ran tests on the fully functional network. Messages across the network were tracked on all servers to verify arrival in the correct order. If a client request was made to the primary server, we checked that it sent a prepare message to the other servers, received promise messages, sent accept request messages, and finally received learn messages before saving. Likewise, the acceptor servers each had to receive a prepare message and an accept request message before sending off the respective promise and learn message.

If a server is already the leader (having sent a previous prepare message and received enough promise messages), then we do not need to repeat the prepare and promise phases and can skip to the accept request phase. Therefore, we perform a few consecutive requests on the same server and check to see that only the first request contains the prepare and promise messages. However, if a client request is made to another server, we need to check to see that it performs the prepare and promise phases with the appropriate round numbers.

As the number of servers increases, so does the number of promise and learn messages required to fulfill the majority quota. With three servers, the proposer only requires one promise and learn message (because it acts as part of the implicit majority). With five, it requires two external promise and learn messages, and with seven servers, three are required.

We also have to ensure that the contents of the messages follow proper protocol. That is, the sequence numbers of different objects should be monotonically increasing, and the round numbers between requests should increase with respect to the length of the number of servers. Also, messages with lower round numbers than the current highest round number should be ignored.

After verifying correct operation with no server failures, we began to introduce faults into the system. Failures were first simulated by simply shutting down Helios on a server and preventing any contact to and

from the server. Shutting servers down during key moments in the Paxos protocol created the test cases that we needed.

The easier faults to test for are failures in the acceptor servers rather than the lead server. Because Paxos only requires a majority of the servers to respond, removing one server should not affect progress. In fact, in a three-server example where Server 1 is the primary server, the second learn and promise messages Server 1 receives from Server 3 are ignored by the lead server, since it will have already achieved consensus with the responses from Server 2, which arrived earlier. Indeed, shutting down a single replica server did nothing to affect the protocol other than require synchronizing after a restart. To extend the example to more servers, a leader in a seven-server Paxos environment can safely ignore up to three messages, meaning the system is unaffected if three acceptor servers crash, since they do not have to undergo leadership change.

A larger problem occurs when a majority of the acceptor servers are taken down. If a proposer times out while waiting for a majority number of responses from the acceptors, it will consider the request a failure. If another request is made, the server should increase its round number (to avoid being ignored by any servers that it managed to contact) and try again until the acceptors come back.

The most complex situations involve failures in the lead proposer server. Regardless of how the new lead is chosen, we need to make sure that all the other servers are aware of the choice through the prepare and promise Paxos phases and the correct round numbers. If the lead server was not about to save a client request, then the situation does not need to be resolved further.

However, in the event that the lead server is in the process of sending accept request messages in response to a client request when its leadership gets taken over, then the original leader needs to inform the new leader of the pending request. To accomplish this, if the proposer has managed to accumulate enough promise messages from the acceptors (or if it is already the leader), it will save the potential request. If another server attempts to take leadership from the current leader, the leader will attach the request in the promise message it sends back to the new server. The new server will execute the previous request with new accept request messages before executing its own requests.

Messages to the proposer were interwoven and interrupted with new requests to verify proper behavior. We created unit tests to assert that certain conditions — sequence numbers, round numbers, message ordering — were fulfilled under arbitrary message arrangements.

4.1.2 Testing Running Time

To determine how Paxos affects the running time of Helios, we tracked the execution time of the overwritten `save()` function from the initial function call to when the object is actually saved into the database. In the physical world, distributed servers are often separated geographically, creating latency between message arrival. We simulated physical network travel by inducing an artificial latency of 50 milliseconds for each message sent across the virtual network. We kept track of five objects throughout the trial runs: AuditedBallot, CastVote, Election, Trustee, and Voter. These objects are class objects located in `helios/models.py` and are automatically created and updated by Helios as the user goes through the election process. The objects are tracked by overwriting their default `save()` function and forcing them to go through the Paxos protocol — the respective class names and objects are passed through the protocol as parameters.

The AuditedBallot object contains the information necessary to post an audited ballot to the public bulletin board. Similarly, the CastVote object is saved when a vote is cast and posted to the ballot tracker. The Election object is the most commonly saved object and includes updates to the election name and description, adding or editing election questions, altering the availability of the election, and general adjustments to the election. Helios automatically saves itself as a Trustee object when an election is created, but it is possible to add additional trustees afterwards. Finally, Helios offers the option to limit the voters in a private election to a select list or to open up the election to the public. If the former is chosen, a Voter object is created for each member of the voter list, and if the latter is selected, the Voter object is saved upon casting the vote.

