An Architecture for a Widely Distributed Storage and Communication Infrastructure

Nitesh Mor
Eric Allman
Richard Pratt
Kenneth Lutz
John D. Kubiatowicz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-130
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-130.html

August 28, 2018

An Architecture for a Widely Distributed Storage and Communication Infrastructure

Nitesh Mor, Eric Allman, Richard Pratt, Kenneth Lutz, John Kubiatowicz

Abstract

With the advancement of technology, richer computation devices are making their way into everyday life. However, such smarter devices merely act as a source and sink of information; the storage of information is highly centralized in data-centers in today's world. Even though such data-centers allow for amortization of cost per bit of information, the density and distribution of such data-centers is not necessarily representative of human population density. This disparity of where the information is produced and consumed vs where it is stored only slightly affects the applications of today, but it will be the limiting factor for applications of tomorrow.

The computation resources at the edge are more powerful than ever, and present an opportunity to address this disparity. We envision that a seamless combination of these edge-resources with the data-center resources is the way forward. However, the resulting issues of trust and data-security are not easy to solve. Toward this vision of a federated infrastructure composed of resources at the edge as well as those in data-centers, we describe the architecture and design of a widely distributed system for data storage and communication that attempts to alleviate some of these data security challenges; we call this system the Global Data Plane (GDP).[1]

The core idea of GDP is a secure single-writer append-only log, which provides a layer of uniformity on top of a heterogeneous infrastructure.[2] This secure single-writer log is a unified storage and communication primitive that represents a refactoring of interfaces. Such a refactoring enables cleaner application design that allows for better security analysis of information flows. Not only cleaner design, GDP also enables locality of access for performance and data privacy—an ever growing concern in the information age.

[1]Note that Global Data Plane is an ongoing project and is evolving continuously. This document is an adaptation of a doctoral thesis proposal and merely documents the ideas at a certain point in time. For more current information, please see the project web-page at `https://gdp.cs.berkeley.edu`.

[2]Since the time of this writing, we have changed the name of 'secure single-writer append-only log' to DataCapsule in the GDP. We believe that this change of terminology better reflects the desired properties and avoids the confusion with a simple log file.

Contents

1 Introduction

The world is moving towards a ubiquitous computation model. Not only do we see richer sensing and actuation capabilities making their ways into everyday objects, we also see a significant amount of computational resources closer to the edge.[3] However, when it comes to running services and applications, we still rely on primarily a cloud-centric model. The model of data-centers as the enormous pools of resources (cloud-computing) has its place, but many have argued that exclusively relying on a cloud-centric model limits the evolution of richer services and applications. A number of people have analyzed rich applications enabled by the widespread sensing and actuation capabilities and concluded that the resources at the edge are necessary to support such new applications [21].

The resources at the edge provide an opportunity for low-latency communication, higher bandwidth and lower overall energy consumption from a networking viewpoint, and improved data security, privacy and isolation by keeping the data within trusted domains from an administrative viewpoint. Even with these advantages of using the resources at the edge, there are very few applications that make use of such resources—most of these applications are highly custom applications running in very specific environments such as a factory floor. We argue that the reason for such a low adoption of edge-computing paradigm is that an application developer's interface to the edge-computing infrastructure asks for too much.

In order to use the resources closer to the edge, application developers need to perform quite a bit of system administration. The high level tasks of resource management to achieve the desired QoS requirements for services/applications and adhering to the security/privacy concerns of the user data quickly translate into low-level mundane jobs such as keeping the systems updated, ensuring correct system configuration, maintaining firewalls, dealing with system/network failures, etc. The large number of smaller and potentially untrusted administrative entities with heterogeneous resources makes the edge-computing ecosystem a much more challenging environment to work with than the cloud-computing model, where a few service providers with homogeneous resources dominate the market. The net result is that just to be able to use the infrastructure and ensure the security of their computation/data, application developers need to be an expert in both distributed systems and computer security.

1.1 Toward Higher Levels of Abstraction

The additional burden of managing systems and infrastructure in edge-computing is somewhat similar to the challenges of Infrastructure as a Service (IaaS) offerings in the cloud computing paradigm [46]. To enable application developers focus on writing applications, various cloud-based service providers offer a Platform as a Service (PaaS) where an application developer works with a higher level interface for individual resources (compute, storage, etc). As opposed to the IaaS approach of forcing application developers to configure and manage systems directly, the *platform* provides application developers with a way to convey the desired performance/security properties to the infrastructure [28, 20].

[3] By edge resources, we don't mean the hundreds of data-centers around the world controlled by some of the most powerful corporations in the world that are at the edge of the cloud [7, 1]. There is another layer of mostly decentralized resources managed by individuals, smaller corporations, municipalities and other organizations that are at the edge of the network. Many of these resources are in homes, offices, public infrastructure and elsewhere; a large fraction of these resources are behind firewalls or NATs and aren't even reachable on the public Internet directly. The resources in this hidden layer are what we call edge resources.

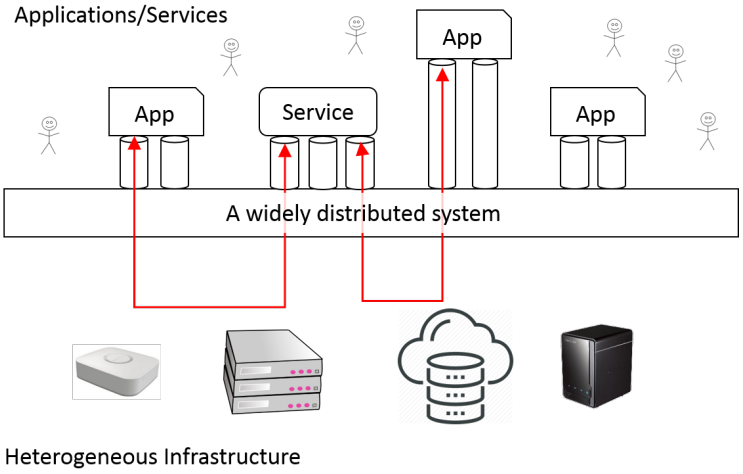


Figure 1: A widely distributed system spread over heterogeneous infrastructure that provides a uniform *platform* for applications (see § 1.1).

Additionally, the principle of separation of persistent state from compute is a very useful principle adopted by various cloud service providers, both in IaaS and PaaS paradigms. Such a separation is useful for the following reasons: the requirements and mechanisms for computation and storage are quite different. Computation is transient which makes it fungible, i.e. one can restart a failed/interrupted computation elsewhere to recreate the desired results. On the other hand, data is persistent; one needs to preemptively replicate data in order to recover from corrupted storage and ensure its security at all times. Hence, it is desirable to replicate data widely for durability, but such a mechanism isn't necessarily needed for computation. Finally, a separation of computation from storage allows service providers to specialize in either storage or computation, and not necessarily both. Thus a separation of compute from state allows for the most flexibility both for application developers and infrastructure providers.

We ask the question whether these two core ideas—higher level *platform* like abstractions and a separation of compute from state—can be applied to the edge-computing ecosystem? In particular, in this document we explore the idea of a storage and communication platform in the form of a widely distributed system spread over the heterogeneous infrastructure managed by a large number of administrative entities, but provides a homogeneous interface to the application developers. We envision that such a system can provide a seamless integration of resources at the edge with those in the cloud, if done right. Other than the typical properties of a typical distributed storage system (scalability, fault-tolerance, durability, etc.), let us take a look at the additional requirements needed to support the edge-computing paradigm:

- **Homogeneous interface:** First and foremost, such a system should provide a homogeneity of interface even though the infrastructure underneath can be quite heterogeneous. This allows application developers to create portable applications and avoid stove-piped solutions. It is also useful if the interface can support a wide-variety of applications natively, or allows for higher level interfaces to be created on the top. Even though many distributed systems achieve this by default, we think it is important enough property to be listed explicitly given the heterogeneous infrastructure underneath.

- **Federated architecture:** Such a system should not be restricted to a single (or a handful of) administrative entities; instead, anyone should be able to contribute their resources and be a part of the platform.[4] As opposed to the cloud ecosystem where only a few large players dominate the market, the edge-computing ecosystem has a large number of administrative entities that vary quite a bit in the amount of resources they control. For a truly federated architecture, reputation should not give an unfair advantage to large service/infrastructure providers; we argue for a baseline of verifiable data-security to make it a fair playing field for smaller providers (see below).

- **Locality:** In addition to a federated infrastructure, it is also important to maintain locality for two reasons. First, using local resources allows one to achieve the desired properties of low-latency and real-time interactions that the proponents of edge-computing have advocated [21]. Second, for privacy of highly sensitive data, it might be desirable to either limit the access to only local clients or use local resources that may be more trusted to not engage in sophisticated side-channel attacks. Finding such local resources relative to a client usually requires a global knowledge of the routing topology, thus such functionality is best achieved when assisted by the network itself (e.g. network assisted *anycast*) which hints towards an overlay network of some kind. However, such global routing state, if not managed properly, can be corrupted by adversaries (see 'secure routing' below).

- **Secure storage on untrusted infrastructure:** The infrastructure should provide a baseline verifiable data security (data confidentiality and data integrity) even in the face of potentially untrusted infrastructure. In the cloud ecosystem, there are few if any commercial storage offerings that provide any verifiable security guarantees. As a result, the cloud ecosystem is powered by trust based on reputation—a model that is favorable to large service providers and provides a significant barrier to entry for smaller, local service providers. Enabling secure interfaces allows for a utility model of storage where smaller but local service providers can compete with larger service providers.

- **Administrative boundaries:** Even though a homogeneous interface is desired from such a system, the system should provide some visibility of administrative boundaries in the infrastructure to an application developer if desired. Ideally, an application developer should be able to dictate what parts of the infrastructure to be used for specific data for two reasons. First, as hinted above, concerns over data privacy and security—especially for highly sensitive data—may require that the data does not leave a specific organization. Second, an application developer should be able to form economic relations with a service provider and hold them accountable if the desired Quality of Service (QoS) is not provided by the service provider.

- **Secure routing:** Data security in transit is equally important as data security at rest and simply encrypting the data is not sufficient in many cases. Encryption does provide a baseline of data confidentiality, however any adversarial entity monitoring communication in real time can learn enough information from the side-channels (such as size and timing of messages) [16]. In a federated infrastructure where the system routes a reader/writer to a close-by resource, it becomes easy enough for third party adversaries to pretend being such a close-by resource and either perform man-in-the-middle attacks or simply drop traffic (effectively creating a

[4] An *administrative entity/domain* in edge-computing could be an individual with a small smart-hub in their home, small/medium business with a closet full of servers, a large corporation with their own small data-centers, a large scale cloud service providers with massive data-centers, or anything in-between.

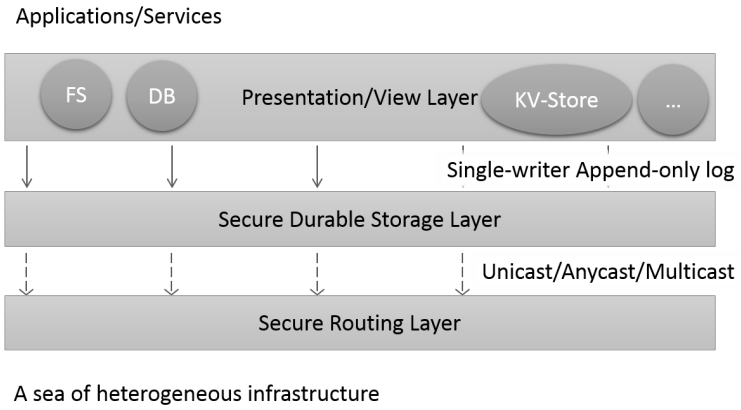


Figure 2: A refactoring of interfaces for a cleaner design of a widely distributed system (see § 1.2). A secure single-writer append-only log interface acts as a bridge between the "presentation/view layer" and the "secure durable storage layer". Similarly, the "secure durable storage layer" utilizes unicast/anycast/multicast to utilize the "secure routing layer underneath". See § 1.2 for details.

black-hole)—a problem well studied in overlay routing schemes [53]. The two requirements—anyone can be a part of the network and adversarial disruption of routing state—would appear to be at odds, and a system should provide a solution or a workaround to such conflicting goals.

- **Publish-Subscribe and multicast:** The usefulness of a storage system increases exponentially when communication is an integral part of the ecosystem; such ideas have been well studied in economics of communication (*network effect*) [32]. Toward this goal, we believe that a publish-subscribe is an important paradigm of communication for a storage system that enables composability of services and applications. It is equally important to have network assisted secure *multicast* schemes for an efficient use of often limited network bandwidth.

A system that meets these high level requirements—security and locality being the two most important—provides application developers a *secure ubiquitous storage* platform. Such a system is general enough to support a wide variety of deployment scenarios: a service provider model with high density of resources; private corporations with restricted data flows; strict control loops in industrial environments with explicitly provisioned resources; a strictly data-center model similar to the cloud; or any combination of such scenarios. Not only is such a federated infrastructure better than stove-piped custom solutions, clear interfaces that separate compute from state allow for a *refactoring* of applications for a better security audit of the *information*—one can reason about the information flows by analyzing well defined entry-points and exit-points of information in an application.

1.2 A System Design by Refactoring of Interfaces

In this document, we provide a high-level architectural design as well as enabling mechanisms for a system that achieves the goals described above. Instead of designing a monolithic system, we use a three-layered approach by performing a refactoring of functionality and interfaces—this ensures that simple applications stay simple and complex applications can be built cleanly and easily. In this refactoring, at the very bottom is the "secure routing layer": a flat address-space

routing network that works across administrative domains while preserving locality and provides a communication fabric for the system. In the middle is the "secure durable storage layer" that provides the core storage functionality of the system. Finally, on the top is the "presentation/view layer" that provides higher level APIs to applications and fulfills application specific requirements (see Figure 2). Let us take a look at these layers in detail:

- *Presentation/View layer* is a layer that provides rich interfaces to applications/services; such interfaces could be that of a file-system, a key-value store, a database, a multi-writer log, and so on. This layer mostly concerns itself with *what* should be written to persistent storage and *how to order* such writes; making the data durable is the task of the middle layer. The interface to the layer below is that of a secure single-writer append-only log. This layer is rather open to systems' integrators and they can put together an interface of their choice that uses these single-writer logs underneath to meet application specific requirements such as a desired semantics for reads/writes, consistency issues, access control, update ordering, etc. The simplest of the applications can even bypass this layer and use the single-writer logs directly.

- *Secure Durable Storage Layer* is where the information is made persistent. The core interface—secure single-writer append-only log—is the unifying homogeneous interface that we envisioned in the previous section. In addition to reads, this single-writer log also supports a publish-subscribe mode of operation, enabling high-performance event-driven applications. The task of this layer is to make the information durable and available to the appropriate readers while maintaining the integrity of data. It uses secure unicast, anycast, or multicast provided by the "secure routing layer" underneath to efficiently replicate and migrate information across domains (as allowed by the policy specifications) and enable locality of access for readers/writers to satisfy their performance requirements.

- *Secure Routing Layer* provides the communication fabric for secure delivery of information. Instead of a host-to-host network, this is a higher level information centric network where one communicates directly with services, data, or in the general case—principals. The names/addresses of such principals are decoupled from the names of physical hosts; each such principal has its own identity that is independent of where it is located or who owns it. The ownership of a principal may define how to access it or who can access it, but not what such a principal truly is.

Why is this refactoring the right thing? Such a refactoring provides two benefits. First, cleaner interfaces benefit application developers and allow them to perform a better security and performance analysis of their applications. For example, the access control issues are handled at the "presentation/view layer", whereas data durability is handled at the "secure durable storage layer". Similarly, the scope of data, i.e. where can data reside, is handled at the "secure durable storage layer", and the "secure routing layer" ensures that the information does not flow through hostile routing infrastructure. Second, such a separation of requirements and concerns benefits infrastructure provider by enabling them to focus on individual aspects of the service they provide to applications (and even specialize in specific tasks). For example, adjusting the level of durability for an application can be handled independently of data integrity or consistency issues.

Why the single-writer append-only log interface? An append-only design is the most natural choice for streams of information. The append-only design enables us to support a publish-subscribe mode of access natively, since append has similar semantics as 'publish'. Further, such an interface

makes the design of system simpler in following two ways. First, append-only design makes old data immutable, which transforms data-consistency problems into a data-freshness problem, thus simplifying the mechanisms for replication on the distributed infrastructure underneath. Second, with a single-writer design, the writer can not only attest individual updates, but also fix the ordering of such updates; this again simplifies the functionality of the infrastructure to simply making the data durable and available to appropriate readers. A single-writer append-only log is a narrow, yet sufficient interface to enable other higher level interfaces, such as that of a file or a multi-writer log that can be created at the "presentation/view layer".

Why unicast/anycast/multicast as first class communication principles? As we just described, the "secure routing layer" is an information-centric network that enables principals (such as bundles of data) to communicate with each other directly instead of relying on a host-to-host communication primitive. Physical hosts in IP-based networks mostly have unique identities and they can work well with one-to-one communication strategies such as unicast. In contrast, the replication of data bundles in the "secure durable storage layer" implies that the corresponding principals being available at multiple locations in the network is the norm and not the exception. Such multiplicity of principals makes anycast/multicast a more appropriate mode of communication and a key enabler for locality.

1.3 Global Data Plane: A Storage and Communication Platform

Inspired by the idea of a *platform* for *secure ubiquitous storage* and the high level design that we just described, we explore the detailed architecture, mechanisms and implementation of a widely distributed and federated storage and communication infrastructure in this document, which we call the Global Data Plane (GDP). The key interface in GDP is that of the single-writer append-only secure log that represents an ordered stream of messages (called records) generated by a single data producer (writer); such single-writer log can be used as a building block for richer interfaces (see Figure 2). At a high level, GDP is to a log is what a distributed file-system is to a file. To reemphasize, the single-writer log concept is a way to refactor the application interface to the infrastructure: the computation and application logic to generate and order the information is handled by applications, whereas the log is simply a way to make such information persistent and available to others in a secure and verifiable way. As we will show in § 3.4, such an interface is also a natural fit for numerous simple sensors producing time-series data.[5]

Logs are identified by a flat 256-bit *GDP name*; such a name separates data from the physical infrastructure underneath providing location independence and also serves as a cryptographic trust anchor for data integrity. Separating data durability from data ordering also allows a writer to decide on the importance of durability for individual updates, thus allowing for applications on both ends of the performance/durability spectrum—a high performance video feed that can tolerate a small number of missing frames as well as a database where individual transaction must absolutely be made durable. In addition to append by a single writer, logs support publish-subscribe mode of operation (where the single-writer is the only publisher), secure replays at a later time (thus providing a time-shift property), and efficient random reads for old data. An important consequence of the publish-subscribe mode is decoupling of a data producer from consumer, which makes a log into a unidirectional real-time communication channel. Thus, a log can be viewed as a unification of data security at-rest and in-transit. As a result, an application can exclusively rely on logs for all its state maintenance and communication with other applications/services, making the log a

[5] Also see Appendix A.

narrow uniform interface to utilize the heterogeneous infrastructure underneath (see Figure 2). We provide a detailed interface of the single-writer log in § 3.2.

