Global Data Plane: A Widely Distributed Storage and Communication Infrastructure

by

Nitesh Mor

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John D. Kubiatowicz, Chair
Professor David Wagner
Professor John Chuang

Fall 2019

Global Data Plane: A Widely Distributed Storage and Communication Infrastructure

Abstract

Global Data Plane: A Widely Distributed Storage and Communication Infrastructure

by

Nitesh Mor

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John D. Kubiatowicz, Chair

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 in a world full of complexity. 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).

The key abstraction in the GDP is a secure cohesive container of information called a DataCapsule, which provides a layer of uniformity on top of a heterogeneous infrastructure. A DataCapsule represents a secure history of transactions in a persistent form that can be used for building other applications on top. Existing applications can be refactored to use DataCapsules as the ground truth of persistent state; such a refactoring enables cleaner application design that allows for better security analysis of information flows. Not only cleaner design, the GDP also enables locality of access for performance and data privacy—an ever growing concern in the information age.

The DataCapsules are enabled by an underlying routing fabric, called the GDP network, which provides *secure* routing for datagrams in a flat namespace. The GDP network is a core component of the GDP that enables various GDP components to interact with each other. In addition to the DataCapsules, this underlying network is available to applications for native communication as well. Flat namespace networks are known to provide a number of desirable properties, such as location

independence, built-in multicast, etc. However, existing architectures for such networks suffer from routing security issues, typically because malicious entities can claim to possess arbitrary names and thus, receive traffic intended for arbitrary destinations. GDP network takes a different approach by defining an ownership of the name and the associated mechanisms for participants to delegate routing for such names to others. By directly integrating with GDP network, applications can enjoy the benefits of flat namespace networks without compromising routing security.

The Global Data Plane and DataCapsules together represent our vision for *secure ubiquitous storage*. As opposed to the current approach of perimeter security for infrastructure, i.e. drawing a perimeter around parts of infrastructure and trusting everything inside it, our vision is to use cryptographic tools to enable intrinsic security for the information itself regardless of the context in which such information lives. In this dissertation, we show how to make this vision a reality, and how to adapt real world applications to reap the benefits of secure ubiquitous storage.

To my parents.

Contents

List of Figures

Preface

We live in a world where computation is deeply embedded in everyday life and is bringing many positive changes to the world; things once considered fantasy and science fiction are now possible. On the flip side, such technological advancement comes at the cost of highly complex systems that are difficult to analyze and reason about, and thus are prone to security issues. We are at a stage where physical objects around us are controlled by potentially vulnerable systems, and extreme attention to security is needed. Complexity is inevitable, but it is extremely important to manage this complexity well in order to prevent IT security disasters.

While we are limited by the infrastructure and the technology of today, what would a world look like if it were designed with a different outlook? Is there a viewpoint and design philosophy that addresses the complexity, and thus safety and security of things around us? The book *"Trillions: Thriving in the Emerging Information Ecology"* by Peter Lucas, Joe Ballay, and Mickey McManus outlines such a vision to tame the complexity. Loosely inspired by this book, we imagine a world around *secure information* in the form of DataCapsules that are analogous to a freight container.

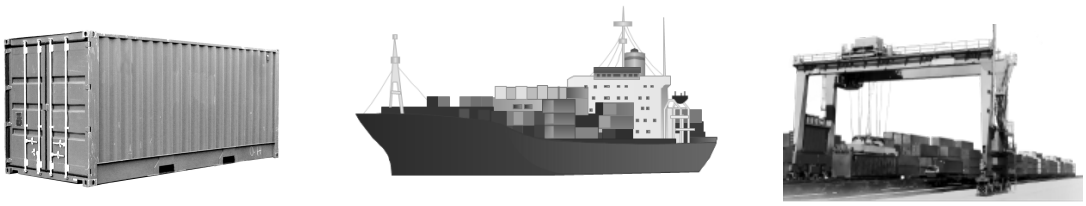


Figure: A freight container as a standardized unit of shipping. Infrastructure, such as container ships and cranes, can be designed to handle the standardized shipping container.

Freight containers revolutionized the shipping industry by bringing a form of standardization. With a freight container as the standardized unit for movement of goods around the world, the rest of the infrastructure can be designed around a relatively simple interface leading to an overall efficient shipping industry. Similar to freight containers, a DataCapsule is a cohesive information container that can be moved around in the digital world as necessary while providing verifiable security guarantees such as confidentiality, integrity, and provenance of contained information. Just like freight containers revolutionized the shipping industry, DataCapsules bring standardization to the information landscape in the form of a simple yet sufficient interface; this standardization is extremely valuable for addressing the complexity and heterogeneity while keeping information secure.

Can we accomplish this vision of DataCapsules through incremental changes to the existing world? Or is this such a bold vision that it requires us to start afresh? We believe that it is indeed possible to bridge the gap between the reality today and a brighter future tomorrow, and this dissertation is an attempt at realizing this vision. Towards this goal, the dissertation delves deep into the design of DataCapsules and the supporting infrastructure in the form of the Global Data Plane (GDP)—the equivalent to the shipping infrastructure that handles the cargo containers. Our agenda is standardization of information around security and trust, and the dissertation embarks upon a journey toward this vision.

How to read the dissertation

The dissertation is divided into three parts:

Part 1 is intended for a general audience: it discusses the rationale for DataCapsules and a system like the Global Data Plane. Without going into too much internal mechanistic details, this part provides a gentle introduction of what DataCapsules and the GDP mean to various stake holders such as end users, application developers, and infrastructure operators. It is divided into two chapters:

Chap. 1 gives the background and general motivation for a system like the Global Data Plane. It also outlines the research challenges that must be addressed for the vision to be realized.

Chap. 2 provides a more detailed, albeit interface-level, introduction to DataCapsules and the Global Data Plane. It provides the operational details for infrastructure providers and interface details for application developers.

Part 2 delves deeper into the research questions outlined in the first part, and goes into the mechanisms for making the system a reality.

Chap. 3 summarizes the *design* of DataCapsules. The design discussion is at a level that even if there were no network and DataCapsules were to be stored and moved around explicitly by sneaker-net, it should still work.

Chap. 4 goes into the *engineering* of how to make the DataCapsule vision work for real applications on a network underneath with a small set of assumptions.

Chap. 5 discusses the *design* of a routing network underneath—called the GDP network—that provides a secure routing fabric for DataCapsules as well as other native applications.

Chap. 6 details the *engineering* needed for a scalable GDP network that actually works in practice, including some real world topologies.

Part 3 describes the implementation of the GDP and DataCapsules, and provides a quantitative assessment of the design.

Chap. 7 provides an overview of our prototype implementation, and how the various components perform in the real world.

Chap. 8 concludes the dissertation. While this dissertation is an exercise in design—and hence just a beginning—this chapter lays the path that engineers and system builders must take to convert the vision to a reality.

Acknowledgments

The ideas presented in this dissertation would not have been possible without the help from a large number of people. The path to a doctoral degree is long and arduous. I would like to thank each and every individual who I met and interacted with during my stay at Berkeley for shaping me into the person I am today, and thus helping formulate this dissertation the way it is. I dedicate the final step of my doctoral journey, this dissertation, to all of you.

Most importantly, I would like to thank my advisor, Kubi (Prof. John Kubiatowicz), for his continuous support throughout the years. He taught me the importance of asking the question *why*. Thank you for treating me as an equal in our discussions, for pushing me to find a research topic that I am passionate about, and just being there when I needed you the most.

I would like to extend a special thanks to my colleagues from Swarm Lab: Eric Allman, Ken Lutz, and Rick Pratt. Without their guidance and mentorship, I wouldn't have been able to find a good direction. Thank you. I hope this dissertation is a fitting tribute to the endless discussions and brainstorming sessions.

I would like to thank Prof. David Wagner, Prof. Anthony Joseph, and Prof. John Chuang for their help and guidance during my qualifying exam and helping formulate the beginnings of a dissertation. I would like to thank Prof. David Wagner and Prof. John Chuang for also being a part of my dissertation committee and reviewing this dissertation.

The early ideas of the Global Data Plane stemmed from discussions with a number of graduate students and faculty members. Notably, I would like to thank graduate students Ben Zhang, John Kolb and Palmer Dabbelt; and faculty members Prof. John Wawrzynek and Prof. Edward Lee for their help in formalizing many core aspects and assumptions of the GDP in the early days of the project. I would also like to thank Prof. Ken Goldberg and post-doctoral researcher Ajay Tanwani for collaboration on the Secure Fog Robotics initiative towards the later parts of the dissertation.

A number of Master's students helped us research and develop individual pieces of the GDP ecosystem. I 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.

I would like to especially thank visiting researchers in our research group: 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, I 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.

I am also 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 I'd especially like to mention Neil Agarwal, Daniel Hahn, Lila Chalabi, Zana Jipsen, Gun Soo Kim, Jordan Tipton, Nicholas Sun, and Siqi Lin for their help in various operational aspects of the system.

Throughout the dissertation, the work was financially supported in part by The Ubiquitous Swarm Lab at UC Berkeley. Early stages of the work were 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. The later stages of the work were supported in part by NSF and VMware under NSF award #1838833 (ECDI: Secure Fog Robotics Using the Global Data Plane). I am thankful to all these agencies and institutions for their generous financial support.

Finally, I would like to thank all my friends who made the journey a lot more enjoyable. A special thanks to Tathagata Das and Moitrayee Bhattacharyya for being there through my moments of joy and sorrow, and being my family away from home. I'd like to thank Ivana Stradner for helping me through the lowest moments in my dissertation journey. A special thanks to Pulkit Agrawal for discussions on almost every aspect of life and work as a friend, a housemate, and a neighbor. Thanks to Kasra Nowrouzi for sharing the journey of writing a dissertation. I'd also like to thank Jack Kolb for being a truly amazing hiking companion; thank you for partaking in the crazy excursions to almost all of Northern California. Thank you Anurag Khandelwal and Shromona Ghosh for the warmth and joy you brought in my life.

I'd like to thank the I-House community for two wonderful years of my life. I am truly grateful for the intercultural experience that I-House provided me with. I cannot think of a better way of sharing experiences and culture with so many amazing people from around the world. Thank you. Life as a graduate student would have been extremely dull without some amazing housemates over the years. I'd like to thank Prachi Shah, Richie Przybyla, Ning Zhang, Steven Callender, Saurabh Gupta, and Pulkit Agrawal for bearing with me as a housemate.

Last but not the least, words cannot express how grateful I am to my parents. Thank you for making me who I am today.

Of course, all the errors in this work are my own.

Part I

The Why and The What: Introduction, Motivation, and Overview

Chapter 1

Introduction

1.1 Edge computing: An evolving computational landscape

The world is moving towards a ubiquitous computation model. In recent years, 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 end users. The cloud computing model of data centers as the enormous pools of resources 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 academics 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 [1], [2]. According to Gartner, around 10% of enterprise-generated data is created and processed outside a traditional centralized data center or cloud as of 2018, but this number is expected to reach 75% by 2025 [3]. The cloud is certainly a part of the big picture, but cloud alone is not enough [2].

In terms of terminology, this trend of computation moving closer to end-users has been called by various names, such as 'edge computing' [4], 'fog computing' [1], etc. When it comes to details, currently there is no single consensus on what the various terms exactly mean and if they are all the same. However, the mere presence of the debate sufficiently acknowledges the existence of this trend. Without necessarily claiming to provide an authoritative definition, we just use the term 'edge computing' as an umbrella term to refer to this trend.

As for physical manifestation of edge computing, we take a broader stance. In recent years, hundreds of modestly sized data centers have started to show up near centers of human population worldwide; the intent is to be able to deliver content quickly to end-users [5], [6].[1] These data centers are portrayed as the edge of the cloud that is closest to the users, and target the attention of existing cloud practitioners by sharing many of the same characteristics of typical cloud computing. However, there is another layer of mostly decentralized resources managed by individuals, smaller

[1] In contrast, the cloud is traditionally characterized by a handful of mega data centers in locations chosen to minimize the cost. Typically, such cloud data centers are far away from end users in the places where land, electricity, and other resources can be procured cheaply.

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. We consider the resources in this hidden layer as an essential part of the edge.

Adoption hurdles for edge computing in a cloud-centric world

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, the current adoption is relatively low [3]; 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. When it comes to running services and applications today, we still rely on primarily a cloud-centric model.

We argue that the reason for 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 of widely heterogeneous resources. The task of provisioning and maintaining resources, which amounts to a relatively simple thing in cloud computing that requires just a few clicks, ends up being extremely time consuming and expensive undertaking for edge computing scenarios. 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 ease of management has been acknowledged as a major factor in the growth of the cloud [7], and the advantages of using edge computing resources must be justified against the additional cost of managing complex systems that require understanding the many knobs and dials available to an administrator.

The complexity of managing a wide range of disparate systems also makes it challenging to secure the infrastructure and more specifically, the information. Traditionally, perimeter security has been the de-facto mode for securing infrastructure. Perimeter security involves drawing a boundary around the infrastructure, filtering everything that crosses the boundary, and treating everything inside as trusted. Perimeter security is not a sufficient model of security anymore: ensuring appropriate security requires covering every possible attack vector, whereas exploiting a system needs only one unattended entry point. Even with homogeneous resources in the cloud where it is relatively easy to use perimeter security approach, breaches are not uncommon. With the complex edge computing landscape where resources are spread over a wider area both inside and outside of traditional data centers, using a perimeter security approach clearly is a non-optimal strategy. While the basics are the same, security challenges in edge computing are just more visible because of the complexity, heterogeneity, and the geographical spread.