Similar to the correctness testing, we ran several iterations of a complete election process, from election process to vote casting. We repeated this process several times to create a reasonable number of data points. When we apply our fault tolerance protocol to Helios, the additional latency depends on the number of

protocol messages sent across the network (which standard Helios does not send) and the network latency. We can estimate the extra latency based on this equation:

$$\#\,servers * ((\#\,messages * network\,latency) + \#\,messages)$$

Our equation estimates the amount of extra time to save an object relative to standard Paxos. The number of servers denotes the number of servers required to form a majority: two servers in a 3-server system, three servers in a 5-server system, and five servers in a 7-server system. The number of messages represents the messages sent with the Paxos protocol. There are four messages: the prepare message, the promise message, the accept request message, and the learn message. Along with the induced 50-millisecond latency, we can estimate the expected delay of Distributed Helios. In a 3-server system, we expect an object to be saved around 8.4 times slower than standard Paxos. In a 5-server system, we expect the latency to about 12.6 times the standard Paxos latency. Finally, in a 7-server system, the latency is expected to be about 21 times that of standard Paxos.

4.2 Empirical Results

A control test of a single server without Paxos was conducted to measure the time taken to save an object to the PostgreSQL database. In the following figures, each dot represents the time it takes from an object to be saved into the database, with each color representing the object in the dot's column. The horizontal jittering is for visual purposes only.

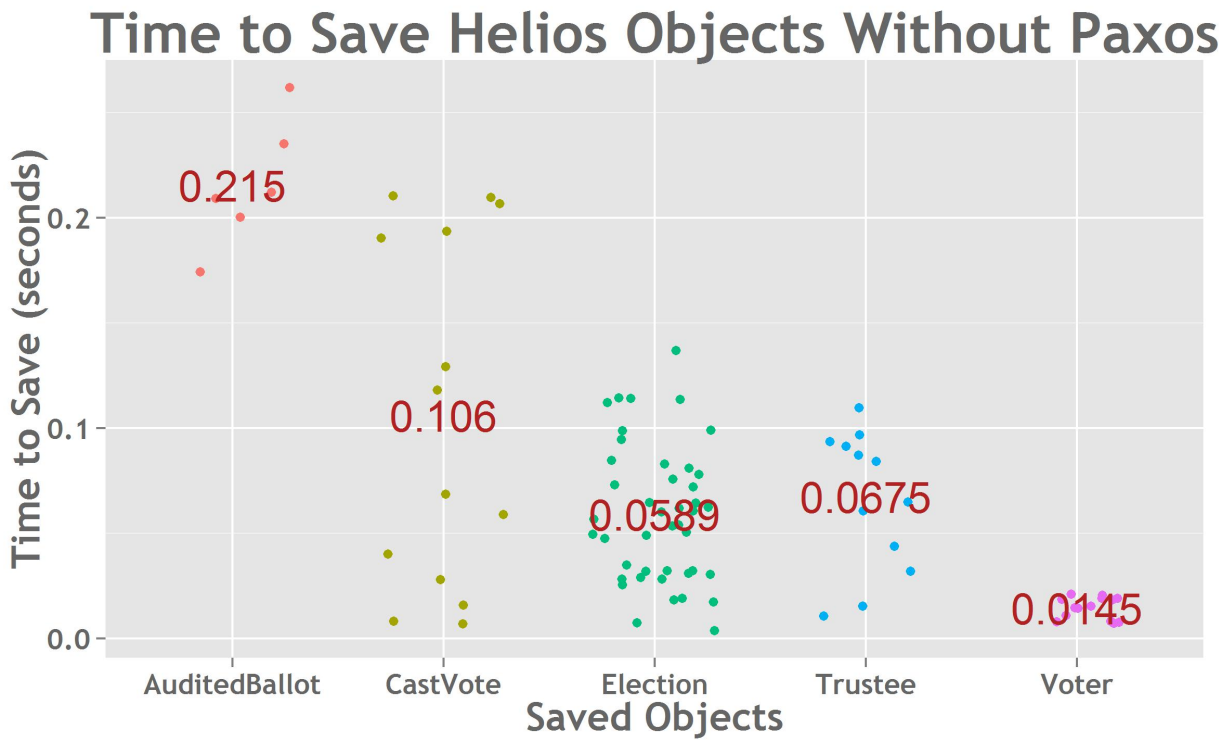

Figure 1: Average Helios Save Times for Single-Server Non-Paxos (Overall: 0.071s)

As shown in Figure 1, without having to participate in Paxos, the average save time of a Helios object is 0.071 seconds. Webpage load times are nearly instantaneous.