At a physical level, GDP is a distributed system of heterogeneous storage and routing nodes; storage nodes (called *log-servers*) provide a physical backing for log storage, routing nodes (called *GDP-routers*, or simply *routers*) perform secure data routing. The physical infrastructure is managed by various administrative domains (organizations); an administrative domain could operate log-servers and play the role of a *storage organization*, or operate routers and create a *routing domain*, or do a combination of both. Just like logs, such organizations and their nodes—log-servers and routers—are also identified by a flat 256-bit name. Using a flat cryptographic name as the trust anchor (as opposed to traditional hierarchical Certificate Authorities) allows for a truly federated system—an infrastructure provider can join the system without going through the bureaucratic process of obtaining a certificate. At the same time, the security of the entire system does not depend on a handful of certificate authorities that may go rogue or be pressured by authoritarian governments to hand over their private keys.

Using cryptographic tools, GDP provides various security guarantees (integrity, confidentiality, etc.) for stored data while requiring little trust from the underlying hardware/software and administrators. A user explicitly delegates the data storage to one or more storage organization by using cryptographic certificates; this allows a user to make economic contracts with service providers and/or hold them accountable if the agreed upon performance/locality requirements are not met. Similarly, the routing scheme relies on cryptographic certificates to ensure that users have control on data flows and scope of data. Users (or storage organizations) can delegate the routing functions to semi-trusted routing domains; however a secure routing scheme ensures that adversarial entities can not compromise the routing state to perform sophisticated side-channel attacks. In addition, GDP works on a datagram based network that can work well with an *anycast* scheme and provide access to the nearest copy of data. To summarize, the cryptographic techniques allow users to use local resources without necessarily trusting them for data security in presence of a threat model that we formally define in § 3.1.

In comparison to existing systems, GDP explores a relatively unexplored design point: a secure single-writer log interface directly exposed to writers/readers with a separation of durability concerns. A number of systems have used logs as a design tool, but either such logs are used internally on the server side, or are used in fully trusted environments. We adopt a single-writer log as a first class data structure, where the infrastructure is only supposed to make it (a) durable, and (b) available to appropriate readers. This enables us to use a leaderless replication algorithm allowing for a quick path between a writer and a reader for real-time pub/sub mode of operation. We take a deeper look at existing academic work in the related areas in § 2.

Although motivated by the needs for secure ubiquitous storage, GDP is not restricted to resources on the edge and can readily be used for traditional computation scenarios. In particular, GDP log-servers and routers can be deployed on existing cloud infrastructure to enable seamless integration of cloud with the edge or "fog". Such seamless integration enables GDP to provide an opportunity to use local resources at the edge wherever possible for low-latency access and using cloud resources for durability. Even though application developers can use a combination of existing tools and technologies to achieve similar functionality as GDP, it is quite a burden on the users to use these tools correctly and requires domain expertise in both computer security and distributed systems (see § 2 for a few such combinations). GDP provides an infrastructure with cross-layer optimization of some of these existing tools focusing on efficiency and a clean application interface, and provides a sufficiently high-level interface to be used by application developers.

1.4 Research Questions and Tasks

We break down the design of the Global Data Plane into following three high level research tasks. Note that the following way of division is not because these tasks are independent of each other, but because each of these tasks targets a specific functionality that can be analyzed and evaluated independently (see § 5). Even though we solve these problems in the context of a specific system, we believe that the ideas and the results can be applicable to other scenarios that demand similar requirements.

Task 1 *Design of a single-writer append-only secure log that can be distributed over a federated infrastructure, and provides data provenance for every single bit of data and the ordering relationships between such bits; this is essentially the design of the middle layer in Figure 2.*

First and foremost, this task involves API specification of log for the readers, subscribers, designated single-writer, and the infrastructure, which we describe in § 3.2 in the presence of a threat model formally defined in § 3.1. We discuss the internal mechanisms in § 4.1 and address two main sub-challenges: (a) design of an efficient log-structure that enables a user to achieve application-specific performance/durability goals while preserving data-integrity, and (b) design of an efficient leaderless replication strategy. We assume there is a secure communication protocol (Task 2) on a secure routing network underneath (Task 3). The outcome of this task is evaluated by using micro-benchmarks that measure the security overhead of log operations (cryptographic operations, required local state, etc.) and macro-benchmarks that compare the effectiveness of a leaderless replication scheme with a few other architectures that we describe in § 2.3.

Task 2 *Design of efficient end-to-end secure datagram based protocol for state update/query operations on logs (an authenticated data structure) that works in a anycast/multicast setting; such protocol bridges the middle layer and base layer of Figure 2.*

This task essentially involves a cross-layer optimization by ensuring there is minimal duplication of functionality (such as integrity protections) already achieved by the log-layer, and looks at efficient mechanisms with minimal state maintenance to support anycast/multicast scenarios. This task involves more engineering than research, but at the same time requires a careful consideration of any side-effect a naive optimization may have. We discuss such analysis and the mechanisms in § 4.2 and hope to address the two sub-challenges: (a) efficient and (mostly) stateless secure log-protocols with secure acknowledgments that work well with *anycast*, and (b) secure and efficient acknowledgments for *multicast*. The effectiveness of this task is measured in terms of cost of communication on the wire (size and number of messages, number of round-trips, etc), as compared to existing tools for secure communication channels.

Task 3 *Design of a datagram based secure routing network with flat address-space (base layer in Figure 2).*

This task involves the design of a scalable and secure communication fabric that provides a datagram based interface for communication between cryptographic principals using primitives such as unicast, anycast and multicast, while respecting administrative boundaries and locality. In § 4.3, we describe the mechanisms to address the two main challenges (a) secure unified mechanisms for advertisements, and (b) routing across administrative domains while providing transitive proofs of routing delegation. The outcome of this task is measured by how well the network behaves under failures when deployed on a large number of nodes in cloud data-centers across the world in a simulated network topology.

In this section, we described a high level overview of the architecture of the Global Data Plane and the refactoring of interfaces it achieves. We also provided a gentle introduction to the system and listed our research goals. In § 2, we provide an overview of the existing work in related fields, followed by a classification system for assessing the strengths of various existing systems objectively; we conclude with some combinations of existing systems and demonstrate how they fail to achieve the goals of the GDP. In § 3, we formally describe the threat model GDP aims to address, followed by the user-interface and an application model as an example for developers.

In § 4, we describe the key enabling mechanisms for the GDP, and elaborate on the individual research tasks introduced in this section. In § 5, we describe an evaluation strategy for the overall system and three research tasks. We finally conclude in § 6. Additionally, in Appendix A, we describe how a system like the GDP can alleviate the security and performance challenges that plague the IoT ecosystem.

2 Related Work

In the previous section, we argued for the necessity of a secure ubiquitous storage platform and how such a platform enables a new set of applications that are too difficult to build otherwise. One might ask why design something new and not reuse existing systems or tools. In this section, first we provide a summary of previous work in categories broadly related to this goal. We then describe a classification system to objectively assess the properties of a given system and show where existing systems/tools fall short in achieving a truly ubiquitous and secure storage interface for application developers. Towards the end of this section, we demonstrate the engineering challenges when attempting to achieve the desired properties by using a few *steel man* systems[6]—which we later use in § 5 to evaluate the effectiveness of our leaderless replication strategy.

2.1 Existing systems

The 'secure ubiquitous storage' challenge pushes the limits of both data security and distributed systems architecture design, and these sub-challenges need to be solved together. Existing systems only work with one sub-challenge at a time. However, when attempting to combine these individual solutions, one finds that such solutions do not compose well and often lead to inefficiencies or practical engineering challenges.

To illustrate this point, recall that verifiable security of data—both at-rest and in-transit—is a necessity for enabling a secure ubiquitous storage platform. Considerable academic research has already been done regarding techniques and systems for verifiable storage of data on potentially untrusted infrastructure. Even if we were to assume for a moment that 'secure storage on untrusted infrastructure' is a solved problem, this is only a part of the solution to the challenge of secure ubiquitous storage. As we will show soon in the next section, existing solutions treat the security of data at-rest separately from security of data in-transit; such systems lack security of data flows or even a notion of *scope* of data. Just side-channels on encrypted data flows can leak enough significant information, and thus it is necessary to control where data flows [16]. Consider a simple *smart* garage door opener that can enable open/close commands using a secure encrypted channel over the Internet; an adversary simply observing such an encrypted channel has enough information to figure out when someone leaves the house.

A slightly different yet related problem is that of finding the appropriate parts of infrastructure to handle a request from a client (writer/reader) is a non-trivial challenge in a federated environment—many existing systems do not address this problem. An adversary that can break this discovery process can place itself as a passive man-in-the-middle and easily analyze traffic at will. Even worse, a man-in-the-middle adversary can just drop all write requests and send spoofed acknowledgments.[7] Such requirements often translate to a secure communication tool being used together with such secure storage systems, which not only does not address the side-channel analysis issue but also leads to inefficiencies and duplicated effort—especially noticeable for small payloads such as in Internet of Things (IoT).[8] With many federated infrastructures, the transport security

[6] A steel man, in this case, is an ideal abstract implementation of a given system. In contrast, a straw man is a hand-picked system that can be used to defend practically any argument.

[7] One might argue that a writer can immediately read the data after writing in order to ensure that it has been stored appropriately by the designated server. Such a defense is probabilistic in nature, requires extra work on behalf of the writer, and can be thwarted by a motivated man-in-the-middle that can reply differently based on who is asking.

[8] A similar inefficiency argument is made in [18] which strengthens our claim, however we address the question of end-to-end security in a much more broader context than in [18].

is even worse than simple IP-based networks, which is why it is important to have mechanisms in place for better control on sensitive data.

Not only do the existing systems lack a full solution to the secure ubiquitous storage problem, the assumption of treating 'secure storage on untrusted infrastructure' as a solved sub-problem isn't completely correct either. Many existing solutions are strictly focused on the data security, and aren't suitable for a distributed infrastructure as it is, leaving out important details such as the consistency issues that distributed storage systems have to solve. The few existing systems that do consider the storage servers as a distributed system usually assume only a single administrative domain; the nuances of a federated infrastructure with data spread over potentially a number of administrative domains makes them unsuitable for our goal of secure ubiquitous storage. As we will discuss shortly, the performance requirement of low-latency requires careful placement of mutually distrustful servers—typically in close proximity of each other—for storage schemes that involve quorum based algorithms. On the other hand, simple master-slave based algorithms require careful master selection to achieve good performance when nodes are spread over a wide geographical area. Thus, an Internet-wide federated system for secure handling of data has to take the implications of its widely distributed nature—both routing latencies and security of data flows—into account.

Rather than designing a new system altogether, can we modify existing systems and applications or use a combination of existing tools, and achieve the desired goal of making them work in a federated environment and untrusted infrastructure? Often, when faced with such a question, application developers present a simplistic viewpoint: 'lets add encryption to an existing system'. Such a viewpoint undoubtedly is very unsettling, since there are a variety of scenarios overlooked by just encrypting individual data items; an approach like this leads to insecure applications in the worst case and very specific point solutions in the best case. A little more acceptable approach is to use a combination of well defined tools as layers of abstraction, but it has two downsides. First, a naive combination of existing tools as various layers of abstraction can often miss the subtle requirements and guarantees from such abstractions, leading to unsolved engineering challenges or broken systems altogether.[9] Second, duplicated functionality across various layers of abstraction often leads to inefficiencies and performance penalties; such inefficiencies become relatively more pronounced for applications with small data items but that require strict performance guarantees (such as those involving tight-control loops). It is our opinion that application developers benefit by having access to well-defined abstractions, as opposed to being burdened with combining such tools themselves; we demonstrate these shortcomings by using a few example combinations in § 2.3.

Let us take a brief look at some broad classes of systems that we borrow ideas from:

Secure storage: A number of existing storage systems provide a high-level interface to the end-users in the face of untrusted storage servers. The exact interface varies across various systems, e.g. SUNDR [38], Oceanstore [36], and Plutus [34] provide a file-system interface on untrusted storage servers; CryptDB [44] provides a database interface; Depot [39] provides a key-value store interface; Antiquity [55] provides a log-based interface; and so on. Additionally, a number of existing systems have specific goals in specific environments; SiRiUS [31] targets users that can not modify remote storage server; CloudProof [43] aims to hold cloud storage servers accountable for violating data security; and so on. The general mechanisms for such systems fall in two broad categories:

[9] As an example: imagine a widely distributed simple content distribution service $\mathbb{S}$ that provides read only copies of data, similar to various existing commercial CDN services with the exception that $\mathbb{S}$ uses roadside cabinets for a true edge-computing style operation as opposed to traditional CDNs that still adopt a data-center approach with restricted physical access. A seemingly simple deployment of HTTPS with session reuse (for performance) can be quite challenging to implement in a way that compromise of a single machine does not result in compromise of the entire infrastructure, especially in a world where certificate revocations are poorly implemented.

cryptographic solutions that rely on on possession of keys and validations of cryptographic proofs,[10] or fault-tolerant distributed systems that assume that a certain fraction of storage servers to be honest at all times.[11] The GDP architecture falls in the former category, which allows us to use an efficient leaderless replication algorithm and avoid communication penalties associated with the architectures in the latter category.

Secure communication: For enabling secure communication between two endpoints in the presence of network level adversaries, a number of high-level tools are available to provide 'secure-channels'; e.g. IPsec/VPN for generic secure tunneling service, SSH for secure shell and other tunneling applications, TLS/DTLS for securing other protocols such as HTTP, SMTP, etc. However, all of these high-level tools assume that the endpoints at the secure channel are trusted, and usually only work well with a unicast scheme. We borrow ideas from such existing tools: the GDP protocol effectively provides a unidirectional secure communication channel for log operations and uses some mechanisms similar to TLS; GDP routing domains have few similarities to the core concept of a VPN.

Publish-subscribe systems: Publish-subscribe (pub-sub) style communication enables decoupling data producers from the consumers, and has been widely recognized as a useful design pattern for enabling composability and micro-services in Internet of Things (IoT) [2] and data-processing in general. In addition to pub-sub systems to handle high volume of data in trusted single-administrator environments (e.g. Apache Kafka [33]), a number of scalable Internet-wide pub-sub systems also have been devised (see [27] for an overview). However, there are a number of security issues that a scalable multi-administrator pub-sub system ought to consider (for an overview of security challenges, see [54]). GDP adopts the pub-sub mode of operation for a log to achieve a high level of composability. In addition to subscribing to real time data, a reader can also read old data in bulk efficiently—something that only a few single administrator systems consider but without any end-to-end data integrity guarantees [33].

2.2 A classification system

Comparing existing systems that are designed for different use-cases is a challenging subjective exercise. In order to assess the applicability of existing solutions to our high-level goal, we identify a broad set of system properties. For each of these properties, we propose a scale that attempts to capture the subjective nature of such properties. Even though this approach is not perfect and misses certain nuances, we believe that it serves the purpose of showing the strengths and weaknesses of a particular system.[12] Note that we use a level '0' for situations where a specific property is trivial to achieve. Additionally, the parameters are specifically chosen to emphasize where existing systems fail to meet the desired goals of secure ubiquitous storage.

[10] Examples of cryptographic tools are: Digital signature by a writer that allow a reader to verify the integrity of a blob of data; Authenticated Data Structures (ADSs)—such as Merkle Hash tree—that generalize this to a data structure and enable a reader to verify data-integrity by means of a proof; encryption that can be used for data secrecy; and so on. [51]

[11] The basic building blocks used are primarily solutions to the Byzantine Generals' Problem; examples include PBFT [22], A2M-PBFT [23], Byzantine-Paxos, Blockchains, etc.

[12] Some of the following may seem comparing apples to oranges. Such an analogy may be true in certain cases, however such comparison ought to be looked at comparing the size, ripeness, or any other property of fruits—apples and oranges just happen to be two such fruits chosen for comparison.

2.2.1 The distributed nature of a system

The move towards a distributed system is typically in order to achieve more abstract properties such as data durability achieved by replicating data, locality of access by introducing caches/replicas, scalability in amount of data by splitting data storage across a number of machines (sharding), scalability in number of requests by introducing replicas, etc. Since such higher level goals are usually intertwined and hard to characterize, we simply look at how widely distributed a system is. Higher levels indicate a wider spread of the system. This classification is in context of storage systems, but could be applied to a wide variety of distributed systems.

D0 No distributed storage; only a single server that stores all the data. There may be caches of data, but ultimately the single server is the master copy of data. Example: NFS, AFS.

D1 Multiple servers in a single data-center (or similar facility) managed by a single administrative domain. The multiplicity of servers is primarily used to achieve scalability, but could also be used for durability. Example: Google File System [29].

D2 Multiple servers distributed across data-centers, but still in a single administrative domain. This is roughly the same as above, but explicitly considers situations where Internet-wide latencies make certain operations challenging. Example: Google Spanner [24].

D3 Multi-cloud architectures that introduce a multiplicity in number of administrative domains. The assumption is that individual administrative domains have investment in the infrastructure (acquires appropriate DNS name, SSL/TLS certificates from well-known CAs, typically assumes distributed highly-available infrastructure within each domain). Example: email infrastructure.

D4 Widely distributed architectures with edge-computing, where each administrative domain could be as small as a Single Board Computer (SBC) like a Raspberry-Pi in a home, or as big as a cloud data-center. Typical Peer to Peer (P2P) architectures fall in this category. Example: OceanStore [36], BitTorrent [45]. GDP also falls in this category.

2.2.2 System Security and Availability

We focus on three high-level properties that a user cares about: data confidentiality, data integrity and the general system/data availability. Data integrity can be further categorized into two subparts: (a) integrity of individual updates (or messages), and (b) ordering of updates. We assume that any distributed system has some kind of fault-tolerance property to address failed or corrupted nodes. However, not all systems can work in presence of malicious/adversarial nodes; various existing system architectures strive to achieve some kind of tolerance for such malicious nodes in the system.[13] Further, various systems differ in the remedial action available to end-users in case of a violation of one of these properties. Based on these high-level characteristics, we define a scale as follows.

T0 The architecture assumes fully trusted nodes. Many single-administrator big data systems running in cloud data-centers fall in this category, such as Google File System [29]. To an end user, such systems do not provide detection/remedy of violation of integrity or confidentiality; a user has to trust the system to do the right thing.

[13] The net result for users is whether trusting the system as a whole implies trusting each node individually, or not.

T1 The architecture assumes fully trusted node, but any single node (and the system as a whole) is not trusted for confidentiality. CryptDB [44], encrypted data in a cloud blob storage service, etc. are examples of such situations; a user can get an implicit proof of confidentiality because the data can be decrypted only by keys possessed by users.

