Exploring a New IoT Infrastructure

Paul Bramsen
Sam Kumar
Andrew Chen
Dibyo Majumdar

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-56
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-56.html

May 11, 2017

Exploring a New IoT Infrastructure

by

Paul Bramsen

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Kubiatowicz, Chair
Professor John Wawrzynek

Spring 2017

The thesis of Paul Bramsen, titled Exploring a New IoT Infrastructure, is approved:

Chair _____ Date _____

_____ Date _____

University of California, Berkeley

Exploring a New IoT Infrastructure

Abstract

Exploring a New IoT Infrastructure

by

Paul Bramsen

Secure channel work from paper originally by Paul Bramsen & Dibyo Majumdar

GDPFS work from paper originally by Paul Bramsen & Sam Kumar & Andrew Chen

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor John Kubiatowicz, Chair

Internet-connected devices are rapidly becoming ubiquitous. This so called Internet of Things (IoT) carries significant security implications thanks to all the sensors, actuators, and computers that make up the devices. If they are not properly secured, IoT devices enable malicious actors to spy on, digitally attack, and even physically harm victims. Unfortunately, many IoT devices are currently chock-full of security holes. We believe that this is largely the result of traditional network abstractions being a bad fit for the IoT. The Global Data Plane (GDP) is a new architecture for global IoT storage and communication that makes data the network's "narrow waist" by presenting users with secure single-writer append-only logs.

In this report, we present some improvements to the GDP and show that the append-only log is an abstraction that can underpin powerful applications which operate on mutable data. First, we present changes to some of the GDP protocols to improve the performance and security of the GDP. We describe these changes in detail and argue for their merit based on empirical data. Next, we present the Global Data Plane File System (GDPFS), a distributed filesystem that expresses mutable files on top of append-only logs and provides efficient data access within files. Because it is built on top of the GDP, the GDPFS has the potential to scale very well and run securely while giving application developers a traditional interface for managing data.

To Mom Bramsen, Dad Bramsen, Cory Bramsen, Maki Bramsen, Bedstemor Bramsen, Uncle Rich Prohaska, Nathaniel Barlow, Josh Ricafrente, Grace Han, Heather Caughell, Nardeen Dawood, Rebekah Inouye, Evan Pope, Risa Balcom, The Workshop men, Dan & Deb Goodson and all the other Berkeley Cru students and staff, The Berthas, The Bakers, The Strykers, and all my other friends and family—in Berkeley, Santa Barbara, and around the world—who have encouraged and supported me through my time as a UC Berkeley student and helped shape me into who I am today. Lastly, Leo the cat Bramsen for his uplifting playfulness and fluffiness and Chad the cat Bramsen for his affectionateness (when he is in his basket).

I can do all things through Christ who strengthens me. —Philippians 4:13 NKJV

Contents

List of Figures

List of Tables

Acknowledgments

We would like to thank the Swarm lab Global Data Plane group for their support throughout this work. In particular, we would like to thank Professor John Kubiatowicz for his architectural guidance as well as Eric Allman and Nitesh Mor for their help in understanding the current GDP implementation and architecture as well as providing us with GDP trace data. Eric and Nitesh were always happy to spend time resolving our confusions and we are indebted to them for their helpfulness. Additionally, we would like to thank Eric for aiding in the process of debugging and patching the GDP C client library and log daemon whenever we ran into bugs that we could not work around. We are thankful to Ken Lutz for helping us review our work and determine which results were most significant. We would also like to thank Professor Vern Paxson for his feedback on the network security portion of our work. Sam Kumar and Andrew Chen joined the GDP filesystem effort at an early stage. They co-designed and co-implemented many of the GDPFS features presented in this report. They also co-wrote the initial GDP filesystem paper that the filesystem chapter of this report is based on. Dibyo Majumdar co-designed GDP secure channels. He also co-wrote the initial GDP secure channel paper that the secure channel chapter of this report is based on. We are thankful for Sam, Andrew, and Dibyo's contributions and very much appreciate that they allowed us to reuse the material in this report. Finally, we would like to thank our faculty readers, Professors John Kubiatowicz and John Wawrzynek, for taking time to review this report.

Chapter 1

Introduction

1.1 The Rise of Ubiquitous Connected Devices

Over the course of the last decade, the number of devices connected to the Internet has exploded, surpassing the global population in 2012 and doubling over the next few years [24]. Cisco believes this number will be as big as 50 billion in 2020 [19]. Much of this growth is the direct result of the rise of the Internet of Things (IoT): the massive network of small Internet-connected sensors, actuators and processors that are rapidly becoming ubiquitous. Everything from your watch to your toaster can now Tweet [1].

This level of hyper-connectedness means that tomorrow we will be able to widely deploy systems that are in their infancy—if they exist at all—today. Systems like university campuses that can automatically detect and alert police to violent crime, energy efficient buildings that automatically learn when they can cut lights and HVAC systems, or cities that efficiently route traffic and automatically detect infrastructure failures. The list goes on and on. The IoT is fundamentally transforming how we think about how we interact with the digital world. No longer is your washer a device that cleans clothes with a small microcontroller attached. Your washer is a computer with a clothes-cleaning device attached. To paraphrase security expert Bruce Schneier: through sensors, actuators, and processing (i.e. the cloud), we are giving the Internet eyes and ears, hands and feet, and a brain. Essentially, we are building a world-size robot [11].

1.2 IoT: Internet of Troubles

Security Woes

This paradigm shift has significant implications in regards to the effects that misbehaving Internet-connected devices can have, particularly when those devices are misbehaving as a result of adversarial intervention. Not too long ago, Internet exploits would, at worst, disgruntle some academics and disrupt their research. Shortly later, the stakes got higher

as multi-billion dollar companies were built with the Internet at their core. The effects of Internet failures and attacks could now have significant economic impact. With the advent of the IoT, such attacks become far more likely and the consequences much more grave. Cheaply made IoT devices can often easily be hacked in great numbers and assembled into massive botnets which can be used to cause significant economic damage [28]. With its vast sensor network, this world-size robot has the ability to eliminate any shred of privacy we have in our modern world . Worst of all, it even has the ability to inflict physical harm—even death—with its actuators. When you think about the consequences of, say, an attacker finding an exploit in your Tesla and taking control of your car while you fly down the freeway at 70mph it is not hard to see just how high the stakes are.

Unfortunately, these issues are not simply the wild prophecies of academic doomsayers eager to dramatize their research. Every week now it seems we see new ways in which the IoT is used to cause harm [16, 27, 30, 37, 29]. Most of these attacks cause economic damage, however, we have also seen cases of physical harm. For example, attackers have taken down critical hospital systems—disrupting the hospital's ability to treat patients—in hopes of exacting a small ransom from the hospital in exchange for ending the digital onslaught [50, 52]. And this is just the tip of the iceberg. The IoT is in its infancy. As the IoT grows, so does the opportunity to perform increasingly sinister attacks.

Given the dangers we have outlined, it is of the utmost importance that our IoT devices are robust and secure. But, unfortunately, they are not. There are a number of reasons why this is the case.

For starters, the market incentives are wrong. There is ever increasing demand for smaller, cheaper devices. This leads to manufacturers churning out new devices at a frenzied pace which requires cutting corners—particularly in software—when developing devices. The end result is devices that have had little thought put into security. Currently, distributed denial-of-service (DDoS) attacks are one of the most common things done with hacked IoT devices. And, as of right now, consumers do not care. And why should they? As long as John and Jane Doe can watch the show they recorded on their Acme DVR there is little incentive for them to do anything about the fact that their device is part of a botnet attacking servers all over the world (they probably do not even know this is happening). Consequently, the Does continue to buy and use Acme DVRs. The Does are much more likely to care if their DVR is used to spy on them. However, it is still fairly uncommon for the average person to actually get burned by such an attack (attackers usually extort the rich and/or famous). Consequently, most people are unwilling to pony up the extra cash for a higher quality more secure device [12].

Even if Acme Corp does get flamed by the media for a security screw up, they have much less to lose than traditional Internet companies. Often, they do not have the same brand value to protect that drives companies like Apple and Google to go to great lengths to make products like iOS and Android secure for fear of losing users should a high profile exploit occur. IoT devices are often produced on thin margins by obscure companies that are not well recognized.

The Cloud's Failures

Another issue with today's IoT is the lack of proper infrastructure for storing and communicating data. Today, most companies developing IoT devices turn to large cloud providers like Amazon [2], Google [3], and Microsoft [4] to support their needs. There are many drawbacks to pure cloud IoT solutions [53]. We highlight a few of them here.

First of all, as we discussed in the previous subsection, the Internet is a scary place when it comes to security and privacy. Transferring data across the Internet for it to sit somewhere in the cloud, nearly always on a server controlled by another party, increases attack surface and the probability that data gets leaked. Another issue is data durability management. There is no way to be 100% sure that anything sent to the cloud will ever be completely destroyed. Thus, whenever possible, data that does not need to be sent across the Internet should not be.

Furthermore, as mentioned previously, the IoT is already huge and rapidly growing so scalability is a big concern. At the end of the day, the cloud puts all data at the edge of the network. As the volume of IoT data grows, so will the bisection bandwidth necessary to support the cloud. IoT application data is often ultimately consumed very close to where it is generated (e.g. Thermostat controlling HVAC system). Additionally, the current Internet was built to optimize downlinks in consumer connections. So far, this has made sense because most consumer Internet use consists of data consumption, not production (e.g. streaming video, downloading software, accessing image-heavy social media, etc.).

On a similar note, latency is an issue that has unique implications in the IoT. Actuators affect the physical world, possibly in irreversible ways, and when they are acting on sensor data it is critical that they get up to date information. Sending sensor data to a cloud server for processing, waiting for computations to happen, then waiting for the actuation command response to come back adds delay to the control loop that should be avoided if at all possible. Yet another reason that the cloud is not a good fit for the IoT.

1.3 The Global Data Plane: A New Architecture

The Global Data Plane [10, 40] (GDP) is a system being developed by the UC Berkeley Swarm lab. The GDP addresses the issues discussed earlier in this chapter by offering IoT developers a standardized authenticated time-series single-writer log abstraction that can run on top of untrusted hardware. GDP logs have globally unique flat names called GUIDs which are SHA-256 hashes of each log's metadata. This flat name structure gives the GDP implementation total freedom to optimize performance and provide durability by moving and/or duplicating data at will. Furthermore, location independence means that IoT devices can seamlessly switch from network to network, a valuable property for small lightweight devices that are likely to move.

GDP logs offer end-to-end object security on all log entries. To achieve this, GDP writers sign each new entry with an asymmetric key. The public key is included in the log metadata

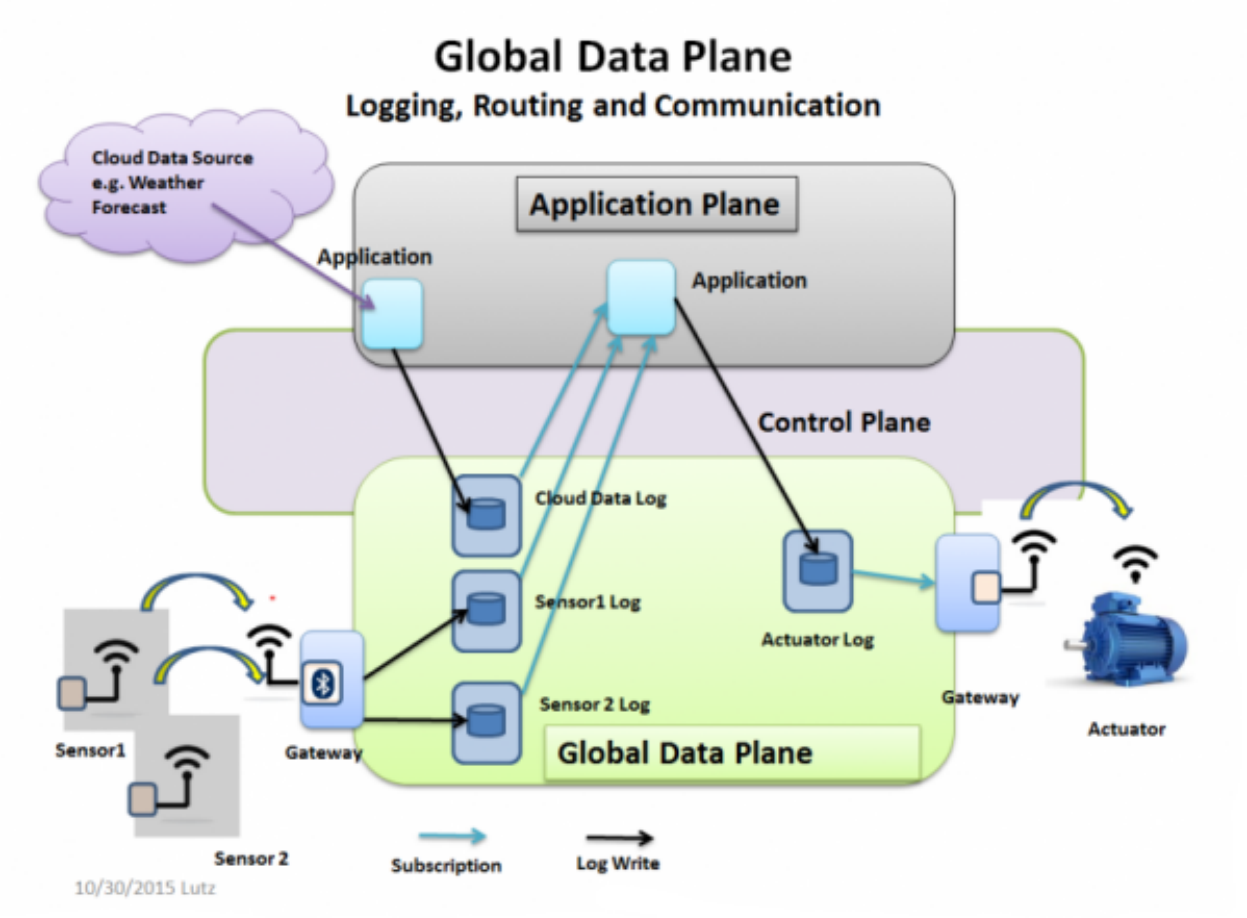


Figure 1.1: Global Data Plane Architecture [33]. Sensors actuators connect can connect to the GDP directly although often they will be lower power and connect through a gateway. Applications can live anywhere (often this means they are local however they can also be pushed to the cloud) and read from/write to GDP logs in order to communicate and store data.