Using our hypothesized estimates and the average save time for single-server non-Paxos, we estimate that the average save time for 3-server Paxos will be about 0.596 seconds. Likewise, we estimate that 5-server Paxos save times will be around 0.895 seconds. Finally, our equation estimates that 7-server Paxos will have a save time of around 1.491 seconds.

3-server Paxos marks the introduction of transmitted messages across the network that create inactivity in the system until the protocol is resolved. To enforce interface correctness, we imposed small load times on the webpages while the protocol was in effect. For 3-server Paxos, there was a small but noticeable half-second delay when saving objects, especially when a new leader needed to be accepted.

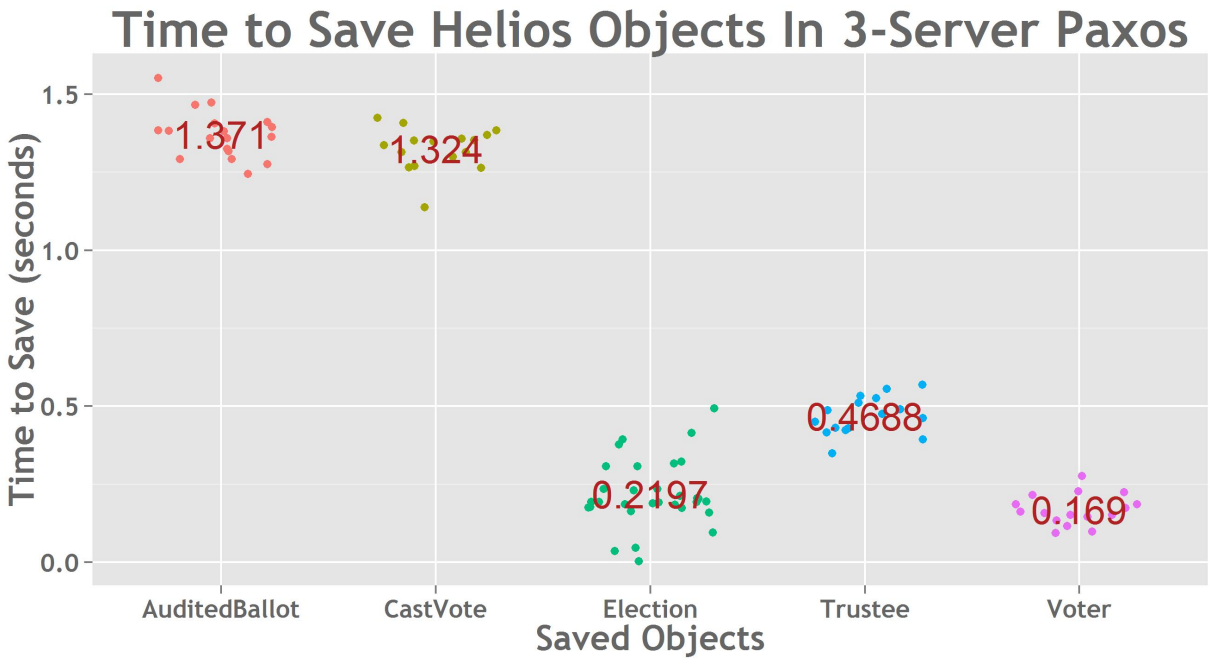

Figure 2: Average Helios Save Times for 3-Server Paxos (Overall: 0.6483s)

As shown in Figure 2, on average, in 3-server Paxos, a server that wishes to fulfill a client request takes anywhere between 0.17 to 1.4s to save an object, depending on object type. These times include the 50ms induced latency we imposed on network traffic. The overall average save time was 0.6483s, and an object takes 9.13 times as long as non-Paxos to save under 3-server Paxos.

As shown in Figure 3, the mean save time for an object under 5-server Paxos is 1.087 seconds, ranging from 0.18 to 2.3 seconds. Under 5-server Paxos, a new object takes 15.31 times as long as non-Paxos to save.

As shown in Figure 4, under 7-server Paxos, the protocol takes 1.588 seconds to save an object, with objects taking between 0.28 and 3.2 seconds to save. With respect to non-Paxos, 7-server Paxos takes 22.36 times as long to save an object.

As shown in Figure 5, overall average save times across environments ranged from 0.071 seconds in the non-Paxos environment to 0.65 seconds with 3 servers to 1.09 seconds with 5 servers and finally to 1.59 seconds with 7 servers.