T2 The system requires all nodes to be trusted for correct operation, i.e. no tolerance for any malicious nodes. However, the system can still provide some integrity/confidentiality guarantees to end-user. Thus, any single node (and the system as a whole) is not trusted for confidentiality or integrity. A user can only detect integrity violation with access to minimal remedial action.[14] e.g. CloudProof [43].

T3 The system tolerates a *fraction* of malicious nodes and still continue correct operation. Any individual node can be compromised, but the system as a whole still provides some guarantees for data integrity and confidentiality as long as the threshold isn't crossed. The system may provide limited remedial means in case a client detects violation of integrity. e.g. OceanStore [36].[15]

T4 The system continues correct operation as long as there are at least a *constant* number of non-malicious nodes in the system. Any number of malicious nodes colluding together can not corrupt the state of the data on a correctly functioning node, however they may actively deny a user access to the system. e.g. Single-writer logs in GDP.

2.2.3 Update Path from Producer to Consumer

Since we are interested in supporting low-latency communication path between a data producer and a consumer, we would like to know about the critical data path for a given architecture. We exclusively consider systems that involve at-least one storage server (or a pub-sub broker) in the path between a producer (publisher) and a consumer (subscriber); multicast schemes that consider storage a secondary goal are out of scope for this discussion. Note that this kind of classification enables one to distinguish the systems that have archival storage as their primary goal. With this goal in mind, the following scale looks at the update propagation path: given the consistency/durability model a system tries to achieve, how many storage entities need to be contacted before an update/write/message from a producer/writer/publisher makes its way to a consumer/reader/subscriber.

C0 The trivial solution: non-distributed systems with a single copy of data, or a master-slave architecture where master acts as the only point of serialization and slaves are merely for backups. Path of data is typically: producer ⇒ server/master ⇒ consumer. Usually, such systems achieve strict consistency without any update conflicts. Note that the master is usually manually assigned; if a master is elected using a leader-election process, then it starts getting close to a distributed consensus system, which we consider in the next level.

C1 Multiple servers with strong consistency: A minimum number of servers are contacted and ordering is decided by the server side. This includes systems where a leader—either manually

[14] Since the system ceases to function correctly in presence of malicious nodes, violation of integrity would typically imply a malicious node leaving little room for remedy.

[15] In OceanStore, a collection of nodes is trusted for ordering and the data is spread over a large number of nodes using erasure codes. A client can read again for corrupted reads (remedial action), however there is no detection or remedy in case the untrusted server threshold was crossed during write ordering.

assigned or explicitly elected—is merely acting as a coordinator, and a number of round-trips are required between the servers. Usually such systems also achieve strong consistency with no conflicts that need to be resolved at a later time. Examples: Two phase commit, three phase commit, Paxos, Raft, Byzantine Fault Tolerant systems. Usually, ordering of updates is combined with durability but there are exceptions where ordering is separated from durability, such as in OceanStore.

C2 One or more servers with weaker consistency/serializability: In systems like Dyanmo [26] or Bayou [52], a writer contacts one (or more) servers, and updates are propagated optimistically. There maybe update conflicts, but the applications can tolerate and resolve such conflicts. The ordering of data is decided by the server side using a number of strategies, and applications can see out of order updates (see [49] for an overview), but it allows for a faster path between a data producer and a consumer, leading to higher performance.

C3 One server with a single-writer data-structure: In a system such as GDP, a single-writer data-structure contains data ordered by the designated single-writer, as opposed to the server-side infrastructure. Only a single server is sufficient to act as an intermediary between a data producer and consumer. Data durability is separated from ordering and applications do not see out of order updates, as opposed to the previous level. Such systems allow for potential loss of data when the applications are tolerant of missing data (e.g. video frames); it is the writer's responsibility to ensure sufficient durability.

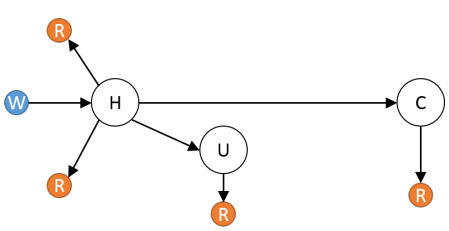
2.3 A Steel Man System

In this section, we create a steel man solution for a *secure ubiquitous storage* using existing tools that roughly follows the discussion in § 1. The novelty argument of GDP implies that (a) no existing systems enable a user to achieve the desired properties out of the box, and (b) a combination of existing systems poses non-trivial engineering challenges. We propose a few system constructions below which illustrate the non-triviality and also serves as a baseline for benchmarks later.

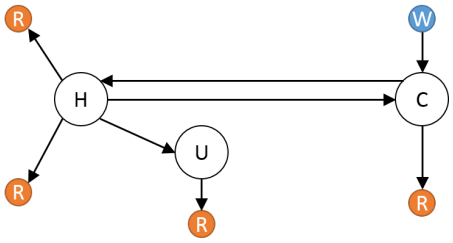
Let us imagine that a user Alice wants to run some applications relying on a *secure ubiquitous storage* platform created out of existing tools. Also, imagine that she has a few readers spread over the country. Ideally, Alice would like to use a combination of local resources and cloud resources for a combination of high-performance and durability. Toward this goal, she has access to infrastructure from three administrative domains—one in her home H where she herself is the administrator, another managed by a utility provider in a nearby city U, and the third one is a cloud data-center C that requires transcontinental communication. Note that each of these H, U and C represent varying degrees of storage and computation capacity, latency, and bandwidth.

Let us imagine that for the purpose of her application, Alice uses a resource identifier *id*; the exact choice of data-structure is left to whatever is easy for her to achieve within the given constraints. We consider the following systems to support her application:

- A designated master (a 'C0' system): Suppose Alice designates H to be the master; all updates to *id* go through H which are serialized and then propagated to U and C; U and C act as a read-only copy and are configured to accept such updates. Any of H, U or C can serve read requests, but any updates/writes received by U or C are sent to H first. At first, this strategy seems to work well, however (a) it requires manual configuration for assigning master and is most optimal only when the writer is located close to H, and (b) a failure of H breaks this strategy.



(a) An optimal topology.



(b) A suboptimal topology.

Figure 3: An optimal and suboptimal topology for a number of administrative domains, writers and readers in the case when H is a manually assigned leader.

- A byzantine fault tolerance (a 'C1' system): Instead of manually assigning a master that can become the single point of failure, suppose Alice runs some kind of distributed consensus protocol among H, U and C.[16] In this scheme, there is no designated master; all updates are ordered by using a distributed consensus protocol. If the consensus protocol can tolerate malicious nodes (such as PBFT [22]), none of H, U or C need to trust each other. This strategy is better than previous because (a) the writer does not need to be closely located to a specific node, and (b) there is no single point of failure. But as hinted earlier, there is a high communication cost among H, U and C which can cause writes to block.[17]

- An eventual consistency model (a 'C2' system): Suppose the above two systems are too expensive purely because the communication cost is too high. Let us imagine Alice opts for an optimistic replication strategy; any of the three sites can receive an update and propagate it to other sites. Such a scheme works well in practice where applications can tolerate conflicts and resolve them on demand (Dynamo [26], Bayou [52], Coda [35]). Even though this scheme solves the performance/fault-tolerance challenges with the previous two mechanisms, existing optimistic replication strategies require nodes to mutually trust each other and are not amenable to a federated architecture.[18]

In addition to inefficiencies, such home-grown combinations of existing tools with poorly defined interfaces leaves a number of engineering issues unanswered; e.g. identities, mutual authentication for administrative domains, the granularity and mechanisms of access-control, clear interfaces for read and write, consistency models for distributed data-structures, etc. Even the very basic but non-trivial questions of locality are unanswered; in the example constructions above, we simply assumed that a client magically finds the closest copy of the resource.[19] On the other hand, GDP strives to address such often overlooked questions and formalizes the log as a unifying interface that can be used by users who are not domain experts in secure system design.

[16] Alice can run two processes in her trusted home domain H_1 and H_2—thus making a total of 4 nodes—for algorithms that can tolerate f malicious nodes out of $3f + 1$ nodes.

[17] We will use SFS/SFSlite (an implementation of PBFT) later for evaluation.

[18] For evaluation purposes, we assume mutual trust and use an existing system such as Apache Cassandra or Project Voldemort.

[19] As we hinted earlier, the ideal way to find the closest server to handle a particular request would be network assisted *anycast*. IP networks do provide anycast support and are heavily used by a number of CDN's, however an average user like Alice realistically does not have a way to convince her ISP to allow a native anycast scheme. Hence, a rather crude technique used by many is to use a redirection service running at a higher level (such as DNS). Such measures, even though not ideal, work only at a coarse granularity and are not necessarily easy to setup since they require a global knowledge of the ever changing network topology.

3 Global Data Plane: An Ecosystem for Logs

In this section, we provide a gentle introduction to the proposed architecture—what is the user interface, what it means for an application developer, what is the threat model, what are the properties that this architecture provides, etc. In addition, we formally define various concepts that we will be using for the rest of this document. Note that unless otherwise specified, 'hash' refers to a SHA256 hash function. Also, 'signatures' refer to ECDSA instead of a non-EC algorithm such as RSA or DSA for the reasons described in [11].

3.1 Threat model: Minimal Trust in Infrastructure

In this section, we formally define a threat model for the GDP. Recall from § 1 that GDP is composed of storage nodes (called log-servers) and routing nodes (called GDP-routers, or simply routers) that are managed by administrative entities playing the role of a *storage organization* and a *routing domain*, respectively. Before we begin, a fundamental assumption is that the typical assumptions for security of cryptographic constructions (secure hash functions, digital signatures, symmetric and asymmetric encryption) apply, and that an adversary doesn't have infinite computation power to launch brute-force attacks against such constructions.

One of the key goals of GDP is to keep data secure in presence of untrusted infrastructure. A log 'administrator' designates a log-server (or a storage organization in general) to accept append requests from the designated single writer, keep the data stored and replicated for as long as the policy specifies, and respond to readers with appropriate proofs of integrity. However, such designated log-server may engage in malicious behavior and may accept bogus writes; modify or reorder stored data to violate the integrity properties; may attempt to use the contents of individual data items for other purposes thus violating data confidentiality; may respond with bad data to the readers; or engage in any other similar behavior that corresponds to deviation from the protocol or violation of data integrity and/or confidentiality.

An important concern is that of data freshness: instead of returning bad data, a log-server may return stale data even when the client asks for latest data—such behavior may be malicious or accidental (when a replica is in fact running behind). A client, by itself, does not have a good way of detecting staleness of data; however GDP mechanisms provide for creation of separate freshness services (see 'heartbeats' in § 4.1), application level defenses, and retries from different parts of infrastructure (see below). For data transport, a client/log-server delegates the routing tasks to a router (or a routing domain in general). However, such designated routers may also attempt to violate data integrity or confidentiality properties by replaying at a later time, tampering with during transit, or anything else.

Even though individual nodes (log-servers, routers, etc.) may be behaving maliciously, the GDP architecture allows for a client to retry and get the desired service, as long as there are alternate nodes available and reachable.[20] In the worst case when there are no alternates available, the client can still detect violation of data integrity. However, the federated architecture of the GDP allows a client to use multiple service providers at the same time and have a workaround for service availability when an entire administrative entities goes rogue. Based on the classification in the previous section, GDP falls in the 'T4' category, i.e. the system continues correct operation as

[20]To illustrate, if there is only a single copy of the data (based on client policy specification) and the log-server that stores data maliciously corrupts data, such data is permanently lost. Similarly, if a client is connected through a malicious router which is the only path to the rest of the infrastructure, the client may have all of its messages corrupted.

long as there are at least a constant number of non-malicious nodes in the system; any number of malicious nodes colluding together can only deny service to a client but not corrupt the state of data on a correctly functioning node.

An important exception to the threat model is that even though the designated infrastructure operators can become curious and look at contents of individual messages, they are trusted to not engage in sophisticated side-channel attacks on data confidentiality by observing access patterns and time/size of messages/requests.[21] Additionally, GDP routers are trusted to provide locality of access wherever possible. Any poorly configured or intentionally faulty router violating policies for locality of access does not directly compromise on data integrity/confidentiality and a client can hold a service provider accountable for not meeting the desired performance requirements. The federated architecture allows users with highly sensitive data to use their own resources and join the infrastructure as a service provider, and still enjoy the benefits of a single common platform.

Third party adversaries, on the other hand, may attempt to do practically anything; such third party adversaries include infrastructure operators that the client hasn't designated any operations to. Such third parties may attempt to impersonate the designated infrastructure providers, or attempt to corrupt the state of honest infrastructure providers by pretending to be an authorized client requesting service. Especially for the case of routing functionality, third-party adversaries may attempt to corrupt global routing state by pretending to provide a better access to the resources (say, a close copy of data nearby). As opposed to the designated service providers, third party adversaries can in fact launch sophisticated side-channel attacks by attempting to insert themselves in the path between a given source and destination. Note that any network level adversary, that just happens to be in the path between a given source and a destination, can still observe traffic flows and deduce critical information. However, GDP architecture ensures that it is not possible for an adversary to inject itself at will in the communication path between an arbitrary pair of endpoints.

Lastly, a reader of the log trusts the designated single writer of that specific log; if the single-writer intentionally corrupts the state of the log or the contents of log itself, the readers do not have a remedy for such corruption. However, they still get a verifiable assurance from the infrastructure that the writer is the culprit to be blamed for such behavior.

3.2 User interface: Single-writer Logs

Recall from § 1, that the key interface to the storage layer in GDP is a single-writer append-only log that represents an ordered stream of messages (records), and is identified by a flat 256-bit GDP name. Also recall that a log supports both publish-subscribe mode of operation as well as efficient reads for old data. In this section, we describe the interface for application developers that want access to the single-writer log layer. In the next section, we describe a bit about higher level services and applications that go beyond the simple single-writer log semantics.

The readers, subscribers and the single writer of a log are collectively referred to as clients. In addition to clients, there is a log 'creator' or 'administrator' tasked with the creation and policy specification for the log, although it is not uncommon to have such a role combined with that of the writer. The 'creator' designates one or more log-servers (or *storage organizations* in the general case) to physically store the data, and respond to 'append' queries by the writers and 'read' or

[21] Note that if A and B have delegated routing functions to routing domains $\mathbb{R}_A$ and $\mathbb{R}_B$ respectively, then both A and B need to trust at least both the routing domains—$\mathbb{R}_A$ and $\mathbb{R}_B$ for not engaging in such traffic analysis.

'subscribe' queries by the readers. The 'creator' is also responsible for the life-cycle management of the log as well as any potential economic relationship with the service provider(s).

The single writer possesses a private signature key specific to the log. For an update (append) to the log, the writer creates a signed 'heartbeat' which is sent along with the actual payload. Based on the desired durability properties, the writer is also responsible for maintaining some local state—usually in a non-volatile memory—which includes at least a few hashes; we will discuss this in more detail later. The 'heartbeat' generated by the writer identifies a particular state of the log; readers can verify the read queries against a given 'heartbeat' thus making a log essentially an Authenticated Data Structure (ADS). These 'heartbeats' are stored by the log-server along with the log, or can also be managed by a separate set of more reliable/trustworthy 'heartbeat-servers'. Encrypted data can *generally* be read by anyone (including untrusted log-servers), but only the authorized readers have the correct decryption key to make sense of data.[22] Clients use digital signatures and encryption as the fundamental tools to put their trust in data than in infrastructure.

At a data-structure level, a log is an ordered list of immutable records; a record is the unit of read or write to the log, and can be variable sized. An 'append' operation is adding new records to the log. 'Read' is fetching existing records by addressing them individually or by a range, while 'subscribe' allows a client to request future records as they arrive. Each log has a special record at the beginning, called the metadata. Like other records, the metadata is also immutable. The flat 256-bit name of the log is derived by the SHA256 hash of the metadata. The metadata is essentially a list of key-value pairs identifying properties of a log. At the very least, the metadata contains the public part of the signature key belonging to the designated single writer of the log, which is used to validate data-integrity and perform write access control. In addition, the metadata may contain policy specifications for desired performance, durability, sources for read access control, etc. We describe these details in § 4.1.

A log can be truncated based on policy decisions, where truncation is marking data older than a threshold 'safe-to-delete'. In addition, there are certain meta-operations, such as 'log-creation' and 'log-attach-request' that we will discuss in detail in next section. Internally, a log can be replicated or distributed over a number of physical machines, and there are various internal operations such as migration, replication, etc. that log-servers can perform on logs in the context of an ecosystem like GDP, however these operations are opaque to a client. Using multiple log-servers (or storage-organizations) allows for fault-tolerance, durability and scalability. The single-writer and append-only nature of the log allows for a leaderless and relatively conflict-free replication algorithm. The level of durability and replication strategy is decided by the writer based on the type of data; e.g. applications that generate data at a high rate but can tolerate some missing data, such as video frames, may adopt an optimistic replication strategy, whereas applications that require high levels of durability may choose a different strategy. Regardless of the durability requirements, the simple leaderless replication algorithm converges without conflicts.[23]

Note that not only logs, but all addressable entities in GDP (logs, clients, log-servers, GDP-routers, storage organizations, routing domains, etc.) are named using a 256-bit flat address derived cryptographically. The communication between various entities is using the GDP-protocol over the

[22]Such a read mode allows easy caching of encrypted data. In case of slightly relaxed threat models, individual logs with sensitive information may be kept isolated at the routing level (requires trust in GDP-routers), or read access can be limited by means of client-side certificates (requires trust in log-servers).

[23]This is assuming correct operation from the single writer, which includes keeping track of state in non-volatile memory. We discuss where this strategy may fail in detail later.

GDP-network. GDP-protocol is a datagram based protocol involving exchange of GDP-PDUs; a PDU[24] for GDP-protocol has an address pair *(dst, src)* along with payload.

The GDP-network is composed of *routing domains* that manage GDP-routers. GDP-routers use overlay routing techniques or various policy decisions to find the most optimum path between a given pair of addresses and deliver the PDUs to appropriate destinations based on locality and other QoS requirements. In case a log is replicated over a number of log-servers, GDP-routers will find the most optimum log-server based on a client's location, thus providing locality of access. To join the GDP-network, an entity finds GDP-router belonging to a *routing domain* of choice and securely advertises one or more names into the GDP-network; usually the choice of *routing domain* is directed by trust, economic relationship, or other administrative factors. Note that both readers/writers as well as log-servers are *clients* to the GDP-network, i.e. they initiate a connection to a GDP-router.

3.3 Beyond a Log Interface: Higher Level Abstractions

In the previous section, we described the single-writer secure log interface that simple applications as well systems' integrators can use. Although a log abstraction shelters developers from low-level machine and communication primitives, many applications are likely to need more common APIs or data structures than an append-only single-writer log interface. In order to support a wide variety of use cases, we describe Common Access APIs (CAAPI) and gateways below.