(recall that a log's name is the hash of its metadata). Thus, with just a log's name, whenever we read from a log we can cryptographically verify that the data we got back was in fact the data that the log's owner wrote. To protect confidentiality, data can also be encrypted before being written to a log. The GDP can be thought of as having two primary components: a network component that handles moving data between clients and servers with opaque SHA-256 and an application component that handles all of the actual data storage and log management. Note the uniqueness and importance of the end-to-end security. When using cloud-based IoT solutions, security is usually only available to/from clients and the cloud; unfortunately, not from client to client. That is, cloud providers (who are also vulnerable to attacks as well as government sleuthing) have the opportunity to inspect and modify all

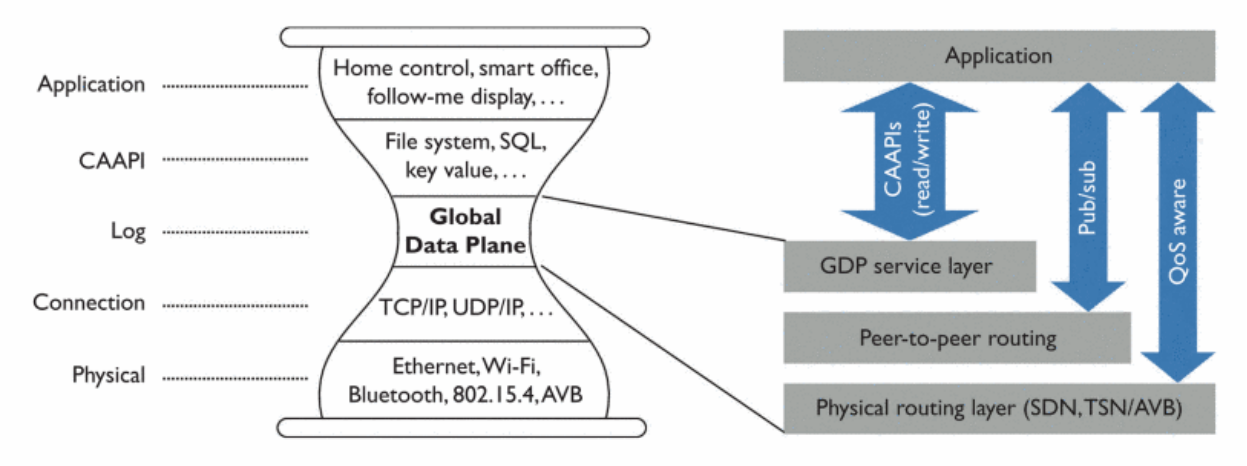


Figure 1.2: The GDP operates above the network layer. Applications can be built either on the raw GDP using the GDP client library or on top of so called Common Access APIs (CAAPIs) that offer traditional data storage and access abstractions to the user (e.g. GDP file system which is presented in chapter 3). [40]

data. In the world of the GDP, this is not the case. Security is achieved from the client writing the data to the client reading the data with no gaps in the middle, even if the read happens much later than the write.

The core components of the current GDP implementation are readers and writers, log servers, and routers. Readers and writers are client applications that read from or write to logs. Log servers are in charge of actually storing the data. And routers are responsible for moving requests around within the GDP (i.e. between a reader or writer and a log server). Figure 1.1 offers a sketch of this architecture. Further architectural details are covered in section 2.2. At the moment, the GDP is used by a number of research institutions around the world.

GDP log servers can be hosted anywhere; locally, in the cloud, or anywhere else a UNIX-style machine is available. This means that not all data has to be stored in the cloud which reduces attack surface and helps mitigate some of the aforementioned scalability issues. GDP data can be stored locally and then replicated to the cloud as necessary. Cisco proposes something similar in their work on "Fog" computing [14]. Keeping data local also gives users tighter control over the data's life. That is, if the data does not leave a user's possession and he wants to destroy it then he can do so and have real confidence that the data is actually gone. The moment anything goes to the cloud, it is impossible to know for sure whether it was ever destroyed. Tight data durability control is a big advantage for storing possibly sensitive IoT data.

Essentially, the GDP makes data the new "narrow waist" for IoT application developers. This is illustrated in figure 1.2. While some of the aforementioned IoT issues may require political and social solutions, the GDP provides a technical solution to others. By being

free, open-source software that offers a simple easy-to-use abstraction, the GDP eliminates the need for IoT application developers to come up with one-off solutions to achieve their goals. While it may be hard to change market pressures to favor security, the GDP's ease of use has the potential to get developers to start using a system that is secure by design.

1.4 This Report

In this report, we build on the GDP. First, in chapter 2, we look at improving GDP performance and security at the network level. Currently there is significant network overhead incurred when using the GDP, particularly when performing small log writes as will many IoT devices. We believe we will have achieved our goal if we can show that our protocol cuts GDP header overhead to a small fraction of what it is now.

Next, in chapter 3, we attempt to build a Unix filesystem using the GDP as the underlying data store. By doing so, our goal is to show that non-trivial applications can be built on top of the GDP. Furthermore, the Unix filesystem is a very well-known API that has the potential to make the GDP easily accessible to a range of application developer. If we can get our filesystem to perform complex tasks—for example, "make"-ing a significant codebase—with performance similar to that of the native filesystem, then we will consider our effort a success.

Chapter 2

Secure Channels

2.1 Introduction

As discussed, the GDP already has a significant userbase made up of researchers. Currently, there are a number of organizations currently looking at either trying out the GDP or upping the level of their involvement in GDP development. That said, the GDP is still quite new and is under active development. Consequently, there are a number of areas in which there are opportunities for performance enhancements to the GDP by rethinking certain protocols. Some GDP users would like to see certain aspects of the GDP improved. For example, Intel's IoT research division is quite interested in a version of the GDP with less header (details below). We addresses this issue and a number of related security issues in this chapter. In particular, we examine how ephemeral secure channels can enable log servers to maintain current security guarantees while reducing the number of asymmetric cryptographic operations required by end devices while also reducing side-channel attacks by developing a strategy for controlling access to log data even though this data is already protected by strong encryption.

2.2 Global Data Plane Network Protocol

In this section, we present the details of parts of the Global Data Plane architecture and protocols–particularly focusing on the portions that deal with the networking component of the GDP–with the aim of facilitating the subsequent sections in this chapter which describe the changes we have made to the architecture. Particularly the GDP router architecture which operates as an overlay network on top of TCP/IP and is described in a previous masters thesis [20].

The basic GDP data structure around which everything else is built is that of an append-only log. For the purposes of this chapter, however, we will focus on how data flows through the GDP. GDP communication is built around 256-bit globally unique IDs. Every GDP endpoint has a GUID that is the hash of the metadata of the log it names. This metadata

includes the log's public key which means that the GUID can be used to bootstrap the verification of various cryptographic authenticity and integrity guarantees. GUIDs exist in a flat namespace which allows separation between a log's locality and its name. An external naming service can be used to map more practical names onto log GUIDs. Such a structure allows for clean separation of mechanism and policy.

Since GUIDs are used as addresses for communication between endpoints and they exist in a flat namespace, GDP routing is tricky. The current implementation of the GDP router simplifies the task by forming a fully connected network. Endpoints (log servers, log readers and log writers) connect to the GDP by connecting to one of these routers. Routers let other routers know about what endpoints can be reached through them through advertisements of new endpoints and notification of withdrawal of endpoints. When an endpoint, say a log writer W, wants to talk to another endpoint, say a log server S, W sends its message to the router it is connected to, say $R1$. Now, one of three things may happen:

1. S is connected to $R1$. In this case, $R1$ sends the message directly to S.
2. $R1$ has received an advertisement for S from another router, say $R2$. $R1$ forwards the message to $R2$, which then forwards it to S.
3. $R1$ does not know about S. It responds to W with a negative acknowledgment to that effect.

Due to the fully connected nature of the routing layer, messages between endpoints must make at most 3 hops, as illustrated above. However there are plans to design a new router based on distributed hash tables since the current implementation is clearly not scalable. Fortunately, our modifications to the GDP protocol are agnostic to the underlying routing algorithms and should work just as well with the new router as they do with the current one.

GDP Protocol Data Unit

All communication on the Global Data Plane is related to either the writing/reading of records to/from a log (hosted by some log server) or else advertising the entry or exit of some log or client endpoint (advertisements are sent exclusively to/between routers in the routing layer). Log readers request specific records and log servers respond with acknowledgments that include the requested record. Log readers might also ask to subscribe to a log, in which case a log server sends log records as they become available. Log writers write signed records to log servers, and log servers send back write acknowledgments.

Since communication on the GDP has limited expressiveness, the GDP Protocol Data Unit has a fairly rigid format (Table 2.1). The Header contains the source and destination GUIDs along with a command or an acknowledgment. For instance, a log reader might send a `CMD_READ` to a log server requesting to read a particular log record, and the log server might respond with an `ACK_CONTENT` PDU with the corresponding record. When the log

0x03 (1 byte)	TTL (1 byte)	Reserved (1 byte)	Cmd/Ack (1 byte)
Destination (32 bytes)			
Source (32 bytes)			
Request Id (4 bytes)			
Signature Info (2 bytes)		Optionals Length (1 byte)	Flags (1 byte)
Data Length (4 bytes)			
Record Number (8 bytes, optional)			
Sequence Number (8 bytes, optional)			
Commit Timestamp (16 bytes, optional)			
Additional optional header fields (variable length)			
Log Record (variable length)			
Signature (variable length)			

Table 2.1: GDP Version 3 Protocol Data Unit (existing format)

reader connects to a GDP router, the router sends a `CMD_ADVERTISE` to all the other routers and when the reader disconnects, the router sends a `CMD_WITHDRAW` to all other routers.

We observe that the limited communication expressiveness of PDUs is a security advantage. It can be used to isolate weak IoT endpoints from the Internet at large by having the last hop of the GDP protocol run over some non-IP network such as Bluetooth LE or Zigbee.

Threat Model

One of the GDP's primary design goals is to be able to securely operate on untrusted hardware. That is, anyone should be able to host a router or log server. Log owners may choose to host their logs on their own log servers, but may also choose to host them on other log servers. Logs may be replicated over multiple log servers to improve availability.

Log records are encrypted when they are sent over the network in a PDU. The log server stores it in this form. Therefore, any reader must have the decryption key to read the contents of log records. Optionally, log writers sign the records they send, thus allowing a log reader or the log server to verify the authenticity of records. By storing a record number or timestamp with the record (that is signed over), log writers can protect against replay attacks.

However, the network is susceptible to a variety of side-channel attacks. A malicious router or log server could extract substantial information from the timing and length of records being written and read. We discuss this further in Section2.4.

Moreover, the network is highly susceptible to a number of Denial-of-Service attacks. A malicious endpoint could read any and as many logs as they wish from a log server, thus

Return Flow Id.	Timeout	ECDH Half-Handshake	Timestamp	Nonce	Signature
4 bytes	4 bytes	64 bytes	4 bytes	4 bytes	78 bytes

Table 2.2: Circuit Establishment Request/Response optional header field.

Channel establishment parameters are colored in gray and are only consumed by endpoints.

- **Return Flow Identifier** is the flow identifier this entity expects on the flow in the opposite direction. That entity will send this flow identifier on a return PDU for this circuit.
- **Timeout** is the timeout that the PDU sender is setting for this circuit. If the circuit is left unused for **Timeout** seconds, it will expire. An entity does not change this field if it is equal to or lower than its own supported maximum timeout.
- **ECDH Half-Handshake** refers to the two 256-bit point coordinates on the elliptic curve used for the ECDH handshake. (The curve used is decided in advance.).
- **Timestamp** is the time at which the circuit establishment request expires, and **Nonce** is a random value. These are used by endpoints to protect against replay attacks. An endpoint only accepts **Circuit Establishment Requests** if the time is earlier than **Timestamp**, and drops any request with the same **Timestamp**, **Nonce**, and endpoints as a previous request. Thanks to the **Timestamp**, endpoints need only keep track of the requests they've seen until the time is later than **Timestamp**.
- **Signature** is over the **Half-Handshake**, **Nonce**, **Timestamp**, and endpoints (source and destination). It prevents a man-in-the-middle attack on the shared secret exchange and is part of the replay protection mechanism when immediate-authentication is used.

overloading a log server. A malicious log server, on the other hand, could stop serving requests for reads and writes to the logs it stores. A malicious router could blackhole all PDUs sent to it, thus taking down that part of the GDP network. We address some of these threats in our design.

2.3 Security & Performance Architectural Enhancements

In this section, we discuss the changes we made to various GDP mechanisms and the rationale behind each change. Our goal is to improve both the security and the performance of the GDP for what we expect to be the common use case.

PDU Compression using Virtual Circuits

One of the major GDP use cases is to support low-powered devices and sensors writing their readings to a log as records (to be consumed at a later time). Each record may only be a few bytes long. However, each GDP PDU has at least 80 bytes of header information (see Table 2.1 for non-optional header fields). A major chunk of that is the destination and source name fields which take up 64 bytes together. Moreover, for verification of authenticity, all records must be signed. This signature is 78 bytes (assuming the default elliptic curve signature and SHA-256 digest are used). There are a number of ways we cut down on the PDU size.

We can avoid sending long endpoint names by creating a virtual circuit between source and destination. Essentially readers, writers, routers and log servers associate a per-hop flow identifier with a particular connection and send the flow identifier instead of the source and destination names. GDP routers store flow information as temporary state in lookup tables. When future PDUs come in that are part of that flow, the router simply looks up the flow ID in the lookup table and sends the PDU out the corresponding port. This is much like the way that IP routers store routing information in lookup tables.

Secure Channels

In addition to establishing a virtual circuit, the two endpoints can also choose to establish a secure channel by deriving a shared secret and using that to authenticate and encrypt the body and record number of all future PDUs (using something like AES-GCM [48] which is an efficient authenticated-encryption protocol). The log server can then be sure that all communication it receives over the circuit is from the original writer/reader. That is, it is not spoofed.

We need to make a number of changes to the current PDU header format to enable circuits. Source and destination fields are now optional. We add a flow identifier optional field. For establishment of a circuit, we require a few different parameters which we pull together into a circuit establishment request/response optional field (Table 2.2). We demonstrate the circuit establishment procedure in Figure 2.1. Lastly, Table 2.3 shows what the GDP PDU looks like after our changes. Essentially, this is a new version of the protocol/PDU (version 4) with breaking changes.

We note that having per-hop flow identifiers instead of one for each half of the circuit complicates our circuit establishment procedure. We decided to do this because with the per-hop approach, entities choose what flow identifiers they want on their hop for a circuit, and therefore there's no chance of conflict between flow identifiers for different circuits. Furthermore, note that the entire circuit establishment process piggybacks on top of what was previously the only way to communicate. This means that if, say, a client is talking to a log server that doesn't want to establish a circuit or the client is making a one-off request, then it does not make sense to establish a circuit. The communication can continue in the old way with minimal to no extra overhead.