4.3 Analysis

Overall, our hypothesized latency increases were very similar to the actual results. In 3-, 5-, and 7-server Paxos, our estimated save times were 0.596 seconds, 0.895 seconds, and 1.491 seconds respectively. The data collected reveals that the actual save times were 0.648 seconds, 1.087 seconds, and 1.588 seconds. In each

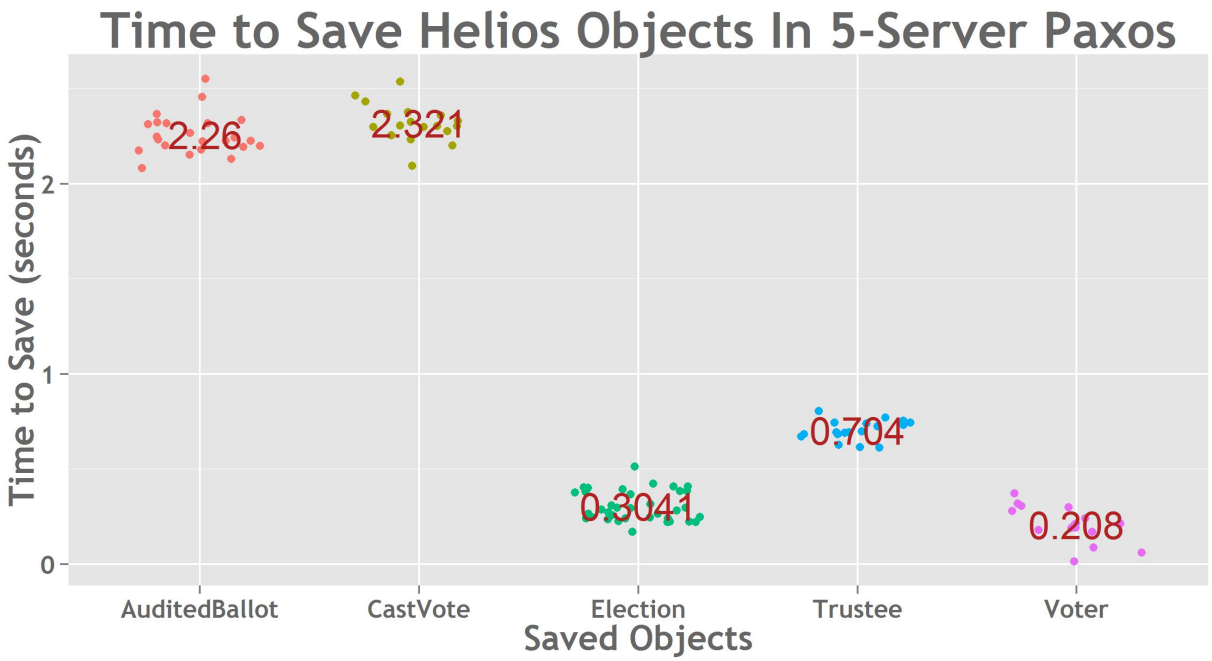

Figure 3: Average Helios Save Times for 5-Server Paxos (Overall: 1.087s)