3.3.1 Common Access APIs: CAAPIs

We argue that single-writer logs are sufficient to implement any convenient, mutable data storage repository. In fact, logs have been used to implement file systems [47]. Various databases internally keep their transactions as logs that enables rollback and recovery. Using the single-writer append-only GDP logs to implement higher level interfaces allows for the separation of concerns we discussed in § 1; because a log serves as the ground truth, the benefit of integrity, confidentiality, and access control are carried over to such interfaces for free. We have built a key-value store with history as well as a file-system[25] on top of the single-writer logs.

3.3.2 Gateways

Not all client are capable of communicating using the GDP protocol natively. These limitations arise because of various reasons, the most common being hardware limitations, custom and proprietary protocols, closed software, and even software engineering in some cases. In order to support a wide-variety of use-cases, we propose a gateway based approach. The task of mapping the operations and devices to appropriate logs and log-operations is taken care of by such a gateway. As an example, we have built a RESTful gateway to enable publishing from a number of sensors that lack the platform support to interact with the GDP directly. In order to enable web applications perform in-browser visualizations, we created a websocket subscription gateway.

Gateways can be as simple as translating an arbitrary protocol to GDP, or as complex as implementing the entire client side functionality on behalf of an extremely limited client. It is expected from a user to understand the implications of using a gateway in their particular environment.

[24] Protocol Data Unit.

[25] Work done by a team of students as their class project.

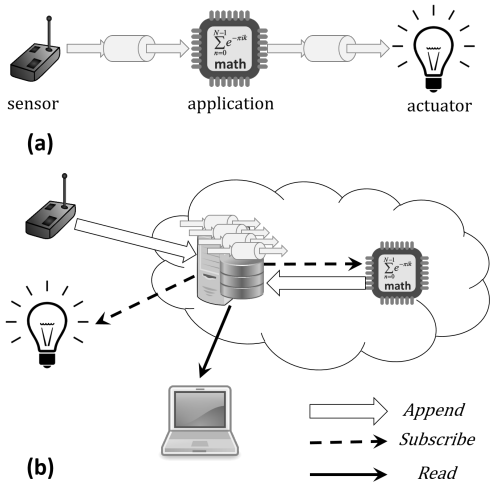


Figure 4: (a) A logical view of the single-writer log when used in the context of Internet of Things (IoT); the log provides an analogy of a pipe with an attached bucket. A sensor appends data to a sensor-log which is consumed by an application, which processes the sensor data and writes it to an actuation-log. An application developer does not need to interact with physical devices; instead the log provides a virtual device with associated history. (b) The actual physical view of the log, where a log-server physically stores data and multiple readers can be subscribed to the same log.

Even though the exact specifics vary based on the gateway functionality and its implementation, a gateway should be included in the trusted computing base of a client in the most general case.

3.4 An Application Model for Internet of Things (IoT)

In this section, we describe an application model for the Internet of Things (IoT)—one of the motivating application scenarios that can benefit from the availability of a secure ubiquitous storage platform like GDP and can use the single-writer append-only logs directly (also see Appendix A). IoT applications deal with a wide variety of sensors continuously generating data and actuators consuming actuation commands; such data requires secure long-term preservation and a secure transport. Keeping the single-writer model of a log in mind, sensors are assigned their own logs at the time of sensor provisioning. Each sensor appends the data it generates to the associated log, thus making a log the only interface to the rest of the world for the sensor. Similarly, each actuator is subscribed to an actuation log from which it gets the actuation commands. The designated single writer of the actuator log is either a pre-configured application, or a service representing the actuator (See Figure 4).

A log interface makes dumb sensors and actuators significantly more functional. Low-power sensors usually only generate data, but can't answer any queries. If data values are written to a log by such sensors, the log can be used as a proxy that supports a much richer set of queries, especially for historical data. A subscription to such a log provides the latest sensor values in almost real time, thus virtualizing the sensor in some sense. Actuators, on the other hand, usually need to maintain some kind of access control – by physical isolation, some authentication method, or a combination of both. Instead, if an actuator were to subscribe to an actuation log to read the actuation commands as we described, access control could be implemented at the log level. This makes actuator design simpler and avoids the pitfalls of ad hoc authentication mechanisms hastily

put together by hardware vendors. Long term retention of actuation commands also provides *accounting* of actions performed.

Further, there's no need to expose the physical devices with potentially questionable standards of software security to the entire world, while still being able to connect things together. This is especially important because it takes the burden of implementing security off the device vendors' shoulders. All a device manufacturer has to do is publish data to a log (in case of a sensor), or subscribe to a log (in case of an actuator).

Representing sensors and actuators with logs separates policy decisions from mechanisms, enabling cleaner application designs. Applications can be built by interconnecting globally addressable logs, rather than by addressing devices or services via IP addresses. Further, with applications running inside containers (Docker, Uni-kernels, Intel SGX enclaves, and so on), forcing data-flows in and out of the container through logs enables any filtering at the log level (for example, access control). Last but not least, the narrow waist provided by globally addressable logs avoids stove-piped solutions and provides for a heterogeneous hardware infrastructure.

3.4.1 Sample applications

Let us describe a few sample application scenarios to understand how logs enable a cleaner solution compared to a traditional design.

Robo-pet: *Imagine using accelerometer data from a smart-watch to perform gesture recognition using machine-learning, and control a robot's movement based on the gestures. One would also like to do offline long-term analytics to estimate wear and tear in various motors.*

Such a scenario combines both the real-time requirements as well as long term analytics. We built a similar application using an off-the-shelf automatic vacuum cleaner controlled by a laptop. High frequency raw accelerometer data from an Arduino based sensor is written to a sensor-log, which a machine learning application subscribes to. The final actuation commands are pushed to an actuator-log, which eventually triggers the robot. A different version of the application used an EMG sensor for gesture recognition, which controls an off-the-shelf robot using a very similar data-flow.

Using two logs enables the application to not worry about integrity of data or access-control issues. In addition, with appropriate credentials for reading, any application can perform wear and tear analysis at a later time. A typical cloud based pub-sub solution requires (1) reliable and fast internet connection, (2) explicit access control on actuation, and (3) additional work to save the data securely for offline analysis.

A Smart-Building: *Imagine a combination of generic off-the-shelf environmental sensors and actuators to create a customized building-automation solution to turn on lights, adjust air-conditioning, etc. One would also like to view long-term trends to understand energy usage.*

Such applications really focus on the composability of sensors and actuators. We deployed a number of sensors in a machine room that write data to their individual logs; such data was used for anomaly detection (such as partial air-conditioning failure) as well as for a web-based visualization tool to understand the general state of the machine room. A conventional application design would use a pub-sub mechanism to make the data from one sensor available to multiple subscribers, which leaves questions of data integrity—both in-transit as well as at-rest—unanswered.

Note that most of these example applications involve static graphs of components/services connected via logs. For dynamic scenarios (e.g. a smart traffic intersection involving vehicle-to-infrastructure communication), we propose an extension in the form of *log-attach request* (see § 4.4.3).

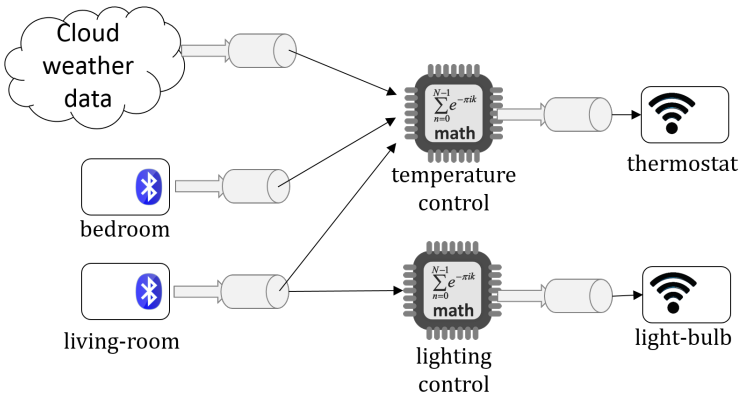


Figure 5: An example building-automation solution, where physical sensors and actuators are represented by logs to applications. Each application can have multiple input/output logs, including those representing external data sources. Each sensor log can be subscribed by multiple applications as well.

3.4.2 A Case Study: TerraSwarm

As a validation of the usefulness of the single-writer append-only logs to application developers, we made a preliminary version of the system available to a group of researchers—all part of a multi-university research project. The preliminary version lacked the security and scalability of the GDP that we hope to achieve, however the user interface provided to the users was that of a single-writer append-only log interface.

Various groups of researchers used the GDP as a data-storage repository as well as a middleware glue to connect heterogeneous sources and sinks of data together. We provided the GDP as a client side software library in C with Python, Java and JavaScript wrappers around it. Additionally, we created a number of gateways to connect with various sensors and actuators such as Bluetooth based environmental sensors, CoAP based mesh networks of sensor nodes, sensors publishing to MQTT, REST gateways, and many more. Some of the applications built on top of GDP included web-based visualization of time-series sensor data from environmental sensors, audio/video streaming applications, real-time control applications for robots, and many more.

In addition to directly accessing the underlying log, we created applications providing richer interfaces, such as a key-value store with history, a client-side caching mechanism for efficient queries for records in a window of time, etc.

In summary, a diverse group of users using the system first-hand provided us with a high-level validation of the user-interface. The experience also demonstrated the weak points of our preliminary version—scalability and durability. With the mechanisms we will describe in the next section, we hope to address these points of weakness for the next version of GDP.

4 Global Data Plane: A First Look at Mechanisms

In this section, we describe a first attempt at the architecture and internal mechanisms of GDP that enables the user interface described in § 3.2. In order to validate these ideas, we use a Python-based research prototype of GDP; this research prototype is a separate parallel software base than the current GDP infrastructure and enables us to do quick implementations without needing to maintain backwards compatibility. Currently, this prototype partially implements the GDP-network and the log-operation using the GDP-protocol. Before proceeding further, let us talk about two important concepts: GDP-name and metadata.

Recall from § 3.2 that all addressable entities in GDP have a cryptographically derived flat 256-bit name; we call this the *GDP name*. Examples of such addressable entities are: logs, log-servers, readers, writers, storage organizations, routing domains, etc. Such a name serves two key purposes. First, it provides a location independent naming scheme which enables a true identity of any *principal* regardless of location, administrative domain, or any other transient property. Second, such a name is derived from the public signature key of the principal; thus the GDP-name also serves as a cryptographic trust anchor.

A GDP-name is derived from a cryptographic hash (SHA256) of an immutable data structure called *metadata*. The metadata is essentially a signed list of key-value pairs, describing immutable information about the corresponding principal. A required item in this key-value list is a public signature key of the principal, the private part of which is used to sign a serialized version of this key-value list to create the metadata. Note that metadata is a public data-structure, and thus does not contain any information that should be kept confidential. As an example of what might be included in this key-value list, a metadata for a log could contain policy specification such as creation time, a human readable name/description, desired performance characteristics, durability requirements, source for read access control, etc.

4.1 Secure Single-Writer Append-Only Logs

In this section, we discuss the design and internal mechanisms of the secure single-writer append-only logs. As we discussed earlier, a GDP log is an authenticated data structure with certain additional properties that allow integration with the rest of the system and achieve the desired performance characteristics. Before proceeding further, let us first do a quick review of Authenticated Data Structures (ADS).

4.1.1 Background: Authenticated Data Structures

An authenticated data structure (ADS) is a data structure where an intermediary can perform operations on behalf of a user and prove the correctness of such operations to the user by using a proof. In the context of storage systems, an ADS can be stored on a remote service provider who is not trusted for data integrity. Using the proofs, the reader can satisfy itself that the result of a read query is correct. Authenticated data structures have been extensively studied and are used in many protocols and distributed systems in the form of hash-trees, hash-chains, or other variants [51, 40, 41]. The single-writer GDP logs inherit the concept of a proof for integrity verification from hash-trees.

Even though a log is essentially an authenticated data structure, there are enough important details that the design of the log merits its own discussion. Using an off-the-shelf hash-tree/hash-chain for GDP is less than ideal because of two reasons. First, applications vary widely in the cost

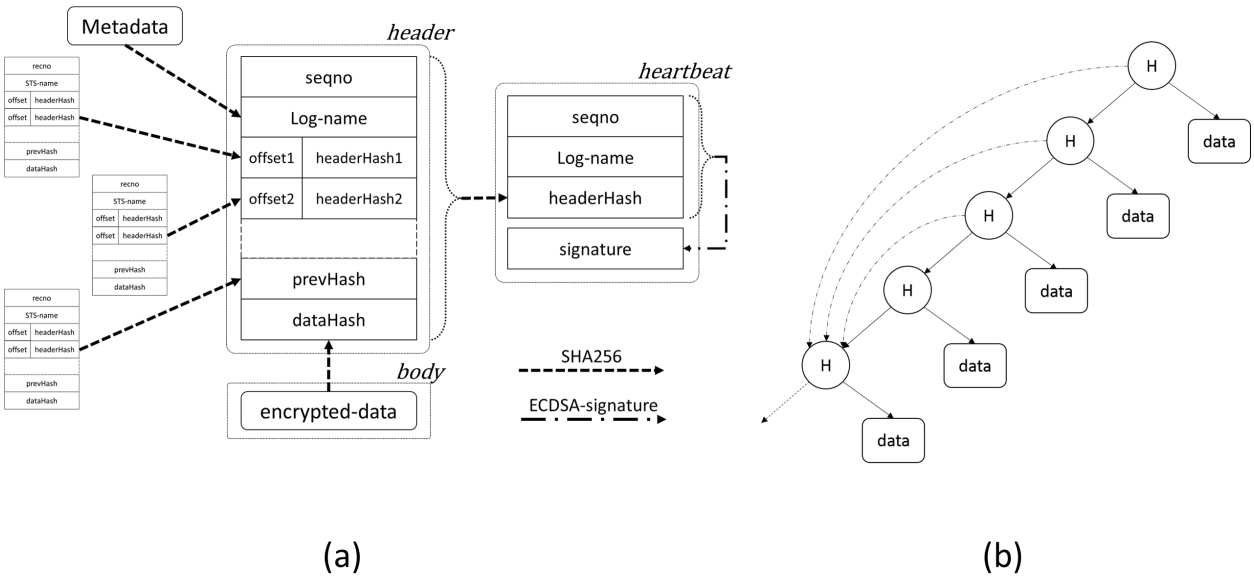


Figure 6: (a) Record structure demonstrating record-header, body and heartbeat. (b) Without hash-pointers *(offset, headerHash)*, the records in a log make a very skewed Merkle tree (essentially a hash-chain). Hash-pointers include additional links and transform the Merkle tree to a Directed Acyclic Graph (DAG).

of proofs (both size and required computation) that they can tolerate. As we will discuss very soon, the log design provides the applications flexibility for very efficient proofs at the cost of requiring the writer to maintain more state (and vice versa). Second, the widely distributed nature of the underlying storage infrastructure comes with various data consistency challenges. A number of applications can work well even in the presence of some irregularities in data (e.g. missing video frames in a video stream). The design of the log allows applications to specify such tolerances to make the most efficient use of the underlying infrastructure and ensure that the data structure is not the limiting factor.

As we will show later, even though the log design provides enough flexibility to meet the goals of a wide variety of applications, such flexibility does not come at a cost of compromise of data-integrity.

4.1.2 Log: Internal Structure

A log is primarily an ordered list of records with the log-metadata at the beginning. A record is the unit of read/write to a log; it is an immutable structure composed of a header and the body, with an associated heartbeat (see Figure 6).

Log-metadata, as the name implies, is the immutable metadata associated with the log from which the log derives its flat 256-bit GDP name. A non-exhaustive list of information in the log-metadata is: a public signature key for the signing records ('writer-key'), a public signature key of the log-creator (if different from 'writer-key'), a source for obtaining decryption key for the data, creation time, truncation policies, a human readable name/description of the log, etc. At the very minimum, metadata contains the 'writer-key', which serves the following purposes: (1) individual records are signed with the private part of 'writer-key', providing data-provenance

and non-repudiation properties; (2) a write-access control mechanism for a correctly functioning log-server, where a signature associated with an 'append' is validated against the 'writer-key'. To ensure the single-writer semantics, the private part of this 'writer-key' ought to be protected by the designated writer and should not be shared.

Record body contains the application level data. This data is opaque to the log-server (or any other intermediate entity) and is padded and encrypted using (preferably) a fast, symmetric encryption scheme, such as AES.[26] The decryption key is communicated to the allowed readers using either an 'log-attach request' (described later), or any out-of-band communication channel suggested in the log metadata.

Record header contains meta-information necessary for integrity verification, data-ordering and storage. At the very essence, the header contains a monotonically increasing integer (`seqno`), the 256-bit name of the log (`log-name`), a variable number of *hash-pointers* to older records in the form of `(offset, headerHash)` pairs, the hash of most-recent record header than the current one (`prevHash`)[27], and the hash of the record body (`dataHash`). Within the namespace of a log, a record could be referenced either by `seqno` or by the hash of the header `headerHash`.

Record heartbeat is a signed stand-alone piece of data associated with a record that contains `seqno`, `log-name`, the corresponding `headerHash` and a signature over the three items using the private part of the 'writer-key'. Heartbeats are small, self-sufficient pieces of data that can be distributed widely in a network (because of the small size) and have the same built-in ordering as the records (because of the included `seqno`). The purpose of 'heartbeats' is to provide data-freshness guarantees and authenticity of the corresponding record.[28] However, as we will describe shortly, signature verification using 'heartbeats' is not the most optimal way for reading old data in bulk.

A simple log without any additional *hash-pointers* is a hash-chain and a very simple ADS. As we will describe very soon, such simple hash-chain suffers from performance issues for reading old data and low-tolerance for any missing records.

4.1.3 Log operations

Log creation: The process of creating a new log involves sending the signed metadata and an 'advertisement certificate' (AdCert) to a log-server. An AdCert represented as $AdCert(A \rightarrow B,\ expire\_at)$ means: "A designates B to advertise for A till the time $expire\_at$". Such a certificate is signed by the private key of A; anyone with the metadata of A can securely get the public key of A and validate the AdCert. Typically, A is the log 'creator' and B is a log-server. B could also be generalized to a storage organization instead of a single log-server; in such a case, the metadata and AdCert are sent to a designated 'creation-service' for B instead of a single log-server and B could then appropriately place such metadata on log-servers it controls. Such AdCerts are renewed periodically and their lifetime can be tweaked by the 'creator' to fit a variety of application and infrastructure scenarios. In some ways, the 'creator' acts as a Certificate Authority issuing a certificate to log-servers or service providers to serve requests on behalf of the log.[29]

[26] For our prototype implementation, we use AES-128 in CTR mode, where the counter is initialized from the log-name and `seqno` (see later). Using CTR mode enables a reader to individually decrypt records and reduces management overhead by deriving an IV implicitly from already available information.

[27] For the first record, `prevHash = log-name`

[28] Note that heartbeat is different than a typical *keep-alive* used in various protocols.

[29] Note that in case the log is placed on multiple log-servers for replication, all log-servers are made aware of each other so that they can keep in sync with each other.