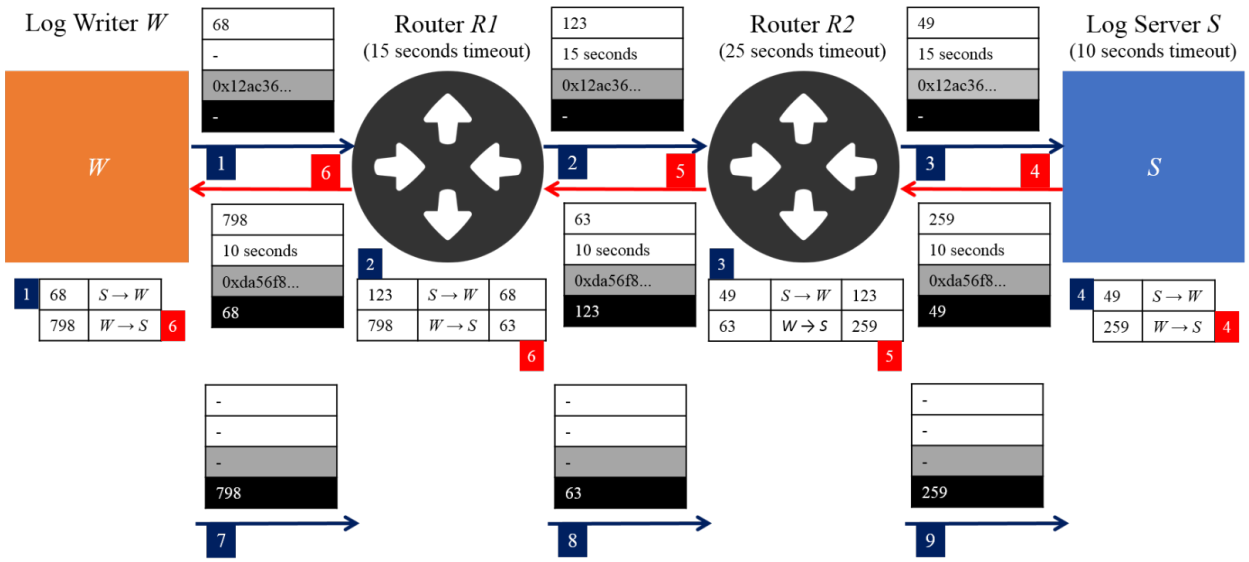


Figure 2.1: Illustration of circuit establishment procedure

The maximum timeout value supported by each entity is shown below their name. The flow tables used to maintain circuit state are shown below the entities. Flow table entry creations and PDU send operations (arrows) are grouped together and their order is noted in the blue and red colored boxes. We describe these operations below. Some of the Circuit Establishment Request/Response fields and flow identifier are shown alongside the PDU sends. (White: Requested Flow Identifier, Timeout / Gray: Secure Channel Setup Parameters / Black: Flow Identifier)

Consider a log writer W that wants to establish a circuit with a log server S.

1. W selects a random value, 68 as the flow identifier it wants for the return flow, places that in its flow table, and sends that along to its router, $R1$, along with the secure channel setup parameters. Note that at this point, source and destination addresses still need to be sent.

2. $R1$ first ensures that it can get to Log Server S. It then selects its own random return flow identifier, 123, and associates it with the half-circuit $S \rightarrow W$ in its flow table. When it receives a PDU with flow identifier 123, it will know that it is for this half-circuit, and will know to forward it with the flow identifier 68. $R1$'s maximum timeout value is 15 seconds, so it sends that along with the return flow identifier 123 to S's router, $R2$.

3. $R2$ sees that it is directly connected to S. It follows the same procedure as $R1$. Its maximum timeout value is 25 seconds which is more than the timeout value on the PDU, and so it does not change that value.

4. S decides that it does want to establish a circuit with W, takes note of the secure channel setup parameters, and places 49 in its flow table. It selects a random value, 259 as the return flow identifier and places it in its flow table. S then sends that with its own half of the secure channel setup parameters back to $R2$. Since its maximum timeout value is less than that on the incoming PDU, it sends its maximum timeout value as the timeout value on the PDU. It also knows to send 49 as the flow identifier. Note that at this point, source and destination names can be dropped.

5. $R2$ sees 49 in its flow table and knows how to route it. It selects a random return flow identifier, 63, and associates that with the half circuit $W \rightarrow S$ in its flow table. $R2$ then sends that over to $R1$ along with the flow identifier 123 it has stored in its flow table for this half circuit. It does not change the timeout value since its already below its maximum timeout value.

6. $R1$ follows the same procedure.

At this point, the circuit has been created. W and S can send PDU's to each other without including the source and destination names. Their PDU bodies are also encrypted with the shared secret they established.

0x04 (1 byte)	TTL (1 byte)	Reserved (1 byte)	Cmd/Ack (1 byte)
Request Id (4 bytes)			
Signature Info (2 bytes)		Optionals Length (1 byte)	Flags (1 byte)
Data Length (4 bytes)			
Destination (32 bytes, **optional**)			
Source (32 bytes, **optional**)			
Flow identifier (4 bytes)			
Circuit Creation Params (158 bytes, optional)			
Record Number (8 bytes, optional)			
Sequence Number Number (8 bytes, optional)			
Commit Timestamp (16 bytes, optional)			
Additional optional header fields (variable length)			
Log Record (variable length)			
Signature (variable length)			

Table 2.3: GDP Version 4 Protocol Data Unit. This is the new version of a GDP PDU. Changes are in bold. Note that the order of some of the fields has changed.

Note that while we believe that most of the time it makes sense to establish a secure channel and a virtual circuit together, it is possible to create one or the other independently. A secure channel can be established without a virtual circuit by setting the timeout in the establishment request to 0. A virtual circuit can be established without a secure channel by setting the signature to 0.

Handling Replay Attacks

Let us consider the security implications of the one way handshake. For a moment, ignore replay attacks (we will come back to them). Without replay attacks, a client can easily authenticate with a secure channel establishment request. As the protocol requires, they just sign the request. The key used to sign the message is either 1) the key of the owner of the log that is being accessed or 2) the key of some other client along with a certificate showing that the log owner has granted access to the client establishing the secure channel.

Now we consider how replay attacks change things. We argue that it turns out replays actually dont matter for idempotent log operations such as reads and writes. This is because, by definition, idempotent operations do not cause any state change when replayed. Furthermore, even if the idempotent operation causes data to be returned, fate sharing tells us the attacker does not learn anything new by replaying the operation. In order to capture the sequence of messages they plan to replay in the first place they had to be eavesdropping on the network. Any information they can gain through replays has already flowed by so the attacker learns nothing new.

To make this clearer, let us think about how this comes in to play with reads and writes. First, consider writes. Thanks to the idempotent nature of a GDP log write, it doesnt matter if the write is replayed by an attacker. In fact, the attacker is helping us out by doing the job of the log migration/consistency/durability service and filling holes. Now consider reads. As with writes, reads are idempotent. The difference is that with reads what matters is the response, not the request. It is true that with single message authentication protocol an adversary can replay a read and get back the encrypted response. However, fate sharing indicates that this does not matter. An eavesdropper who can record a read request can also record the servers response to that request (which must be encrypted if confidentiality is required). So the eavesdropper doesnt learn anything new by replaying the read request.

For non-idempotent operations, however, things are different. For example, subscriptions. If an attacker can replay a sequence of messages to get a subscription to a log, then they can get timing data which essentially leaks information to the attacker. To handle non-idempotent operations, we require what we call immediate-authentication. What immediate-authentication means is that the log-server knows that they are talking to a live client and not an attacker replaying a previous sequence of messages. A client requests immediate-authentication from a log server with a new command, `CMD_IMMEDIATE_AUTHENTICATION`. This command can only be sent as part of a secure channel (either within a secure channel construction message or else in an existing secure channel). The server responds to this command with a nonce. The client must include the MAC of this nonce with the shared

secret as an additional optional header field in their next message. Once the log server verifies the MAC, it sets a flag indicating that immediate-authentication has been established and allows non-idempotent operations to proceed.

Cumulative signatures

Secure channels also allow us to reduce the number of signatures that have to be generated by allowing a log writer to avoid signing every record. By signing the first record they send over during/after circuit creation, a log writer verifies its identity to the log server. The server no longer needs to verify subsequent signatures it receives over that circuit since the PDUs are sent encrypted with the shared secret established during circuit creation.

However, a potential reader might still want to verify the authenticity of the records without having to trust the log server. So, the writer must keep sending signatures periodically, say, every 100 records. And, the signature must be over all records written since the last signature.

Essentially, this means that the log server can buffer a set of writes that are not yet signed without being concerned about denial of service attacks. Without a secure channel providing identity and authenticity, buffered unsigned writes would be an obvious resource exhaustion attack vector. Again, it is important to note that the data only becomes officially committed to the log once the log server receives a signature over the chain of thus far unsigned log records.

We achieve this through iterative hashing. Given records $r_1..r_n$, our choice of hash function H and signing function S, the algorithm we use to obtain our signature s is

$$h_0 = \text{“”},\ h_k = H(r_k||h_{k-1})\ \forall k \in 1..n$$

$$s = S(h_n)$$

The size of n used above is a function of the frequency at which new records are created and written. If records are sent infrequently, n should be small so that a subscriber does not have to wait long to verify the authenticity of the most recent records.

The writer must also necessarily sign the last record it sends over the circuit, so there exists a cumulative signature for every record. Note that since a fixed-length hash function is used, there is no need to worry about ambiguity due to the concatenation.

Access Control

The creation of a secure session during circuit establishment means that all future communication on the circuit is authenticated. It also means that the log server can verify the identity of the reader or writer (henceforth, accessors) from the signature sent as part of the circuit establishment request. But how can the log server verify that the accessors should be allowed to access the log? As we discussed earlier, all data in the log is encrypted, but a number of side- channel attacks are possible.

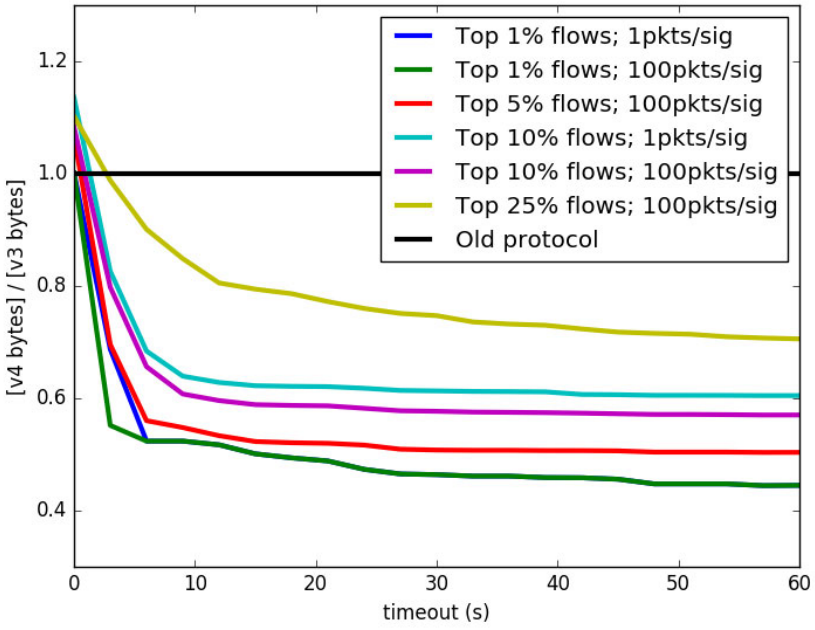


Figure 2.2: Overhead ratio (new version / old version) vs. circuit timeout value for different percentages of flows using circuits, and different signature frequencies (calculated based on trace). *A ratio of 1 means no increase or reduction in overhead.*

If the accessor is also the log owner, this is trivial. The owner has the private key corresponding to the log's public key which is stored in the log metadata. The log server can verify the circuit establishment request signature against this.

Otherwise, we require that the log accessor obtain a certificate from the log owner certifying that the accessor (ie. the entity with the accessor's public key) may read from or write to the log. The certificate may optionally have an expiry date set by the log owner. The log accessor can then send this certificate as part of their circuit establishment PDU (and include it as part of what is signed over).

We note that a log owner may choose to disable this check and essentially make the log public.

2.4 Evaluation

We evaluated our proposed modifications to the GDP architecture using 3 approaches: analyzing a trace of existing GDP traffic, considering our modifications from a theoretical perspective, and by working on a prototype. In the following section we describe our findings.

	gdp-01 (1.5h)	gdp-02 (41h)	gdp-03 (41h)	gdp-04 (41h)	Total gdp-{02,03,04}
Total Bytes (L2 traffic and up)	885 MB	93.1 MB	194 MB	391 MB	678 MB
Total GDP Bytes	829 MB	87.8 MB	182 MB	312 MB	582 MB
Total GDP Bytes % (of total bytes)	93.6%	94.3%	93.8%	79.7%	85.8%
GDP Client/Server Data Bytes	557 MB	1.04 MB	10.3 MB	132 MB	144 MB
GDP Client/Server Data Bytes % (of total GDP bytes)	67.2%	1.32%	5.66%	42.3%	24.7%
# GDP PDUs	1.58m	31.7k	74.1k	760k	866k
Average PDU size (including header)	525 B	2,367 B	2,456 B	411 B	672 B
# GDP log reads	1.31m	2.76k	1.53k	316k	321k
% GDP log reads w/ signature	99.9%	0.00%	2.48%	99.6%	98.3%
# GDP log writes	1.15k	2.99k	23.8k	1.42k	28.2k
% GDP log writes w/ signature	92.7%	50.6%	99.3%	0.56%	89.2%
Reads per write	1,140	0.922	.0643	223	11.4
Client ↔ client traffic %	98.8%	2.38%	9.36%	72.0%	41.9%
Client ↔ router traffic %	0.138%	2.06%	10.8%	0.182%	3.79%
Router ↔ router traffic %	1.02%	95.6%	79.8%	27.8%	54.3%

Table 2.4: GDP trace data. Note: we analyzed approximately 1.5 hours worth of data for `gdp-01` and 41 hours of data for `gdp-{02,03,04}`. All three 41-hour traces were collected during the same time window. To avoid combining apples and oranges, we exclude the `gdp-01` data from the Total column.

Trace Analysis

GDP log servers, routers, and clients are currently-deployed in and used by several research labs around the world. Consequently, there is already a fairly large amount of GDP traffic being generated. We were able to collect a trace of the network traffic flowing through several

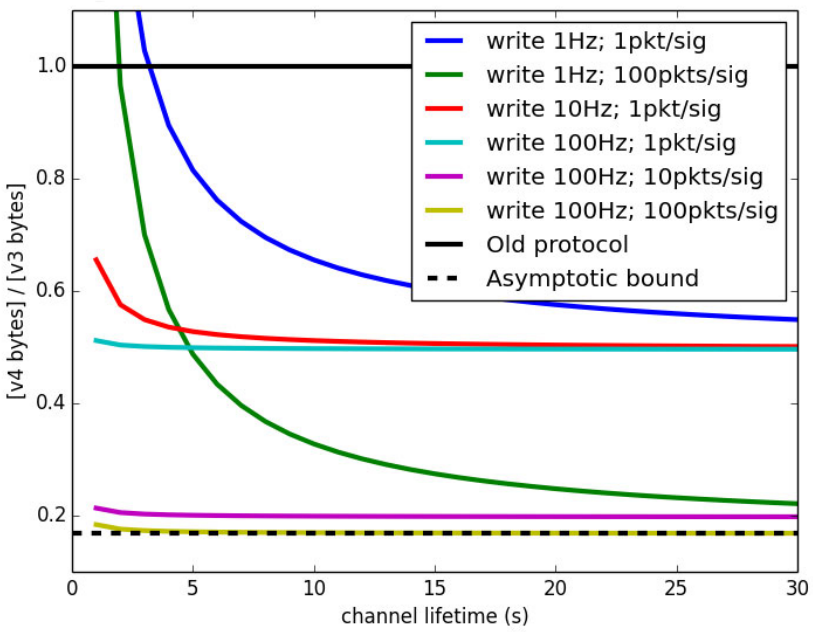


Figure 2.3: Overhead ratio (new version / old version) vs. circuit lifetime for different frequency of writes, and different signature frequencies (calculated based on trace). *A ratio of 1 means no increase or reduction in overhead.*

currently-deployed GDP routers, designated `gdp-{01,02,03,04}`. We analyzed 1.5 hours of traffic data from `gdp-01` and 41 hours from `gdp-{02,03,04}`. The purpose of this analysis was 2-fold: 1) to get a sense of what the current GDP traffic profile looks like (summary in Table 2.4) and 2) determine what sort of impact we could expect our modified version of the GDP protocol to have on the observed traffic (results in Figure 2.2). Below we describe our procedure for analysis and highlight some of the interesting things we found.