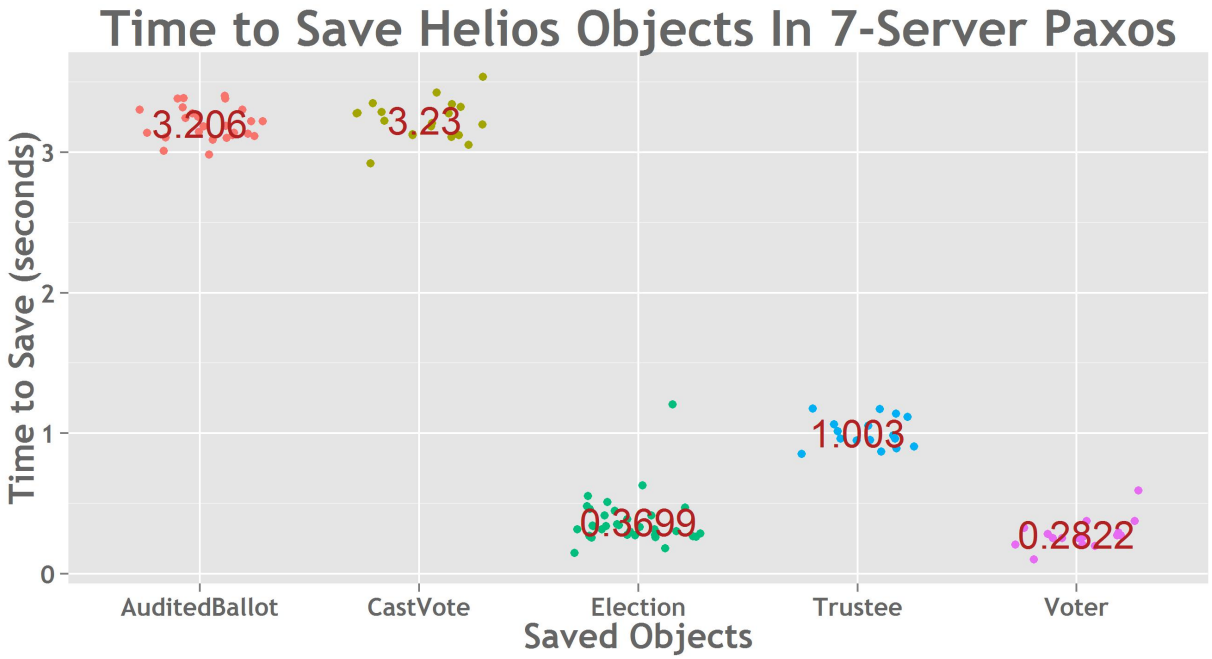

Figure 4: Average Helios Save Times for 7-Server Paxos (Overall: 1.588s)

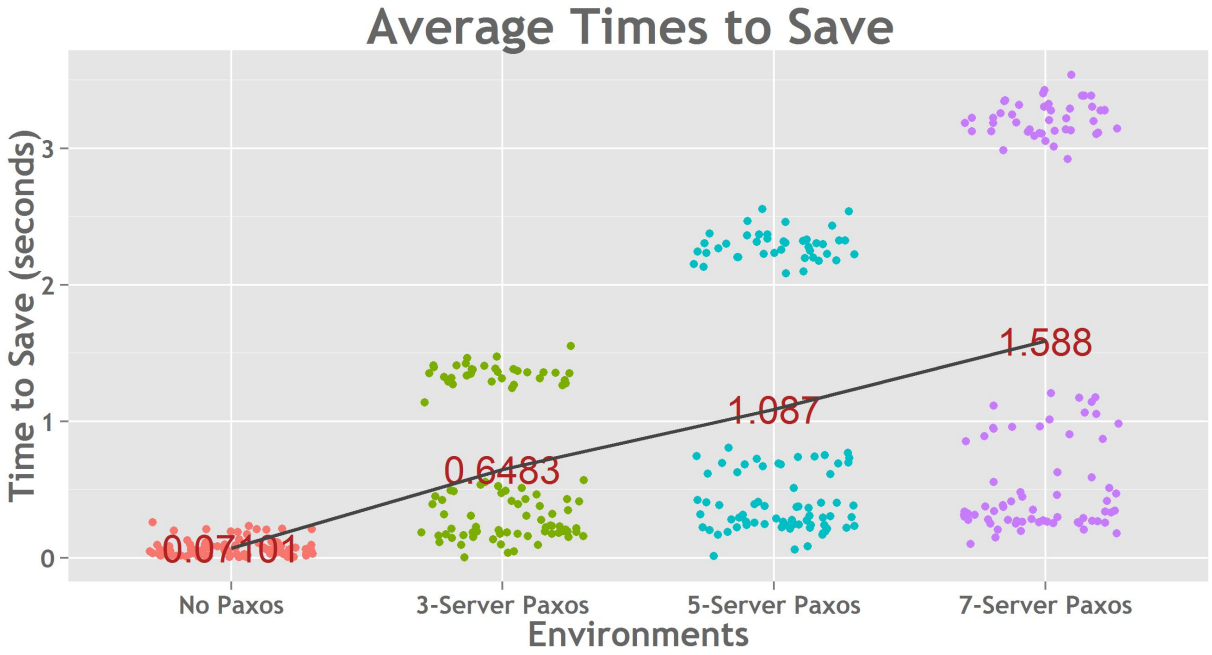

Figure 5: Average Helios Save Times Across Environments

case, our estimated delay was slightly too low. We attribute the small discrepancy to the overhead involved with maintaining the Paxos state inside each server.

Apart from the non-Paxos control environment, adding additional servers increased the save time linearly, with each single server adding approximately 0.23 seconds to the average save time. The insertion of three servers creates a 912.97% increase in running time, up to a 2236.36% increase with seven servers, and the data seems to indicate a linear increase in save time as servers are added.