Why delegate a specific organization/host for a log? As we will describe in detail in the design of GDP-protocol, any state update operation performed by a writer ought to be acknowledged securely. Otherwise, an active man-in-the-middle can simply drop any 'append' operations and send a spoofed acknowledgment to the writer, thus framing an honest log-server.[30] The identity of the physical server embedded in the AdCert allows a writer to ascertain that it received an acknowledgment from the designated log-server.[31] However, note that the AdCerts can be short lived allowing for a log to be migrated to a different service provider. Neither the writer nor the readers need to know about the identity of the log-server in advance; in fact, as we will discuss in § 4.2, all operations from writer/reader are addressed to the log, instead of a physical server.

The role of single writer: The single-writer append-only design enables the single-writer to be the point of serialization. However, it comes with two additional costs. First, we use signatures with the writer's private key to maintain write access control and provide data-provenance properties to a reader. In addition to adding to the message size, signatures are computationally expensive and can incur significant burden on the writer. Second, the writer needs to maintain the most recent state of the log in a local non-volatile memory. This state includes at least the `headerHash` of the most recent record as well as any records that have not been made sufficiently durable. Failing to do so may result in permanent data loss in the form of *holes* or divergences called *branches* in the otherwise neat hash-chain; we discuss these issues in the next section.

Append: To 'append' data to a log in the simplest case of a single log-server (no replication), a writer forms the appropriate record structure; the writer is free to include *hash-pointers* to any older records as it seems fit. The exact nature of the linking structure created by writer has a number of performance and durability implications, which we will discuss shortly. Once the writer signs the record, it first updates its local state stored in a non-volatile memory, and then sends the 'append' request along with the signature included in 'heartbeat' to the log; the GDP-network finds an appropriate log-server that has securely demonstrated to the GDP-network that it is authorized to advertise for the given log. On receiving the append request, the log-server performs a basic sanity checking on the received record structure and verifies the signature before accepting the 'append'. Any subscribers to the log are notified of the new data along with the 'heartbeat' for verification, and a secure acknowledgment is sent back to the writer. We describe the 'append' operation with replication in the next section.

Reads and integrity verification: Readers can query old records either by their `headerHash` or `seqno` and ask for a proof relative to a more recent record r. If the original query for the record is by the `headerHash`, no explicit proof is needed and the client can simply verify the received data against the `headerHash`. If the client uses a `seqno` instead, the server provides a proof of integrity along with the requested record(s). The proof is generated using a combination of a chain of hashes and/or a signature from the 'heartbeat'. Ignoring the *hash-pointers* for a moment, the record structure described earlier is essentially a hash-chain (a very skewed Merkle tree, with each new record as the root of a new tree) (see Figure 6(b)). The log-server locates the record r—the more recent record relative to which the client is seeking a proof—and uses it as a root of the Merkle subtree to generate a proof of integrity by traversing the hash tree to the queried record. If the client didn't specify a record r, then the log-server picks an appropriate record r itself and includes the corresponding 'heartbeat' (which includes a signature from the writer) along with the proof.

[30] Even the log creation operation is securely acknowledged by the log-server.

[31] An alternate strategy could be to include the designated log-server in the metadata itself instead of an AdCert, which could work in certain situations. However, because the metadata is immutable, this alternate strategy has the downside of fixing the log to a particular log-server for eternity.

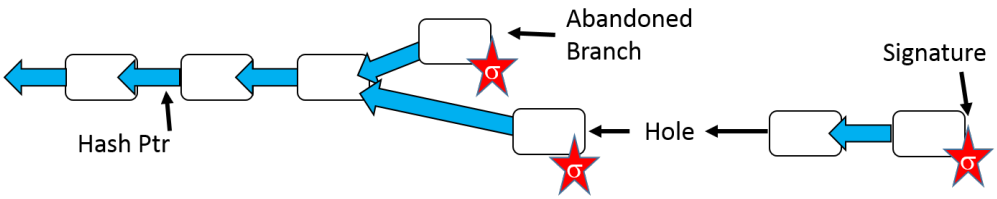


Figure 7: A log with holes and branches. See § 4.1.4.

Integrity verification: hashes or signatures? Even though a 'heartbeat' is associated with each record, signatures are orders of magnitude more expensive to compute and verify[32]. Because of this heavy cost, a log-server uses hashes as much as it can for all integrity verification. With the design of a record header described above, one single signature verification at a particular instance in time allows a reader to verify everything up to that time in the past with only hash-verification. Not only using hashes reduce the computation cost, they also free the log-server from the burden of storing signatures. To reduce storage costs, the log-server can get rid of all the signatures except the most recent signature and generate hashes on demand.[33]

4.1.4 Replication and Durability: Holes and Branches

In order to achieve durability, it is crucial that data be replicated across multiple log-servers. Replication is an active focus of research; the exact mechanisms presented here are still in development. We consider two modes of durability: server-driven or writer-driven.

Server-driven mode of durability is essentially using optimistic replication: a writer uses *anycast* to send an append request and considers the append to be complete as soon as it receives an acknowledgment from any single log-server. The log-server performs a best-effort replication of data by simply forwarding any appends to other log-servers responsible for the given log. In such a mode of operation, the writer trusts the server to make the data durable by replicating it to other log-servers. However, even if the log-server arbitrarily delays the replication process, data integrity is not compromised; since the ordering is decided by the single writer, any update conflicts can be easily and securely resolved without any client intervention.

Writer-driven mode of durability is where the writer considers an 'append' to be done only when it receives a desired number of acknowledgments. We consider two potential schemes toward this goal: the writer still uses *anycast* to contact only a single log-server, but the log-server collects secure acknowledgments from multiple log-servers on behalf of the writer and returns them together; or alternatively, the writer uses *multicast* for 'append' and receives acknowledgments from log-servers directly.

The durability mode chosen by the writer is usually dependent on the desired performance goals and the type of data—whether a reasonable application can tolerate potentially missing data. The server-driven durability mode is typically faster than client-driven mode at the cost of a potential for permanent data loss, thus resulting in a *hole* in the log. Such permanent loss can occur if the

[32]One-time signature schemes are much cheaper than traditional digital signatures, however they suffer from excessively large data-sizes. Our space-time equation can not be too biased towards data-size, because it needs to be transferred over network.

[33]The storage cost of signature is to be looked in relative terms—what is the size of an average record's payload as compared to the size of an individual signature? Records in a typical IoT application, such as ambient temperature measurements, are only a few bytes long and often smaller than the size of the signature.

first log-server that the writer contacted is malicious, or if there are failures where the log-server crashed permanently before the data could be made sufficiently durable. There are many types of data where a small probability of loss of individual data items does not make the entire data-stream useless; such as time-series data values for ambient temperature, or high bandwidth video-streams with individual frame per record. On the other hand, there are applications where a single missing update can make the entire data stream useless and thus it is important to ensure that every single update is made durable, e.g. a file-system on top of a log without any checkpoints. For applications in the latter category, writer-driven replication is preferable.

Note that the role of the writer in either mode—server-driven or writer-driven—is more than simply contacting the log-servers; the writer is also required to maintain some local state regarding the state of the log, preferably in some non-volatile memory. For applications where *holes* are acceptable, simply keeping the `headerHash` of the most recent and a few other older records is sufficient. However, for applications where the durability is of high importance, a writer must maintain the contents of the records as well as the hashes in the local state till the data is made sufficiently durable. In the extreme cases where a writer does not get sufficient acknowledgments, the writer is stalled.

In case a writer does not maintain the local state appropriately and recovers after a crash, it may query a log-server about the most recent record and re-populate the local state. However, in presence of accidental or malicious stale servers, the writer may not get the correct state of the log; the writer may start off from an older record, thus resulting in an abandoned *branch*. We consider such recovery after a catastrophic failure, where the writer lost all its non-volatile state, to be rare and require readers to be aware of the guarantees from the writer (potentially expressed in the log-metadata). Nonetheless, the single-writer log primitive works well even in case of such failures.

In addition to *on-line* replication, log-servers also perform periodic background synchronization to fill any holes or branches. The local state of the log, that may be full of branches and/or holes, can be efficiently and uniquely described in the form of a list of pairs of hashes. The log-servers can use such compact representations to efficiently carry out state reconciliation using anti-entropy gossip protocols. Such a state reconciliation is typically done in two steps: first to fill holes, and second to synchronize branches.

4.1.5 Beyond a Hash-Chain: Managing Application-Specific Requirements

Note that the structure of the log, as we have described thus far, is a very skewed Merkle tree and is essentially a hash-chain (see Figure 6b). The integrity proof generation scheme described earlier can result in very long proofs with such a simple hash-chain. In order to limit the size of proofs, we use *hash-pointers* in the record header; hash-pointers are pointers to any arbitrary record older than a given record. Using hash-pointers makes the graph of pointers a Directed Acyclic Graph (DAG) instead of a simple hash-chain and provides for configurability of the structure of a given log. However there are certain implications of using such hash-pointers: (a) The writer needs to either keep a local cache of any `headerHash`'s it might need in future, or query them from a log-server on demand causing a performance penalty. (b) For a reader reading old data, the overhead of verification is dependent both on the linking structure and the access pattern. (c) The tolerance for failures increases with denser links, however too many links may adversely impact archival storage of data on log-servers.

Generating an optimized integrity proof in a generalized DAG: When queried for a record m against a more recent record n (that a reader may have obtained by verifying a signature from

a 'heartbeat'), a log-server has to find the shortest path from n to m in a weighted DAG, where the weight on an edge is calculated as the size of the record-header where the edge is pointing to. However, for most practical purposes, a simple greedy strategy works well enough, where starting from n, one picks an edge to the oldest record more recent than m.

Why not make a balanced Merkle tree? Instead of using arbitrary *hash-pointers*, an obvious design choice would have been to make a balanced Merkle tree rather than a skewed tree. We do, in-fact, use *hash-pointers* to create a rather balanced Merkle tree for certain use-cases. However, the reasons to not restrict the structure to only a balanced tree are: (1) individual applications may consider some records more important than others and may want optimized proofs for such special records, e.g file-system checkpoints; (2) a Merkle tree does not work very well with log-truncation (imagine a situation where we just like to keep last 10 records around); (3) a strict tree structure forces one to require extra data transfer in the simplest of the cases; (4) Merkle trees require relatively larger state to be maintained on writer, which may not be available on all devices.[34]

The appropriate strategy for picking hash-pointers for a particular application is still an active focus of exploration. The choice of 'hash-pointers' is primarily that of the writer to control the performance; all invariants and proofs works regardless of the structure of 'hash-pointers'. At a high level, the goal is to find an appropriate trade-off between the cost of 'append' and integrity proofs for 'read'. We are currently looking at three generic strategies:

1. The simple linked list (hash-chain) with a constant number of back-links. In addition to just the `prevHash`, each record may include links to the last n records as well (where n is a pre-configured parameter, and allows for a hole of size up to $n-1$). This is the simplest strategy where the writer needs to maintain a constant number of `headerHash`'s, but the the integrity proof is as long as the number of records between the queried record and an already known record. However, this simple linked-list design is very efficient in range queries; a range of records is self-verifying with respect to the newest record in the range. Such a design is a good fit for applications that require "all the data starting from time t to now".

2. A type of binary tree, where a record has more links to nearby records. The exact strategy works as follows: the writer expresses the record number (the actual count of records, and not the `seqno`) in its binary expanded notation in, but in decreasing order of powers. As an example, $27 = 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$. Then the record contains back-links to older records as represented by cumulative sums of the terms; we also need to ignore duplicates, self-pointers, and pointers to immediately previous record (which is included as `prevHash` already). To illustrate, 27 will have back-links to record numbers $1 * 2^4 = 16$, $1 * 2^4 + 1 * 2^3 = 24$, $1 * 2^4 + 1 * 2^3 + 0 * 2^2 = 24$, $1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 = 26$, $1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 27$. Since we count 24 only once and ignore 26 and 27, the list of back-links is $[16 \leftarrow 27, 24 \leftarrow 27]$. Such a strategy ensures that the writer can determine what `headerHash`'s to keep in local state for use later. The proof sizes are logarithmic in nature.

3. A checkpoint strategy. Consider an application that performs checkpoints every n records by dumping its local state to the log; each checkpoint has a link to the previous checkpoint, and non-checkpoint records have links to the most recent checkpoint. Such a strategy allows for flexibility while providing good performance and tolerance for holes. Our version of a

[34]This becomes especially important in case of commercially available microchips with hardware ECDSA support but very limited storage, e.g. Atmel ATECC108A that ships with only 10Kb EEPROM.

key-value store CAAPI (see § 3.3.1) uses a similar checkpoint strategy. Note that in the most general case, n does not need to be constant and can vary within a single log.

4.1.6 On-Disk Storage on Log-Server

A log-server is free to choose the on-disk data representation as it seems fit and such choices are dependent on the specific implementation. The only contract it has to satisfy is to be able to serve the requests that are not older than the truncation threshold. In order to understand the implications of a given strategy and to provide a reference design to developers, we consider the on-disk format design to be an area of future exploration. We are currently leaning toward the following 'extent' based strategy as used by many existing systems.

A single log-server can not store arbitrarily large logs. For this reason, we introduce the concept of 'extents', a large enough chunk of data representing a range of records in a log that can be independently replicated or migrated. Extents are opaque to the clients; a log-server may move certain extents to remote systems or slower but larger storage media depending on the access pattern. The append-only design ensures that old data is read-only, making way for easier replication/durability of data. Extents could be used for optimization of log-truncation. Since truncation is merely a 'safe-to-delete' flag anyways, instead of doing per-record cleanup, a log-server can perform such cleanup at the extent level.

An important trade-off that a log-server has to make is the storage vs computation costs for extents. Recall that a log-server does not need to keep older signatures for each record; for providing integrity proofs to the client, it can simply use hashes and the most recent signature. Since the hashes can be re-computed at a later time, the question a log-server must answer is: whether to store record hashes along with the data or regenerate them when an extent is loaded into the memory? Recomputing hashes at a later time allows for extremely efficient application-specific compression; in specific scenarios, the amortized storage overhead of security mechanisms (hashes and signatures) can be less than 10 bytes per record.

4.2 A Secure Datagram Protocol for Anycast/Multicast Operations on Logs

The single-writer log described in the previous section allows a client to use remote storage resources that are not necessarily trusted. However, how can a client communicate with such remote log-servers securely in presence of potential network level adversaries? Even if the remote log-servers are honest, an adversarial man-in-the-middle can frame such honest infrastructure providers. To answer this question, we describe how log-operations are sent over GDP-network securely and efficiently using the GDP-protocol: a secure datagram based protocol that works well even with anycast/multicast.

4.2.1 Background: Secure Communication Channels

There are a number of communication protocols/tools that provide an end-to-end secure channel, both for generic message exchange (TLS, VPN, etc) or specific protocols (e.g. HTTPS, SSH, etc). Such secure messages assume two things: a stream interface underneath (typically TCP), and a host to host communication. There are a few secure communication protocols that attempt to work with a datagram oriented protocol (e.g. DTLS), however such protocols are still for a host-to-host communication. In most protocols, the two communicating hosts establish some shared state

explicitly involving multiple round-trips of communication before any actual data transmission can commence.

GDP-network is an information-centric network where one communicates directly with the appropriate principal and not a physical host. Especially in case of logs that may be replicated across a number of administrative domains, the communication mode is often either anycast or multicast. Recall that the log operations from a writer or a reader are directly addressed to the log and not a host; the network finds the closest physical node advertising for the given log and delivers the request to such node. The existing security tools that rely on a stream based unicast communication to a specific host do not naturally fit in this new ecosystem.

One could argue that anycast communication is quite similar to unicast and that a datagram based tool such as DTLS could be made to work with the GDP-network. However, if the underlying network switches a client communicating with a server A to a different server B advertising for the same, the client must pause and do explicit state negotiation with the B before it can start communication again—an expensive process that itself requires multiple round-trips. Further, such a negotiation may never finish in the simple case of the two servers getting client messages alternatively—a typical load balancing strategy.[35]

4.2.2 Design Principles

Before discussing details of the protocols, we need to understand the type of messages for log operations. Viewing the log as a storage interface, various log operations can be classified in two very distinct categories:

1. Writes, or in a more generalized viewpoint, operations that involve change of persistent state. In the context of logs, examples include log-creation, appends, messages for synchronization among log-servers, etc.

2. Reads, or operations that query existing state. Log operations that fall in this category are: reads, subscribes, queries for metadata, etc.

For any log operation, there is always an active entity that initiates communication by sending a request; such request is fulfilled with a response by some other active entity. These requests or responses can last an arbitrary amount of time and can span multiple messages sent back-and-forth. In the most generalized sense, we can designate entities initiating requests as clients, and those that respond as servers.[36]

With the generic classification above, the actual messages can be classified in the following four categories: (1) state update requests, (2) responses (acknowledgments) corresponding to state update requests, (3) state query requests, and (4) responses corresponding to state query requests. We follow certain basic principles in order to secure these types of messages:

1. *Secure state update:* Any request to update persistent state should have appropriate information for the server to verify the identity of request creator and the contents of the request.

[35] A separate argument is that many HTTPS websites are internally served by an array of servers, and they must have solved this kind of challenge. However, note that such services are in the same administrative domain with cooperative front-end load-balancers carefully configured specifically to prevent such issues.

[36] This terminology may create some confusion with GDP-clients and GDP log-servers, but we hope this will be clear from the context. Note that a GDP-client is always a client, but a GDP log-server can be either a client or a server, depending on the particular operation. An example where a log-server is acting as a client is during replication.

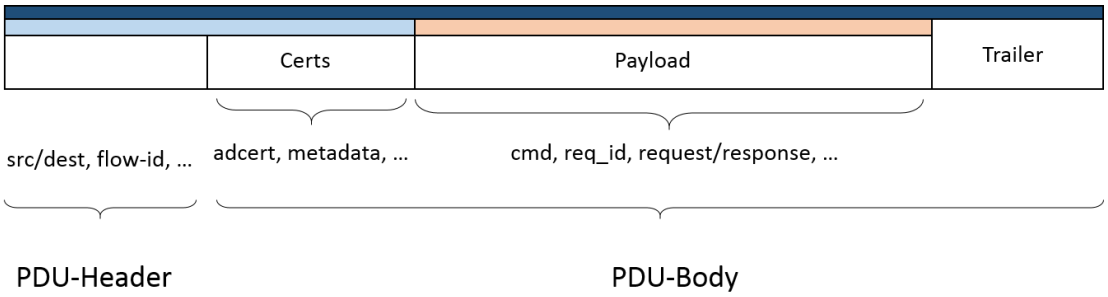


Figure 8: GDP protocol PDU. PDUs are split in two parts: a *PDU header* and a *PDU body*. PDU Header primarily contains 'source' and 'destination' information and is used by the GDP-network to route messages. The PDU body is split into three parts: *certs*, *payload* and *trailer*. *Certs* contains certificates and metadatas for key-exchange; *payload* contains the actual request/response to/from a log-sever; *trailer* contains an HMAC (or a signature) from the log-server.