Analysis Data Generation Methodology

The GDP trace data was collected using tcpdump and stored in a pcap file. We pulled the dump in to Python using Scapy [5], an open-source library for manipulating and decoding network data. Since the current version of the GDP is implemented as on overlay network on top of TCP, we reassembled the TCP streams and then parsed them in to Python object representations of PDUs. This made extracting data (see section 2.4) and analyzing our findings as easy as manipulating a collection of Python objects. In particular, this allowed us to reconstruct PDU flows and analyze the impact of our modified protocol.

One challenge we faced when parsing PDUs was determining where the PDUs started. Since we captured data live, some of our TCP streams started in the middle of a PDU. To deal with this, we ended up coming up with a heuristic to match the start of PDU headers

which included looking at the PDU version number and the zeroed out reserved byte. All in all the flexible, user-friendly nature of Python made developing and debugging our analysis code easy. When there was a problem, it was very easy to drop into the Python REPL and quickly identify the issue. While not polished, we believe someone doing a similar study of the GDP network layer could learn from and possibly reuse chunks of this code. It can be found on Github [6].

Yet another issue we had to overcome was the limited computational resources we had at our disposal. We had conflicting goals of trying to process a large amount of data and not wanting to spend more time than necessary optimizing our data analysis code. Ideally, all data from a single trace could be stored in memory at once. The in-memory data representation used by Scapy added to our challenge by further bloating the data size. None of our computers were going to be able to handle this large volume of data. Thankfully, we ended up finding a solution in Google Cloud Platform [7]. We were able to rent an Ubuntu machine with 52 GB of memory. This was sufficient for our needs. While we were able to get $200 of student credit for free, we only actually used a few dollars. It is amazing how cheap it is to rent computational resources today!

Observed Trends

Table 2.4 shows a summary of some of the interesting data that came out of our analysis. Before trying to extract meaning from the data, it's important to observe that there is a high level of variation in the data from router to router (*e.g.* overhead, log reads per write, average PDU size). Most likely, this is a result of the GDP currently having a fairly small number of users. Since there is a limited amount of traffic flowing through each router, observed behavior can easily be influenced by a couple dominant clients (*e.g.* a flaky log server that regularly joins and leaves the network generating a large number of advertisements and, consequently, proportionally more routing traffic). As a result, any inferences made from this data must be taken with a grain of salt. That said, there are a few trends that stood out clearly enough that we felt they were worthy of mention.

First of all, it is clear that using the GDP incurs significant network overhead. In the best case scenario (`gdp-01`), we see a cost of 28.2% over a standard TCP/IP stream. This makes it clear that work must be done to reduce overhead if the GDP is to be scalable.

We also found it interesting that each of the routers tend to be dominated by either read traffic or write traffic. This lends credence to our theory that each router's traffic characteristics are dominated by a small number of clients.

Traffic Replay Results

Our primary reason for collecting a trace of GDP traffic was to determine how our modified version of the protocol would have impacted overhead for existing GDP traffic. To this end, we took the Python PDU objects that we had reconstructed and reassembled them into GDP communication flows. We then replayed the flows, calculating proportional header

overhead vs circuit timeout for various PDUs per signature rates. We limited our evaluation to the top X% of flows that benefited most from our changes for various levels of X. We feel justified in doing so because our changes only benefit flows with consistent traffic and were never intended to be applied to, say, one-off reads or irregular massive writes. The results are presented in Figure 2.2.

Theoretical Analysis

As discussed in Section 2.3, our protocol modifications bring security advances over the previous version.

Primarily, we greatly reduce the opportunity for side-channel attacks. Currently, there is no access control on logs. Data protection relies completely on encryption. This opens the door to attacks that glean information from timing or traffic volume. For example, an attacker might learn the name of a log that, say, actuates a user's bedroom lights, subscribe to the log, and infer when the user was home/leaving/going to bed based on observed traffic. Since our access- control method requires that the log owner give people permission to fetch data from a log, such attacks will no longer be possible.

Additionally, we further increase overall system security by adding channel security on top of object security. Once a channel has been established a client or log server might consider sending no-ops down the channel to further throw off anyone who might be trying to snoop on GDP traffic timing.

Importantly, we gain these security advances while increasing performance (discussed in 2.4) without giving up on any of the existing security guarantees. It's still possible to get a proof of integrity for any log entry (albeit at the price of a slightly increased delay).

Simulation Results

Of course, we also hope that our modifications will reduce the required overhead to use the GDP, increasing appeal for end users. To help us better understand how our changes might affect various traffic patterns, we present a plot showing the relative overhead of relevant headers vs. the lifetime of a channel (*i.e.* how much time passes before the channel needs to be reestablished). We consider various traffic patterns by plotting several levels of write frequencies (*i.e.* how many writes are performed per second) and signature rate (*i.e.* once a channel has been established, how often must a client compute an asymmetric signature). Figure 2.3 shows our results. Encouragingly, the results in Figure 2.3 seem to correspond with what we saw in Figure 2.2. It's also interesting that savings converge to their asymptotes fairly quickly. This supports the notion that flow state can be treated as soft state with a fairly small timeout and without giving up significant benefit.

2.5 GDP Secure Channel Related Work

Zhang *et al.* made the case for why the cloud model doesn't fit IoT applications [53]. Mor *et al.* also make a similar argument that also provides a justification for an initial analysis of the Global Data Plane in particular [40]. These serve as motivating arguments for our work.

The Global Data Plane has a unique architecture. This means that our contributions are comparable to previous work done on other projects but adapted to work for the GDP's unique set of properties and constraints. In this section we discuss these other works.

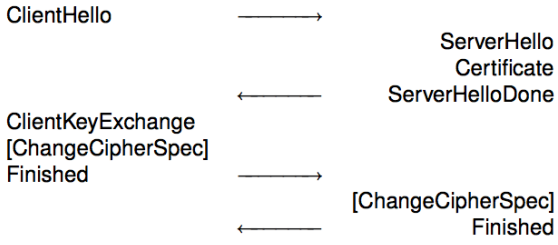


Figure 2.4: DTLS channel establishment [38]

Datagram TLS (DTLS) [44, 38], a version of TLS with the minimal number of changes necessary to operate over a datagram protocol, is similar to our secure channel establishment protocol. However, by taking advantage of domain specific knowledge—for example the idempotent nature of certain operations as discussed in the subsection on handling replay attacks that can be found in section 2.3—we are able to establish channels with fewer message exchanges than possible with DTLS which needs to be more general. The DTLS channel establishment protocol is illustrated in figure 2.4 and can be compared to figure 2.1 which illustrates GDP secure channel and circuit establishment.

We look at alternate approaches using symmetric cryptography. TinySec [25] is a link-layer security architecture specifically built for low-powered devices, but it requires that keys are exchanged ahead of time. Eschenauer *et al.* [18] discuss an intricate scheme for the offline distribution of keys to low-powered devices ahead of time that are later used for symmetric encryption between devices. This provides authentication, but requires that a large number of keys be stored on the sensors (and replenished periodically). The μTESLA protocol presented by Perrig *et al.* [41] requires that each device is issued only one key initially, and keys don't have to be replenished. Periodically, a new key is generated by applying a pseudo-random function on the previous key making this scheme more computationally expensive than the previous scheme. We note that in each of these cases, the key exchange problem still remains. Since we require that any log writer or reader be able to access a log on a log server without prior communication on other channels (provided they are authorized to access the log), distribution of keys ahead of time is not feasible.

Kerberos [54] is a protocol that allows nodes communicating on an insecure network to prove their identity to one another by obtaining tickets to access various network services from a centralized trusted Key Distribution Center (KDC) service. Such a service does not make sense for the Global Data Plane since it's completely decentralized and all entities on the GDP are, by design, untrusted.

We look to the asymmetric cryptography space to solve the key exchange problem. The

Sizzle architecture [22] brings end-to-end encryption and authentication using an elliptic curve cryptography-based scheme instead of RSA which requires significantly less computation than RSA [21]. This is the least computationally intensive among asymmetric cryptography schemes. We therefore go with this approach in our work.

One other approach that we considered is offloading encryption and authentication to a more powerful network gateway device such as a desktop computer that can then perform asymmetric encryption [13, 17]. However, device communication with this gateway device is in the clear. This is not ideal.

Chapter 3

Global Data Plane File System

3.1 Introduction

The Global Data Plane (GDP) [10] provides a substrate for storing data that can run securely on untrusted hardware. The basic primitive that it exports to applications is a verifiable, append-only, single-writer log. With such a log interface, it is possible to provide atomicity and consistent replication with little overhead (where a log entry is the basic atomic unit). Traditionally, systems that aim to provide rich data semantics first build the functionality that they need, and then layer schemes to enforce such data semantics on top of the functionality. Database Management Systems are a good example of this. First the system implements its main functionality, namely allowing the creation of relations and the retrieval of data from them. Then, to achieve the desired data semantics, it may employ schemes such as Write-Ahead Logging [39], Two-Phase Locking [42], Key-Range Locking [32], etc. on top of the existing functionality in the system.

In contrast, the GDP provides a primitive, namely a single-writer log, for which these desirable properties can be supported more easily than in a database. Because the logs are *append-only*, replicas can be kept consistent without a write-ahead logging scheme. Furthermore, it is easy to verify authenticity of the logs because each entry will have a fixed signature at write time.

As data-collecting sensors become pervasive, the GDP is obviously much more well-suited to storing their data than a traditional DBMS, for the simple reason that a DBMS must support much richer data semantics than are needed for this task. Beyond being more efficient, the GDP enables a new paradigm for building complex distributed data systems; rather than first building the system and then adding locking or logging schemes to achieve the desired consistency and durability properties, one may choose to build a system on top of GDP, which can cheaply achieve these properties, in such a way that some of these properties shine through in the final system.

The main difficulty in this approach is one of the advantages mentioned earlier: the append-only nature of logs in the Global Data Plane. In addition to being well-suited for

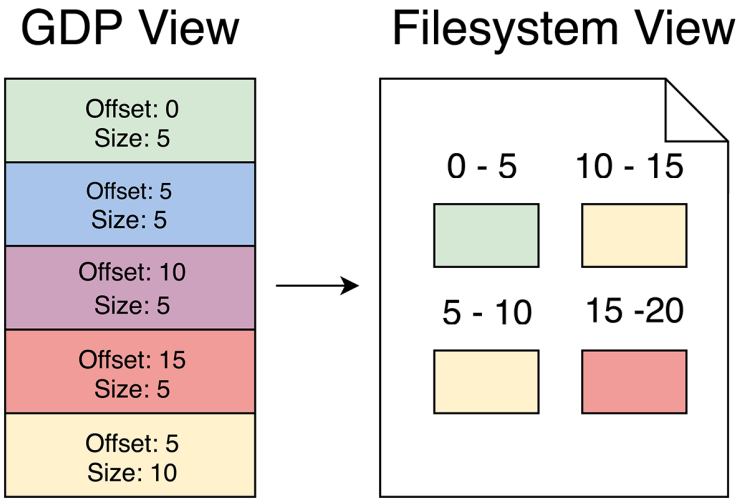


Figure 3.1: Basic layout of a GDPFS file

use cases in which data is only added, and never removed, the GDP is aimed more generally at providing secure data transport for cloud services. One natural way the GDP could be used to this end is as support for a log-based messaging scheme. Apache Kafka [31], for example, is a publish-and-subscribe messaging service based on a log. However, it is also desirable to provide richer semantics such as data mutability on top of the GDP. In particular, we wish to provide the abstraction of a shared file that can be written by its owner, and read by other entities[1]. The append-only nature of logs in the GDP makes this a nontrivial task; while adding new data to a file is easy, our goal is to record *mutations* to data in an append-only log while simultaneously providing a means to efficiently retrieve that data.

In this chapter, we present the Global Data Plane File System (GDPFS)[2], a traditional distributed filesystem built on top of the GDP log abstraction. Our motivation to do this is twofold. First, we would like to open the atomicity and security properties of the GDP to a wider variety of use cases. Second, we would like to evaluate the effectiveness of the GDP as a primitive for building complex distributed systems, based on both the performance of the resulting filesystem and our experience in creating it.

3.2 Overview of the GDPFS

As with most filesystems, the basic unit of storage in the GDPFS is a file. All other filesystems constructs—such as directories, symbolic links, etc.—can be built on top of basic files.

[1]The single-writer restriction stems from the fact that the GDP provides a *single-writer* log as its main abstraction.

[2]The GDPFS code is open-source and available on Github [8].

Thus building a basic file which could be efficiently read from and written to in a way that guaranteed desired ACID semantics was our primary focus when building the GDPFS. The design we settled upon is described in this section. The implementation details of this design are described in Section 3.3.

Each file in the GDPFS is backed by a single log in the GDP. Files are modified by appending entries to the log specifying a range of bytes and the new bytes that that range is to take on. This means that new writes can eclipse old writes simply by specifying an overlapping offset and size. Since the GDP accepts appends to logs atomically, all writes to files are guaranteed to be atomic.

Reads are satisfied by scanning the log backwards and retaining bytes that fall within the range of bytes being read as they are seen. Note that it is essential that the log be scanned backwards. The same logical byte may have been written multiple times but we are interested in the most recent version. See Figure 3.1 for a depiction of how this works.

While the previously described system is correct, it is not efficient. Writes can be done in time constant in the number of log entries and logical file size. Unfortunately, it may be necessary for a read to examine every log entry (for example when the read needs data that is in the first log entry) so reads can take time proportional to the length of the log. In order to solve this issue, we implemented a number of caching and indexing strategies.

As a first step toward performance, we cache reads and writes on the local filesystem. This gives us gains on reads, in two ways. First and most importantly, if we have a cache hit we can avoid going to the GDP at all, which completely eliminates the need to hit the network even once. This advantage is magnified if multiple network accesses would have been necessary in order to locate and retrieve relevant blocks. Second, we no longer spend CPU time reconstructing blocks based on log entries. We use a write-through asynchronous policy, allowing readers to be kept up-to-date without incurring the network latency on the writes. See Section 3.3 for additional details.

If the bytes of interest are not in the cache, then clearly we have to hit the GDP to retrieve the necessary data. This once again brings up the problem of reads taking time linear in the size of the log, which is not acceptable performance. We solve this through use of a special tree index structure called a FIG Tree, described in detail in section 3.3. A FIG tree allows bytes to be found in time *logarithmic* in the size of the *file*, much better than *linear* in the size of the *log*. The FIG tree is updated on writes and periodically[3] pushed to the GDP in a special checkpoint log entry. Updating the FIG tree adds a small cost to writes, no worse than logarithmic in the size of the file. Since we expect frequent access of FIG tree indexes for files in active use, we keep an in-memory partial representation, pulling in subtrees as they are accessed. Furthermore we cache the checkpoint log entries as they are read in since it is likely that related subtrees will live in the same checkpoints. Figure 3.2 illustrates how the system fits together.

[3]Due to time constraints, our current implementation does not checkpoint files periodically; rather it writes the checkpoint entry when the in-memory representation of the file is discarded.