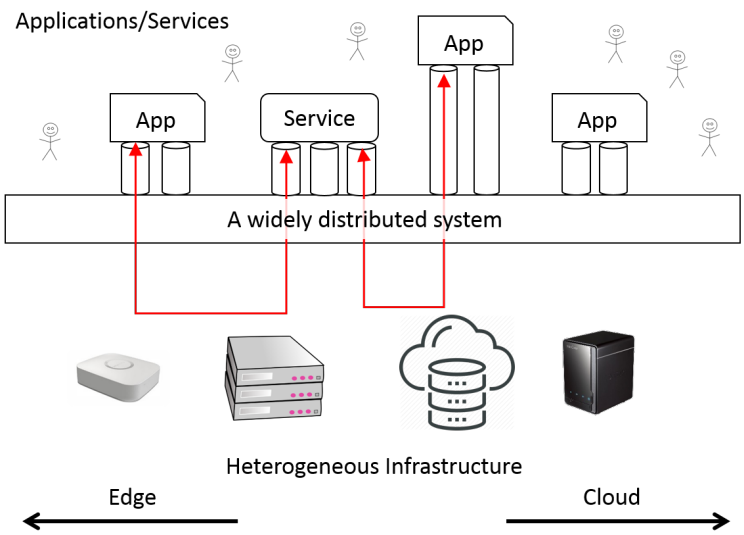


Figure 1.1: A widely distributed system spread over heterogeneous infrastructure that provides a uniform *platform* for applications (see section 1.2).

Summary: 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. Maintaining information security in a heterogeneous world remains a laborious task that requires careful planning and expertise.

1.2 Case for a ubiquitous storage platform

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 [8]. Even for IaaS, some have argued that cloud IaaS has now achieved a point where developers are burdened with managing a plethora of resources just like the pre-cloud days of running and managing your own infrastructure; instead of managing physical serves, disks, network cables, routers, and firewalls, developers now have to manage the virtual equivalent of these in IaaS [9].

To enable application developers focus on writing applications instead of managing individual resources, various cloud-based service providers also offer a Platform as a Service (PaaS) where an application developer works with a higher level interface for individual resources (compute, storage, etc) [10]. In more recent years, the proliferation of *serverless* computing in the form of Function as

a Services (FaaS) has taken this platform approach to a completely new level [9].[2] 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 [11], [12].

It is important to draw the platform abstractions at the right level. If the level of abstraction is too low, for example in IaaS, it leaves much work for the users to do. On the other hand, too high level of abstraction makes the platform extremely specific to only a limited set of users and applications. We do believe that a platform with a narrow and well specified interface at the right level of abstraction is not only possible, but is the right approach for addressing the complexity introduced by the proliferation of resources in a somewhat manageable way.

In addition to addressing the complexity question, a platform enables decoupling of concerns. A fundamental principle adopted by IaaS offerings but perfected by PaaS conceptualization is the principle of separation of persistent state from compute. Such a separation is useful for at least two reasons. First, a separation of computation from storage allows service providers to specialize in either storage or computation, and not necessarily both. And second, 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 proactively 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.

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 dissertation we explore the idea of a storage and communication platform in the form of a widely distributed system. Such a system provides a homogeneous interface to the application developers, even though the system itself could be spread over a heterogeneous infrastructure managed by a large number of administrative entities. We envision that such a system can provide a seamless integration of resources at the edge with those in the cloud, if done right. While computation is also a necessity in making our full vision a reality, we merely provide hints on how to integrate computational elements in the picture considering computation as out of scope of this dissertation.

Properties required of the platform

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 that are needed to support the edge-computing ecosystem:

- **Homogeneous interface:** First and foremost, such a system should provide a homogeneous interface even though the infrastructure underneath can be quite heterogeneous. This requirement 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

[2]Even though the proponents of *serverless* computing may propose it to be a groundbreaking new abstraction, we still consider it to be just another PaaS offering specialized for computation.

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.[3] A system designed with a fundamental assumption of a large number of administrative entities naturally leads to a system architecture with minimal trust in the administrative entities. Further, as opposed to the cloud ecosystem where only a few large players dominate the market, the edge-computing ecosystem is, in fact, comprised of 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 [1]. 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. Ideally, an application developer should be able to dictate which parts of the infrastructure should be used for specific data. There are two reasons for this: 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.

[3] 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.

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.

- **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 adversary monitoring communication in real time can learn information from the side-channels (such as size and timing of messages) [13]. In a federated infrastructure where the system routes a reader/writer to a close-by resource, it becomes possible 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 *black-hole*)—a problem well studied in overlay routing schemes [14]. The two requirements—allowing anyone to be a part of the network while preventing 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*) [15]. Toward this goal, we believe that *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; predictable control loops in industrial environments with explicitly provisioned resources; a data-center model similar to the cloud; or any combination of such scenarios. We also argue that a federated architecture brings openness to the platform; it allows participants to join the system on their own terms: they could bring in their own infrastructure for private use, make the infrastructure available to others in exchange for money, or use the infrastructure for services open to the world.

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.

1.3 Refactoring the world around secure information

Inspired by this idea of a *platform* for *secure ubiquitous storage*, we explore the detailed architecture, mechanisms and implementation of a widely distributed and federated storage and communication infrastructure in this dissertation. We call this infrastructure the Global Data Plane (GDP). The key abstraction in the GDP is a DataCapsule, which we introduce next.

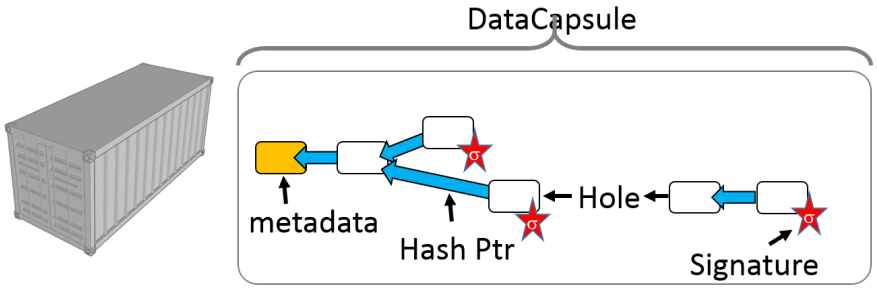


Figure 1.2: A DataCapsule as a digital equivalent of a freight container with standardized characteristics that enable the infrastructure to handle it easily. Internally, a DataCapsule is a cohesive container of information secured via cryptographic primitives.

1.3.1 DataCapsules: A container for information

A DataCapsule is a secure, globally named container of information. A DataCapsule represents a sequence of transactions in a specific order with appropriate attribution to the origin of information. A DataCapsule is a way for applications to maintain secure persistent state in *portable* form, i.e. verifying the integrity and provenance of information requires minimal assumptions from the infrastructure. DataCapsules are identified by a cryptographically derived flat 256-bit name; this name serves as the cryptographic trust anchor for verifying the membership, integrity, and provenance of information it contains. Further, the name of a DataCapsule is a location independent name that is fundamentally decoupled from the physical infrastructure underneath, thus enabling portability. This portability property for information security is the enabling feature for taming the complexity of heterogeneous distributed systems.

The inspiration for a DataCapsule comes from freight containers: a concept that revolutionized the cargo industry. A freight container provides a standardized way of transporting goods around the world. Even though each container is handled by a number of different operators throughout its lifetime, the contents are reasonably secure, trackable, and can be handled by the infrastructure without worrying about specifics of what a container contains. Before the standardization of a freight container, goods were typically shipped in loose sacks where they were prone to theft, loss, and damage. Further, moving goods from one mode of transport to a different mode required repackaging the goods, which was laborious. Freight containers simplified the transportation and handling of goods. Similarly, DataCapsules enable the infrastructure to handle information securely and migrate it elsewhere if needed.

The equivalence of a DataCapsule to a freight container breaks slightly in two important ways. First, unlike physical items, making copies of digital information is rather straightforward. As such, a DataCapsule can be replicated easily. When we refer to a DataCapsule, we don't mean a specific replica. Instead, the DataCapsule is a virtual object represented by a combination of all the replicas. Second, unlike freight containers that are limited in physical size, there is no inherent limit on the amount of information a DataCapsule can contain. A single DataCapsule can be just

a few kilobytes representing a small document, or it can contain terabytes of data that represent the output of a large scale physics experiment. The standardization that a DataCapsule provides is in the form of interface to infrastructure to support DataCapsules as well as the interface for applications. For a freight container, the fact that the physical dimensions, weight limitations, etc. are well specified isn't important in itself; instead, fixing such properties leads to a fixed interface around which the infrastructure can be designed.

While the analogy of a DataCapsule to a shipping container has limits, thinking of information handling in the form of containers enables a unified way of reasoning about information in-transit and at-rest, even when such information transits through infrastructure owned and operated by different stake holders. Just like freight containers, DataCapsules standardize a narrow interface that is simple enough for the infrastructure to handle, yet sufficient for applications to use it. The application interface for a DataCapsule is sufficiently narrow to enable both applications and infrastructure to reason about the information and derive further higher level application-specific properties.

In terms of interface to applications, a DataCapsule is primarily a single-writer append-only data structure.[4] For readers, DataCapsules support subscribing to new information, secure replays at a later time (thus providing a time-shift property), and efficient random reads for old information. Such an interface may seem limiting, but it is a natural fit for anything that can be represented as a stream of information, e.g. an IoT device generating sensor readings, events such as financial transactions, etc. Further, the DataCapsule interface is sufficiently rich to fulfill the needs of a wide variety of general purpose applications.

An important consequence of being able to subscribe to new information in real-time (or in a time-shifted manner) is that it enables a publish-subscribe mode of operation, which further enables decoupling of a data producer from a consumer. A DataCapsule can thus be viewed as a unidirectional communication channel. Since DataCapsules are designed for information security, publish-subscribe mode benefits from the same information security for free. Thus, DataCapsules enable a unification of data security at-rest with data security in-transit. Applications can, if they wish, rely on DataCapsules for all state maintenance and communication with other applications/services.

DataCapsules provide a narrow uniform interface to the heterogeneous infrastructure underneath. We provide a detailed discussion on the DataCapsule interface in the next chapter. But let's first see what the supporting infrastructure looks like.

1.3.2 The Global Data Plane: An ecosystem for DataCapsules

DataCapsules provide a way to refactor applications around information. The computation and application logic generates and orders the information, whereas DataCapsules keep the information secure in a consistent way. The physical infrastructure provides a backing for DataCapsules: it

[4] We consider the consequences of violation of the single-writer assumption when we discuss mechanisms in chapter 3.

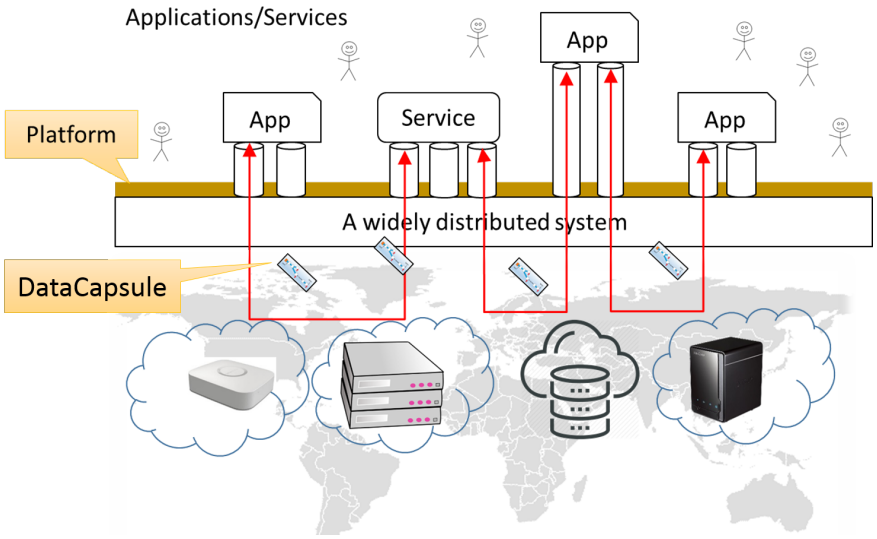


Figure 1.3: The Global Data Plane (GDP) enables an ecosystem for DataCapsules. The GDP provides maintenance of persistent state for applications and services via the DataCapsule interface.

makes the information persistent and available to others in a secure and verifiable way. We call our implementation of the necessary ecosystem and infrastructure around DataCapsules the Global Data Plane (GDP). The GDP is to DataCapsules what a distributed filesystem is to files. In addition to providing persistent storage for DataCapsules, the GDP also exposes the underlying communication fabric to applications that need it; we will explain this in later chapters of the dissertation.

Continuing with the analogy to a shipping container, the GDP can be viewed as the supporting infrastructure that can handle standardized containers of information in the form of DataCapsules. In the shipping industry, such infrastructure comprises trains, trucks, ships, cranes, and other physical equipment owned and operated by multiple administrative entities. Similar to the way providers in the shipping industry are hired for transportation (and sometime storage), the GDP is designed with a focus on a utility-provider model of computing.[5]

In our utility/service provider model, users and applications package their data into DataCapsules, which are made durable and available wherever necessary by the service provider. Users maintain either explicit or implicit economic relations with the service providers in exchange for the services. To achieve greater availability and redundancy, a user may use multiple service providers at the same time for the same DataCapsule.

An example of such a utility is already available to us in the form of the Internet. The Internet is a utility enabled by a federation of Internet Service Providers (ISPs) who provide packet transport

[5]While the distinction between a 'service provider' and a 'utility provider' is thin and is used interchangeably even in this dissertation, we prefer the term 'utility provider' with the rationale that a *service* can be arbitrary complex, whereas a *utility* is a smaller, well defined task (or set of tasks). The GDP enables a decoupling of trust from durability/availability and thus reducing the responsibilities for the infrastructure—it only makes sense that we use the term *utility*.

services. Users do not rely on ISPs for security; instead, they have various tools for transport security at their disposal (e.g. TLS, HTTPS, VPN, SSH, etc.). In a similar way, DataCapsules decouple security from durability and availability; while a DataCapsule interface enables security, the concerns of durability and availability are left to the infrastructure. More importantly, durability and availability are the only concerns the infrastructure needs to handle. As a result, users can acquire both storage and communication services from GDP utility providers without trusting them for data security.[6]

Other than enabling a utility provider model, the separation of security from durability and availability has another benefit: it allows a writer to decide on the importance of durability by means of policy specification, thus allowing for applications on both ends of the performance/durability spectrum to use the same common interface—a high performance video feed that can tolerate a small number of missing frames as well as a database where every individual transaction must be made durable.

At a physical level, the GDP is a distributed system of heterogeneous storage and routing nodes; storage nodes (called *log servers*) provide a physical backing for DataCapsule storage, routing nodes (called *GDP routers*, or simply *routers*) perform secure data routing. The GDP organizes resources based on ownership. The physical infrastructure is divided into various administrative domains (organizations) based on resource ownership; 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. A key component of the GDP is an underlying routing fabric called the GDP network, which serves as the message transport layer for the GDP.