Such requests can be forwarded by arbitrary entities without tampering, and everything should still work. e.g: a log-server forwards the append request created by the a writer to other log-servers during replication.

2. *Secure acknowledgments:* Acknowledgments for persistent state update requests should be created in a way that a client can verify that the state update request has reached the desired components in the infrastructure. Note that the acknowledgments do not guarantee that the requested update has been performed; a server can lie to a client and send false acknowledgment. However, an adversarial man-in-the-middle should not be able to spoof acknowledgments and frame an honest server.

3. *Proof of correctness:* A response to a request querying state should always include a proof of correctness/integrity. The original request may include some additional information to optimize the proof in certain cases.

4. *Idempotent messages:* A request replayed at a later time should not affect the persistent state. For example, if a correctly signed append request is replayed at a later time (either duplicated by the routing layer, or by a third party adversary), the state of the log should not be affected.

4.2.3 The GDP Protocol

With the above design principles and the fact that the log is an authenticated data structure, the key task of the GDP protocol is to ensure *secure acknowledgments* and facilitate any key-management. One way to achieve secure acknowledgments is that the server signs the acknowledgments. However, a client has no a priori information about the identity of the log-server(s); any request from the client is targeted to the log and a log-server is merely serving on behalf of the log. The only private key that a log-server has is its own private key.[37] Thus, if the log-server signs the acknowledgments

[37]This is the private key whose corresponding public key is used to derive the GDP-name of the log-server.

using its own private key, then the client needs to (1) obtain the corresponding public key, and (2) verify that this key belongs to a log-server authorized to host the log in question.[38]

The GDP protocol ensures that such key-setup between a client and a log-server can be done in parallel with the actual data transfer and does not require any explicit round-trips. This is a notably different design then, say TLS, where key-exchange is required before any actual data is sent. At a high level, the key exchange happens as follows: the log-server provides its own metadata and an AdCert that it presumably received from the log-creator at creation time (which includes the GDP name of the log-server). The log-server metadata contains the public key of the log-server, and the metadata should hash to the 256-bit name contained in the AdCert. Additionally, for an optimization that rids of the expensive signature computation/verification, the client can also include its own metadata along with a request; the client and the server can then establish a shared secret key derived using EC Diffie Hellman (using each other's public keys), which is then used by the log-server to perform an HMAC instead of the signature.[39].

An additional goal of the GDP protocol is to enable efficient communication by reducing the number of bits sent on the wire. Toward this goal, the protocol employs two techniques: flowIDs and payload compression. But before, let's take a quick look at the GDP PDU format (see Figure 8). The PDUs defined by the GDP-protocl are split in two parts: a *PDU-header* and a *PDU-body*. PDU-header primarily contains 'source' and 'destination' information and is used by the GDP-network to route messages. PDU-body is further split into three parts: *certs*, *payload* and *trailer*. *Certs* contains AdCerts and metadata; *payload* contains the actual request/response to/from a log-sever; *trailer* contains the HMAC/signature from the log-server.

FlowID: Including two 256-bit addresses in each PDU is a significant data overhead. For efficient communication, we use a hop-by-hop flowID; a flowID is a small integer negotiated between two consecutive nodes on the path between a source and a destination. For each hop on the path in the overlay network, only the first PDU for a given ('dst', 'src') pair contains the full addresses and any future messages include a flowID, thus avoiding the penalty of large addresses. Note that the use of flowID is merely an optimization and individual nodes may choose not to participate in such optimization.

Payload compression: GDP-payload contains compressed, network efficient versions of 'log-create', 'append', 'read', 'subscribe' and their corresponding responses. With the key idea being not to transmit information that can be securely generated on the other side, it is just a careful encoding of information. As an example, for an 'append' request, the actual data sent to the server does not contain any hashes, since they can be generated locally by the server (only offsets for 'hash-pointers'); the log-server fills in the appropriate hashes that the client skipped transmitting on the wire and then performs signature validation. However, note that any signatures/hashes are over a serialized version of the in-memory representation, and not the on-the-wire bytes. This makes sure that any accidental/intentional incorrect assumption of shared state does not compromise security.[40]

Achieving the two conflicting goals—payload compression and a completely stateless protocol—is a challenging process. A completely stateless protocol, for example, requires AdCerts, metadatas,

[38]The actual protocol involves the signature over both the original request and the corresponding response (and not just the response). Additionally, an important bit of information included in GDP-payload is a requestID, which is a unique 32-bit increasing number that (1) the client uses to correlate responses with requests, and (2) ensures an acknowledgment by the server for one request can not be replayed as a response to another request.

[39]We assume that client and server already agree on ECDH shared parameters

[40]The same also applies for the flowIDs described earlier; the HMAC is calculated over the original source and destination address (and not the flowID).

and other similar information to be included with all PDUs, resulting in excessive overhead. Any request failure because of lack of a state negotiation results in extra round-trips, which is also undesirable. However, with the help of appropriate 'hints' for such state negotiation, we believe it is possible to minimize retransmissions and duplicated state information; this is an active area of exploration and requires some more engineering work.

4.2.4 Secure Acknowledgments with Multicast

Recall that in the writer-driven mode of replication, the writer receives multiple acknowledgments from a number of log-servers; such acknowledgments are received by the writer directly or collected by the first log-server and then passed on to the writer. Both the strategies are less than ideal. The direct multicast from the client requires the client to wait for acknowledgments from individual messages. On the other hand, a server collecting acknowledgments on behalf of a client can not use the HMAC optimization—individual servers must use signatures that are expensive to compute and verify.

We would like an efficient acknowledgment scheme for a writer-driven mode of replication. An active area of exploration is a cryptographic construction that lets a log-server collect secure acknowledgments on behalf of the client.

4.3 A Secure Routing Framework for Flat Cryptographic Names

In this section, we present the design of a secure and scalable routing network with flat cryptographic names, called the GDP-network. GDP-network is composed of a large number of *routing domains* that do not necessarily trust each other, yet they coexist together even when there is no single administrative entity performing namespace allocation as in IP-based networks. As described earlier, the GDP-network is an information centric network—instead of simply routing messages between hosts, the GDP network routes information between cryptographic principals. Unlike host based networks where anycast and multicast are rarely used, such modes of communication are the norm rather than the exception.

The routing problem is: given a GDP PDU with a specific (*src, dst*) pair, how to efficiently and securely route the PDU in the presence of adversarial entities who may claim to possess any given name and even control parts of the network (see a formal threat model in § 3.1). In our architecture, the *src* and *dst* are both flat 256-bit addresses. Additionally, the GDP-network should provide native support for not just unicast, but also anycast and multicast.

4.3.1 Background: Security Challenges of Routing in a Flat Address Space

Flat address-space routing has been extensively studied in the past, and a number of distributed storage systems use structured peer-to-peer networks [58, 48]. Many flat address-space routing schemes follow a variation of the Plaxton routing scheme as follows [42].

When a node joins the network, it follows a joining algorithm which results in the node being connected to a number of other nodes. A list of these nodes is maintained in a data-structure typically called a neighbor table. On receiving a message, a node follows a simple algorithm: it first makes a decision whether this is a message intended for itself, or a message to be forwarded. If the message is to be forwarded, then it picks the node from the neighbor table that minimizes the distance between the destination address and the next node's address based on some distance metric. Various existing schemes differ in exact details of the joining algorithm and the way they

maintain the neighbor table in face of nodes joining or leaving. The distance metric also varies across the various routing schemes; a common distance metric is based on the common prefix length.

This simple routing scheme described above suffers from a few security challenges (see [53] for a detailed survey). In the GDP context, the biggest challenge posed by such security challenges is that an adversary can advertise for a name of a victim; such adversary can influence the routing state of honest routing nodes and attract traffic for any given target toward itself. The adversary can then either simply drop the traffic, thus causing a *black-hole* for data; or simply send it back in the network to the correct destination, effectively being a man-in-the-middle. Even worse, since the federated nature dictates that anyone can start their own routing domain and be part of the global infrastructure, an adversary can create an arbitrary number of routing domains and GDP-routers. This makes it incredibly easy for an attacker to insert itself in the communication path between a given pair of endpoints with a high probability and perform sophisticated passive network traffic analysis without them even knowing (see § 2 and § 3.1).

It is important to note that these problems are not specific to our architecture, but many other systems with flat address-space suffer from these problems. Most systems either ignore this problem completely or have application-level workarounds for an active man-in-the-middle. There are a few probabilistic solutions as mentioned in [53], however there is no universal scheme that solves the challenge of a passive man-in-the-middle simply observing traffic. We envision that our solution to the routing problem, that we describe below, can also be applied to these other systems.

4.3.2 Routing Problem: A Two-Part Solution

We partition the problem into two parts: route lookup and actual data transfer; our scheme provides a deterministic solution to an adversarial man-in-the-middle given certain trust properties. It is still possible that an adversary prevent communication to a specific destination with a small probability, but the adversary can not insert itself in the communication path. In this scheme, we explicitly delegate the routing functions by using a cryptographic certificate based mechanism; our routing scheme is quite different than Plaxton routing scheme described above and borrows the structural and administrative hierarchies of the IP networks. Before we begin, note that both log-servers and readers/writers for a log are *clients* to the GDP routing network. Recall that GDP routing is composed of routers that are parts of administrative entities called routing domains. Additionally, routing domains can be arbitrarily nested and there can be routing sub-domains, routing sub-sub-domains, and so on.

A client advertises one or more names in the routing network by joining a routing domain; such names can be either local names for which the client has the associated private key for, or delegated names that one of the local names is authorized to advertise for (via an AdCert). Typically, a simple reader/writer only has a local name, whereas a log-server has a local name of its own and a number of delegated names—one for each log it hosts. When a client joins a routing domain, it delegates the routing functions to that particular routing domain. Routing functions primarily include ensuring messages sent by a client reach the intended destination and that the messages sent to the client are delivered to the client. Additionally, a client may specify the scope of the advertisements: whether to advertise globally, or within the routing domain, or any routing sub-domains, or something else.[41]

[41] We would ideally like to also support client specifications for a list of routing domains, however we don't quite have the mechanisms for such capability yet.

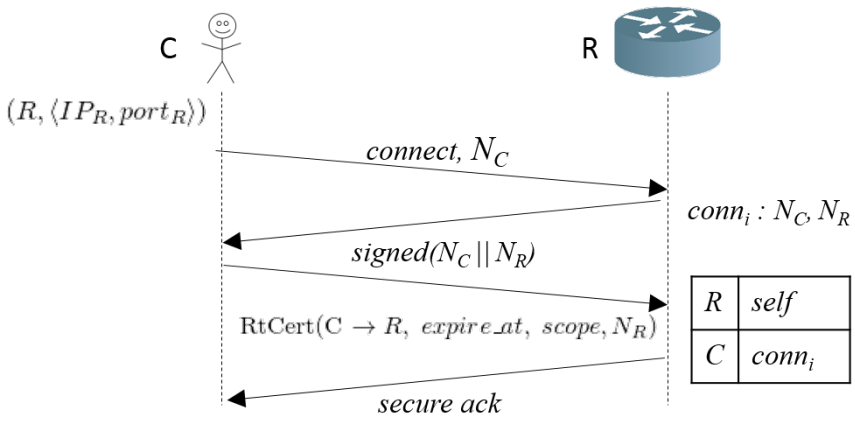


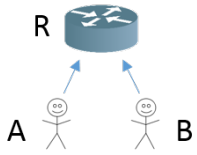
Figure 9: Local advertising process: A client C sends a routing certificate to a router R. After certificate validation, R updates its local forwarding table.

Routing certificate: A routing certificate is a secure delegation of the routing functionality to an individual GDP-router or an entire routing domain. A routing certificate represented as $RtCert(A \rightarrow B,\ expire\_at,\ scope,\ N_B)$ means: "A designates the routing functionality (sending and receiving messages on behalf of A) to B till the time $expire\_at$". Such a certificate is signed by the private key associated with A. Since the name A is derived from the hash of A's metadata, anyone with the possession of A's metadata can validate the certificate.[42] N_B is a nonce from B, its meaning will be clear when we discuss the advertisement process below. Routing certificates can be chained to achieve transitive delegation, e.g. $RtCert(A \rightarrow B,\ ...)\ ||\ RtCert(B \rightarrow C,\ ...)$; *scope* allows a certificate issuer to control the allowed entries in the prefix in such a chain.[43] Note that instead of delegating routing to other GDP-names, a routing certificate can also be issued to a network address from the underlying IP-based networks; such a routing certificate looks as follows: $RtCert(R \rightarrow \langle IP_R, port_R \rangle,\ ...)$. Such routing certificates are issued typically by routers and not clients; we will describe the use of such routing certificates later.

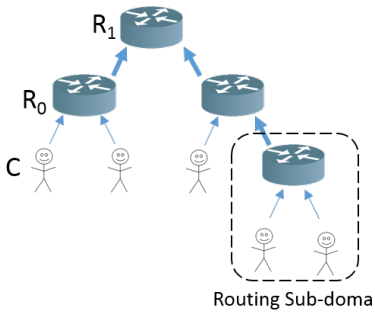
Secure advertisement: In order to join the network in the simplest case, a client C (local name) connects to a router with GDP-name R and network address $\langle IP_R, port_R \rangle$, and sends a nonce N_C. $(R, \langle IP_R, port_R \rangle)$ is the trust anchor a client has to start from in order to secure the routing. On a connection request, the router R presents the client with a signed message $N_C||N_R$, where N_R is a nonce chosen by server (see Figure 9). The client then issues a short lived routing certificate $cert = RtCert(C \rightarrow R,\ expire\_at,\ scope, N_R)$, and then sends *cert* and C's metadata to the router R. If the router verifies that the certificate is valid, it adds a routing entry for C in its local routing table. Otherwise, the router simply disconnects. C can additionally send a number of delegated names $C_0, C_1, ...$ (along with their corresponding metadata and AdCerts) to R—either in the same message or at a later time—which are also added to the local state of R.

[42] One might argue that the routing certificate is no different than an advertising certificate (see § 4.1). Even though the two types of certificates may appear similar, the routing certificates are limited only to the routing layer, whereas the advertising certificates are visible to endpoints and enable setting up shared key for HMACs. In addition, the lifetime of the certificates varies quite a bit; routing certificates are typically short-lived whereas advertising certificates can be arbitrarily long-lived.

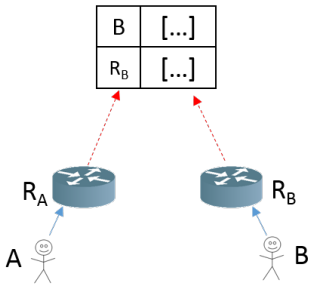
[43] The *scope* description is still an active area of research work; a simple example of *scope* could be the maximum suffix length in the chain.



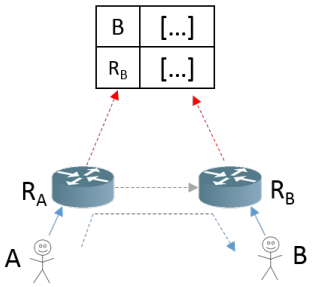
(a) A standalone router R that can securely route messages between clients A and B



(b) Making a tree topology for intra-domain routing using a 'default path' for routers.



(c) GDP routers configured for a lookup service



(d) On demand connection from R_A to R_B.

Figure 10: Intra-domain routing.

4.3.3 Router Configurations

With the secure advertisement process above, a router R that has verified advertisements for two names A and B can easily route messages unicast and anycast messages locally in a verifiable correct way (see Figure 10a). However, in order to be a part of the global routing network, a router can have two more additional configurations: (a) with a default path, and (b) a remote untrusted lookup service for destinations not in the local forwarding table.

Default path: A router R_0 can have a default path to another router R_1; i.e. any PDU with a destination that R_0 does not know about gets sent to R_1 (see Figure 10b). For such a configuration with a default path, R_0 connects to R_1 at startup time as if R_0 were just a regular client. This involves the same secure advertisement process as above including issuance of an $RtCert(R_0 \rightarrow R_1,\ ...)$. Additionally, any client C that connects to R_0 and issues a $RtCert(C \rightarrow R_0,\ ...)$ is also a potential candidate to be forwarded to R_1 depending upon the *scope* requested by C. Thus, over the time, R_1 may see a number of routing certificates $RtCert(C_i \rightarrow R_0,\ ...)$, which combined with the original $RtCert(R_0 \rightarrow R_1,\ ...)$ result into R_1 delegated to perform routing operations on behalf of C_i. We assume that appropriate metadata's are passed along with a certificate such that anyone can verify the certificates using the correct public key. Note that it is easy to generalize this scheme to allow for multiple default paths, one of which is chosen based on some policy specification. Also note that a router must send any multicast PDUs to the default path as well.

Untrusted lookup service: The other configuration with an untrusted lookup service, we consider a service that acts as a repository of (key, value-list) pairs; key is a GDP-name and value-list is a list of certificates (either RtCert or AdCert) corresponding to the given GDP-name. Anyone can publish a valid certificate to such lookup service, and anyone can query for a given GDP-name and

get the corresponding value-list. Such a lookup service is not required to be trusted for correctness, since a querier can verify the correctness of such a service. However, such a service can return partial results. Additionally, such a service does not need to be particularly highly available, since such service is only queried once for any routes not found in local forwarding table of a router.

With such a lookup service, a router can create additional routes on demand. Let us imagine a client A that wants to send a PDU to another client B, where A and B are connected to separate routers R_A and R_B respectively. We also assume that both R_A and R_B are configured to use such a lookup service S; it is also assumed that such routers also publish their own routing certificate as well as their local routing/forwarding information to S as long as such action is permitted by the requested scope of corresponding table entries. Thus, to begin with, S has two entries:

$$R_A \Rightarrow [RtCert(R_A \rightarrow \langle IP_{R_A}, port_{R_A} \rangle,\ ...)]$$

$$R_B \Rightarrow [RtCert(R_B \rightarrow \langle IP_{R_B}, port_{R_B} \rangle,\ ...)]$$

Once both A and B are securely advertised to R_A and R_B, S contains two additional entries (see Figure 10c):

$$A \Rightarrow [RtCert(A \rightarrow R_A,\ ...)]$$

$$B \Rightarrow [RtCert(B \rightarrow R_B,\ ...)]$$

When A sends the first PDU with destination B to R_A, the local lookup fails. Since R_A is configured to use the lookup service S, it recursively queries S until it reaches a point where it knows how to handle the PDU. For each of the queries, R_A receives a list of values—R_A ideally chooses a path based on locality policies, potentially traversing a tree in either breadth-first or depth-first manner.[44] In this particular case, R_A performs two queries: first for B and then for R_B, after which it knows how to contact R_B. For a multicast PDU, however, R_A must use all the path that it learns.