We generated a linear regression model for the relationship between the number of servers (ignoring the non-Paxos case) and the mean time to save an object. To accomplish this, we ran the `lm()` linear model fitting function in R using a dataset comprised of all the data points across the different environments, disregarding non-Paxos. The output of the `lm()` function describes the coefficients for the y-intercept and slope of the line that fits the regression model of the dataset. In other words, the result is a line of best fit. The equation is:

$$y = 0.2352x - 0.06885$$

where x is the number of servers and the resulting y is the mean time to save. We assume an odd number of servers in our model, and if x is even, we use $x + 1$ servers instead to estimate the running time. Using this linear model, we can extrapolate the save time for larger numbers of servers. Each server adds about 0.235 seconds to the overall save time of an object. For example, a 9-server implementation would have about a 2.048-second delay in saving an object.

Thus far, our simulations of failed servers only account for heavy denial of service attacks that brought down all traffic to and from the servers. There are times, however, when the degree of the attack may vary. A server under a light attack may only drop a small percentage of messages rather than all of them. Although hindered, these servers can still contribute to the Paxos protocol. To accommodate servers slowed down by attacks, we specify a timeout period while waiting on messages. By utilizing TCP to send messages, we are guaranteed ordered, reliable messages even in a slow network.

We can use our linear regression model to set an appropriate timeout for the number of servers involved; in our case, we set the timeout to be 1.5 times the estimated average save time of a CastVote or AuditedBallot

object for a given number of servers, as those two were the objects that took the longest to save. In this way, our system is capable of handling servers under varying levels of attacks: uncommunicative servers can be safely ignored as long as a majority exists, while at the same time, slow servers are allowed a grace period in which to respond to messages.

A small optimization we can make to the execution of Paxos is to parallelize the messages sent by the leader to the acceptor servers. However, the bulk of the delay in Distributed Helios lies in waiting for the correct amount of response messages from the acceptor servers. Thus, while parallelizing leader messages might slightly improve efficiency in the prepare and accept request phases, it will not significantly impact the overall average delay.

4.4 Limitations

The Paxos protocol that we employed in our implementation is non-Byzantine [LSP82]: that is, members do not maliciously lie or attempt to undermine the protocol, and messages are delivered without corruption. Protocols that deal with Byzantine faults [CL02] are needed for arbitrary faults such as corrupt servers or messages. Note that Byzantine protocols typically require $3f + 1$ replicas for each failure f that the quorum can tolerate, in contrast to $2f + 1$ for Paxos and crash failures, and of course a corresponding increase in latency due to increased computation and communication. Because our focus is on crash tolerance, and minimizing latency, our implementation is not resilient to Byzantine faults.

Additionally, we replicate only the servers containing the ballots. The processes themselves are not replicated, including the process of encrypting the vote. However the encrypted vote itself is saved to each replicated database. The verification of the correct encryption is also not replicated, but in reality is optional and can be performed at any point by any of the servers as long as the encrypted ballot was saved and replicated. This means that if a server gets taken down in the middle of the process, the process can just be repeated on another server. If a ballot is re-encrypted, it will replace the old ballot so there should be no conflicting ballots.

5 Conclusions and Future Work

In this paper, we explored the dangers of distributed denial of service attacks on single election servers. Malicious attacks on network servers are not only simple, they can be issued from remote locations. A single successful attack on an election server can bring down the entire service indefinitely, a major issue given the limited timetable of an election. Our solution was to replicate Helios, an open-source online election system, into a multiple server network. We addressed the problem of maintaining state across the different servers in a potentially faulty network by augmenting Helios with Paxos, a fault tolerance protocol meant to ensure safety and progress in an imperfect environment. Additionally, testing confirmed correctness of the Paxos protocol: client requests are only honored after being acknowledged by a majority of the network. Each additional server increases the latency linearly by less than a quarter of a second, and this latency can potentially be lessened with optimization. Distributed Helios is a promising first step in protecting election servers from distributed denial of service attacks.

The largest concern with using Distributed Helios over standard Helios is the latency cost associated with enforcing a safe protocol across an unsafe network. Without optimizing the system, a three-server setup averages a 0.65 second delay to save. At the upper end, a seven-server configuration can take over 3 seconds to save an object. High latency can sometimes cause a negative effect on the usability of a system. As Neilsen points out, "1.0 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay ... 10 seconds is about the limit for keeping the user's attention focused on the dialogue" [Nie93]. Longer delays often persuade users to divert their attention to some other task or give up completely, especially without feedback about the delay. In fact, any delay longer than one second causes the user to feel that the interface is sluggish.