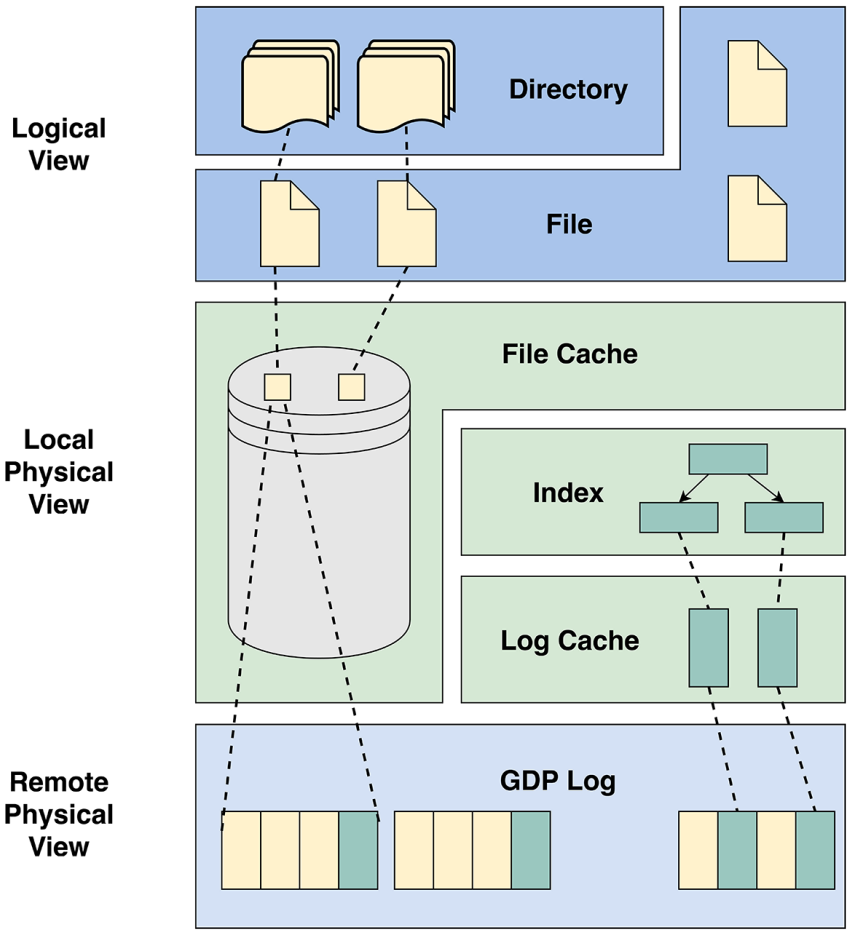


Figure 3.2: Overview of the GDPFS structure. Data flows between vertical boundaries.

3.3 Implementation

Since our goal was to build a working performant prototype of the GDPFS while being efficient with our time, we had to carefully choose tools to speed development while incurring minimal performance overhead (e.g. FUSE). This section describes the process we went through in selecting these tools and gives an in-depth explanation of the techniques we used to achieve our performance goals.

FUSE

The implementation of the GDPFS is entirely built upon the Filesystem in Userspace library (FUSE) [43]. This library enables programmers to implement entire filesystems in userspace and works by redirecting all syscalls issued to a FUSE mounted filesystem to a special user level process.

For example, when a user tries to open a file, the FUSE library in kernel space will forward this syscall to the handler running in our user level process. In this handler for open we first enforce the permissions on this file and then tell FUSE to make all subsequent requests to this specific file with a file handle that we specify. Finally, FUSE will return the results of a successful open with a process specific file descriptor that is different than the file handle that we have specified. Later, when that same process makes another syscall with this file descriptor, the FUSE library will recognize this (PID, FD) pair and call the correct user level handler with the appropriate file handle which we specified on the open. Upon returning from this user level handler, the results of that syscall will be shuffled back into the kernel and then finally to the process which issued that syscall.

We chose to build our filesystem on top of FUSE ultimately because it allowed us to avoid writing any kernel level code. This way it was much easier rapidly iterate on our filesystem without worries of a bad line of code bricking our computer. However, building a filesystem this way is not without its limitations. For example, in FUSE each filesystem syscall incurs an additional pair of user space to kernel space transitions. This makes FUSE inherently slower than filesystems built in the kernel. Also, because our FUSE-based implementation is running in userspace, it is impossible to directly access the block-store. Instead, any sort of disk accesses we make must go through a separate filesystem. Although our current prototype of the GDPFS is built using FUSE due to its ease of use, a more stable and permanent version would foreseeably be written directly in the kernel.

File Caching Layer

Reads to files in the GDPFS first check a cache stored on the local disk. The underlying implementation of this cache is a file on the local filesystem which mirrors the logical view of the file on the GDPFS.

The way the cache works is very simple. Each read first checks to see if it can be satisfied by the file cache. If this is not possible then we must go to the remote log to find out the contents of this range. Upon reading from the remote log we would populate the appropriate parts of the file cache. On writes, we simply populate the appropriate portion of the cache and issue an asynchronous write to the GDP before returning. Because of this, all writes can return without waiting for any network I/Os[4].

An alternative caching scheme that we could have used to locally store data in files is to treat the local disk as a general cache for log entries read from the GDP. However, we believe that materializing files directly, as described above, is strictly better for two reasons. First, storing the log entries directly would also store stale data (i.e., data that has since been overwritten), wasting space in the local disk. Second, it let us use the local filesystem

[4]The correctness of this scheme depends on asynchronous requests being processed in the same order that they are made. For the current implementation of the GDP we believe this to be the case. If this changes, we could create a bounded buffer of requests, and a worker thread that processes them synchronously. Rather than making a request to the log daemon asynchronously, we would just enqueue the request into the buffer, achieving the desired semantics.

(ext4), which is well-optimized for performance, to store parts of the same file close to each other on disk.

Sparse Caching

Because we only cache portions of a file that have been accessed, the cached version of a file may be incomplete and have portions which are not valid.

Originally, each per-file cache was implemented through two files on the local filesystem. One file would be responsible for storing the actual contents of the file whereas the other would be a bitmap which could be used to tell which bytes in the cache were actually valid. The nth bit of the bitmap and the nth byte of the locally cached file correspond to the nth byte in the logical view of the file. Thus, reads to the cache would first check the bitmap to make sure the cache was valid and then would read that range of bytes in the mirrored cache file. Because it is possible to write past the end of the file in ext4 without actually allocating the blocks for the empty regions, this method was an acceptable way for us to create partially cached copies of GDPFS files without paying the space for the entire file.

However, having a bitmap for every file adds a significant amount overhead to the space of our cache. Specifically, for every n cached bytes, we pay an additional $\lceil \frac{n}{8} \rceil$ bytes to keep track of the bitmap. To avoid this overhead, we attempted to use a feature of modern filesystems intended for use on sparse files. In addition to not allocating blocks for holes created by writing past the end of the file, ext4 also keeps track of the positions of these holes internally. For example, using the SEEK_HOLE option with lseek will move the file offset to the next hole greater than or equal to the argument offset. Because of this functionality, finding out if a range $[a, b)$ in our cache is valid should theoretically be reduced to checking if the result of lseek(a, SEEK_HOLE) is greater than or equal to b. However, the granularity to which ext4 tracks these holes is at the blocksize level so holes inside a partially filled in block will not be reported using lseek(SEEK_HOLE).

Throughout our testing, we have not yet run into any issues with this bug since it is unusual for a process to write a part of a block on a cold cache and then read a different part of this block later. Because of time limitations, we have decided to revert to keeping a bitmap for each file because our performance is not bound by checking ranges on the bitmap.

For future work, it may be desirable to combine the bitmap and the SEEK_HOLE functionality in order to proverbially get the best of both worlds. For example, one simple way to use both is to first check using SEEK_HOLE if the block we are reading is valid. If it is not, then we can entirely avoid having to check that portion of the bitmap. Otherwise, we must still check the bitmap since there may be portions of this block that are invalid.

File Cache Coherency

One other aspect that we considered is the consistency of our file cache. Because we the single-writer semantics of the underlying GDP through to the GDPFS, we avoided dealing with a majority of adversarial cases that would leave our cache inconsistent. However, it is

possible that a single private key could be used on two separate hosts. In this case, although there is one entity in the security sense of the word, we must maintain a pair of consistent caches across two separate hosts. One idea that we are considering for future work is to create a service that provides the filesystem to multiple users, and serializes the writes to the logs as a single writer (because the GDP logs are *single-writer*). Many of the same consistency issues we would see in such a setting also arise in the case of a single writer mounting the same GDPFS in multiple places and interacting with the separate mounts concurrently. Therefore, we provide in this section a discussion of some of the difficulties that arise in such a case.

Suppose Alice and Bob both open a file, as writers, on the GDPFS. Consider, as a simple case, the scenario where Alice and Bob concurrently write different values to the same range of byte in the file. To make the example concrete, consider the case where Alice writes byte sequence A to the first 100 bytes of the file, and Bob writes the byte sequence B to the first 100 bytes of the file. The standard way in which the GDP allows one entity to be informed of new writes to a log is via a *subscription* to the log. In the example, Alice first writes A to her cache, and Bob first writes B to his cache; meanwhile, they both asynchronously make requests to the GDP log server. The log server will then choose some serial ordering for these writes and append both entries to the log; then, Alice and Bob will be informed of each other's writes via the subscription. Alice will write B to her cache, and Bob will write A to his cache. After this is finished, Alice will think that the first 100 bytes of the file contain B, whereas Bob will think that the first 100 bytes of the file contain A. In particular, either Alice's cache or Bob's cache will be incorrect until he or she remounts his or her filesystem.

One way to achieve consistency is to treat the order in which updates are sent due to the subscription to a file as describing the ground-truth ordering of writes to the file. After making an asynchronous request to append to a log, a client receives first an application-layer ACK from the log server, and then the same log entry in response to the client's subscription to the log. Upon receipt of the ACK, Alice would check its record number to see if it is what she would expect if she were the only writer; if it is what she expects, then she can be sure that no writes happened in between. If the record number is higher than expected, then Alice can conclude that a separate writer made a write to the file before her write. Alice then must replay all writes starting from the first write made by another writer when she receives the datum from the subscription. Although this method maintains a strongly consistent view of the cache it has very poor performance.

Another way to solve this problem is to use a weaker consistency model in return for better performance. For example, we could have that the file's cache is only up to date when the file is first opened as part of the semantics of our filesystem. Implementing this would be as simple as maintaining a separate cached version of each file for each process instead of treating all processes open file as the same entity.

We currently do not support the case of having two separate hosts writing to the same file since such a use case is somewhat rare. However such a system could be built using the methods discussed above.

In-Memory Caching

The GDPFS does not maintain an explicit in-memory cache of file contents. The reason why we chose to forego any caching layer in memory is that local filesystem's buffer cache will maintain an in-memory copy of commonly accessed parts of a file's disk cache. Leveraging this fact allowed us to not worry about the details of memory management of cached files. However, we do maintain an in- memory copy of part of the index because we felt that it is unlikely for the buffer cache of the local filesystem to be well-optimized for the read and write patterns required to maintain B Tree structure (more details in Section 3.3).

Efficient Data Retrieval with a Cold Cache

Indexing Strategy

Although a file cache allows efficient access to recently read data, or data written during the current session, reads are still inefficient when the cache is cold. There are multiple reasons why reads, in the case of a cold cache, ought to be optimized. First, if a reader mounts the filesystem to read a large file, it is unacceptable for the reader to have to read through the entire log backing the file. Second, a writer (represented by a single keypair) may mount the file system from multiple computers, meaning that they may read a file on a computer where the cache is out-of-date.

Our solution is to *checkpoint* files, by writing a log entry that does not contain any new data, but rather is a tree that allows efficient retrieval of data from the log. In particular, our index solution guarantees that the log entry number at which a byte is stored can be retrieved in $O(\log n)$ time, where $n = \min\{\text{number of entries in the log}, \text{number of bytes in the file}\}$. For very large files, this tree could grow quite large. Therefore, each checkpoint log entry contains either (1) the entire tree, if the file has never been checkpointed before, or (2) the *diff* of the tree from the previous checkpoint, containing only new and modified nodes in the tree, referencing nodes in previous checkpoints that are still in the current tree. In that sense, the tree is copy-on-write; if one node is modified, then all nodes in the path to the root need to be rewritten in the next checkpoint.

An important advantage to only storing part of the checkpoint in each log entry is that the entire checkpoint need not be stored in memory for any given file. In particular, if only part of a file is accessed, only the nodes relevant to that part of the file may be stored by the client at all. The remaining parts of the tree are loaded lazily as they are needed.

When a file is first opened, the client first reads the underlying log backwards until the first checkpoint log entry. The tree nodes in that log entry are stored in memory as the index for that file. Then, the log entries after the checkpoint entry are applied as diffs to the index, allowing the index for the current version of the file to be materialized. *Note that the entire tree need not be stored by the client at this time!* In particular, nodes in the tree that were not written in the last checkpoint, and which were not touched when the later log entries were applied to the index, will not be stored by the client, and will be loaded lazily by the client as they are needed. Because reading the log backwards, and applying the

later log entries as diffs can be cumbersome and time-consuming, our implementation makes the optimization that *every file is checkpointed before its in-memory state is discarded*. This means that as long as the client terminates normally, the last entry in a file's backing log will be a checkpoint.

Because each checkpoint entry contains multiple nodes, it makes sense to locally store checkpoint nodes in case they are needed again, for a different node in the same checkpoint. This may be quite common, because each checkpoint contains an earlier snapshot of a subtree; if one node needs to be loaded, its children are likely to be needed soon. Therefore, we maintain an on-disk cache of recently read *checkpoint* log entries. Note that we do not maintain such a cache for recently read *data* log entries! If a log entry containing data is read, all of the relevant data in that log entries is read into the File Cache and will never be requested from the log server again; therefore a log entry cache for data-containing log entries would provide absolutely no benefit.

Possible Index Implementations

In this section we describe possible designs for an indexing strategy that achieves the above properties. Then we explain the design that we finally chose for our implementation.

One indexing scheme used by many file systems is that of an inode. Files are split into fixed-size blocks (often the size of a block on an underlying hard disk), and are stored in a tree, where the data blocks are leaves. However, we decided against such a scheme for two reasons. First, it requires all writes to be block-aligned. In particular, a small write, of just a few bytes, would require the entire block to be copied as a new log entry so that it can be referenced as a leaf in the inode tree. This adds a significant overhead in network bandwidth for all readers, since the entire log entry must be read, even though most of it is common with the previous state of the file. We view the block-granularity of leaves in an inode tree to be an artifact of block storage such as disks, not an advantage in and of itself. The only possible advantage to performing block-aligned writes is to decrease fragmentation of file data; however, there are better ways to achieve this[5].

Furthermore, while an inode-based index guarantees logarithmic time lookup in the size of the file, it does not take advantage of cases where the log itself is much smaller. We observed, when running compilation jobs on our filesystem, that writes were often much bigger than the block granularity of a disk. Although the log could be much larger than the file it backs, so that in the worst case the indexing scheme should scale with the size of the file, not the size of the log, it is desirable for the indexing scheme to take advantage of cases where the size of the log is actually small.