After such a series of queries from S, R_A then initiates a connection to R_B using the network address (see Figure 10d). This connection is immediately followed by a secure advertising with limited scope—R_A issues a certificate $RtCert(R_A \rightarrow R_B,\ ...)$ with a very limited scope that prevents R_B from using this certificate anywhere else. Next, R_A sends the already created certificate $RtCert(A \rightarrow R_A,\ ...)$ from before to R_B; this allows R_B to verify that R_A is, in fact, allowed to send/receive data on behalf of A. In parallel, R_A also populates its local tables to add an entry for B and R_B. Note that R_A treats the connection to R_B slightly specially than any other client: (a) it can be reused for any other addresses B_i that R_A needs to route PDUs for, (b) only specific certificates/advertisements are sent through this connection to avoid leaking routes, and (c) such connections are cleaned up after certain timeouts.

4.3.4 Intra-Domain and Inter-Domain Routing

Intra-domain routing: A routing domain $\mathbb{R}$ can have a number of routers that are connected to each other in some arbitrary topology. We consider a simple tree structure as shown in Figure 10b to begin with, which we later generalize to an arbitrary mesh. In the tree structure, each of the nodes other than the root node is configured with a default path. Each of the subtrees can be considered a routing sub-domain. By default, each node in the tree has the entire knowledge of the subtree underneath, since a router sends all its advertisements to its default path. However,

[44] The potential implications of traversing large trees and identifying local paths is an active area of research. A router may have limits for resources devoted to such queries and a potential cache for the results.

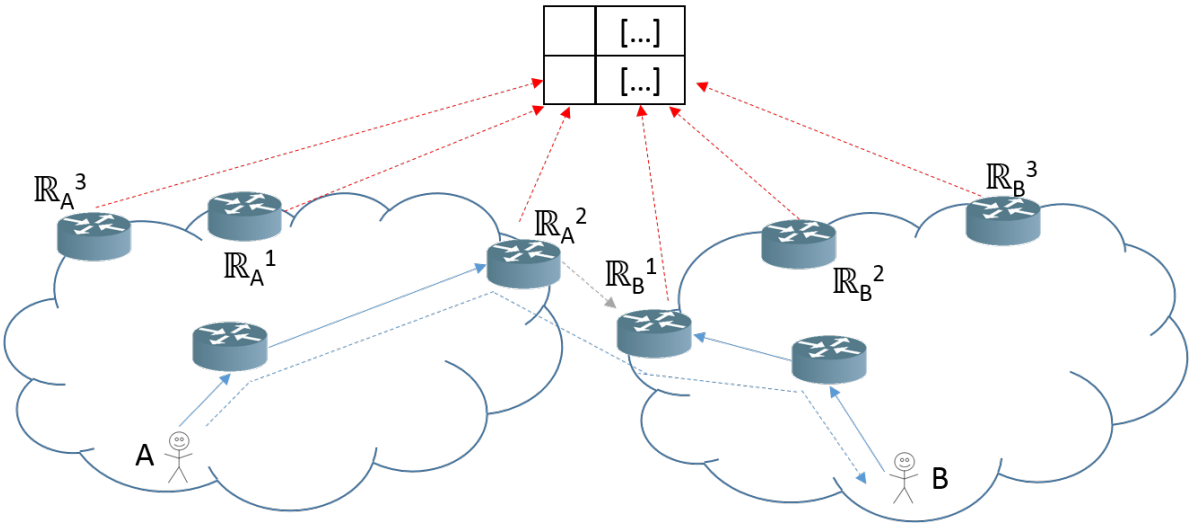


Figure 11: A client A in $\mathbb{R}_A$ initiating communication to another client B in $\mathbb{R}_B$. The communication is facilitated by the border routers $\mathbb{R}_A^2$ and $\mathbb{R}_B^1$ with the help of a global lookup service.

using appropriate *scope*, a client can control how far up its advertisement flows. Two downsides of such a strict tree hierarchy are (a) the root of the tree can run into scalability issues with large number of clients, and (b) each node is a single point of failure for routing to/from clients in its subtree. Thus, such a tree hierarchy is useful only for small and simple routing domains. To avoid the two issues with the tree hierarchy, one could instead use a domain specific lookup service $\mathbb{R}_S$; the routers can be configured to use $\mathbb{R}_S$ and create connections on demand. A step further could be to use an arbitrary combination of the two strategies—default path or lookup service—to achieve any arbitrary mesh within the routing domain $\mathbb{R}$.

Inter-domain routing: To enable inter-domain routing, we imagine a global lookup service $\mathbb{S}$. Each routing domain $\mathbb{R}$ designates one or more *border routers* that can be used to reach clients connected to $\mathbb{R}$. Such border routers are designated with the task of handling all the incoming and outgoing traffic through the given routing domain $\mathbb{R}$, as well as publishing connected clients of $\mathbb{R}$ to the global lookup service $\mathbb{S}$ based on the scope of advertisements of individual clients. In addition, such border routers should be capable of accepting incoming connections from other domains and routers. Figure 11 demonstrates an on-demand connection creation between $\mathbb{R}_A^2$ and $\mathbb{R}_B^1$ to enable communication from A to B.

4.3.5 Discussion

From a first look, it might appear that this routing scheme is source routing, but it is not. In a typical source routing scheme, a sender partially or completely specifies the path a datagram should take in the network. However, in our routing scheme, a client delegates the routing functionality to a router (or a routing domain) which can further delegate its routing functionality to another

router (or a routing domain) as it sees fit. Any network changes, such as a failed link somewhere in the path of a sender and a recipient, only affect the adjacent nodes.[45]

A scalable lookup service: So far, we have assumed that a scalable lookup service is available. The route lookup hinges on the service availability of such a lookup service. When considering a very large number of nodes, the design of such a lookup service itself is challenging. Further, such a lookup service can selectively deny service to specific destinations. However, such problems can be avoided by using a typical Distributed Hash Table (DHT)—a close sibling of the flat address-space routing systems [50, 19]. The scalability of such DHTs is well studied, and these could be run in federated architectures themselves. Note that the probabilistic solutions to the security issues of DHTs (see [53]) can be applied easily for such a lookup service; at worst an adversary may be able to convince enough parts of the network to deny lookups for a given victim and deny route lookups, however at no point can it perform man-in-the-middle attacks.

A scalability argument: At first, it might appear that such a routing scheme has scalability issues that IP routing protocols try to address by using prefixes. Since our scheme does not use any prefixes, a router needs one entry per address in its local forwarding table. Such a state explosion is a matter of concern, for example in Border Gateway Protocol (BGP) where the number of unique prefixes is growing to the order of a million [4]. However, note that as opposed to BGP routers that maintain the entire forwarding information locally and precomputed, our routing scheme works slightly differently—a router only populates a forwarding table entry for a given remote destination by querying the lookup-service when it receives a PDU for that specific destination. Thus, the state required at a GDP router is not the total number of names/addresses in the system, but much smaller. This scalability is achieved at the cost of a small delay incurred before the first PDU can be routed. We evaluate this scalability/cost trade-off in § 5.

4.4 Discussion and Future Work

In this section, we describe some of the less developed ideas to improve the usability, security, or performance of the GDP ecosystem.

4.4.1 Query by Time

A much desired feature in IoT landscape is querying for information by time (or a range of time). We refer to 'time' as the notion of time relative to the log-writer.[46] An encoding of timestamps of appropriate resolution into monotonically increasing integers representing `seqno` achieves this property of query by timestamp with only a minor change to the read-requests.

However, a naive encoding of timestamps leaks quite a bit of information to a third party without a decryption key. The problem is worse for log than a purely communication based streams, because the data storage aspect of log increases the window of opportunity for a malicious third party interested in this information to more than just the duration of communication. In order to achieve some confidentiality, an Order Preserving Encryption scheme can be used to separately encrypt the `seqno` before creating 'append' without any loss of functionality [15].

[45] For example, consider a path like $src \to A \to B \to C \to dst$. If B fails but there is an alternate path $A \to D \to C$, only A, D and C need to negotiate routing certificates and other authentication information before the data flow $src \to dst$ can commence; src and dst do not need to be aware of any such failures. It is true that each such flow $src \to dst$ requires a separate certificate transfer/verification, but such exchange can be done on demand for active flows which effectively results in a fixed small cost per flow.

[46] Issue of absolute correctness of time on the log-writer is out-of-scope of the current work.

4.4.2 Encryption Key Rotation

The decryption keys are a type of *capability* for reading the data. It is a reasonable assumption that not all readers with access to a certain log will be absolutely secure and trustworthy to protect the decryption keys. Especially for long-running communications, this can lead to serious data compromises. To alleviate this inevitable issue, we recommend that the decryption keys be rotated periodically. The new keys can either be generated by the writer itself and communicated to the designated readers using the log-attach request (see below), or it can be distributed by a key-broker (indicated in the metadata) to both the writers and readers. Such key-rotation mechanisms can be used to provide selective access to ranges of data, or implement a revocation scheme where future access to data is denied.

4.4.3 Log-Attach Request

Logs, as described so far, are a perfect fit to address a static graph of services. In order to address dynamic scenarios, a client can use a *log-attach* request. Log-attach request is a special type of request addressed to a principal that's not a log (a service, application, user); it specifies a number of input and/or output logs that the principal should read from/write to. Such a request acts as a glue to enable composability of logs without compromising on security.

Log-attach request uses the general GDP-protocol, and proceeds with a DH-key exchange as in the regular GDP-protocol. However, the key is used to encrypt the payload instead of just calculating an HMAC. In addition to providing a way to compose services, a log-attach request also provides for a way to perform in-band exchange of decryption keys that are required to access the logs included in the request.

4.4.4 Log-Truncation

Log-truncation is marking old data 'safe-to-delete'. The notion of 'old' can be either described with respect to time, or number of records. Ensuring data is truly deleted from a remote storage server is probably an unsolvable problem; truncation only provides a hint to the server that the data is safe to delete. Log-truncation significantly affects the design choice for *hash-pointers* in the record headers and one of the reasons a simple balanced Merkle tree is not the optimal choice for certain use-cases.

4.4.5 Transient Logs

Taking log-truncation to an extreme level, a creator can designate a log to only require a small number of records. Such kind of log can be purely held in memory of a collection of collaborative log-servers, without ever needing to commit data to a physical disk. Such optimizations can improve the performance significantly, extending the communication potential of logs.

4.4.6 Signing Frequency

Signature creation/verification is one of the most expensive operation for writers and readers. Even though more and more devices include hardware-support for cryptographic operation, certain low-end devices might not be capable of (or need) such real-time signing. A tuning parameter for logs is the signing frequency, where a writer does not sign each individual record as it is generated. With

a slightly relaxed threat model and for applications that do not require real-time communication, a writer can choose not to sign every 'append' request. A log-server tentatively accepts such 'append' and sends a secure acknowledgment as usual. However, if the writer can not validate a secure acknowledgment, it must then resend the 'append' request with the appropriate signature. Such optimizations are very well suited to low-power radio based sensors that send multiple records in batches to conserve power.

4.4.7 Read Access Control

Even though encryption is the prime mechanism for read access control, such a scheme is less than ideal because of the side-channel information leakage described earlier. In slightly relaxed threat model, a log-server could be tasked with enforcing read access control with a certificate based scheme; a reader must include a signature on the read request and include an expiration time to reduce the potential time-window for a replay of such signed request by an adversary. Alternatively, in a slightly different relaxation of threat model where routers can be trusted to enforce access control policies, the access to sensitive logs could be limited to *private* routing domains that require a certificate to join.

4.4.8 Preventing Side-Channels

The information-leakage problem is especially challenging for logs because a third party can monitor communication in real-time by just subscribing to the appropriate log; they don't even have to be in a special position in the network to observe the traffic pattern. The time-shift nature and the storage aspect of log also makes things easier for an adversary; an adversary doesn't even have to subscribe in real-time to get this meta-information. Instead, it can read records in bulk at a later time and get at the very minimum size information.

In the particular case of logs, a privacy centric smarter writer can partially alleviate this side-channel leak use 'pointer-records' to overlay a single log ('primary' log) over a collection of 'secondary' logs hosted on mutually distrustful servers. Pointer-records can be used to describe a schedule of N future records using a secure pseudo-random number generator (PRNG) initialized with a seed s that generates integer values in the range $[1, m]$ (m is the number of secondary logs). A pointer record contains N, s and log-names for each of the m logs, all this information encrypted using the encryption key that the writer would otherwise use to encrypt records in a normal log.

Only a reader with the appropriate decryption key can decrypt the information in the pointer-record, and follow the pointers to the secondary log to retrieve the real payload. An adversary subscribed to just the primary log only obtains the information about new pointers, where the information leak can be minimized by randomizing N and applying a random padding to obfuscate size of pointer records. An adversary subscribed to just a single secondary log can obtain the size and time of interleaved records, however the adversary does not know how many records it missed. A more powerful global adversary subscribed to a subset of all possible logs can perform some correlation of individual logs as constituent log of an overlay log, however it can only never be sure of the missing information.

5 Proposed Analysis and Evaluation

So far, we have reasoned that the proposed architecture meets the required security properties under the given threat model. Assuming that GDP achieves the desired security, we provide an evaluation strategy for numerical evidence of performance (and appropriate comparisons with baselines). The purpose of this evaluation is two-folds: (a) measure the performance of the proposed architecture to judge its strengths and weaknesses, and (b) validate the scalability claim. Additionally, GDP provides various tuning knobs by providing the writer control over durability and performance by the choice of the durability strategy or by tuning the linking structure. With the following evaluation, we also hope to provide some insights to a GDP user with the pros and cons of each of these strategies.

As evidenced by the steel man systems earlier in § 2.3, a hand-tuned system can certainly achieve optimal use of resources in the absence of any failures or adversaries. GDP provides a higher layer of abstraction that doesn't require such hand-tuning and can withstand adversarial situations quite well. Using the hand-tuned system as baseline, we measure what is the added cost of security introduced by the GDP. Since the deployment scenarios vary quite a bit and individual systems have many many tunable parameters, a single experiment followed by a numerical comparison with any existing system is not only unfair but also can be potentially misleading. Thus, the goal of this section is merely to provide an intuition into the effectiveness of the cross-layer optimization under a variety of scenarios, and not to evaluate software engineering effort, hardware acceleration features, or network speeds.

5.1 Applications and infrastructure

We consider the following application scenarios: (1) small time-series data values with low durability requirements (e.g. temperature data), (2) small control commands with high durability requirements (e.g. actuation commands), (3) large time-series data values with low durability requirements (e.g. video frames), (4) large data with high durability requirements (e.g. file-system on a log).[47] For each of these application scenarios, we use real-time subscription as well as reading old data, both randomly and sequentially to mimic real application workloads.

In addition to the durability requirement for the above application scenarios, another parameter we would like to vary is the linking structure in the log; we propose to measure at least three different linking strategies described earlier in § 4.1: (a) a simple hash-chain with constant number of back-links, (b) a tree-like topology that includes $O(log(n))$ back-links and allows for better tolerance to holes, and (c) a checkpoint-style back-link strategy, where an application decides certain records as being more important—such as file-system checkpoints.

For most of the experiments, we assume the three administrative zone infrastructure as mentioned in § 2.3: we use a local machine in a lab setting at UC Berkeley to simulate H, we use a cloud data-center within 100 miles as U[48], and we use a transcontinental cloud data-center for C.[49] In addition, we conduct certain micro-benchmarks where all the relevant processes are running on the same machine to rule out any network latencies.

[47] Note that we assume that for writer-driven replication for higher durability, the writer gets separate acknowledgments from the log-servers (as discussed in § 4.2). If we find a better cryptographic solution for combining the acknowledgments from multiple servers, we will adjust the following experiments accordingly.

[48] AWS N. California Region with average round-trip latency of 5ms from UC Berkeley.

[49] AWS N. Virginia Region with average round-trip latency of 72ms from UC Berkeley.

5.1.1 Micro-benchmarks

We take a deeper look at the cost of an individual operation in the above application scenarios (such as append, read, subscribe, etc). We measure the *cost* as both in terms of data size (both on network as well as local state) and computation time (in terms of number of operations).

Writer: The role of a writer is crucial in GDP, since the writer is also the point of data ordering. As discussed earlier, a writer is also the decider of the durability properties of the data. We measure the following parameters: number of cryptographic operations; bytes sent/received in addition to the size of the paylaod; number of blocked operations and pending writes with a given data generation rate; and the amount of state stored locally in terms of number of hashes and pending writes. Further, we measure these parameters in experiments with the application scenarios described above with varying size and durability requirement, three different types of linking strategies, and potentially with a varying number of artificially introduced server failures.

Log-server(s): For the above experiments, we measure the work done (cryptographic operations, state required and data sent/received) by a log-server for accepting writes and sending acknowledgements; replicating to other log-servers or being the replica server (replication performance during filling holes or reconciling branches, as well as bulk sync after a recovery from crash); serving read requests and generating proofs; serving real-time subscriptions; storing data for short term as well as long term by generating extents; and (re-)advertising to the network and storing certificates on behalf of logs.

Reader(s): For the above applications, we measure the work done (cryptographic operations, state required and data sent/received) by a reader for subscription vs reading old data with a number of patterns involving sequential reads and random reads; size of proof as compared to payload; number of hash verifications vs signatures; tolerance for holes/branches; stale vs fresh data—when using primary log-server or secondary replicas.

Router(s): Even though we do evaluation of a routing network separately (see below), we record the following parameters and report them if we notice any interesting pattern: number of cryptographic operations; forward latency per hop; overhead bytes per connection. State (in bytes) per advertisement; state maintenance protocol; round trip time for the closest server as seen by a reader/writer; route failure recovery (advertisement traffic, etc).

5.1.2 Macro benchmarks

In addition to running GDP nodes on the infrastructure H, U and C, we also setup the steel man solutions as discussed in § 2.3. We use these hand-tuned steel man systems to show the effectiveness of GDP's leaderless replication algorithm. The exact software implementation chosen for such deployment is dependent on the availability and ease of deployment for the target applications, but for the moment potential candidates are (a) a file syncing service for a 'C0' system, (b) SFS/SFSlite or LogCabin for a 'C1' system, and (c) Apache Cassandra or Project Voldemort for a 'C2' system.

For each of the applications, in addition to application specific benchmarks, we compare end-to-end latency and other relevant metrics such as: latency from writer (append) ⇒ log-server ⇒ writer (ack), writer ⇒ log-server ⇒ subscriber, and writer ⇒ log-server ⇒ replica log-server ⇒ subscriber. We also measure total long-term data storage requirements for the various steel-man solutions and the GDP.

5.2 Routing scalability experiment

In addition to the applications described above, we propose to do a large scale routing scalability experiment on AWS with a simulated routing topology. The goal of this experiment is to show the scalability of GDP routing network, measure the tolerance to artificially introduced link/node failures, and measure the latency introduced by GDP routing as compared to the actual network link latency (both for the first message of a flow as well as for the steady state). Additionally, we would also like to measure the overhead of our secure advertisement strategy by varying the various parameters, such as the depth of certificate chain, lifetime of certificates, etc. In particular, with a global knowledge of the routing network based on the topology devised and latencies over individual links, we can compare the efficiency of GDP routing network—both for intra-domain routing as well as inter-domain routing.