With a standardized DataCapsule interface, service providers may implement their systems and infrastructure as they see fit. Service providers in the GDP are analogous to autonomous systems (AS) in the IP world. Just like the current Internet, the GDP is enabled by collections of resources owned by individuals, organizations, and governments. While we present the GDP design with a public utility provider model as the most common scenario, this same model encompasses private corporations or even individuals deploying resources for their private use.

Although motivated by the needs for secure ubiquitous storage, the 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 the GDP to provide an opportunity to use local resources at the edge wherever possible for low-latency access and using cloud resources for durability.

[6]Service providers only see encrypted data, and we trust service providers not to leak such encrypted data to third parties. If such a leak happens, either because of a malicious infiltration of service provider or because the service provider itself acts maliciously, the reduction in data security is limited to the additional information revealed through a side-channel analysis on size and timing of encrypted information.

1.3.3 Novelty claims of the GDP

Rather than designing a new system, could we modify existing systems and applications or use a combination of existing tools to 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 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 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 can often miss the subtle requirements for such tool, leading to unsolved engineering challenges or broken systems.[7] Second, duplicated functionality across various layers of abstraction often leads to inefficiencies and performance penalties; such inefficiencies become more pronounced for applications with small data items 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.

Even though application developers can use a combination of existing tools and technologies to achieve similar functionality as the GDP, it is a burden on developers to use these tools correctly and requires domain expertise in both computer security and distributed systems. By providing standardization around information security that is suitable for a wide variety of applications, the GDP rules out the need for ad-hoc combinations of tools. The GDP provides an infrastructure with cross-layer optimization focusing on efficiency and a clean application interface, and providing a high-level interface to be used by application developers.

In addition to standardization, the GDP focuses on multiplicity of resource ownership as well as information ownership. Most existing ecosystems—especially ad-hoc systems created for a specific environment—do not have to worry about the multiplicity of resource ownership; a single administrator system has much more flexibility in design choices than the GDP. As an example, a closed system can more easily enforce a centralized admission control policy to keep bad actors out of the system. However, a system like the GDP must allow anyone to join the system, but still contain the damage that bad actor can do. The generality requirement of the GDP forces us to go back to first principles and keep a higher standard of security with minimal assumptions in the operating environment.

The 'secure ubiquitous storage' challenge pushes architectural limits of distributed storage system design, data security mechanism, and real-time communication. These sub-challenges

[7] As an example: imagine a widely distributed 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.

must be solved together. The GDP, however, is inspired by many existing systems. 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 [16], Oceanstore [17], and Plutus [18] provide a filesystem interface on untrusted storage servers; CryptDB [19] provides a database interface; Depot [20] provides a key-value store interface; Antiquity [21] provides a log-based interface; and so on. Additionally, a number of existing systems have specific goals in specific environments; SiRiUS [22] targets users that cannot modify remote storage server; CloudProof [23] 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: cryptographic solutions that rely on possession of keys and validations of cryptographic proofs,[8] or fault-tolerant distributed systems that assume that a certain fraction of storage servers to be honest at all times.[9] 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 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) [27]. In addition to pub-sub systems to handle the high volume of data in trusted single-administrator environments (e.g. Apache Kafka [28]), a number of scalable Internet-wide pub-sub systems also have been devised (see [29] 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 [30]). The GDP adopts the pub-sub mode of operation 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 [28].

[8] 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. [24]

[9] The basic building blocks used are primarily solutions to the Byzantine Generals' Problem; examples include PBFT [25], A2M-PBFT [26], Byzantine-Paxos, Blockchains, etc.

The GDP as a system does fill some gaps in what one can do with existing systems. Notably, the GDP provides (1) a widely distributed storage infrastructure; (2) minimal security assumptions from the infrastructure underneath that does not have a centralized authority and is partitioned into a large number of administrative domains just like the Internet; (3) support for real-time operations, i.e. a native communication platform. Rephrasing, the design of the GDP aims to achieve three key properties: (1) distributed storage, (2) security, and (3) real-time communication.

The GDP aims to achieve the best in all these metrics by targeting a relatively unexplored design point: a secure DataCapsule interface directly exposed to writers/readers with a separation of security from durability and availability. We adopt the DataCapsule as a first class data structure, where the primary task of the infrastructure is to make DataCapsules durable and available to appropriate readers. This design paradigm 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.

1.4 Research questions and tasks

We organize our work into two high level research tasks that we introduce here. This specific organization is not because these tasks are completely independent of one another, but because each of these tasks targets a specific functionality that can be analyzed and evaluated independently. Further, separating the tasks this way forces us to design our components in a somewhat modular way. Even though we solve these problems in the context of our specific system, we believe that the ideas and the results can be applicable to other systems that demand similar requirements.

Task 1 *Design of DataCapsules to be transportable over a federated infrastructure, while providing integrity and provenance for every single bit of data and enforcing the ordering relationships between such bits.*

First and foremost, this task involves API specification of DataCapsules for writers, readers, subscribers, and the infrastructure in the presence of a threat model where the infrastructure is only trusted for service, but not for security. We discuss these interface level details in chapter 2. In chapter 3 and chapter 4, we discuss the internal design and engineering required for the DataCapsule vision. We specifically try to address three sub-challenges: (1) design of an efficient DataCapsule that enables a user to achieve application-specific performance/durability goals while preserving data-integrity; (2) design of an efficient leaderless replication strategy with suitable consistency guarantees that works well when a DataCapsule is placed in a distributed infrastructure; and (3) design of a secure communication protocol that translates the operations on a DataCapsule to bits on the network.

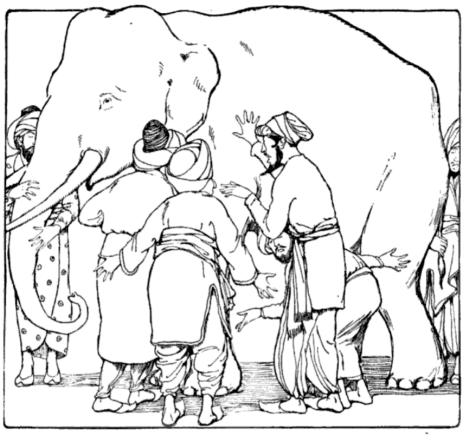
Task 2 *Design of a secure routing network with flat address-space that serves as the enabler for a federated, service-provider model of infrastructure deployment.*

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. We call this communication fabric the GDP network. We describe the interface briefly in chapter 2. We

discuss the design and engineering in chapter 5 and chapter 6, where we describe the mechanisms to address the two main challenges (1) secure unified mechanisms for advertisements, and (2) routing across administrative domains while providing transitive proofs of routing delegation.

We should emphasize that the contributions of this dissertation is not merely solving these two research problems as independent units. Instead, the key contribution of the dissertation is the design of a comprehensive system whose viability happens to depend on addressing these two key problems.

Perspectives



A group of blind men heard that a strange animal, called an elephant, had been brought to the town, but none of them were aware of its shape and form. Out of curiosity, they said: "We must inspect and know it by touch, of which we are capable". So, they sought it out, and when they found it they groped about it. In the case of the first person, whose hand landed on the trunk, said "This being is like a thick snake". For another one whose hand reached its ear, it seemed like a kind of fan. As for another person, whose hand was upon its leg, said, the elephant is a pillar like a tree-trunk. The blind man who placed his hand upon its side said the elephant, "is a wall". Another who felt its tail, described it as a rope. The last felt its tusk, stating the elephant is that which is hard, smooth and like a spear.[10]

The Global Data Plane means different things to different stake holders, depending on what aspect of the system they are interested in and how they use it. But primarily, it allows decoupling of concerns. Let's see how.

- For **infrastructure owners**, the GDP enables *decoupling* of data security from service availability; the GDP provides infrastructure owners with a way to offer their services to other users without necessarily having to worry about data security on their own. While standard operational procedures should be followed for maintaining system security, providing GDP service enables infrastructure owners to convince their users that an infrastructure breach does not compromise users' data.
- For **application developers**, the GDP is a useful middleware that enables *decoupling* of information from infrastructure by providing a secure information centric abstraction. The GDP enables application developers to simplify design of complex applications by working at the abstraction of uniquely named streams of information. More importantly, this simplification does not come at the cost of security.
- For **end users**, the GDP provides a way to have better control over their data for privacy and performance by *decoupling* deployment/execution from application development. While application developers design applications, in many cases, they are deployed and executed

[10] A famous Indian fable. Illustration from 'Martha Adelaide Holton & Charles Madison Curry, Holton-Curry readers, Rand McNally & Co. (Chicago), p. 108'. Text from https://en.wikipedia.org/wiki/Blind_men_and_an_elephant.

by end users, e.g. IoT sensors. By designing applications in a way that GDP names can be provided as runtime parameters, the GDP enables enough controls for end users to get the desired security and performance from the system by selecting which parts of the global system ought to be used for servicing specific data, presumably by selecting appropriate service providers.

The Global Data Plane aspires to solve interesting research challenges and provide rich opportunities in various domains. Let's see how.

- For **distributed systems' researchers**, the GDP demonstrates how to make a distributed system work in a federated ecosystem where infrastructure is owned and operated by a number of real world entities that do not necessarily trust each other. Many existing big-data systems are designed to work within a single administrative domain—most often limited to a single data-center—and it is not straightforward to handle situations where either resources, or data, or both are owned by multiple stake holders.
- For **security researchers**, the GDP demonstrates how to design a secure platform with minimal trust among participants, where the only rationale for various parties to coordinate is an explicit economic contract. Further, it shows how to unify data security at-rest with data security in-transit—something that's especially important to handle dynamically evolving information streams. We take a less than idealistic view of the world from a security perspective; while many problems we work on are considered *solved* security problems, they still are commonplace in the real world. The GDP explores how to mitigate these problems with a better system architecture.
- For **application researchers**, the GDP provides opportunities to refactor their applications around trust in data (and not physical hosts) by using a verifiable information stream. Further, it gives them opportunities for designing new applications that provide privacy by design. The GDP provides a platform that enables development of applications that combine the best of both worlds—using the resources close-by for achieving low latency and using the cloud for the practically infinite pool of resources. The security guarantees provided by the GDP enable new applications.

Chapter 2

DataCapsules and the Global Data Plane

In the previous chapter, we presented a motivation for DataCapsules and a system like the GDP. In this chapter, we first provide a motivating application example that guided the initial design of the GDP and DataCapsules. We then describe the threat model that the overall system (the GDP and DataCapsules) address. Based on the motivating example and the threat model, we then provide a gentle introduction to the proposed architecture—what is the user interface, what it means for an application developer, what is the role of system administrators and service providers, etc. In addition, we formally define various concepts that we will be using for the rest of this dissertation.

This chapter should give a sufficient overview of the system for a reader without necessarily going into details of internal mechanisms. We discuss internal mechanisms in later chapters. Also note that this chapter merely describes *how* the overall system looks like to users, application developers, and administrators. We discuss the in-depth rationale for our design choices for DataCapsules and the GDP network in next few chapters.

2.1 The GDP and DataCapsule for Internet of Things (IoT)

In this section, we describe an application model for the Internet of Things (IoT)—one of the motivating applications that guided our initial design of the interface that the GDP and DataCapsules provide to users. Later in this chapter (section 2.5), we describe our experiences with real-world deployments and actual users for DataCapsule-based IoT applications.

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. Such use cases benefit from the availability of a secure ubiquitous storage platform like the GDP, and can use the single-writer append-only DataCapsules directly (also see Appendix A).

Recall from the previous chapter that a DataCapsule is primarily a single-writer append-only data structure. For readers, DataCapsules support subscribing to new information, secure replays at a later time (thus providing a time-shift property), and efficient random reads for old information.

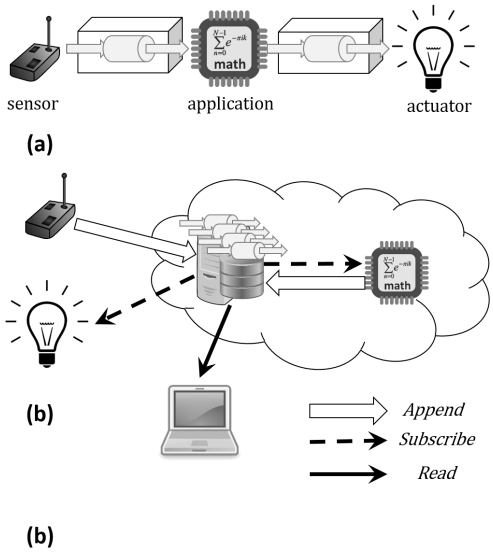


Figure 2.1: (a) A logical view of the single-writer DataCapsule when used in the context of Internet of Things (IoT). The DataCapsule provides an analogy of a pipe with an attached bucket. A sensor appends data to a sensor-DataCapsule which is consumed by an application, which processes the sensor data and writes it to an actuation-DataCapsule. An application developer does not need to interact with physical devices; instead the DataCapsule provides a virtual device with associated history. (b) The actual physical view of the DataCapsule, where a log server physically stores data and multiple readers can subscribe to the same DataCapsule.

Keeping the single-writer append-only model of a DataCapsule in mind, sensors are assigned their own DataCapsules at the time of sensor provisioning. Each sensor appends the data it generates to the associated DataCapsule, thus making a DataCapsule the only interface to the rest of the world for the sensor. Similarly, each actuator is subscribed to an actuation DataCapsule from which it gets the actuation commands. The designated single writer of the actuator DataCapsule is either a pre-configured application, or a service representing the actuator (See Figure 2.1).