A vanilla B Tree, that maps a single byte index to a log entry number is not a desirable index either. To write a range of k bytes would require k insertions into the tree, one for each byte in the range. Some file systems use a B Tree with fixed-size blocks. OceanStore [45] uses such an indexing scheme, with a block size of 8 KB. Although OceanStore uses a

[5]One such way would be to periodically defragment files; this does not incur the overhead of reading and copying parts of file on every write.

```
/* Initializes a Fig Tree. */
void ft_init(struct figtree* this);

/* Sets the bytes in the range [START, END]
 * to correspond to VALUE. */
void
ft_write(struct figtree* this,
         byte_index_t start,
         byte_index_t end,
         figtree_value_t value,
         gdpfs_log_t* log);

/* Returns an iterator to read over the
 * specified range of bytes. */
struct figtree_iter*
ft_read(struct figtree* this,
        byte_index_t start,
        byte_index_t end,
        gdpfs_log_t* log);

/* Deallocates the resources for THIS. */
void
ft_dealloc(struct figtree* this);

/* File-Indexed Group (FIG) */
struct fig {
    struct interval irange;
    figtree_value_t value;
};

/* Gets the next FIG from the Fig Tree
 * Iterator and populates NEXT with that
 * result. Returns true if there are
 * additional FIGs in the iterator. */
bool
fti_next(struct figtree_iter* this,
         struct fig* next,
         gdpfs_log_t* log);

/* Deallocates the specified Fig Tree
 * Iterator. */
void fti_free(struct figtree_iter* this);
```

Figure 3.3: Interface to a FIG Tree.

B Tree rather than an inode, the fact that it uses fixed-size blocks means that it has similar problems to those discussed above.

A quick fix would be store in the B Tree one key-value mapping for each range, rather than one key-value mapping for each byte. For each range that is written, an entry is added to the B Tree mapping the first byte in the range to the record containing data for that range. However, writing a large range would still require the removal of all of the ranges it overlaps with, which could be linearithmic ($O(n \log n)$) in the size of the range.

FIG Tree Index

In this section we explain the type of index we used in our final implementation. We first introduce some terminology. Each write by the client corresponds to one additional log entry added to the log backing the file; we call this log entry a *file group* because it represents a group of bytes that can be found by reading the same log entry. A file group may partially overlap with previous file groups; the values of these bytes in old entries are said to be *stale*. Abstractly, the job of our index is to efficiently find, for any byte index, the most recent file group containing that byte, avoiding stale entries.

We call our data structure a File-Indexed Group (FIG) Tree. The principle of a FIG Tree is to map ranges to records, instead of individual bytes to records. When a range of bytes (a file group) is written, an entry representing that file group is added to the FIG Tree. An entry consists of a range of bytes $[a, b]$ mapped to an identifier of the immutable record containing those bytes. A FIG Tree does not delete the stale intermediate ranges contained within the range of bytes written; instead it puts the entry describing the newly written range higher in the tree so that any queries for bytes in the range will find the new range entries first, and will never find the stale mappings.

While simple in principle, this results in some edge cases when reading and writing data, that are addressed below.

FIG Tree Query Algorithm

Querying a single byte is done in the same way as is done in a B Tree. Starting at the root, we check if the queried byte is in one of the entries at that node. If it is, the record containing the byte has been found. Otherwise, we recurse on the appropriate subtree.

This method can be extended to range queries; traverse the subtree normally, making sure to avoid stale entries. This can be done by keeping track of an interval containing the bytes that are valid at each node. Initially, this range is $(-\infty, +\infty)$. When entering a subtree between entries containing intervals $[a, b]$ and $[c, d]$, the interval gets restricted to $[b + 1, c - 1]$. We backtrack up the tree either when we have finished traversing a node, or when we reach the end of the valid interval.

FIG Tree Insertion Algorithm

To do an insert, we begin by performing the Query Algorithm, with the following modifications. We keep track of the interval containing valid bytes at each node. At each node, we prune the node by "trimming it" to the valid range: this means removing all entries and subtrees that lie completely outside the range, and trimming entries that lie partially within the range (if a subtree lies partially outside the range, then we do not bother trimming it; we never have to traverse any of a node's subtrees in order to prune it). It is important to prune each node as described above in order to prevent stale entries from being pushed up the tree on an insert. Furthermore, we stop early when we find a node where at least one entry in the node overlaps with the range that we are inserting. If we reach a leaf node without this happening, then we just insert the new entry normally and split and push up entries normally as in a B Tree.

If we stop at a node early, then the entries that overlap with the range we are inserting are consecutive entries in that node (since the entries in each node are in sorted order). Let $[x, y] \to Z$ be the group that we are trying to write, and let $[a, b] \to C$, $[d, e] \to F$, $[g, h] \to I$, and $[i, j] \to K$ be the entries that overlap with it. First, we replace all four of the entries in the node with a single entry $[x, y] \to Z$, whose left subtree is the subtree that used to be to the left of $[a, b] \to C$, and whose right subtree is the subtree that used to be to the right of $[i, j] \to K$. The subtrees between $[a, b] \to C$ and $[d, e] \to F$, $[d, e] \to F$ and $[g, h] \to I$, and $[g, h] \to I$ and $[i, j] \to K$, are completely deleted. If $x > a$, then let the new group $[a, x-1] \to C$ be the left continuation. If $x < j$, then let the new group $[y+1, j] \to K$ be the right continuation. We then insert the left and right continuations, if they exist, into the tree normally; these insertions will be done at the leaves of the tree, and not at some intermediate node (remember to use a and j, rather than $x - 1$ and $y + 1$, to compute the valid intervals when pruning the left and right subtrees of the newly inserted group $[x, y] \to Z$).

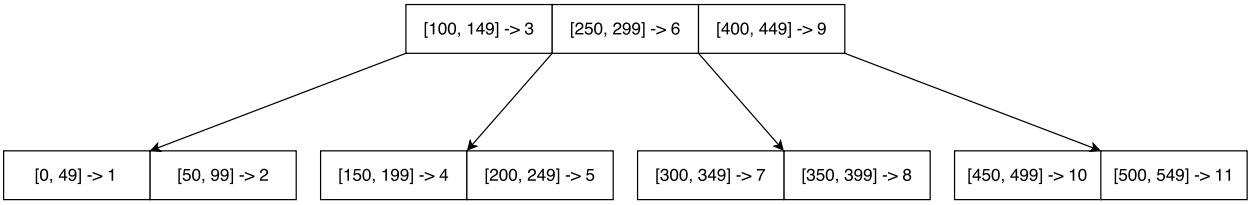
See Figure 3.4 for concrete examples of the above rules.

Additional Optimizations

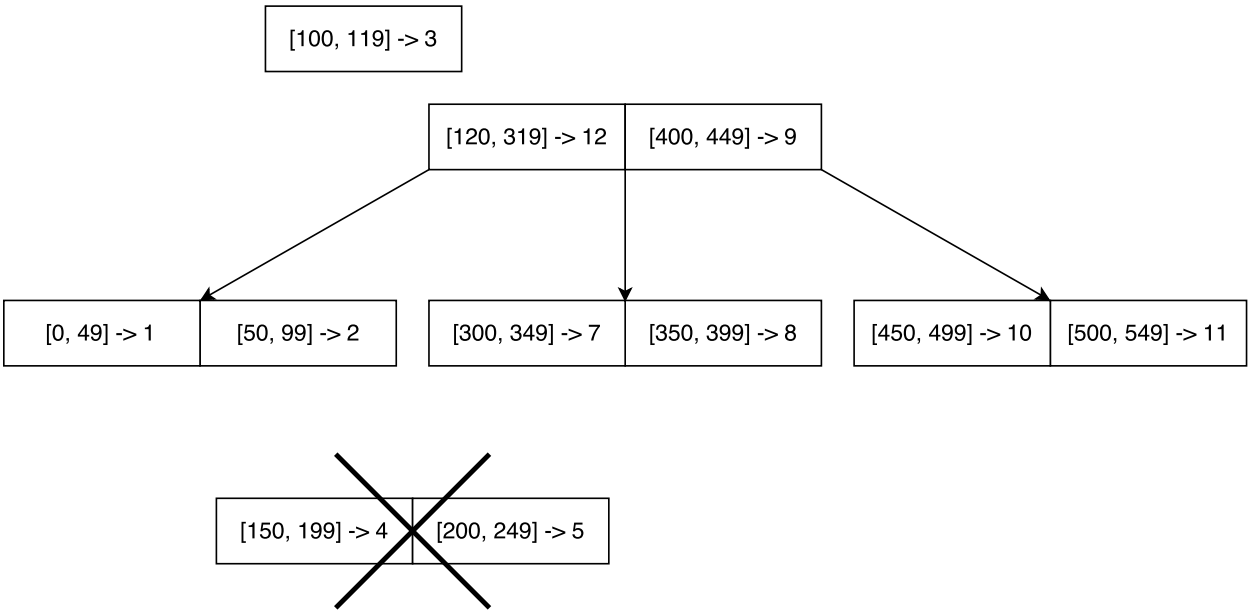
Besides materializing files on the local disk and checkpointing files with FIG tree indices, we perform additional optimizations to improve performance of the GDPFS.

Precreation of Logs

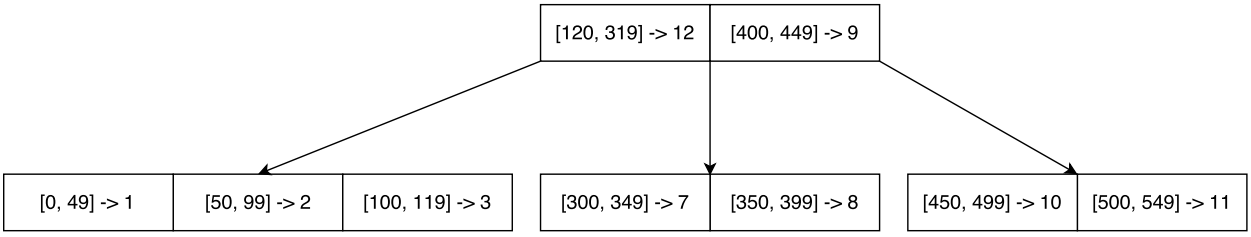
One performance aspect that we quickly noticed when working on the filesystem is that file creation has high overhead. This stems from the fact that reads from a file are generally fast due to caching, and that writes to the GDP can be done asynchronously; in contrast, log creation must be done synchronously and is a bottleneck for workloads that create many files. Furthermore, logs cannot be created concurrently in the current version of the GDP, exacerbating this problem.



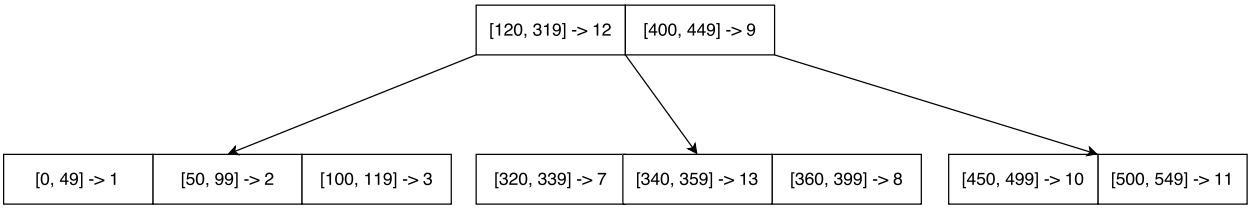
(a) A FIG Tree constructed by creating an empty file and then extending it with 11 writes of 50 bytes each.



(b) When a write of bytes 120 to 319 (inclusive) is performed, it replaces all entries in the root node that it overlaps with. The leftmost entry only partially overlaps with the range, so bytes 100 to 119 form a *left continuation* that is inserted separately into the tree.



(c) The final FIG tree, after bytes 120 to 319 are written.



(d) The final FIG tree after the additional write of bytes 340 to 369. Observe that the leaf node is pruned to the valid interval when it is written.

Figure 3.4: Example insertions into a FIG Tree.

To alleviate this issue, we precreate logs ahead of time and add them to a queue, so that files can be created without having to wait for a synchronous remote procedure call to the log server to return. We implemented this as a synchronized bounded buffer, with a worker thread spawned on initialization of the filesystem that creates new logs and adds them to the bounded buffer whenever it is not empty. Threads that create files remove an element from the bounded buffer, or wait for the worker thread in case it is empty.

Second-Chance List for Files

One artifact of our file system is that opening a file for the first time is an expensive operation that requires multiple reads from the log server. In particular, it requires a synchronous read of the most recent log entry, which becomes a bottleneck when all reads are satisfied by the cache and all writes are done asynchronously. Therefore, we would like to mitigate this delay where possible.

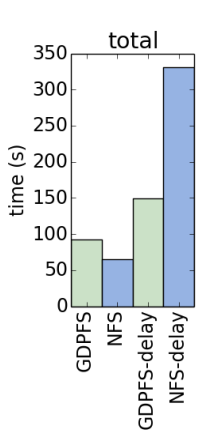
One pattern that, on the surface, seems reasonable, is to checkpoint a file and deallocate the in-memory state of a file when all processes have closed it (it reaches a reference count of zero). However, we found that during compilation jobs, it is common for a file to be opened and closed many times. It is inefficient to deallocate a file when it is closed, only to perform a synchronous remote procedure call to rebuild that state when it is opened again.

Our solution to this problem was inspired by the demand paging algorithm used by the VAX operating system. Because the VAX operating system did not have hardware support for detecting when pages are used, it maintained "hot" pages in memory in a FIFO list and a second-chance list in memory as an LRU list of pages marked "invalid" in the page table. Pages that reach the end of the FIFO list of "hot" pages are given a second chance in the LRU list before they are paged out to disk; a page that is accessed very rapidly will periodically enter the second-chance list before the "hot" pages are in a FIFO list, but will never be paged out to disk because it will be "revived" from the second-chance list on the next access.

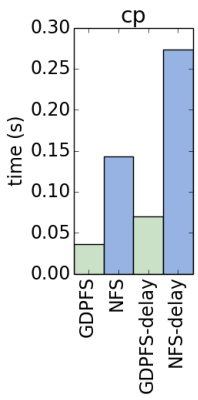
Similarly, rather than deallocating the in-memory file structure when its reference count hits zero (i.e., when it is closed by all threads that opened it), we place the file on a second-chance list and maintain its in-memory state. If the file is opened soon after it is closed, then we no longer have to make synchronous remote procedure calls to rebuild its state, as we can simply "revive" it from the second-chance list. To eventually reclaim resources, we stipulate that the second-chance list has a maximum size, and deallocate the in- memory state of a file when it reaches the end of the second-chance list. While we were developing the filesystem, we found that this optimization gave us a 100% speedup for compilation jobs.

3.4 Performance Evaluation

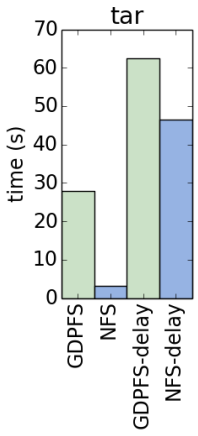
Benchmark measurements were made on two hosts on the same LAN. The log server was run on a set of 2 Intel(R) Xeon(R) E5-2667 2.9GHz CPUs (6-Core, HT, 15MB Cache, 130W)



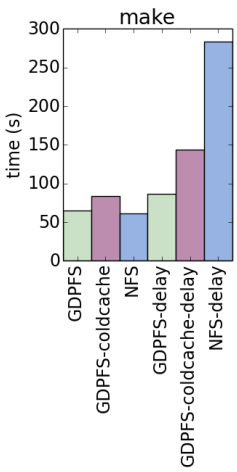
(a) Time to compile Redis in given filesystem. Results averaged over 3 trials.