For this simulated Internet topology, it should be possible to launch 1000-10000 spot-instances for an hour for less than $100 with the current pricing of AWS spot instances and the limits on an individual account. With a global network of data-centers and of the order of 100-1000 clients per instance, it should be possible to do a wide-scale experiment involving $10^5 - 10^7$ GDP-names that mimics the typical Internet latencies. In addition to measurements related to the critical data path, such large scale experiments also allow us to validate our assumptions regarding the scalability of the lookup service used for storing routing state.

6 Conclusions

Motivated by the need of a secure ubiquitous storage infrastructure, in this document, we describe the architecture and design of a widely distributed and federated infrastructure for data storage and communication called the Global Data Plane (GDP). The three key takeaways from this document are:

- A refactoring of interfaces and separation of concerns for cleaner application design, and the use of a secure single-writer append-only log as a unifying communication and storage primitive to build higher layer services on top.
- The design of the secure single-writer append-only log in the presence of potentially untrusted infrastructure, and the use of a leaderless replication strategy.
- The design of a scalable and secure locality aware routing network for inter- and intra-domain routing in the presence of mutually distrustful routing domains with a flat address space.

Many of the architectural details are still evolving, and we hope to have a solution for the remaining pieces of the puzzle soon. We also hope that the performance and overhead evaluation will provide us a constant feedback for improvement in various aspects of the system.

7 Acknowledgments

The ideas presented here would not have been possible without help from a large number of people over the course of a number of years. The early ideas of the Global Data Plane stemmed from intellectual discussions with a number of graduate students and faculty members. Notably, we would like to thank graduate students Ben Zhang, John Kolb and Palmer Dabbelt; and faculty members John Wawrzynek and Edward Lee for their help in formalizing many core aspects and assumptions of the GDP in the early days of the project.

A number of Master's students helped us research and develop individual pieces of the GDP ecosystem. We thank Nikhil Goyal for his contributions to a Click-based GDP-router implementation, Griffin Potrock for his ideas regarding multicast, and Paul Bramsen for FlowID optimization for the GDP protocol.

We would like to especially thank our visiting researchers: Kazumine Ogura (NEC Corporation, Japan) for his contributions to data replication strategies, and Hwa Shin Moon (Electronics and Telecommunications Research Institute, Republic of Korea) for her ideas about key-broker services. Additionally, we would like to thank UC Berkeley staff members Christopher Brooks, Mark Oehlberg and Alec Dara-Abrams for their support in software development and user support.

Finally, we are thankful to a large number of undergraduate students over the years who have helped us with implementation of various individual pieces of the system and maintaining a small infrastructure at UC Berkeley. It is a long list, but we'd especially like to mention Neil Agarwal, Daniel Hahn, Lila Chalabi, Zana Jipsen, Gun Soo Kim, Jordan Tipton, Nicholas Sun, and Siqi Lin.

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STAR-net phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work was also supported in part by The Ubiquitous Swarm Lab at UC Berkeley.

A IoT: A Case Study

Outsourced computation and storage has become a wide-spread and commonplace computation model in the past decade. The emerge of 'cloud computing' and the growing computation needs have been a major proponent for this paradigm shift. This outsourcing model has enabled data-centers become large resource-pools and the clients shrink in size but grow in numbers. The transformation of traditional desktop computing to mobile computing is an example of such trend, and now Internet of Things can only be considered as a continuation of such trend. In particular, the proliferation of Internet of Things (IoT) would not have been possible without this capability to offload computation and data storage to more powerful machines.

Even though there is a wide-spread disagreement about whether IoT is purely a marketing phrase or a fundamentally different computing paradigm to merit a special treatment, the integration of computation in day-to-day life has certainly enabled application developers to create far richer applications than ever before—applications that could benefit from a secure ubiquitous storage infrastructure. In this section, we take a deeper look at the challenges in the IoT ecosystem—*both* applications *and* devices—and demonstrate how an infrastructure like Global Data Plane can help alleviate some of the challenges.

We believe that there is a strong *network effect* when it comes to the usefulness of IoT ecosystem (both devices and applications), i.e. the value of the network increases with the number of connections [32]. The current typical practice is to use the cloud as not only a management platform but also an intermediary for inter-device (or inter-service) communication. In § A.1, we look at the performance challenges of IoT applications; we argue that the current architecture of the cloud as an intermediary in the communication path is not the perfect fit for the growing number of devices, and that certain M2M applications are limited by the latencies and the performance characteristics imposed by such a mode of communication. A separate concern with IoT is that of the security of individual devices, which we look at in § A.2. We argue that the desire to increase inter-device connectivity, combined with the heterogeneity of devices, makes the security problem non-trivial to address.

A.1 A Cloud-Centric Model for the IoT: Performance Challenges

The cloud-centric model for the IoT involves connecting everything to the cloud and using the cloud as the interconnecting hub for the various sensors and actuators [37]. Such a model has proved to be tremendously useful in the growth of the IoT industry. It has enabled hardware vendors and users to collect sensor-data at scale and use the cloud as the centralized data hub for management and processing of the collected data [17]. A number of everyday IoT devices have a cloud counterpart that enables end-users to interact with their devices and the device manufacturers to use user-data to better understand the market needs. Responding to the market needs, cloud vendors have also started focusing on the specific needs of IoT vendors, exemplified by the various IoT specific APIs [2, 3, 5, 6, 8]. The availability of large scale data from such varied devices has also fueled the growth of a wide variety of applications.

This cloud-centric model certainly has its merits: the availability of the cloud resources is a liberating feature for the application developers, who no longer have to work within the constraints of the actual physical device. As an example, even though a tiny sensor continuously generating data may lack persistent local storage, an application developer can use the seemingly infinite storage provided by the cloud to store every bit of data generated by the sensor for eternity. In addition, the cloud model provides a centralized *dashboard* that reduces the management overhead for application

developers/administrators considerably, allowing one to control hundreds of thousands of devices, manage the interconnected nature of the devices, etc.

Even though there are certain advantages of the cloud-centric model, there are a number of issues with this approach. However, in order to understand the challenges better, let us first look at a broad classification of the IoT applications:

Long term data analytics: This category includes *big data* applications that perform machine learning, statistical analysis, or some other kind of data processing on sensor data collected over a period of time, or a across a number of devices, or both. For example, building monitoring system [25], air quality monitoring [57], etc. These kind of applications are typically useful for understanding long-term trends; such understanding can further be used for various purposes such as market development, resource planning, public policy, etc. In many cases, such applications require a significant amount of computational resources and the accuracy of such computations increases with the amount of data.

Real-time applications: These are the applications that involve some kind of real-time actuation. These applications could be as simple as turning on a smart light-bulb based on human input, or as complicated as connected vehicles with real-time decision making [21]. Typically, there is a control-loop involved in these applications, and humans may or may not be a part of such control loop. Quite often, there are strict latency bounds for a normal operation of such applications.

It is not uncommon to see the same data being used for multiple applications that span across the above two categories. We believe that the two types of applications require vastly different type of performance characteristics from the underlying infrastructure. The applications involving large scale data analytics require relatively larger computation resources but don't have strict latency bounds. On the other hand, applications involving real-time actuation do have strict latency bounds, but are often less resource intensive.

While data-center scale resources are a good fit for performing long-term analytics, applications involving real-time machine-to-machine (M2M) communication and tight control-loops can not always rely on the cloud. Even though the cloud is portrayed at the center of the network graph, the reality is that data centers are not as densely located as the human population. Typical Internet scale latencies (50-100ms) are acceptable for human in-the-loop computation, but not for M2M communication. Since the communication latency is directly proportional to distance, the only way to avoid such latencies is to use edge resources for M2M communication. The importance of using local resources becomes even more pronounced when reliable Internet connection isn't available–a reality often ignored by technologists but faced by *billions* of people.[50]

With the performance requirements aside, the security of data and communication is more important than ever as well. This is especially important in a world where adversarial entities have the capability to influence the physical world by controlling real things. With security as a necessity, the lack of appropriate mechanisms for trust is a hurdle in enabling use of resources at the edge–users who want to use multiple service providers to better support low latency communication across a wide geographical range have to use ad-hoc management schemes to make their application work. Our proposed infrastructure of secure ubiquitous storage enables a user to achieve verifiable data security without needing to rely on the reputation of a service providers. This opens the opportunity for an application to seamlessly use a combination of a local resource hub for low-latency communication and far away resources for durability. In theory, such resource discovery process could even be automated by taking the economic factors into account.

[50] A smart house with sufficient local resources should be able to provide *almost* similar functionality whether it is in San Francisco or in a remote village in a third world country.

A.2 Security of IoT Devices: A Heterogeneity Challenge

The general state of security of IoT devices isn't very good [14, 10, 30, 12]. With varied kind of smart devices present in every aspect of life, the impact of a security breach could range from a minor annoyance to failure of critical infrastructure. Not only security, such smart devices lead to endless new privacy issues that didn't exist before [13]. Securing IoT devices is as important as securing any other computer system, if not more so.

Before looking at the general landscape of security of IoT devices, we need to understand the challenges first. Except for the pervasive nature of devices, is there any fundamental difference between IoT devices and traditional computing devices? The Open Web Application Security Project (OWASP) has compiled a list[51] of the top 10 IoT vulnerabilities [9]. None of these security challenges are specific to IoT landscape. So what makes securing the IoT so hard? We attribute the security challenges to two broad reasons:

1. Heterogeneous Systems: Designing an absolutely secure system is challenging, time-consuming and costly. Any system involving more than a few hundred lines of code is prone to software bugs and security issues. The heterogeneity of devices/software in the IoT landscape leads to a large number of unique designs and code-bases; statistically a significant fraction of these systems will have security issues. In traditional computing, de-facto standard tools and software libraries have helped focus the efforts to create better and well maintained software that get regular security updates. Such practices are hard to realize in the heterogeneous IoT world, especially when there is little financial motivation for a device vendor to provide long-term security patches. Lack of software re-usability and market pressure to quickly release products leads to the necessary security features often being considered 'optional'. In addition, any resource-intensive security shims around potentially vulnerable software that work reasonably well for traditional computer systems (e.g. sand-boxing, firewalls, intrusion-detectors) are impractical for most of the IoT devices.

2. Management Overhead and Usability: Another security challenge for the IoT devices is usability. Better security usually is at odds with usability, especially when it comes to the management overhead of authentication and authorization (e.g. password management). In the case of IoT (or any machine-to-machine communication), connectivity and composability of devices and services is a significant driving factor. Good security practices are usually neglected in favor of such usability goals, especially in order to make stove-piped solutions talk to each other. In addition, securing a wide variety of devices requires one to be an expert of all possible systems, which, combined with reduced usability, leads to human errors [56].

Even if these are not the only challenges for IoT security, these definitely are some of the distinguishing challenges for IoT security. We argue that GDP-logs provide a standardized narrow waist with reduced attack surface for communicating with physical devices (see § 3.4). Logs can be used to represent a stream of data generated by a sensor, or consumed by an actuator, or input/output of an application processing data, thus virtualizing the physical devices in some sense (see Figure 4). Using logs, any access-control, firewall, intrusion-detection, etc can be applied to the stream of data living outside the device, thus reducing replication of software functionality on individual devices. In addition, such standardized streams of data incorporate good security practices and enable easy composability and hopefully end-to-end security in the IoT.

[51] OWASP Top 10 IoT vulnerabilities: (1) Insecure web interfaces, (2) Insufficient authentication/authorization, (3) Insecure network services, (4) Lack of transport encryption, (5) Privacy concerns, (6) Insecure cloud interface, (7) Insecure mobile interface, (8) Insufficient security configurability, (9) Insecure software/firmware, (10) Poor physical security.

References

[1] Akamai: Facts and figures. https://www.akamai.com/uk/en/about/facts-figures.jsp.

[2] AWS IoT. https://aws.amazon.com/iot/.

[3] Carriots. https://www.carriots.com/.

[4] CIDR Report. https://www.cidr-report.org/as2.0/#General_Status.

[5] GroveStreams. https://www.grovestreams.com/.

[6] Samsung SAMI. https://developer.samsungsami.io/.

[7] The Cloudflare Global Anycast Network. https://www.cloudflare.com/network/.

[8] Xively. https://xively.com/.

[9] Internet of Things Top Ten: OWASP. https://www.owasp.org/images/7/71/Internet_of_Things_Top_Ten_2014-OWASP.pdf, 2014.

[10] Hackers Remotely Kill a Jeep on the Highway, With Me in It. http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/, 2015. [Online; accessed 27-March-2016].

[11] RSA vs ECC Comparison for Embedded Systems. http://www.atmel.com/Images/Atmel-8951-CryptoAuth-RSA-ECC-Comparison-Embedded-Systems-WhitePaper.pdf, 2015.

[12] Samsung smart fridge leaves Gmail logins open to attack. http://www.theregister.co.uk/2015/08/24/smart_fridge_security_fubar/, 2015. [Online; accessed 27-March-2016].

[13] The government just admitted it will use smart home devices for spying. http://www.theguardian.com/commentisfree/2016/feb/09/internet-of-things-smart-devices-spying-surveillance-us-government, 2016. [Online; accessed 27-March-2016].

[14] Internet of Things security is hilariously broken and getting worse. http://arstechnica.com/security/2016/01/how-to-search-the-internet-of-things-for-photos-of-sleeping-babies/, 2016. [Online; accessed 27-March-2016].

[15] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (2004), ACM, pp. 563–574.

[16] Apthorpe, N., Reisman, D., and Feamster, N. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. *arXiv preprint arXiv:1705.06805* (2017).

[17] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[18] Bagci, I. E., Raza, S., Chung, T., Roedig, U., and Voigt, T. Combined secure storage and communication for the internet of things. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on* (2013), IEEE, pp. 523–531.

[19] Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Looking up data in p2p systems. *Communications of the ACM 46*, 2 (2003), 43–48.

[20] Boniface, M., Nasser, B., Papay, J., Phillips, S. C., Servin, A., Yang, X., Zlatev, Z., Gogouvitis, S. V., Katsaros, G., Konstanteli, K., et al. Platform-as-a-service architecture for real-time quality of service management in clouds. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on* (2010), IEEE, pp. 155–160.

[21] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing* (2012), ACM, pp. 13–16.

[22] Castro, M., Liskov, B., et al. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[23] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 189–204.

[24] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

[25] Dawson-Haggerty, S., Jiang, X., Tolle, G., Ortiz, J., and Culler, D. sMAP: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems* (2010), ACM, pp. 197–210.

[26] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 205–220.

[27] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM computing surveys (CSUR) 35*, 2 (2003), 114–131.

[28] Gass, O., Meth, H., and Maedche, A. Paas characteristics for productive software development: an evaluation framework. *IEEE Internet Computing 18*, 1 (2014), 56–64.

[29] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 29–43.

[30] Ghena, B., Beyer, W., Hillaker, A., Pevarnek, J., and Halderman, J. A. Green lights forever: analyzing the security of traffic infrastructure. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014).

[31] Goh, E.-J., Shacham, H., Modadugu, N., and Boneh, D. Sirius: Securing remote untrusted storage. In *NDSS* (2003), vol. 3, pp. 131–145.

[32] Hendler, J., and Golbeck, J. Metcalfe's law, web 2.0, and the semantic web. *Web Semantics: Science, Services and Agents on the World Wide Web 6*, 1 (2008), 14–20.

[33] Kafka, A. A high-throughput, distributed messaging system. *URL: kafka. apache. org as of 5*, 1 (2014).

[34] Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., and Fu, K. Plutus: Scalable secure file sharing on untrusted storage. In *Fast* (2003), vol. 3, pp. 29–42.

[35] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 3–25.

[36] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices 35*, 11 (2000), 190–201.

[37] Lee, E. A., Rabaey, J., Blaauw, D., Dutta, P., Fu, K., Guestrin, C., Hartmann, B., Jafari, R., Jones, D., Kubiatowicz, J., et al. The Swarm at the Edge of the Cloud.

[38] Li, J., Krohn, M. N., Mazières, D., and Shasha, D. Secure untrusted data repository (sundr). In *OSDI* (2004), vol. 4, pp. 9–9.

[39] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS) 29*, 4 (2011), 12.

[40] Merkle, R. C. Protocols for public key cryptosystems. In *Security and Privacy, 1980 IEEE Symposium on* (1980), IEEE, pp. 122–122.

[41] Merkle, R. C. A certified digital signature. In *Conference on the Theory and Application of Cryptology* (1989), Springer, pp. 218–238.

[42] Plaxton, C. G., Rajaraman, R., and Richa, A. W. Accessing nearby copies of replicated objects in a distributed environment. *Theory of computing systems 32*, 3 (1999), 241–280.

[43] Popa, R. A., Lorch, J. R., Molnar, D., Wang, H. J., and Zhuang, L. Enabling security in cloud storage slas with cloudproof. In *USENIX Annual Technical Conference* (2011), vol. 242, pp. 355–368.

[44] Popa, R. A., Redfield, C., Zeldovich, N., and Balakrishnan, H. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 85–100.

[45] Pouwelse, J., Garbacki, P., Epema, D., and Sips, H. The bittorrent p2p file-sharing system: Measurements and analysis. In *IPTPS* (2005), vol. 5, Springer, pp. 205–216.

[46] Rimal, B. P., Choi, E., and Lumb, I. A taxonomy and survey of cloud computing systems. *NCM 9* (2009), 44–51.

[47] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[48] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001* (2001), Springer, pp. 329–350.

[49] Saito, Y., and Shapiro, M. Optimistic replication. *ACM Computing Surveys (CSUR) 37*, 1 (2005), 42–81.

[50] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[51] Tamassia, R. Authenticated data structures. In *Algorithms-ESA 2003*. Springer, 2003, pp. 2–5.

[52] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29, ACM, pp. 172–182.

[53] Urdaneta, G., Pierre, G., and Steen, M. V. A survey of dht security techniques. *ACM Computing Surveys (CSUR) 43*, 2 (2011), 8.

[54] Wang, C., Carzaniga, A., Evans, D., and Wolf, A. L. Security issues and requirements for internet-scale publish-subscribe systems. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on* (2002), IEEE, pp. 3940–3947.

[55] Weatherspoon, H., Eaton, P., Chun, B.-G., and Kubiatowicz, J. Antiquity: exploiting a secure log for wide-area distributed storage. *ACM SIGOPS Operating Systems Review 41*, 3 (2007), 371–384.

[56] Whitten, A., and Tygar, J. D. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Usenix Security* (1999), vol. 1999.

[57] Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. Internet of things for smart cities. *IEEE Internet of Things journal 1*, 1 (2014), 22–32.

[58] Zhao, B. Y., Kubiatowicz, J., Joseph, A. D., et al. Tapestry: An infrastructure for fault-tolerant wide-area location and routing.