2.1.1 Benefits of using DataCapsules

A DataCapsule 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 DataCapsule by such sensors, the DataCapsule can be used as a proxy that supports a much richer set of queries, especially for historical data. A subscription to such a DataCapsule 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 DataCapsule to read the actuation commands as we described, access control could be implemented at the DataCapsule level. This makes actuator design simpler and avoids the pitfalls

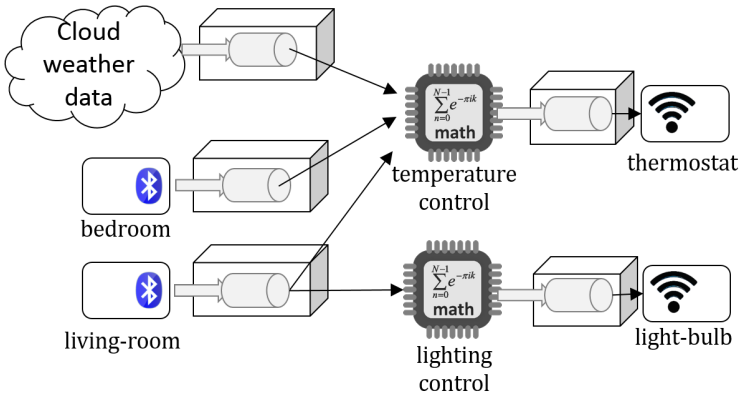


Figure 2.2: An example building-automation solution, where physical sensors and actuators are represented by DataCapsules to applications. Each application can have multiple input/output DataCapsules, including those representing external data sources. Each sensor DataCapsule can be subscribed by multiple applications as well.

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 DataCapsule (in case of a sensor), or subscribe to a DataCapsule (in case of an actuator).[1]

Representing sensors and actuators with DataCapsules separates policy decisions from mechanisms, enabling cleaner application designs. Applications can be built by interconnecting globally addressable DataCapsules, rather than by addressing devices or services via IP addresses. Further, with applications running inside containers (Docker [31], [32], Intel SGX enclaves [33], and so on), forcing data-flows in and out of the container through DataCapsules enables any filtering at the DataCapsule level (for example, access control). Last but not least, the narrow waist provided by globally addressable DataCapsules avoids stove-piped solutions and provides for a heterogeneous hardware infrastructure.

2.1.2 An example IoT application

Let's see a sample IoT application—a smart building—and see how DataCapsules enable a cleaner design compared to traditional custom ad-hoc solutions. 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

[1] Of course, the devices do need to include some functionality for interacting with DataCapsules, but this is rather small common piece of code across a wide variety of devices.

understand various patterns and the *typical* state of the building in a given season. We deployed a number of sensors in a machine room that write data to their individual DataCapsules; 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.

Such an application really illustrates the power of composing sensors and actuators. A conventional application design would use a pub-sub mechanism to make the data from one sensor available to multiple subscribers. Further, the data from sensors must be stored somewhere for analysis later. To accurately perform any modeling of building's behavior, it is also important to record the state of actuators and any actuation decisions. Using custom ad-hoc pub-sub and storage leaves questions of data integrity—both in-transit as well as at-rest—unanswered. Further, any bad data injected into an ad-hoc design is almost impossible to detect later.

A typical cloud based pub-sub solution, as proposed by for example AWS IoT [27], 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. On the other hand, a DataCapsule based design enables a rather straightforward design with a comprehensive data security model while ensuring that the building administrators also *own* their data (see Figure 2.2). Using DataCapsules enables the application to not worry about integrity of data or access-control issues at the device level. In addition, with appropriate credentials for reading, any application can perform analysis of, say energy consumption in a building, at a later time. In general, any application that can be represented as a static graph of interconnected components can be directly translated to DataCapsule.

2.2 Threat model: Decoupling security from availability

In this section, we describe the general threat model that the GDP together with DataCapsules attempt to address. This is only a broad overview of threat model for the entire system (the GDP network and DataCapsules), and we discuss the threat model that individual components are designed to protect against in later chapters.

Throughout the dissertation, we adopt typical assumptions for security of cryptographic constructions (secure hash functions, digital signatures, symmetric and asymmetric encryption), and that an adversary doesn't have infinite computation power to launch brute-force attacks against such constructions. Another assumption is that various entities protect their secret keys. The GDP and DataCapsules do not provide sufficient protection in case of compromised keys of users, administrators, or other infrastructure components.

Security goals, i.e. what does 'data security' mean? The key goal of the GDP is to keep data secure in presence of untrusted infrastructure. Our data security goals are at least three fold: (1) confidentiality, i.e. no information should be leaked to anyone other the authorized producers and consumers of information; (2) integrity, i.e. nobody other than the authorized producer/consumer of the information should be able to alter the information in any way without being detected; and (3) provenance, i.e. every bit of information can be traced back to the producer of the informa-

tion.[2] DataCapsules directly enable integrity and provenance, and enable applications to achieve confidentiality by providing support for encryption.

A Trust in the writer and the owner of DataCapsule

Readers of a DataCapsule trust the designated single writer of the DataCapsule. If the single writer intentionally or accidentally corrupts the state of the DataCapsule (either the contents of DataCapsule itself, or the ordering of the updates), then readers don't have a remedy for such corruption. As we will describe in section 2.4, the single writer is also responsible for keeping some information locally about the state of the DataCapsule. If the writer violates these assumptions, then the readers still get a verifiable assurance from the infrastructure that the writer is the culprit to be blamed for such behavior.

Somewhat similarly, the owner of a DataCapsule is trusted to make appropriate policy decisions. These policy decisions include, but are not limited to, choosing trustworthy service providers that will provide the desired quality of service, and setting policy decisions for whether a DataCapsule must be visible only within a subset of the infrastructure, or visible globally, or something in-between; etc.[3]

Finally, readers are expected not to leak information to unauthorized parties. However, malicious readers can only violate the confidentiality property but not the integrity or provenance.

B Distrust in the infrastructure

Broadly, the infrastructure is not trusted for data security. Even if the infrastructure operators start actively attacking their own users, data security is not compromised. However, the infrastructure *is* trusted for making the negotiated service available; such negotiations and their enforcement is performed out-of-band via economic and legal frameworks. In case the infrastructure operators act maliciously, or in case of an adversarial infiltration, the worst that can happen is a service outage.

Security of stored information: A log server that stores a DataCapsule replica may engage in malicious behavior and may accept bogus writes from unauthorized writers; 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. Regardless of malicious behavior from log servers, data security is not compromised.[4]

[2] As an example in real world: Alice sends a text message to Bob. 'Confidentiality' means that nobody other than Alice and Bob know the contents of text message; 'integrity' means that Bob can assert that the contents of the text message hadn't been changed by an intermediary; and 'provenance' means that Bob can verify that the text message in fact was generated by Alice and not an adversary Eve pretending to be Alice.

[3] It is certainly possible for the policies to be too restrictive that result in unavailability of services. A real world example of 'too restrictive' policy would be that in a small home network, the only available ISP is not to be trusted for routing information, thus resulting in the information being locked inside the home network.

[4] Returning bad data on read requests, however, amounts to the service provider not providing *the promised service*.

Security of information in transit: The underlying GDP network provides a datagram transportation service, but it does not provide generic transport-level security of information. Users must connect only to trusted service providers; malicious service providers can do practically anything with the datagrams that are either generated by or are intended for their users. The very limited guarantees for routing security that the GDP network provides is that an adversary cannot influence the communication path between an arbitrary pair of GDP endpoints.[5] However, when information transits through the GDP network wrapped inside a DataCapsule (or a subset of DataCapsule), the additional DataCapsule-level protections still apply to ensure that the information in the DataCapsule is secure.

Isolation from other users: We assume that service providers enforce appropriate isolation for shared resources; one user should not be able to infer any information about any other user that is being served by the same log server, for example. However, if such isolation is not maintained, a malicious user cannot compromise data security beyond the information leaked through side-channels.

Service availability and data freshness: 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 via *heartbeats*. If an individual node (log server, router, etc.) is behaving maliciously but there are alternate nodes that are available and reachable, the GDP architecture allows for a client to retry and get the desired service by connecting through a different part of the infrastructure.[6] 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 entity goes rogue.

Scope of protection from side channels: An important exception to security guarantees from the infrastructure is that even though 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 or size of messages.[7] 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 data integrity/confidentiality and a client can hold a service provider accountable for not meeting the desired performance requirements.

[5] In other words, the GDP network does not protect against on-path adversaries, and it merely ensures some protection against off-path adversaries. On-path adversaries can analyze traffic or tamper with it at will. However, off-path adversaries cannot corrupt the state of the network and insert themselves in the communication path of an arbitrary pair of GDP endpoints (which could further lead to man-in-the-middle attacks, for example).

[6] 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.

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

C Untrusted third parties

Broadly, third parties may attempt to do practically anything to compromise the security guarantees that the GDP and DataCapsules provide. As such, the GDP and DataCapsules must provide a strong defense against such adversaries.

The federated nature of the GDP allows anyone to join the system and start operating their own routing domain or storage organization, as long as they provision appropriate resources. This makes it easy for malicious entities to appear and disappear at will and acquire a different name each time they come up. Thus, third party adversaries include not just individual rogue attackers but also infrastructure owners that don't have anything to do with hosting or servicing/transiting requests in any way for a given DataCapsule. Further, the utility provider model means that malicious actors may create short term economic contracts with specific service providers with the sole purpose of attacking benign users served by the service provider. Third parties may also 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. The GDP provides protection from all such attempts by third party adversaries.

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). 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, the GDP network architecture ensures that information can be routed only via explicitly trusted paths, and thus it is not possible for an adversary to inject itself at will in the communication path between an arbitrary pair of endpoints.[8] We elaborate this in more detail when we discuss the threat model specific to the GDP network.

However, if a DataCapsule owner decides to make the DataCapsule publicly available, third parties *can* in fact examine encrypted information and potentially get some side-channel information on the mere existence of information, as well as size and timing of updates to the information—the GDP and DataCapsules do not provide any defenses for such public DataCapsules.

2.3 The Global Data Plane: An overview of infrastructure

In the GDP, the key organizing principle for infrastructure is that of resource ownership. Just like in real world, using ownership as the organizing principle in the GDP enables a very clean way of identifying stake holders.[9] Based on resource ownership, the infrastructure is divided into

[8] If an adversary were to be able to inject itself in the communication path of an arbitrary pair of GDP endpoints, such adversary can: (1) observe all traffic and perform side-channel attacks based on the size and timing of information, (2) tamper with traffic (including reordering and replaying of information), or (3) drop all traffic altogether. In a system like the GDP, even something as naive as (1) can lead to an adversary being able to, for example, enumerate all clients that interact with a DataCapsule, leading to a privacy disaster.

[9] Subtle issues, for example privacy, are extremely dependent on policy which is hard to codify in a computer-readable form. Instead, we tackle a different problem: codify the resource ownership in a computer-readable form, and make users choose specific resource owners that they want to interact with based on out-of-band negotiations.

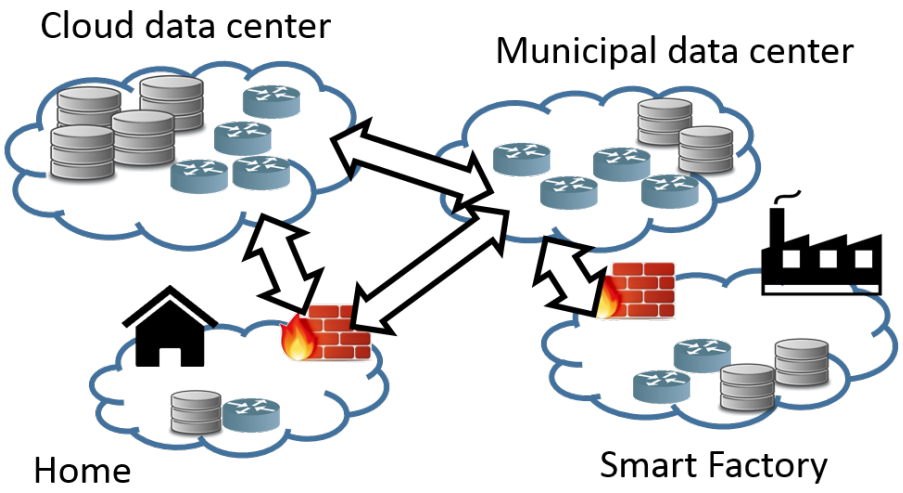


Figure 2.3: An organizational view of the GDP.

organizations; these organizations are somewhat analogous to administrative domains as used in a number of other systems and real-world organizations. Organizations in the GDP can decide their internal policies in the way they want, but they still interact with other organizations by adhering to a well defined interaction framework; in this way, these organizations also bear a resemblance to Autonomous Systems (ASes) in the Internet. In the real world, these GDP organizations can be operated by governments, municipalities, for-profit/non-profit organizations (both large and small), and even individuals (see Figure 2.3 and Figure 2.4).

Just like DataCapsules, such organizations are identified by a unique flat 256-bit name. In fact, all addressable entities in the GDP have a cryptographically derived flat 256-bit name. We call these names as *GDP names*.[10] In the GDP, anyone who can provision resources can start an organization; this makes the GDP an open federated system for anyone to join. All that a new infrastructure operator needs to do other than provisioning resources is to create a GDP name for the organization. To create a GDP name, the infrastructure operator first creates a cryptographically signed list of key-value pairs called *metadata*. The GDP name is then 'calculated' as the cryptographic hash of this metadata.[11] This metadata describes immutable information about the corresponding entity. A required item in this key-value list is the public signature key of the principal, the private part of which is used to sign the metadata and kept safe by the infrastructure operator.

The ownership relation between an organization and the resources it owns (which themselves have their own unique 256-bit names) is specified cryptographically in the form of $OwnCerts$. An $OwnCert$ is a type of *delegation* granted by a domain administrator to a resource (e.g. a physical host), and allows the given resource to claim that it is operating on behalf of a specific organization.

[10]Examples of such addressable entities are: DataCapsules, readers, writers, log servers that provide the physical storage for DataCapsules, GDP routers that route GDP traffic between various entities, and even organizations.

[11]Unless otherwise specified, 'hash' refers to a SHA256 hash function throughout the dissertation.