One long-standing method of mitigating the short attention span of users is implementing a percent-done indicator for any action requiring more than 10 seconds [Mye85]. The indicator's benefits are threefold: it

assures the user that progress is being made and that a crash has not occurred; it gives an estimation of completion time to the user; and finally, it alleviates the annoyance of waiting by providing the user with something to look at while waiting. Although the latency of Distributed Helios is less than 10 seconds, a five- or seven-server configuration will have enough noticeable delay that a percent-done indicator, or even just a quick animation to represent "loading" time, may be useful to maintain the user's attention.

The other real-world fault-tolerant systems discussed, Google Chubby and Zab, both use between three to seven replica servers in practice [CGR07, RJ08], which is one of the reasons behind our experiment environments. With optimization, Distributed Helios, which is relatively small, may be able to run even faster.

Distributed Helios is not highly recommended for small-scale elections that are unlikely to be attacked, as the latency cost would not outweigh the safety advantage of a distributed system. For example, small school elections are probably not going to be targeted, making the strong safety guarantees of Paxos excessive, especially considering the fact that it will likely only have one server. However, if interest in safety takes precedence over latency, Distributed Helios is highly recommended. In important elections, such as government elections, safety takes the highest priority. In this situation, it is worthwhile to sacrifice some user convenience in favor of robustness.

5.1 Future Work

There are several potential directions our current implementation of Distributed Helios could take in the future. Other than expanding the threat model to include Byzantine failures, the primary drawback at this point is efficiency, so improvements to the running time can greatly improve the viability of Distributed Helios. For example, Fast Paxos is a modification to Paxos that seeks to reduce end-to-end message delay in basic Paxos [Lam06]. By passing over phases in basic Paxos, overall running time can be greatly reduced. However, the client is forced to send its request to multiple destinations. In the same vein, another improvement to Distributed Helios would be the capability to select different variations of Paxos, such as Fast Paxos over basic Paxos, depending on the situation. This would greatly aid in various environments, where a different version of Paxos might contain an optimization to the protocol that would allow for superior efficiency. The Raft consensus algorithm [OO14] also shows promise. One of the foremost principles in the design of Raft is to be simpler to understand than Paxos, making its implementation theoretically easier than Paxos' implementation. However, all of the alternatives to Paxos utilize some sort of two-phase commit protocol at some point in their algorithms to accomplish consensus. In Distributed Helios, a large portion of the delay occurred during the two-phase commit section of the algorithm, meaning that using another protocol might not increase the efficiency by much. However, it may still be worthwhile to experiment with various protocols in an effort to reduce latency as much as possible.

Of course addressing Byzantine faults may compound this issue even further, but a wide variety of research efforts are being made to reduce latency in Byzantine protocols as well [KAD+09, DMPZ14, AGK+15].

From a broader perspective, real-world Paxos has been elusively difficult to implement properly. In the Google Chubby paper, the authors note that "[t]here are significant gaps between the description of the Paxos algorithm and the needs of a real-world system" [CGR07]. Even though there exists a large body of literature concerning fault tolerance and consensus, implementing the algorithm in a practical system remains complex and difficult. To convert Paxos into a real-world application, Google engineers had to transform a protocol that could have been written in a page of pseudocode into a multi-thousand line implementation. The proposed algorithms remain largely inaccessible despite their apparent simplicity, and building tools that would allow easier implementations of fault tolerance would go a long way to furthering this work.

Acknowledgements

Our sincere thanks to Ben Adida, the author of Helios, for his responsiveness and helpfulness to our (many) questions about Helios, and updating Helios to support Google's new authentication system.

This research is based on work supported by the National Science Foundation under Grant Number CCF-1018871 and the National Institute of Standards and Technology award O-60NANB13D165 to the University of California at Davis. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation or the National Institute of Standards and Technology.

References

[Adi08] Ben Adida. Helios: Web-based open-audit voting. In *USENIX Security Symposium*, volume 17, pages 335–348, Berkeley, CA, USA, July 2008. USENIX Association.

[AGK+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems*, 32(4):12, 2015.

[AKBW15] Claudia Z Acemyan, Philip Kortum, Michael D Byrne, and Dan S Wallach. From error to error: Why voters could not cast a ballot and verify their vote with helios, prêt à voter, and scantegrity ii. *USENIX Journal of Election and Technology and Systems (JETS)*, 3(2):1–25, August 2015.

[Ben06] Josh Benaloh. Simple verifiable elections. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop*, pages 5:1–5:10, Berkeley, CA, USA, August 2006. USENIX Association.