(b) Time to cp zipped Redis tar into given filesystem. Results averaged over 3 trials.



(c) Time to unzip and untar Redis in given filesystem. Results averaged over 3 trials.



(d) Time to make Redis in given filesystem. Results averaged over 3 trials.

Figure 3.5: Macrobenchmark results.

with 64 GB (8 x 8GB DDR3-1600) of memory and 5 × 1 TB Seagate Constellation.2 (6Gb/s, 7.2K RPM, 64 MB Cache) 2.5" SATA drives in a RAID 6 configuration on a MegaRAID SAS 9260-16i controller. The remote client of the GDPFS was run on an Intel Core i7 processor, with 8 GB of memory and a 5400 RPM hard drive. The average latency between the two hosts is about 5 ms. In order to simulate running our file system in a Wide Area Network, we also ran experiments where we used the Linux's netem (Network Emulation) utility to artificially inject latency into our system. For these experiments, we added a latency of 10 ± 2 ms.

First, we discuss a macrobenchmark which tests our filesystem as a complete system and then we dive into microbenchmarks on file creation, sequential read, and sequential write performance. For each of our benchmarks, we compare against NFS a popular distributed filesystem. The NFS server is configured on the server with default settings and also mounted with default settings.

Macrobenchmark: Redis

The macrobenchmark we ran was to compile a popular key value store called *Redis*. There are three steps to this benchmark: copying the tar file to the file system, untaring the archive, and finally making Redis. As a whole, we believe that these three steps accurately mimic a large proportion of use cases for our file system.

cp

In Figure 3.5b, we present the results for first copying the compressed tar into both the GDPFS and NFS. We found that in both the normal and the simulated WAN case, the GDPFS copied the tar in faster. We suspect that the cause is that the GDPFS returns from writes as soon as they hit the local filesystem, pushing them to the GDP asynchronously, whereas NFS may have to do some network I/Os to maintain cache coherency. That said, this is not a fair comparison because NFS supports more general semantics than the single-writer GDPFS. Because we assume a single writer, it is easier for the GDPFS to maintain cache coherence.

tar

NFS significantly outperformed the GDPFS in the tar step. This is because untarring requires creating many small files and the GDPFS is particularly bad at file creation, primarily for two primary reasons. First, since the GDPFS maintain a global mapping of file handles to file structs, the GDPFS must lock this structure when opening files. Thus when a large number of files are created and opened, there is a great deal of contention around the lock. Second, a bug in the current GDP implementation prevents us from creating logs concurrently so we are forced into synchronous creation of logs and we need one for each new file. We attempted to solve this problem by precreating a bounded buffer of logs, but we found that due to lock contention on the buffer we weren't using up the precreated logs. For further discussion, see Section 3.4.

make

Redis make times are comparable on NFS and the GDPFS in the simple case. However, in the simulated WAN case, we found that the GDPFS is significantly better than NFS. This led us to suspect that NFS was making more network round trips than the GDPFS, possibly to maintain a consistent cache.

We also tested make times for the GDPFS with a cold local cache. In this benchmark, the files were copied and untarred as normal, but then the GDPFS had its cache cleared before making. In all the other tests, reads and writes could be satisfied by the local cache. This benchmark was run in order to test the effectiveness of the FIG tree. The results demonstrate that the FIG tree keep the times for make at an acceptable level even with a cold cache. For more detailed results, see Figure 3.5d.

Total Times

In total, we found that the time it takes to run all three stages (cp, tar, and make) on the NFS is slightly faster than on the GDPFS. However, when we added network latency into the benchmark we found that the GDPFS was significantly faster. This is due to the stricter single writer semantics of the GDPFS.

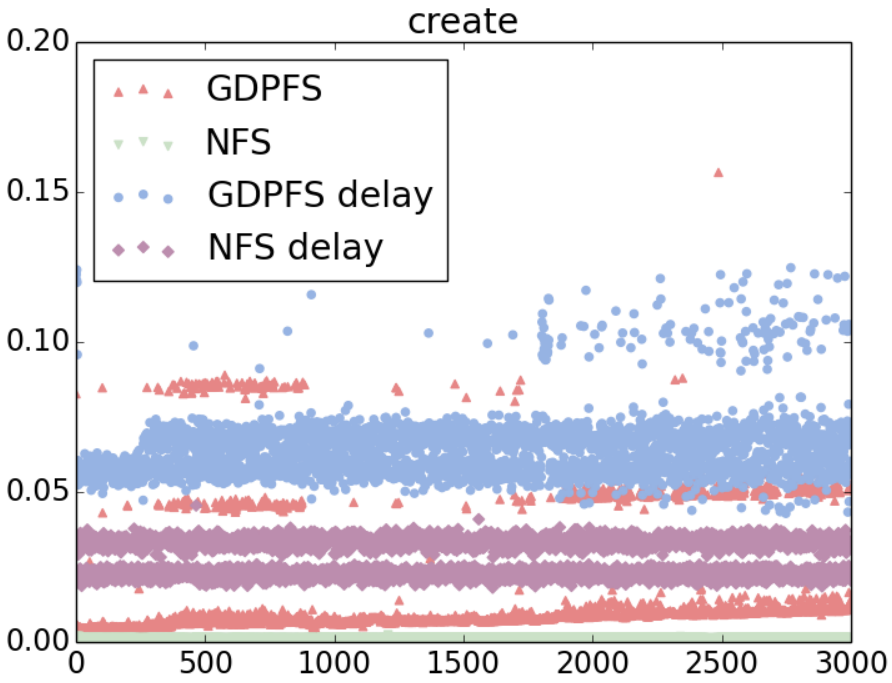


Figure 3.6: Distribution of times to create a file over 3000 files.

In summary, this macrobenchmark concludes that it is plausible to build a performant file system on top of append-only logs if enough caching and indexing layers are put in.

Microbenchmarks

In order to get a more fine-grained view of the performance of the GDPFS we also ran three tests each of which stressed a main feature of file systems. These are: file creation, sequential writes, and sequential reads. These three tests were done in the same client and server as the macrobenchmarks. In addition, each of the tests also measures the performance in face of additional simulated latency.

Creation Test

In the creation test, we created a sequence of 3000 files and timed each call to the "open" function, which we used to create files. We found that in both the normal and the delayed case, the NFS was faster at creating files. Because we are sequentially creating files in this microbenchmark, the earlier discussion about the file table lock is not applicable.

We originally hypothesized that the explanation for this result was that we were running through the logs from our buffer of precreated logs. However, when we measured the size of

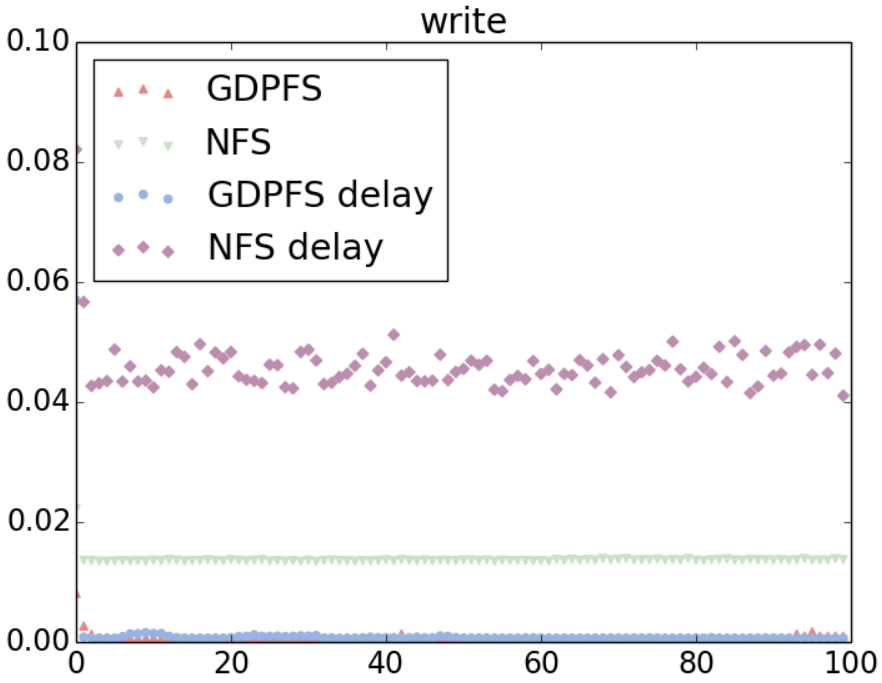


Figure 3.7: Distribution of times to write 32 blocks (128 KiB) over 100 writes.

our buffer we found that this was not the case. In fact, in all cases the buffer was either full or one away from being full. We suspect that this is because of lock contention on the bounded buffer. In the presence of multiple threads creating files (for example, in a compilation job), precreation of logs would likely improve performance.

Write Test

On each iteration of the write microbenchmark, we opened a file, appended 128 KiB of data, and closed the file. This process was repeated sequentially 100 times. We opened and closed the file on each iteration was to ensure that the filesystems were persisting the data and not just storing updates in an in- memory buffer. The reason we chose this chunk size is that FUSE prefetches reads in chunks of this size, and we wanted to make sure that the prefetching did not cause irregularities in our results.

Our results for this microbenchmark, which can be seen in figure 3.7, reflect those seen in the cp macrobenchmark displayed in Figure 3.5b. As stated in the section on cp, we believe this phenomena to be a result of the GDPFS returning from writes as soon as they hit the cache and then persisting them to the GDP asynchronously. In contrast, NFS must hit the network cache consistency before returning from a write.

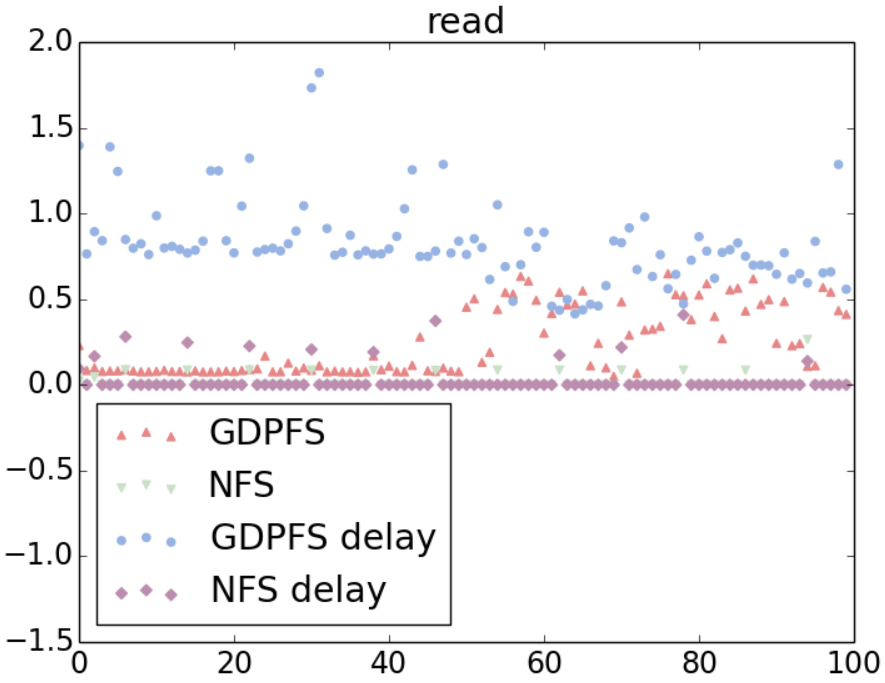


Figure 3.8: Distribution of times to read 32 blocks (128 KiB) over 100 reads.

Read Test

In the read test, we unmounted and then remounted the filesystem in which we ran the write test. We then sequentially read the same 100 blocks from the file we wrote in the previous microbenchmark ensuring that each read still had the same data which was written.

In both cases, where we added network delay case and where we did not, the NFS significantly outperformed our GDPFS. There are two reasons for this. First, it seems that the NFS may have cached all of the data for the file upon opening it whereas the time to access the file is spread out across each read in the GDPFS. Second, because the NFS is working on top of a mutable backing store it is much easier for it to fetch the appropriate bytes. In the case of the GDPFS, since we had unmounted the file system earlier, each read had to be satisfied by a tree traversal and GDP synchronous read.

3.5 GDPFS Related Work

Log based filesystems have been built before, most notably the Log-Structured Filesystem (LFS) [47]. That said, we differ from LFS in a number of key areas. First, we are motivated to use logs because the GDP gives them to us with network access, distributed durability, atomicity and does this all with strong security guarantees over untrusted hardware. LFS

uses logs because they allow writes without seeks. Second, every file in the GDPFS exists in its own log. Third, we do absolutely no modification in place. Everything is built around append only logs. LFS uses modify-in-place semantics in several areas including the checkpoint region. Fourth, we do away with traditional inodes and store file metadata in the log entries for each file. This means that a file in our system has no tie to the filesystem that it exists in and thus could theoretically be linked to in any other GDPFS mount. It also means that each file in the filesystem could potentially be owned by a different user[6].

Looking at file systems in general, the Network File System (NFS) [49] and the Andrew File System (AFS) [23], being distributed file systems, bear great resemblance to the GDPFS. One of the most interesting contributions of AFS is its relatively weak semantics for concurrent modification of the same file. files are written back to the AFS server when they are closed, and the last close wins. These weak semantics are motivated by the observation that it is unlikely for multiple users to concurrently modify the same file. This same idea partially justifies the *single-writer* nature of files in the GDPFS.

Because files are backed by logs, it is possible to "go back in time" and restore an older version of a file. Our decision to use a separate log for each file allows files to be separately reverted to older versions. Some filesystems, such as the B-Tree Filesystem [46], use a Copy-on-Write strategy to cheaply snapshot files. However, the semantics of these snapshots are weaker than the versioning semantics achieved with a log.

There are a few other similar filesystems such as TahoeFS [51] and the Fast Secure Filesystem (FSFS) [15]. However, they lack the copy on write and versioning advantages that our system provides. The GDPFS is novel because it is built solely on append-only logs, it gives strong atomicity guarantees, and can guarantee strong data integrity and confidentiality while running on a network at global scale and consisting solely of untrusted hardware.

Ultimately, we decided that of systems similar to the GDPFS, NFS was the best to do a detailed comparison against. There were two primary factors that lead us to this decision. First, NFS is very widely deployed and is well-known which makes the comparison interesting and relatable. Second, NFS and the GDPFS implement most of the same features and serve similar purposes but each with a slight twist. NFS's unique feature is that it supports multi-writer semantics. But the GDPFS is able to run over wide area networks on untrusted hardware. In section 3.4 we show the results of our comparison benchmarks. The GDPFS and NFS are neck and neck in terms of the total number of benchmarks won. However, the GDPFS had the advantage when network delay is added (3/4 wins) while NFS has the edge without added delay (3/4 wins) (see section 3.4 for a more detailed discussion). This is what we would hope for given that the GDPFS is targeted at a wider area network than NFS.

[6]We did not implement this functionality and thus will leave it to future work.