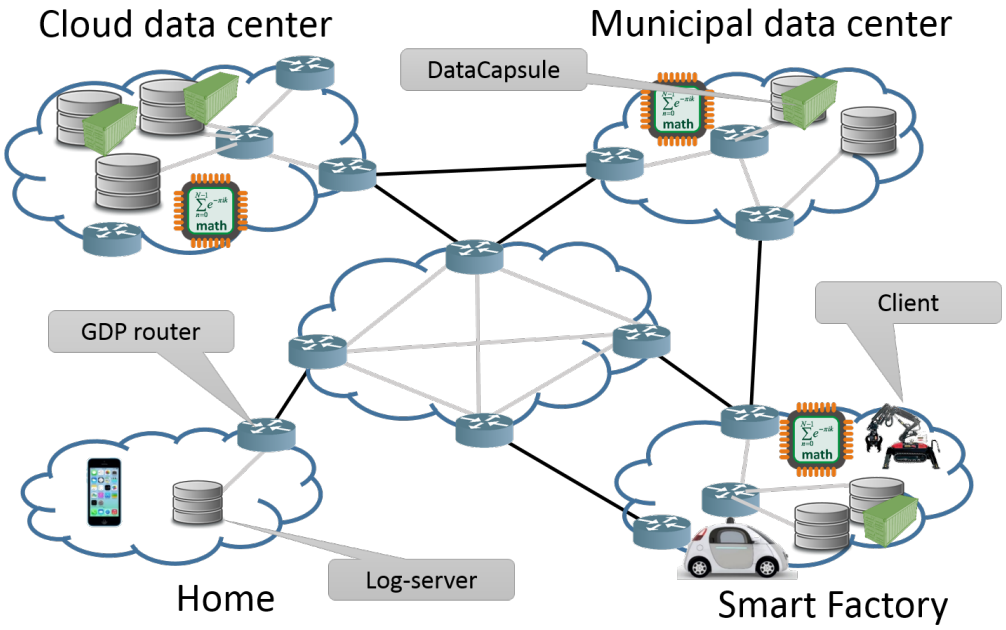


Figure 2.4: A more elaborate organizational and physical view of the system.

Broadly, *delegations* are cryptographically signed statement linking two (or more) GDP names and represent a specific type of responsibility or authorization granted by an issuer to one or more issuees; such issuers and issuees are represented by their GDP names.[12] The issuer uses the private key associated with its GDP name to sign such statement, thus making the GDP names a trust anchor that anyone can use to verify such delegations. Delegations are managed and stored by the recipient (issuee), and presented to others whenever needed.[13] The flat cryptographic names serve as a trust anchor to verify relationships—encoded in the form of delegations—between various physical entities without relying on trusted third parties such as traditional hierarchical certificate authorities.

An organization can provision resources either for its private use, or make them available to others—enabling a service provider model.[14] The service provider model of the GDP assumes that users enter into economic agreements with infrastructure operators to obtain services. Thus, these organizations enable what we loosely call as a 'trust domain'. Contrary to what the name might

[12]We use four different types of delegations in this dissertation: *OwnCert*, *JoinCert*, *AdCert*, and *RtCert*. We will describe them in detail as we go along.

[13]A delegation is like a passport, where the issuer asserts that it has vetted the issuee in some way. Because delegations are cryptographically signed, a malicious issuee cannot forge a bogus delegation. A malicious issuer can certainly issue bogus delegations, but because it is the issuee that must present a delegation to authorize itself in a given context, honest issuees can simply ignore bogus delegations.

[14]Note that a service provider model is a generalization where there are at least two distinct roles: the owner and the user. With such a model, reasoning about resources deployed for private use essentially means that the role of owner and user are played by the same entity.

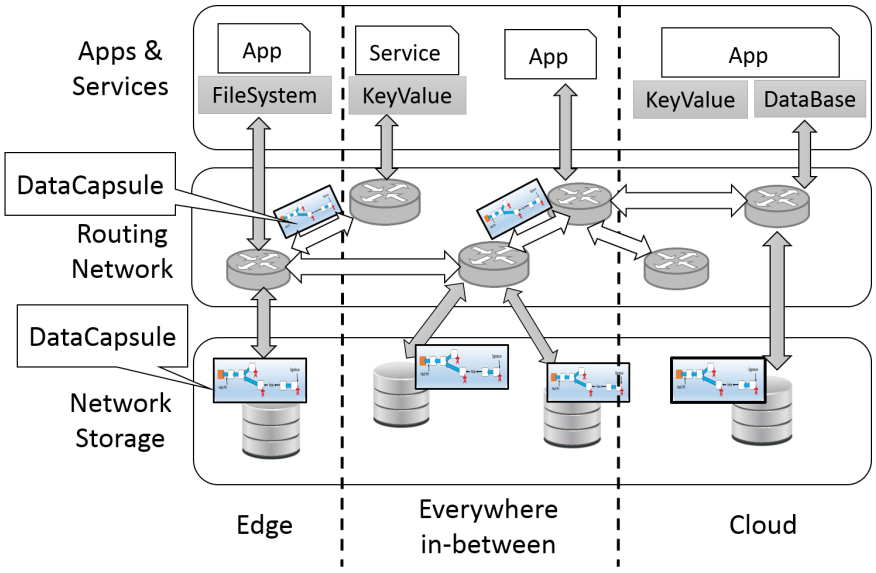


Figure 2.5: A logical view of the system with the GDP network at the core.

suggest, a trust domain is *trusted* only for maintaining infrastructure and providing some service; trust domains are not blindly trusted for the actual security of information. We use the terms 'trust domain' and 'organization' interchangeably.[15]

Individual organizations could specialize in providing a specific type of service. There are two important types that we will describe shortly: 'storage organizations' that specialize in providing storage services and operating *log servers*, and 'routing domains' that specialize in providing communication services and operates *GDP routers*.

A Routing domains: Providing communication as a service

Routing domains are organizations that specialize in providing message transport as a service; they are somewhat equivalent to Internet Service Providers (ISPs) in the current Internet. Collectively, all the routing domains constitute the message routing fabric called the GDP network that is at the core of the GDP. The GDP network enables message delivery between various entities by using their location independent flat 256-bit GDP names (see Figure 2.5). At a very high level, the GDP network provides an interface similar to a datagram oriented communication much like UDP, except that datagrams handled by the GDP network use GDP names instead of IP addresses and provide a number of security guarantees that traditional IP networks lack.

The GDP network is influenced by the traditional IP infrastructure in many ways. The GDP network can be partitioned into routing domains in the same way as the IP network can be

[15]To be specific: organizations are ownership domains that *enable* a trust domain. In this context, a trust domain is a set of infrastructure trusted for service availability in exchange for money, and is enabled by an organization (or a number of organizations cooperating with each other) acting as a service provider.

partitioned into Autonomous Systems (ASes). At a physical level, the GDP network is essentially an interconnect of GDP routers—an equivalent of typical IP router—owned and operated by various routing domains. Similar to ASes, routing domains maintain border GDP routers and internal GDP routers (see Figure 2.4). Routing domains not only provision GDP routers, but also provide supporting infrastructure and ensure connectivity with other routing domains. The simplest routing domains may be stand-alone independent domains that rely exclusively on overlay routing by creating direct TCP connections to other routing domains when they need to reach remote resources. More complex and extensive routing domains may enter into peering agreements similar to IP routing.[16]

To join the GDP network, a user finds a GDP router belonging to a routing domain of choice and securely advertises one or more GDP names into the GDP network.[17,18] Users can indicate policies on *scope* of their names—whether such names be made available globally or restricted to subset of the GDP network, etc. GDP routers then propagate the advertised GDP names in the GDP network appropriately. On behalf of users, GDP routers deliver datagrams to appropriate destinations by using overlay routing techniques to find the most optimum path between a given pair of addresses.

The GDP network differs from the IP routing in a significant way: as opposed to IP networks where typically the network temporarily *leases* addresses to participants, GDP names are created and permanently owned by users who created them. Such name ownership is defined by possession of private key corresponding to the public signature key included in the metadata used to construct the GDP name in question. A user can advertise its own name in the GDP network together with names on behalf of others; however, it must cryptographically prove to the GDP network that it either owns the name or that it has a *delegation* from the owner of the name to advertise such a name. This ensures that malicious adversaries cannot claim to possess arbitrary names and launch routing attacks by corrupting the routing state of the network; we call this as the *delegated secure flat routing* scheme and describe it in detail in later parts of the dissertation.

Routing domains that provide services to end users are analogous to Internet service providers, except that they provide routing between GDP names and not IP addresses. Similarly, there may be routing domains that act as transit providers and enable connectivity between other routing domains. Depending on the ownership of a routing domain and its intended purpose, it could either be open for anyone to join or be restricted to specific participants. Private domains that only allow a limited set of participants are a crucial feature to enable control on resources; such private domains require that anyone attempting to join a routing domain present a *delegation* by

[16]In this dissertation, we primarily demonstrate our mechanisms using the former type of domains, since this is more favorable to an overlay network and directly applicable to a GDP network implementation in the existing Internet. We then briefly describe how to extend such model to the latter type of routing domains that demand native peering.

[17]Here, a user is any active process that is interested in initiating or receiving communication. A DataCapsule is *not* a user because it is not an active process. Instead, a log server that hosts a DataCapsule is the user that advertises the GDP name for the DataCapsule along with its own name.

[18]Usually the choice of routing domain is directed by trust, economic relationship, or other administrative factors.

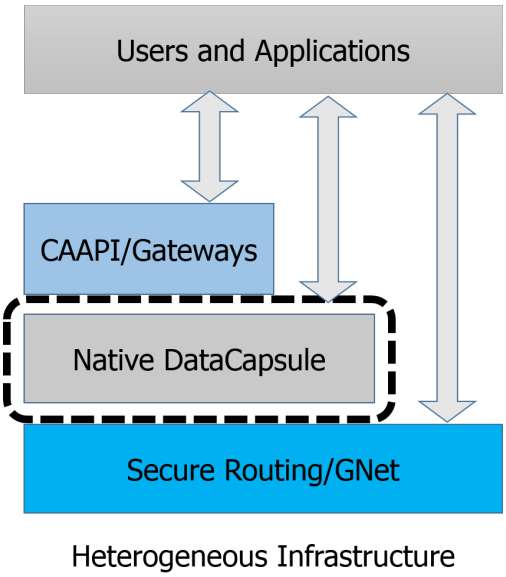


Figure 2.6: User interface to the GDP and DataCapsules. Users can use the native DataCapsule interface directly (see subsection 2.4.1), or they can use higher level abstractions built on top of DataCapsules (see subsection 2.4.2), or if need be they can directly tap into the underlying GDP network (see subsection 2.4.3).

the domain administrator. This kind of a delegation is essentially a cryptographic authorization to join the given domain, and is only granted to valid clients.

B Storage organizations: Providing persistence as a service

Storage organizations are broadly tasked with persistent storage for DataCapsules. They accept and validate write requests, make information durable, and service read requests from authorized readers. To enable such persistent storage, storage organizations provision and operate *log servers*.

A log server stores a DataCapsule replica. It advertises for the DataCapsules it hosts in the GDP network, and serves write and read requests from appropriate writers/readers. It also communicates with other log servers to ensure that it is in sync with other DataCapsule replicas (which may be stored by log servers belonging to other storage organizations). A single log server may host a replica for many many DataCapsules depending on the availability of local resources. Log servers are free to choose the on-disk data representation for DataCapsules as they seems fit. Such choices are implementation specific. For an arbitrarily large DataCapsule, a log server may not be able to hold the entire contents locally. However, given that a DataCapsule is an append-only data structure, the log server may offload older parts of DataCapsules to remote machines to enhance scalability. Such scalability concerns are internal to a storage organization's operations, and we don't discuss such details in this dissertation.

The task of infrastructure operated by storage organizations is to make information durable, and not police the order of updates. As long as an update meets the admission control criteria for a DataCapsule (typically a valid signature as we will describe shortly), the infrastructure may not permanently reject the update.[19] Doing so would be a violation of the agreement between a DataCapsule owner and the service provider. A service provider must ensure that such violations are minimized by provisioning sufficient capacity, bug-free software, etc.

DataCapsule owners explicitly delegate DataCapsule storage to one or more storage organization by using delegations called $AdCerts$. Such explicit delegation has two implications: (1) it allows a user to make economic contracts with service providers and/or hold them accountable if the agreed upon requirements of performance, locality, and durability are not met; and (2) it allows log servers operating on behalf of storage organizations to prove to the GDP network that they are in fact authorized to advertise names for the given DataCapsule.

2.4 Applications' view of DataCapsules and the GDP

Recall from chapter 1 that DataCapsules provide stable state to applications. Applications developers must only concern themselves with the DataCapsule API, whereas infrastructure operators (administrators) are responsible for providing service. In this section, we first describe the interface for direct interaction with DataCapsules. Then, we describe higher level services and interfaces that go beyond the native DataCapsule semantics, as well as the interface provided by the underlying GDP network for applications (Figure 2.6).

2.4.1 Native DataCapsule user interface

A DataCapsule is a cohesive information container identified by a location independent flat 256-bit GDP name. At a data-structure level, a DataCapsule is an ordered list of immutable records; a record is a variable sized read or write to the DataCapsule.

A DataCapsule is a *single writer* data structure that supports 'append' as the only mode of writing. An 'append' operation adds new records to the DataCapsule. The single writer is responsible for both the contents and the ordering of updates in a DataCapsule. There can be multiple readers that can perform random reads for old data, or they can 'subscribe' to updates. 'Read' fetches existing records by addressing them individually or by a range, while 'subscribe' allows a client to request future records as they arrive.

The readers, subscribers and (single) writer of a DataCapsule are collectively referred to as clients. Clients do not need to worry about placement or other policy decisions for DataCapsules;

[19] In other words, properly formed update requests may only fail with a 'Service Unavailable' or similar status indicating a transient failure.

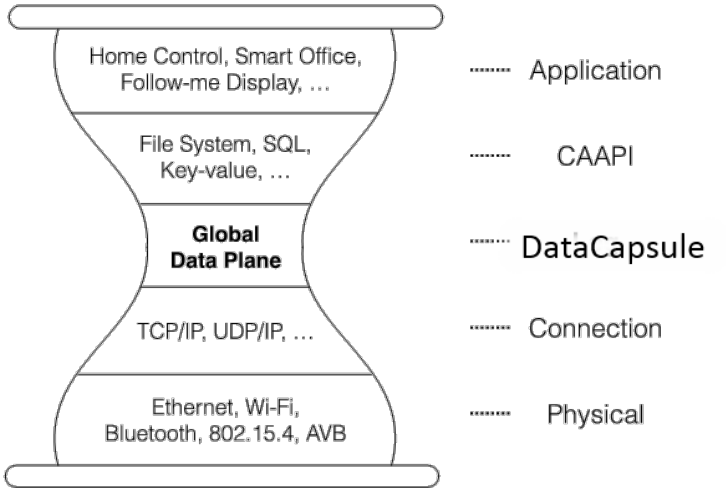


Figure 2.7: For applications, a DataCapsule provides a narrow waist interface of secure information that is sufficient to build higher level services on top.

they direct their requests to the DataCapsule and not to the log server hosting DataCapsules.[20,21] A DataCapsule may be replicated widely; the GDP network delivers the requests to an appropriate replica of the DataCapsule based on locality, quality of service requirements, or any other policy-based constraints imposed at the network level.