[Ben13] Josh Benaloh. Rethinking voter coercion: The realities imposed by technology. In *Proceedings of the 2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, pages 82–105, Berkeley, CA, USA, August 2013. USENIX Association.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, Berkeley, CA, USA, November 2006. USENIX Association.

[CCC+08] David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L Rivest, Peter YA Ryan, Emily Shen, and Alan T Sherman. Scantegrity ii: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes. *EVT*, 8:1–13, 2008.

[CGR07] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407, New York, NY, USA, August 2007. ACM.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[DMPZ14] Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. Bchain: Byzantine replication with high throughput and embedded reconfiguration. In *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*, volume 8878 of *Lecture Notes in Computer Science*, pages 91–106, Switzerland, December 2014. Springer International Publishing.

[Eps13] Jeremy Epstein. Internet Voting Security: Wishful Thinking Doesn't Make It True. `https://freedom-to-tinker.com/blog/jeremyepstein/internet-voting-security-wishful-thinking-doesnt-make-it-true/`, Apr. 3, 2013.

[FCC12] FCC. Eighth broadband progress report. FCC 12-90, Federal Communications Commission, Washington DC, USA, August 2012.

[FCC15] FCC. 2015 broadband progress report and notice of inquiry on immediate action to accelerate deployment. FCC 15-10, Federal Communications Commission, Washington DC, USA, February 2015.

[Ha.09] Ha.ckers.org. Slowloris HTTP DoS. `http://ha.ckers.org/slowloris/`, 2009.

[JRSW04] David Jefferson, Aviel D Rubin, Barbara Simons, and David Wagner. Analyzing internet voting security. *Communications of the ACM*, 47(10):59–64, 2004.

[KAD+09] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems*, 27(4):1–39, 2009.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[Lam06] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MJ13] Hein Meling and Leander Jehl. Tutorial summary: Paxos explained from scratch. In *Proceedings of the 17th International Conference on Principles of Distributed Systems*, volume 8304 of *Lecture Notes in Computer Science*, pages 1–10, Switzerland, December 2013. Springer International Publishing.

[Mye85] Brad A Myers. The importance of percent-done progress indicators for computer-human interfaces. *ACM SIGCHI Bulletin*, 16(4):11–17, April 1985.

[Nat15] National Conference of State Legislatures. Electronic transmission of ballots, July 2015.

[Neu13] Peter G. Neumann. Risks to the public. *ACM SIGSOFT Software Engineering Notes*, 38(1):20–26, January 2013.

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, New York, NY, USA, September 1993.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Berkeley, CA, USA, June 2014. USENIX Association.

[RBH+09] Peter YA Ryan, David Bismark, James Heather, Steve Schneider, and Zhe Xia. Prêt à voter: A voter-verifiable voting system. *IEEE Transactions on Information Forensics and Security*, 4(4):662–673, 2009.

[RJ08] Benjamin Reed and Flavio P Junqueira. A simple totally ordered broadcast protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 2:1–2:6, New York, NY, USA, September 2008. ACM.

[SFD+14] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J Alex Halderman. Security analysis of the estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 703–715, New York, NY, USA, November 2014. ACM.

[SJ12] Barbara Simons and Douglas W. Jones. Internet voting in the u.s. *Communications of the ACM*, 55(10):68–77, October 2012.

[TM03] Caroline J Tolbert and Ramona S McNeal. Unraveling the effects of the internet on political participation. *Political Research Quarterly*, 56(2):175–185, 2003.

[TPLT13] Georgios Tsoukalas, Kostas Papadimitriou, Panos Louridas, and Panayiotis Tsanakas. From helios to zeus. *USENIX Journal of Election Technology and Systems*, 1(1):1–17, August 2013.

[VAL04] Bernard Van Acker and Generaal Lemanstraat. Remote e-voting and coercion: A risk-assessment model and solutions. In *Proceedings of the Workshop on Electronic Voting in Europe Technology, Law, Politics and Society*, Lecture Notes in Informatics, pages 53–62, Germany, July 2004. Gesellechaft für Informatik.

[Wag13] David Wagner. Remote voting: What can we do? Keynote Address to the 2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections, August 2013.

[WP07] Adam Wierzbicki and Krzystof Pietrzak. Analyzing and improving the security of internet elections. In *ISSE/SECURE 2007 Securing Electronic Business Processes*, pages 93–101, Wiesbaden, Germany, 2007. Springer Vieweg.

[ZJT13] Saman Taghavi Zargar, Jyoti Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE Communications Surveys & Tutorials*, 15(4):2046–2069, 2013.