Chapter 4

Wrapping Up

In this chapter we wrap up by highlighting what we learned and discussing opportunities for building upon the work presented in this report. We go a bit beyond what is in this report to touch on plans for the GDP Network layer as a whole. We then conclude.

4.1 Future Work & Lessons Learned

GDP Secure Channels & Network Layer

Bigger picture, there is currently a great deal of effort being put into redesigning the GDP network layer as a whole due to the limited scalability of the current design. A new full time staff member recently joined the Swarm lab GDP group largely for the additional manpower required to implement these changes. The protocol changes presented in chapter 2 are part of this redesign. Consequently, there are plans to continue to refine the protocol and begin implementing it in the canonical GDP client and router libraries in the coming months (as part of a wider set of changes, discussed below).

Since the GDP router will essentially need to be completely rewritten, we have the opportunity to start with a clean slate for the GDP network layer technology stack. Thus a good deal of consideration is being put into evaluating various technologies supporting the layers of the network stack as well as the overall architecture. One important realization is that the GDP router currently performs two duties: high level route calculation and low level forwarding. In the next generation GDP network, these operations will be much more decoupled than they are today. Therefore, for the rest of this section, we will refer to the forwarding portion of the GDP router as the forwarder and the routing portion as the router.

The forwarder is of particular interest in this context as it will need to be able to efficiently store temporary forwarding state as channels come and go. Initially, we thought a lot about using SDN to support this functionality and looked into systems like OpenDaylight [36] and OpenFlow [35]. Unfortunately, we found that SDN systems tend to be quite TCP/IP geared and don't give us the flexibility we need. This has caused us to return our focus to The

Click Modular Router [26] which is how the current GDP Router is implemented. One of the problems with the current implementation of the GDP Router is that—although it was built on top of Click—it was built in a very non-Click way. To the extent possible, Click applications are supposed to be made up of small modules that are tied ("Click"-ed) together to create more complex programs. If written properly, a Click program can be pushed form user space down into Kernel space and be very performant. Click instances can even be run as extremely lightweight efficient VMs thanks to ClickOS [9, 34], which offers further scaling flexibility. We believe that a large part of what made the current Click-based GDP router turn into a large monolithic user space program was that it tried to do everything. By pulling out the higher level routing functionality, we believe a very efficient forwarder can be implemented in Click which will pass queries up to a higher level routing system when necessary. This will aid in integrating our chapter 2 work into the currently distributed GDP package.

GDPFS

Our goals in creating the GDPFS were (1) to make the GDP accessible to more applications, and (2) to evaluate the GDP as a useful primitive to create distributed systems.

We believe that we were fairly successful in achieving the first goal. Given that we only had limited time, we were unable to create a bug-free filesystem that was fully featured (with things such as hard links, symbolic links, etc.); however, we have made significant progress towards this end—we have created a filesystem that is stable enough to run compilation jobs of nontrivial systems. We believe that a few more months of development could turn the GDPFS into a fully usable file system.

The second goal was to evaluate the usefulness of the GDP itself. First of all, it must not be forgotten that the GDP is itself a research project under active development. Furthermore, as far as we know our use case has put more stress on the GDP than anything anyone has previously done. That is, compared to other GDP applications we create more logs more quickly, do more frequent and larger batches of asynchronous reads and writes, and open and close a greater number of logs at a greater rate. Given this new stress, it is not a surprise that we found several new bugs. Much of our GDP-related friction was because of these bugs, so an obvious first step in improving the GDP's usefulness in building complex systems like ours is to fix these issues. That said, the fact that with a semester and a half of part time work we were able to construct a distributed filesystem that, with a few minor tweaks[1], has the ability to securely operate over untrusted hardware, speaks to the advantages of using the GDP as a substrate for creating complex systems.

While the GDP enforces single writer semantics, there is no inherent reason why this restriction must be elevated to the level of the GDPFS. It would be interesting to figure out a way to restructure our system such that this limitation was removed. One could imagine

[1]We didn't quite implement all the security features in the GDPFS due to time constraints however the foundation is all there and we don't expect any performance changes since the GDPFS is by no means CPU bound.

a service, owned by a single entity (keypair), that was the "single writer" of the filesystem, that provides the filesystem as a service to multiple logical writers. However, such a service would have to solve the cache coherency problems with multiple writers mentioned earlier. Furthermore, it would have to be replicated in order to be fault- tolerant, and, in order to scale at the level of the backing GDP, it would need to be able to run securely on untrusted hardware. In short, this system would have to implement much of the functionality provided by the GDP in such a way as to support multiple writers. Given that it would have to implement this functionality on its own anyway, we are unsure how it would benefit from using the GDP. The problem here is the *single-writer* nature of GDP logs. While it provides convenient security properties, it is a shortcoming in that a multi-writer system needs to re-implement much of the functionality of the GDP to scale at that level.

The GDPFS could also be improved by implementing a system for accessing earlier versions of files. It would also be interesting to devise a key-sharing permission scheme to restrict which versions of a file are available to which readers.

4.2 Conclusion

By giving the Internet arms and legs, hands and feet, and a central brain to control it all, the IoT is fundamentally transforming the Internet as we know it. Consequently, more than every before, it is of vital importance that we take steps to make the computers, particularly cheap IoT devices, secure. The Global Data Plane offers IoT app developers an easy to use way to write secure IoT applications by making data the fundamental abstraction of IoT applications.

In chapter 2, we proposed a number of changes to the Global Data Plane architecture at the network level. The changes are intended to make the GDP network more secure and performant. Our modifications include the addition of access control lists to reduce side-channel attack surface by limiting who can read from logs. They also add secure virtual circuits that allow headers to be compressed and fewer log records to be signed (albeit at the cost of increased object security latency). Outside of what we improved, our changes maintain existing GDP security properties. We evaluated our changes by looking at them from a theoretical perspective, as well as by using a trace of current GDP traffic. The trace evaluation demonstrates that our PDU header compression and signature reduction techniques can lead to a significant reduction in overhead for applicable flows, which was our goal. In doing so, we provide a protocol which improves GDP usability and opens the door for additional researchers to start using the GDP and aiding in its development.

In chapter 3, we presented the GDPFS, which lifts the single-writer, append-only log abstraction provided by the Global Data Plane to a single-writer filesysystem. Our filesystem uses Unix semantics and is therefore a very well-known API. This makes the GDP much more accessible to many application developers. We showed that non-trivial applications can be built with the GDP as the storage core. We also showed that our filesystem is performant and able to complete complex tasks such as "make"-ing Redis. We found that using the

GDP enabled us to create a filesystem that scales extremely well and runs on untrusted hardware—assuming a bug-free and production-ready GDP. To our knowledge, no other distributed filesystem has been designed with these objectives in mind.

Bibliography

[1] URL: https://twitter.com/mytoaster.

[2] URL: https://aws.amazon.com/iot/.

[3] URL: https://cloud.google.com/solutions/iot/.

[4] URL: https://www.microsoft.com/en-us/internet-of-things/.

[5] URL: http://www.secdev.org/projects/scapy/.

[6] URL: https://github.com/paulbramsen/gdp_virtual_circuit_simulator.

[7] URL: https://cloud.google.com/.

[8] URL: https://github.com/paulbramsen/gdpfs.

[9] Mohamed Ahmed, Felipe Huici, and Armin Jahanpanah. "Enabling dynamic network processing with clickOS". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, pp. 293–294.

[10] Eric Allman, Ken Lutz, and Nitesh Mor. *The Global Data Plane Prototype*. Poster presented at the Berkeley EECS Annual Research Symposium (BEARS). Feb. 2015. URL: http://terraswarm.org/pubs/508.html.

[11] Amber Ankerholz and Bruce Schneier. *Bruce Schneier on New Security Threats from the Internet of Things*. Mar. 2017. URL: https://www.linux.com/news/event/open-source-leadership-summit/2017/3/bruce-schneier-new-security-threats-internet-things.

[12] Brian Barrett. *How To Stop Your Smart TV From Spying on You*. Feb. 2017. URL: https://www.wired.com/2017/02/smart-tv-spying-vizio-settlement/.

[13] R. Bonetto et al. "Secure communication for smart IoT objects: Protocol stacks, use cases and practical examples". In: *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. June 2012, pp. 1–7. DOI: 10.1109/WoWMoM.2012.6263790.

[14] Flavio Bonomi et al. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16.

[15] Nicola Cocchiaro. *FSFS: The Fast Secure File System*. http://fsfs.sourceforge.net/.

[16] John Costello. *New Mirai Variant Leaves 5 Million Devices Worldwide Vulnerable – High Concentration in Germany, UK and Brazil*. https://www.flashpoint-intel.com/new-mirai-variant-involved-latest-deutsche-telekom-outage. 2016.

[17] C. Doukas et al. "Enabling data protection through PKI encryption in IoT m-Health devices". In: *IEEE 12th International Conference on Bioinformatics Bioengineering (BIBE)*. Nov. 2012, pp. 25–29. DOI: 10.1109/BIBE.2012.6399701.

[18] Laurent Eschenauer and Virgil D. Gligor. "A Key-management Scheme for Distributed Sensor Networks". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS '02. Washington, DC, USA: ACM, 2002, pp. 41–47. ISBN: 1-58113-612-9. DOI: 10.1145/586110.586117. URL: http://doi.acm.org/10.1145/586110.586117.

[19] Dave Evans. "The internet of things: How the next evolution of the internet is changing everything". In: *CISCO white paper* 1.2011 (2011), pp. 1–11.

[20] Nikhil Goyal, John Wawrzynek, and John D. Kubiatowicz. "Global Data Plane Router on Click". MA thesis. EECS Department, University of California, Berkeley, Dec. 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-234.html.

[21] Vipul Gupta et al. "Performance Analysis of Elliptic Curve Cryptography for SSL". In: *Proceedings of the 1st ACM Workshop on Wireless Security*. Atlanta, GA, USA: ACM, 2002, pp. 87–94. ISBN: 1-58113-585-8. DOI: 10.1145/570681.570691. URL: http://doi.acm.org/10.1145/570681.570691.

[22] Vipul Gupta et al. "Sizzle: A standards-based end-to-end security architecture for the embedded Internet". In: *Pervasive and Mobile Computing* 1.4 (2005). Special Issue on PerCom, pp. 425–445. ISSN: 1574-1192. DOI: http://dx.doi.org/10.1016/j.pmcj.2005.08.005. URL: http://www.sciencedirect.com/science/article/pii/S1574119205000568.

[23] John H Howard et al. *An overview of the andrew file system*. Carnegie Mellon University, Information Technology Center, 1988.

[24] *IoT: number of connected devices worldwide 2012-2020*. Statista, 2014. URL: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[25] Chris Karlof, Naveen Sastry, and David Wagner. "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks". In: *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. Baltimore, MD, USA: ACM, 2004, pp. 162–175. ISBN: 1-58113-879-2. DOI: 10.1145/1031495.1031515. URL: http://doi.acm.org/10.1145/1031495.1031515.

[26] Eddie Kohler et al. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[27] Brian Krebs. *Akamai on the Record KrebsOnSecurity Attack*. https://krebsonsecurity.com/2016/11/akamai-on-the-record-krebsonsecurity-attack/. 2016.

[28] Brian Krebs. *Krebs on Security*. Oct. 2016. URL: https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/.

[29] Brian Krebs. *New Mirai Worm Knocks 900K Germans Offline*. https://krebsonsecurity.com/2016/11/new-mirai-worm-knocks-900k-germans-offline/. 2016.

[30] Brian Krebs. *Researchers Find Fresh Fodder for IoT Attack Cannons*. https://krebsonsecurity.com/2016/12/researchers-find-fresh-fodder-for-iot-attack-cannons/. 2016.

[31] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: NetDB. 2011.

[32] David B. Lomet. "Key Range Locking Strategies for Improved Concurrency". In: Morgan Kaufmann Publishers, Jan. 1993. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=68374.

[33] Ken Lutz. URL: https://swarmlab.eecs.berkeley.edu/projects/4814/global-data-plane.

[34] Joao Martins et al. "ClickOS and the art of network function virtualization". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association. 2014, pp. 459–473.

[35] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[36] Jan Medved et al. "Opendaylight: Towards a model-driven sdn controller architecture". In: *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*. IEEE. 2014, pp. 1–6.

[37] Michael Mimoso. *Mirai-Fueled IoT Botnet Behind DDOS Attacks On DNS Providers*. https://threatpost.com/mirai-fueled-iot-botnet-behind-ddos-attacks-on-dns-providers/121475/. 2016.

[38] Nagendra Modadugu and Eric Rescorla. "The Design and Implementation of Datagram TLS." In: *NDSS*. 2004.

[39] C Mohan et al. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging". In: *ACM Transactions on Database Systems (TODS)* 17.1 (1992), pp. 94–162.

[40] Nitesh Mor et al. "Toward a Global Data Infrastructure". In: *IEEE Internet Computing* 20.3 (2016), pp. 54–62.

[41] Adrian Perrig et al. "SPINS: Security Protocols for Sensor Networks". In: *Wirel. Netw.* 8.5 (Sept. 2002), pp. 521–534. ISSN: 1022-0038. DOI: 10.1023/A:1016598314198. URL: http://dx.doi.org/10.1023/A:1016598314198.

[42] Nathan Goodman Philip A. Bernstein Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Microsoft Research, 1987.

[43] Nikolaus Rath. *FUSE (Filesystem in Userspace)*. https://github.com/libfuse/libfuse.

[44] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. `http://www.rfc-editor.org/rfc/rfc6347.txt`. RFC Editor, Jan. 2012. URL: `http://www.rfc-editor.org/rfc/rfc6347.txt`.

[45] Sean Rhea et al. "Pond: The OceanStore Prototype." In: *FAST*. Vol. 3. 2003, pp. 1–14.

[46] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-tree filesystem". In: *ACM Transactions on Storage (TOS)* 9.3 (2013), p. 9.

[47] Mendel Rosenblum and John K Ousterhout. "The design and implementation of a log-structured file system". In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 26–52.

[48] Joseph Salowey, Abhijit Choudhury, and David McGrew. *AES Galois Counter Mode (GCM) cipher suites for TLS*. Tech. rep. 2008.

[49] Russel Sandberg et al. "Design and implementation of the Sun network filesystem". In: *Proceedings of the Summer USENIX conference*. 1985, pp. 119–130.

[50] Tom Spring. *Conficker Used in New Wave of hospital iot device attacks*. `https://threatpost.com/conficker-used-in-new-wave-of-hospital-iot-device-attacks/118985/`. 2016.

[51] Brian Warner, Zooko Wilcox-O'Hearn, and Rob Kinninmont. *Tahoe: A Secure Distributed Filesystem*. https://tahoe-lafs.org/ warner/pycon-tahoe.html.

[52] Kim Zetter. *Why Hospitals Are the Perfect Targets for Ransomware*. Mar. 2016. URL: `https://www.wired.com/2016/03/ransomware-why-hospitals-are-the-perfect-targets/`.

[53] Ben Zhang et al. "The Cloud is Not Enough: Saving IoT from the Cloud". In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. Santa Clara, CA: USENIX Association, July 2015. URL: `https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/zhang`.

[54] L. Zhu and B. Tung. *Public Key Cryptography for Initial Authentication in Kerberos (PKINIT)*. RFC 4556. RFC Editor, June 2006.