A DataCapsule creation

DataCapsule creation involves creating the DataCapsule metadata and delegating the hosting of the DataCapsule to service providers.[22] At the very least, the metadata contains the public part of the signature key belonging to the designated single writer of the DataCapsule. In addition, the metadata may contain policy specifications for desired performance, durability, sources for read access control, etc. This metadata can be considered a special record at the beginning of the DataCapsule. The name of a DataCapsule is derived from the hash of this metadata.

The metadata is created by the 'creator'. The 'creator', 'administrator', or 'owner' of a DataCapsule is tasked with the creation, provisioning, and policy specification for the DataCapsule. 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 subscribe operations by the readers. The creator is also responsible for the life-cycle management of the DataCapsule

[20]This is done with the help of GDP library that takes care of translating higher level function calls into network operations. The library also takes care of creating a names for the client, advertising it onto the network, and translating operations on a DataCapsule into messages on the network. We describe the details later when we discuss mechanisms.

[21]Because users interact directly with the information (and not the host), the GDP network fits the definition of an Information-Centric Network [34]. We will discuss this in detail later.

[22]This is the same metadata that we described earlier in this chapter, and is a list of key-value pairs identifying properties of a DataCapsule.

as well as any potential economic relationship with the service provider(s). For higher availability, the creator can use a redundancy model by recruiting service from a number of service providers all at the same time that hold replicas of a DataCapsule.

To answer who should handle the creator/administrator role (i.e. who should *own* a DataCapsule in a given application), developers have a few choices based on who *owns* the data: it can be either left to end users of the application (e.g. a user provisions a simple home IoT device and creates a DataCapsule for the said IoT device), or handled by the developers in the case of *fully managed* applications (e.g. an IoT device that comes pre-configured with a unique DataCapsule whose hosting and maintenance is included as part of a subscription service), or can be handled by a completely separate entity.

B DataCapsule writes, reads, and subscription

The single writer possesses a private signature key specific to the DataCapsule; the public part of this key is what is included in the DataCapsule metadata at the time of creation.[23] The writer also possesses appropriate symmetric encryption keys used to encrypt the data. The writer is responsible for keeping these secret keys secure. The single writer is also responsible for making the decisions of what data should go in the DataCapsule and in what order. As part of this responsibility, the writer must maintain 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 smallest unit of read or write to a DataCapsule is called a *record*. A record is user-supplied encrypted information together with some meta information needed to uniquely place the record in the DataCapsule with respect to other records. Each record has two key properties that uniquely identify the record in a DataCapsule: a hash value and a monotonically increasing integer called `seqno`.

For an append to the DataCapsule: the single writer encrypts the data, creates records with appropriate meta information, creates some signatures using its own private key, puts the newly created information inside an append request, and sends this append request to the DataCapsule. The append request is handled by the log servers delegated to serve the specific DataCapsule. The log servers validate the request, update the local replica of DataCapsule if validation succeeds, and send an acknowledgment back to the writer.

Note that from an interface perspective, a record is created and committed for eternity as part of the abstract DataCapsule as soon as the writer signs the records (even before it is sent out on the network); whether the record has been delivered to a replica is a separate question. Individual replicas may be out of sync if they haven't received all the records that are part of the abstract DataCapsule, but since the admission control is done at the writer level, all the replicas of a DataCapsule will eventually be in sync.[24]

[23] We assume that the DataCapsule owner has secure access to the public key of the single writer. For the simplest applications, the same entity could play the role of both the administrator and designated single writer; in such a case, the public key of the writer is readily available to the owner.

[24] We discuss more details related to replication and consistency semantics in the next section.

The append operation created by the writer also includes a 'heartbeat'—this is a small piece of data that contains a signature and other appropriate ordering information that can be disseminated widely in the network if needed. This heartbeat uniquely identifies a particular state of the DataCapsule.[25] These heartbeats are stored by the log server along with the DataCapsule, or can also be managed by a separate set of more reliable/trustworthy *heartbeat-servers* if need be.

Readers can request records either by referring to the corresponding hash value that they can extract out of a heartbeat, or by referring to the unique `seqno`; they can verify the result of read queries by asking for a proof from the log server against a given heartbeat.[26] Subscribers, on the other hand, are forwarded the append requests by the log servers as they receive it; they can validate the append requests in exactly the same way as a log server does and get the information as it is generated.

DataCapsule creators/administrators may also specify policies on the *scope* of data, i.e. whether the given DataCapsule should be restricted to some subset of the infrastructure, made available globally, or something in-between. Trust domains are trusted to enforce such scope restrictions and make the data available only to authorized readers, but encryption is the final line of defense even if such trust domains do not perform their tasks correctly. Only the authorized readers have the correct decryption key to make sense of data. Thus, even if encrypted data is made available to an unauthorized entity (for instance, in the case of an infrastructure breach), the loss of confidentiality is limited. Clients use digital signatures and encryption as the fundamental tools to secure their data rather than trusting the infrastructure.

C DataCapsule replication and consistency semantics

A DataCapsule can be replicated or distributed over a number of physical machines, and there are various operations such as migration, replication, etc. that log servers can perform on DataCapsules in the context of an ecosystem like the GDP. Using multiple log servers (or storage organizations) allows for fault-tolerance, durability and scalability.

The single-writer and append-only nature of the DataCapsule 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.[27] While it may seem an extremely limiting interface, it is sufficient to address a wide variety of use cases and rich applications. In later chapters in the dissertation, we will relax this requirement slightly later on for applications that can work with weaker ordering of updates.

[25] Note that a 'heartbeat' is different than a *keep-alive* used in various protocols. Instead, a 'heartbeat' is more like a version identifier similar to commit hashes in `git`.

[26] Asking for records based on the hash value results in an implicit proof, as we will describe in next section.

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

Since a DataCapsule with replicas is a distributed data structure, it is important to consider the consistency guarantees. In the most general case, DataCapsules can be exploited to provide sequential consistency guarantees to readers. As we will discuss in more detail in chapter 3, the DataCapsule design allows readers to detect out of order or missing records on client side, but they may still get stale data which can happen when service providers do not fulfill their obligations to provide service—such violations of service level agreements can be enforced out-of-band. For readers with stricter consistency requirements, they *can* achieve a strong consistency without any changes to the infrastructure or the writer, but such readers are at the risk of stalling in case of a certain class of infrastructure failures as we will describe in a later chapter. It is important to note it is the same underlying infrastructure and the common DataCapsule interface that can support these varying levels of consistency guarantees depending on how readers and writer interact.

2.4.2 Beyond a DataCapsule interface: Higher level storage abstractions

The native DataCapsule interface shelters developers from low-level machine and communication details by providing the location independent secure storage/communication interface. Simple applications can directly use the native DataCapsule interface and get the benefits of security and simplicity. However, many applications are likely to need more common APIs or data structures than an append-only single-writer DataCapsule interface provides. In order to support a wide variety of use cases, we next describe other ways for applications to use a DataCapsule.

A Common Access APIs: CAAPIs

We argue that an append-only DataCapsules interface is sufficient to implement any convenient, mutable data storage repository. In fact, append-only logs have been used to implement file systems [35]. Various databases also keep their transactions internally as append-only logs to enable rollback and recovery. In the GDP, we call such a layering of a more familiar interface on top of a DataCapsule as Common Access API (CAAPI).

CAAPIs are distributed as libraries that users can include in their applications; such a library translates the familiar interface into operations on a DataCapsule. In addition to translation of API calls, a CAPPI can perform application specific client-side caching and other optimizations. Because a DataCapsule serves as the ground truth, the benefit of integrity, confidentiality, and access control are carried over to such interfaces for free.

We believe that such CAAPIs are crucial to a widespread adoption of DataCapsules and the GDP. As a proof of the CAAPI concept, we have built a key-value store with history, a filesystem based on FUSE[28], and a TensorFlow [36] file system plugin on top of the single-writer DataCapsules.

B Multi-writer DataCapsules

While the basic DataCapsule interface we consider assumes a single-writer model, a number of applications scenarios greatly benefit from a multi-writer interface. For example, consider a number

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

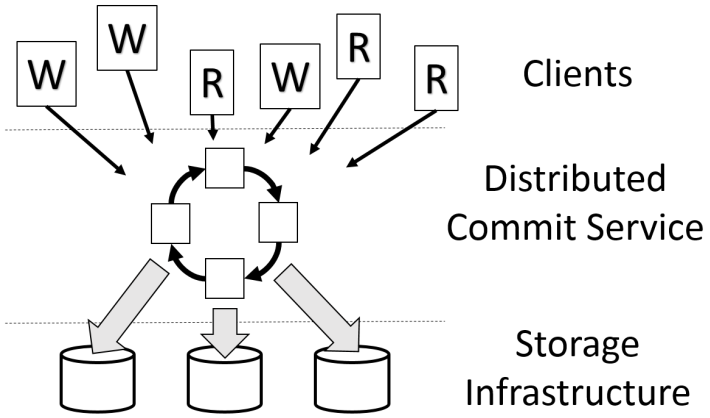


Figure 2.8: A distributed commit service that accepts writes from multiple writers and serves as the single writer for a DataCapsule. The service is responsible for ordering the updates received from multiple writers based on some application specific logic.

of temperature sensors deployment in a single room, where a user may be interested in seeing a unified view of temperature readings from all the sensors instead of dealing with individual sensors. Another example could be a database which accepts not just reads but also writes from multiple writers.

A multi-writer interface begs the fundamental question: who performs the ordering of writes from multiple writers? For such patterns with multiple concurrent writers, we recommend a service that accepts writes from multiple writers, and then performs an application specific update ordering or aggregation such as a running average, a median, minimum and maximum, etc. (see Figure 2.8). Such a 'commit service' could be as simple as a process on a single physical node, or a distributed commit service running on a number of physical nodes that use some well known consensus protocol (e.g. Paxos, RAFT, etc.).

In a later chapter, we also discuss relaxing the single writer criteria slightly in the following way: instead of requiring a true single writer, an application may allow multiple writers as long as they are not concurrently writing. Any deviations from the 'no concurrent writers' model may result in weaker consistency guarantees. Further, it is essential that these multiple writers perform synchronization of state before starting to write. An example scenario where this pattern is useful is a DataCapsule that can be mounted as a personal file systems using a CAAPI on multiple devices; this situation can work with a *smart* CAAPI that uses some application-specific heuristics to ensure that only a single device is writing at a time.

C Gateways and protocol translators

Not all clients are capable of communicating using the GDP protocol natively. These limitations arise because of various reasons such as limited hardware capabilities, custom and proprietary protocols, closed software, and even software engineering effort needed in some cases. In order

to support a wide variety of use cases for DataCapsules, we propose a gateway based approach. The task of mapping the devices and operations to appropriate DataCapsules and DataCapsule-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. As another example, we created a websocket subscription gateway that enables web applications to create in-browser visualizations.

Gateways can be as simple as translating an arbitrary protocol to the 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. 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.

2.4.3 Raw interface to the GDP network: Delegated secure flat routing

While most applications should be able to just use either the native DataCapsule interface or higher level interfaces built on top of DataCapsules, the lower-level routing infrastructure provided by the GDP network is also available to applications. In fact, DataCapsules and the persistent storage provided by them can be viewed as an example application hosted by log servers and enabled by the GDP network.

The GDP network is a flat namespace routing network focused on routing security and targeted at a service provider model for edge computing and general Internet. At its core, the GDP network supports datagram oriented communication between 256-bit long cryptographically derived end-points. The endpoints represent addressable entities such as DataCapsules, hosts, service instances, etc. The security guarantees of the GDP network are twofold: (1) protection from adversaries attempting to insert themselves between a given pair of addresses by polluting the routing state of the network, and (2) enabling controls on *scope* of a GDP name, i.e. restricting whether a GDP name is visible within a routing domain, or available globally.

In order to achieve security property (1) above, the GDP network follows a slightly different model than many existing networks. Instead of network assigned addresses (as those in IP networks), users bring the names they *own* along with them and mark them active when they join the GDP network. Further, users can even delegate the names to service providers for defined durations; these service provider provide storage or computation resources and advertise these delegated names, thus enabling remote access to these resources.

Recall that the underlying infrastructure that supports the GDP network is partitioned into *routing domains* based on resource ownership. However, unlike IP-networks, the administrative boundaries in the form of routing domains are explicitly exposed to the users. This allows users to specify policies for what parts of the network should these names be visible in, and achieve the security property (2) above.

In addition to supporting unicast, the GDP network supports anycast and multicast at the network level. Native support of anycast and multicast, coupled with a service provider model, makes it easier to support multiple instances of a given service replicated widely across service providers for redundancy, scalability, and low latency access.

Note that the GDP network does not by itself provide general end-to-end transport security in the most general case. All that the GDP network guarantees is that datagrams are delivered to the correct destinations, and that arbitrary off-path adversaries cannot tamper with the delivery of such datagrams. Any end-to-end security guarantees that provide protection from on-path adversaries must be produced by applications running on top of the GDP network. In fact, we will demonstrate with the help of an example protocol, called the GDP protocol, how to do this in the case of DataCapsules.

A Why use something like the GDP network?

One of the original motivations for the GDP and DataCapsules is to use geographical diverse computing resources to support rich applications that can't rely solely on Cloud data centers. However, the heterogeneity and the geographical spread of computing resources presents numerous challenges. One such prominent challenge is how to name and address services and content. The traditional way of naming, i.e. by a URL that invariably includes a host name, does not necessarily work in edge systems that have far greater dynamism and heterogeneity than existing cloud systems, especially when replicas of the same resource are hosted by a number of different edge service providers in different physical locations. To address the naming and addressing challenge, we believe that it is crucial to separate naming and identity from location. Such separation provides higher-level abstractions for application developers and enables decoupling placement decisions from identity and addressing.

As we will discuss in more detail when we review related work, such idea is hardly new; a number of schemes have been proposed in the past two decades to use the identity of the object as its network address. A large fraction of existing work involves using flat location independent identifiers derived cryptographically. We believe that such flat names are especially suited in the context of computation that happens outside data centers, since they provide for an opportunity to use the location-independent name as a root of trust for further interaction with an object.

While much work has been done in the past, flat namespace routing networks remain prone to relatively low-cost routing attacks.[29] For instance, in a number of networks that propose *pervasive caching* of named content, a malicious node can simply lie about the possession of content with a given name and receive the traffic intended for the specific content [14], [34]. The GDP network asks a different question: instead of verifying the possession of specific content, can named objects be explicitly *delegated* to specific service providers who are tasked with hosting such objects and potentially paid for such hosting? In situations where an entity has a valid delegation but doesn't actually have the object, it would be a violation of the service provider agreement that can be

[29]Note that attacks on network routing can lead to availability attacks (e.g. routing black hole), man-in-the attacks, and so on.

handled out-of-band. We call this as the "Delegated Secure Flat Namespace Routing" problem, where we embrace the potential of someone advertising content that they don't have, but there are economic consequences for it.

While the GDP network builds on the knowledge of a vast breadth of research, the key innovation of the GDP network is to use cryptographic delegations for name advertisement and policy specification that tie well with a flat namespace enabling the root of trust to start from the destination address itself. Further, the GDP network allows one to reason about the security properties of the system by appropriate chaining of cryptographic delegations generated by different parties.

B What is the GDP network good for?

In addition to supporting DataCapsules, the GDP network enables new application scenarios as well as provides a cleaner architecture for many of today's applications that, for instance, rely on somewhat fragile combinations of IP anycast and DNS. the GDP network enables secure association of identity to the corresponding objects, extremely fine-grained provisioning that empowers edge computing, multiplicity of service providers (such as multi-cloud architectures), and the control over scope of information that is highly desirable in edge computing. Other than providing the routing fabric for the GDP, consider the following as mere examples of what is possible with the GDP network:

- The GDP network enables unlocking the full potential of DataCapsules. A user may deploy his own storage infrastructure in his house to store sensitive data in a DataCapsule restricted to private resources but archival data in a separate DataCapsule provisioned on a municipal-level storage provider, or maybe even in a far away cloud provider—each DataCapsule is referred to directly by its flat name regardless of the service provider underneath. Similarly, multi-cloud applications are possible natively.
- A user may connect to an IoT device by simply using its flat name (identity) instead of going through a level of indirection (a DNS-based URL). The GDP network allows for changing service providers underneath while keeping the same address (and associated security properties derived from the identity of the object).
- A machine-learning enthusiast may deploy a microservice for real-time object detection in video frames; end-users can simply refer to a running instance of the service by a flat name, whereas the developer ensures that the running instances are replicated with multiple compute providers in close proximity to end-users for quick response time and scalability. Even further, extremely fine grained provisioning of services, even at a building level, can be done much more easily where different instances of a service are hosted by different service providers.
- Organizations can enable network isolation based on policies specified by end-users, as opposed to solely decided by administrators.

2.5 Application case studies

In this section, we describe two case studies on application of the GDP and DataCapsules to real-world applications. Both of these case studies helped us refine the design of the GDP and DataCapsules, and enabled us to get invaluable feedback from users.

2.5.1 TerraSwarm

As a validation of the usefulness of the single-writer append-only DataCapsules to application developers, we made a preliminary version of the system available to about 8 group of researchers. These various research groups were all part of a multi-university research project, and spanned a number of research areas such as discrete event simulation, machine learning, hardware design, IoT application design, etc. 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 DataCapsule 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 to users as a client side software library in C with Python, Java and JavaScript wrappers around it, whereas we operated the server-side infrastructure. 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 the 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 DataCapsule, 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 later chapters, we believe to have addressed these points of weakness.

2.5.2 Secure 'Fog Robotics'

In order to continue the GDP design as guided by application requirements, we recently started a new collaboration with a group of robotics and machine learning researchers. We call this initiative as "Secure Fog Robotics".

The recent boom of machine learning has enabled new uses for robotics. With more sophisticated sensing of the environment powered by cameras and other rich sensor modalities, robots are now becoming an integral part of production environments, factory floors, machine shops, and even homes and businesses. The robots of today are highly connected machines that rely on not just on-

board resources but remote resources as well. This sub-field of robotics that deals with integration of cloud resources and Internet connected robots is dubbed as 'cloud robotics' [37]. While such integration expands the resources available to a robotics application beyond the on-board resources on a robot, there are a number of privacy, security, latency, bandwidth, and reliability issues that must be addressed.

'Fog robotics' is an emerging sub-field of robotics that takes into account the continuum of heterogeneous resources present between a robot and the cloud, in addition to the usage of cloud resources. As an example of the fog robotics vision, mobile robots on a factory floor can rely on reliable fixed infrastructure in the same building, thus minimizing on-board resources leading to better battery life and reduced unit cost. As another example, multiple robots in a home can collaborate with each other by sharing information directly instead of relying on the cloud as a rendezvous point, thus leading to better control on information that leaks outside a user's home.

Fog robotics is more than merely moving resources closer to where they are to be used. Lower latency, higher bandwidth, and enhanced reliability are certainly the most obvious effects of using resource closer to a robot. In the examples scenarios above, fog robotics introduces additional benefits as well. Going one step further, one could even claim that these other benefits, such as reduced unit cost in case of mobile robots in a factory floor or enhanced user privacy for collaborating home robots, *are* the driving factors of fog robotics. While fog robotics enables new applications for robotics, there are a number of security and privacy concerns that must be solved. Robotics applications of today are highly data-driven, and this data needs to be protected. The same perimeter security limitations that make edge computing challenging present themselves again for fog robotics.

DataCapsules and the GDP can help alleviate the security challenges by providing secure information management for robotics. At a very abstract (and superficial) level, if one were to consider a robot as a computational unit augmented with wheels and motors, then the same principles of separating computation from state maintenance can be applied to robotics as well. Examples of the state that robotics applications deal with are: data generated by on-board sensors and cameras, machine learning models for operating actuators based on what a robot senses, any remote actuation commands, diagnostic logs, etc. DataCapsules are an ideal mechanism for such state management. Using the GDP and DataCapsules allows for users to assert more control on the ownership and security of the information.

As a small demonstration of use of the GDP for fog robotics, we considered the use of a mobile robot for surface de-cluttering of objects in a machine shop environment [38]. By using on-board cameras, a robot can detect objects in front of it, plan a grasping of the object, and place it in appropriate bins. Such application requires extensively trained machine learning models on the set of objects that a robot may be presented with. There are a number of requirements that make this an interesting problem from both machine learning and system/data-flow design perspective. While it is desirable to use practically unlimited resources in the cloud to do such training that can be reused over and over again, it requires the potentially proprietary images of objects to be sent to the cloud. Further, once trained, such model needs to be updated periodically. And finally, trained models must be shared across a number of robots. In summary, selective sharing and appropriate resource

placement are what makes this problem challenging, and it is a good candidate for demonstrating application design and a system like the GDP to support the applications.

Our collaborators from robotics research developed a way to train such models in the cloud by using synthetic 3D models of a large collection of generic objects. These generic models are then distributed to the local site, and then refined locally for use in the specific environment by using real images (see [38] for full details). Once refined, such models can be distributed to individual robots. DataCapsules provide a vehicle for secure dissemination of trained models and any updates, the data generated at the robots, and any other diagnostics information in the form of a secure, verifiable, and auditable history that can be verified by each individual party. Further, one can assert control on the information flows by putting appropriate restrictions on the scope of DataCapsules at an administrative level.

To help with the use of DataCapsules and the GDP for this surface de-cluttering application and other machine learning applications in general, we developed a CAAPI for use with TensorFlow [36].[30] TensorFlow is a popular machine learning framework that is used by a number of robotics applications. Our TensorFlow CAAPI is in the form of a C++ library that can be loaded at run time and works with existing TensorFlow code. The CAAPI allows a user to the GDP and DataCapsules for all file system access by merely specifying a GDP path instead of a local file system path. We describe the internal details of this CAAPI in the next part of the dissertation.

While DataCapsules promote a separation of secure persistent state and enable secure management of information for use in fog robotics, it is only a part of the solution for an end-to-end secure fog robotics. DataCapsules provide information security while the information is at rest or in transit, but not when the information is in use. We envision that other techniques complimentary to DataCapsules, such as hardware enclaves or cryptographic techniques for operating on encrypted data, can fill the gap. However, a more extended discussion of these techniques is out of scope of the dissertation. Use of DataCapsules for secure fog robotics is merely the beginning of another research project in collaboration with other researchers, and many interesting research problems around the use of DataCapsules for fog robotics are yet to be solved.

[30] Recall from subsection 2.4.2 that a CAAPI (Common Access API) is a library that provides a more familiar interface on top of a DataCapsule.

Part II

The How: Internal Mechanisms

Chapter 3

DataCapsules: The Design

In this chapter and the next, we discuss the design and internal mechanisms of DataCapsules. Note that we focus more on the key design aspects of DataCapsule in this chapter; the next chapter is where we discuss how to solve a number of engineering challenges related to making this design work in the context of the overall system (the GDP). This is our attempt to tackle the first research task towards making the GDP vision a reality, which we identified in section 1.4. Recall that the task is:

Task 1 *Design of DataCapsules to be transportable over a federated infrastructure, while providing integrity and provenance for every single bit of data and enforcing the ordering relationships between such bits.*

This chapter is structured as follows. First, we discuss background and motivation showing where DataCapsules fit in the context of existing academic work, and highlighting the threat model that DataCapsules address. Then, we present the design for DataCapsules as standalone data structures that can work even in the absence of a network. And finally, we discuss the replication and durability/consistency implications of hosting a DataCapsule in a distributed environment with multiple copies.

3.1 Background and related work

A DataCapsule provides answer to the question of how to store data in a secure and portable yet distributed manner on an infrastructure that cannot be trusted for data security. A wide variety of related work exists in this domain. We specifically point out two key areas of previous research and practice: (1) secure storage on untrusted infrastructures, and (2) distributed storage systems that support replication for durability and high availability.

A DataCapsule is a data structure that can be easily hosted on a distributed but untrusted infrastructure, and as such, uses ideas from both of these two broad areas. Categorizing existing systems in these two broad categories echoes the tussle for placement of responsibility (which is what makes the DataCapsule design a hard problem): secure storage systems often lean toward

a design where a client is responsible for implementing additional mechanisms necessary for security,[1] whereas distributed storage system architectures often assume that the clients are as simple as possible and that the infrastructure components handle the complexity introduced by various kinds of failures.

DataCapsules address this tussle by targeting a relatively unexplored design point: a separation of durability and availability concerns from security concerns. As we will discuss throughout the chapter, clients are responsible for ensuring the security of the information, and the DataCapsule interface enables them to do so. In addition, DataCapsules enable a simple leaderless replication algorithm that works with minimal responsibilities from the infrastructure. Clients can recover from abnormalities introduced by infrastructure failures or malicious behavior from infrastructure components.

Let's review existing work in the relevant fields.

3.1.1 Secure storage on untrusted infrastructure

Outsourcing data storage to remote but untrusted storage servers has been an area of active research for more than two decades. A number of academic systems have been proposed to address some version of the problem; a few of the representative systems are: SUNDR [16], Plutus [18], SiRiUS [22], Oceanstore [17], etc. CloudProof [23] gives a good overview of such representative systems.

Almost all existing systems deal with a core data structure—a file, a database, key-value store, or something else—as the interface that they provide to users; such interface varies across systems. For example, SUNDR, Oceanstore, and Plutus provide filesystem interface on untrusted storage servers; CryptDB [19] provides a database interface; Depot [20] provides a key-value store interfaces; Antiquity [21] provides a log-based interface; and so on. We argue that a DataCapsule provides a lower level interface: the infrastructure is tasked with making information durable and available, whereas the contents of a DataCapsule are entirely determined by the client side. Thus, DataCapsules provide a narrow waist interface that is sufficient to build any of these other storage interfaces.

Existing systems can broadly be divided in two categories: (1) systems that perform ordering of updates strictly on the clients with little involvement from the server side (e.g. using purely cryptographic primitives), and (2) systems that preform ordering of updates on the server-side (e.g. by using a Byzantine fault-tolerant quorum of servers and assuming that only a fraction of servers can be malicious). While the latter category is appropriate for certain use cases, it requires that there be a minimum number of mutually distrustful servers and a superlinear number of messages exchanged between storage servers for every update. The GDP and DataCapsules are motivated by the proliferation of edge computing ecosystem where satisfying such minimum number of

[1] Of course, the server side must do additional work as well, but the onus of verification and security ultimately falls on the clients.

mutually distrustful servers may not be feasible at times. As such, we mostly designed the GDP and DataCapsules by using purely cryptographic primitives.

Note that using cryptographic primitives for secure storage is a bit more involved than 'just encrypt the data'. A trivial solution of 'just using encryption' is actually a non-starter. Simple encryption *may be* sufficient to guarantee confidentiality if the core data structure that a storage system exposes is read-only (e.g. a read-only file). But handling confidentiality, integrity, and provenance of information becomes challenging when such data structures can be updated over time. Handling updates in the presence of untrusted infrastructure *is* the main challenge that requires a careful design.

As such, any system offering secure storage must solve two key problems: (1) keeping information safe from being tampered with when it isn't being updated, and (2) handling updates. The handling of updates involves some way of finding the state of the data structure and dealing with freshness and consistency issues especially when the storage servers may lie about the current state of the given data structure. Depending on how a system handles these two main challenges, it may also have to handle some additional challenges, such as key management, etc.

To solve problem (1), many protocols and distributed systems have used the concept of an Authenticated Data Structure (ADS) [24]. An authenticated data structure 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; [39] describes the process of generating proofs. Authenticated data structures have been extensively studied and used in the form of hash-trees, hash-chains, or other variants [24], [40], [41]. DataCapsules inherit the concept of a proof for integrity verification from hash-trees. In fact, DataCapsules can be viewed as a custom designed ADS suitable to our use case.

Solutions to problem (2) vary quite a bit across existing systems. A number of previous systems have used the concept of *freshness* [23], [42]. DataCapsules and the GDP use a single-writer model in which the writer keeps local state to ensure that it can work without trusting the infrastructure to provide the correct current state of a DataCapsule. Readers must rely on freshness as provided by the infrastructure which is achieved by out-of-band contractual guarantees. Any further freshness mechanisms needed for specific use cases can be built on top, if need be.

Why not simply use an existing ADS, such as a Merkle hash-tree?

While a number of ADSes have been proposed (including the widely used Merkle hash tree), quite a few questions such as key management, update ordering, etc. are left unanswered by a simple Merkle hash-tree; the overall system still must provide necessary mechanisms for making these ADSes usable. Even after addressing these engineering details, using an off-the-shelf hash-tree/hash-chain for the GDP is less than ideal because of (1) performance, and (2) resilience to faults. In terms of performance, the usage pattern of applications can result in a wide variation in the cost of proofs (both size and required computation). As we will show in section 3.4, the DataCapsule design provides applications flexibility for very efficient proofs at the cost of requiring

the writer to maintain more state (and vice versa).[2] As for the fault resiliency, 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 DataCapsule design 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 DataCapsule design provides enough flexibility to meet the goals of a wide variety of applications, such flexibility does not come at a cost of weakening data integrity guarantees.

3.1.2 Distributed storage systems

Distributing and replicating storage is one way to go about achieving high availability, fault tolerance, and scalability. Distributed storage systems have been around for almost two decades. The core idea seems simple: replicate a given piece of data to multiple servers. However, keeping the replicas consistent is a major challenge that any distributed storage system must solve.

To reason about the guarantees provided by these distributed systems in a formal way, a number of consistency models have been developed over the years. These consistency models range all the way from strong consistency to eventual consistency. A number of general techniques and algorithms have emerged that address various points in the consistency spectrum. A few noteworthy examples are consensus algorithms such as Paxos [43] and Raft [44]; transaction protocols such as 2PC and 3PC; data structures based approaches such as version vectors and Conflict-free Replicated Data Types (CRDTs) [45]; and timestamp based approaches. Existing systems employ these existing techniques to achieve the desired consistency mode for a specific application. A few noteworthy examples of various consistency models in practice can be seen in systems like Chubby [46], Google Spanner [47], LogCabin [48] that rely on strong consistency models by using a consensus algorithm like Paxos or Raft internally. A number of other systems like Tardis [49], Bayou [50] and Dynamo [51] provide eventual consistency.

Stronger consistency models are useful to applications, since users can work with a simpler model of storage that is closer to a non-distributed storage system. However, these stronger consistency models come at a cost of complex system design. CAP theorem [52] suggests that only two out of three desirable properties—consistency, availability, and partition tolerance—can be achieved at a time. In different words, when there is a partition, a system must pick either consistency or availability. Further refinements, such as PACELC [53], suggest that even when there are no partitions, a system must prioritize between latency and consistency.

On the other hand, weaker consistency models are easier to implement as a system, but harder to use from applications. The application must do something about potential conflicts detected when replicas are synchronized. The ease or difficulty of performing conflict resolution varies across

[2]For example, using a Merkle hash tree is a good choice for random access workloads like a file system. But for representing a practically infinite stream of events where recent events are of more interest, a Merkle hash tree would incur too much overhead.

applications.[3] A number of existing techniques, such as version vectors, timestamps, etc. have been devised to effectively handle conflicts in distributed storage. Further, Conflict-free Replicated Data Types (CRDTs) [45] have been proposed that ensure applications are designed to play well with the underlying distributed storage; a CRDT is a data structure where there are no conflicts when replicas are synchronized by the very design of the data structure.

Where are the GDP and DataCapsules in the distributed storage space?

With DataCapsules, the single writer provides a point of serialization. As such, the discussion of consistency semantics is slightly different than a typical multi-writer distributed storage system. A DataCapsule is merely a way of keeping a history of transactions—as decided by the single writer—in a single cohesive unit that can be stored on servers not trusted for data security.[4,5]

The GDP exposes primitives of distributed storage to readers and writers allowing users to get their own desired operational semantics by either doing the necessary extra work themselves, or specifying policies for the infrastructure.[6] In this way, DataCapsules and the GDP provide the thinnest possible layer between an application and the persistent storage in form of disks on a remote server, but can incorporate a wide variety of existing mechanisms such as version vectors, timestamp-based 'last write wins', CRDT, etc. to implement application-specific conflict resolution. This kind of design is in contrast with many other distributed storage systems that prescribe a specific method for reads and writes, which leads to specific consistency or availability semantics in case of failures. We will discuss this in more detail towards the end of this chapter.

3.2 DataCapsule threat model

DataCapsules have a modest security goal: provide secure storage even in presence of untrusted infrastructure. In terms of security properties, DataCapsules primarily aim to provide data integrity and provenance, and enable confidentiality by providing support for encryption at record level. DataCapsules provide these security properties when the data is at rest and when under modification (i.e. handling updates).

The threat model discussed in section 2.2 broadly applies to DataCapsules as well. Readers trust the DataCapsule owner and the designated single writer to do the *correct* thing based on

[3] For example, conflict resolution in a shopping cart application is rather easy because the application is mostly agnostic to the order of operations, whereas handling conflict resolution in an encrypted file system is extremely challenging because the order of operations matters and the system has little visibility into the changes introduced by each operation.

[4] One could even say that a DataCapsule is a generalized Git repository with a focus on security. While this simplified view misses the nuances of the design and engineering work needed to make DataCapsules practical, it does server as a great mental model for the DataCapsule abstraction.

[5] Even though DataCapsules are an ADS that require proof generation, etc., they are specifically designed to perform well when replicas are not completely in sync.

[6] A number of systems consider, for example, a 'write' operation to be complete only after it is replicated on N replicas. The DataCapsule interface by itself does not impose any such restrictions. Applications convey their desired requirements to the GDP by means of policy specification, which can lead to a different definition of 'write' for different DataCapsules.

the negotiated expectations. The infrastructure is distrusted; log servers may tamper with data in any way they see fit: they may shuffle bits, change the order of updates, replay information, reorder updates, etc. Third party adversaries may attempt to insert bad information by, for example replaying requests, etc.

The only exception to the general threat model is that DataCapsules assume a correctly functioning routing layer underneath (provided by the GDP network). The GDP protocol is designed specifically to augment security properties of the GDP network by providing security in transit. Even if the routing layer doesn't provide necessary security guarantees, DataCapsules may not operate efficiently, but data security never gets compromised.

3.3 Anatomy of a DataCapsule

In this section, we first describe the internal details (i.e. how a DataCapsule looks in memory) and the rationale for the DataCapsule design purely at a data structure level. Then, we discuss a byte-efficient and self-sufficient format for transport or archival storage for DataCapsules (or even parts of a DataCapsule). Finally, we describe how operations on DataCapsules are materialized.

3.3.1 In-memory structure of a DataCapsule: Records and metadata

Recall that a record is the unit of read or write to a DataCapsule, and that DataCapsule metadata is the special record at the beginning of a DataCapsule. Records as well as DataCapsule metadata are immutable structures that are linked with each other using *hash-pointers*. A DataCapsule is primarily an ordered list of records with the DataCapsule metadata at the beginning. The simplest DataCapsule is a hash-chain—a very simple ADS—of records in the order they were generated with the DataCapsule metadata at the beginning (see Figure 3.1). By using additional *hash-pointers* and various strategies of interlinking records, DataCapsules enable better performance for generating proofs for readers and higher tolerance for failures. We will describe these generalizations later.

A DataCapsule metadata

DataCapsule metadata, as the name implies, is the immutable metadata associated with the DataCapsule from which the DataCapsule derives its flat 256-bit GDP name. A non-exhaustive list of information in the DataCapsule metadata is: a public signature key for the signing records (writer-key), a public signature key of the DataCapsule 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 DataCapsule, etc. Note that the writer-key is a required field in DataCapsule metadata.

DataCapsule metadata serves an important purpose: it creates a secure association between the DataCapsule name and the public signature key for the writer; since the DataCapsule name is the hash of DataCapsule metadata which contains the writer-key, anyone can verify the accuracy of the writer-key simply by calculating appropriate hash and comparing it with the DataCapsule name.

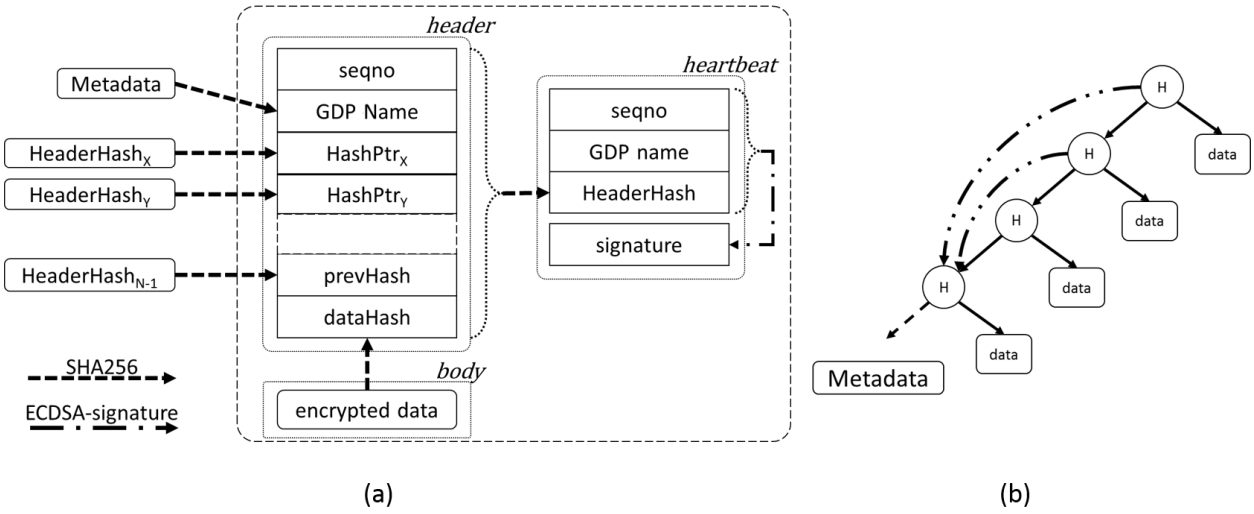


Figure 3.1: (a) Record structure demonstrating record-header, body and heartbeat. (b) Without hash-pointers *(offset, headerHash)*, the records in a DataCapsule make a very skewed Merkle tree (essentially a hash-chain). Hash-pointers include additional links and transform the Merkle tree to a Directed Acyclic Graph (DAG).

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 that can be verified by anyone; (2) it provides a write-access control mechanism for a correctly functioning log server that can validate a signature and assert that a write request is in fact created by an authorized writer.

B Records

A record is an immutable structure composed of a header and the body, with an associated heartbeat (see Figure 3.1).

The *record body* contains 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. An important requirement for the encryption scheme is that a reader should be able to decrypt records individually.[7] The decryption key can be securely communicated to the allowed readers using any out-of-band communication channel suggested in the DataCapsule metadata.

[7]For our prototype implementation, we use AES-128 in CTR mode, where the counter is initialized from the DataCapsule 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.

The *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`),[8] the 256-bit name of the DataCapsule (`DataCapsule 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`),[9] and the hash of the record body (`dataHash`). Within the namespace of a DataCapsule, a record could be uniquely referenced either by `seqno` or by the hash of the header `headerHash`. We loosely refer to the `headerHash` as the 'hash' of the record for the rest of the dissertation.

The *record heartbeat* is a signed stand-alone piece of data associated with a record that contains `seqno`, `DataCapsule name`, the corresponding `headerHash` and a signature over the three items using the private part of the writer-key. Heartbeats are small pieces of data that can be distributed widely in a network (because of their small size) and have the same built-in ordering as the records (because of the included `seqno`).[10] A reader, subscriber, or even an infrastructure component can compare a number of heartbeats, and use the newest heartbeat to incrementally update its local state about the state of corresponding DataCapsule. Heartbeats enable service providers to build infrastructure with data-freshness guarantees: a service provider could provision a 'data-freshness service' that collects and delivers the latest heartbeats for DataCapsules served by the service provider, and end-users can verify the authenticity of such heartbeats. However, as we will describe shortly, signature verification using heartbeats is not the most optimal way for reading old data in bulk.

To summarize, record body contains the actual payload; record header describes the ordering of a record with respect to other records in the given DataCapsule; record heartbeat represents the authenticity and authorization for a record to be a part of DataCapsule. Separating the heartbeat from the header allows old signatures to be discarded if needed, allowing for low-overhead archival storage of DataCapsule. Separating the body of a record from header and heartbeat ensures that the proofs of data integrity can be generated without needing the full body, which enables a few useful properties. For large DataCapsules, the bulky parts are in the body; a reader or a log server needs to keep only the headers in memory to verify the structure of a DataCapsule. This same argument also implies that proofs can be small and readers can validate old records without needing to fetch arbitrary sized payloads of intermediate records. Further, a log server can serve read requests along with appropriate proofs even when it has only a partial replica of the data without the bodies of all intermediate records.

3.3.2 DataCapsules in transit and archival storage: RecContainers

While the in-memory structure we presented fits the needs of an authenticated data structure, it is not the most byte-efficient way for representing a set of records due to the inherent redundancy of

[8] We relax the requirement for `seqno` to be monotonic in the case of failures on the writer. We discuss this failure mode and the associated consequences in subsubsection B.

[9] For the first record, `prevHash = DataCapsule name`

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

information. For example, storing/transmitting both data as well as a hash of data (for use in hash-pointers) is wasteful because the hash can be regenerated from the data when needed; signatures need not be included with every record in a set of consecutive records as one can generate the integrity proofs using hashes instead; etc. To reduce such redundancy, we introduce the concept of *RecContainers*. A RecContainer is a way of representing a collection of records in coherent but compressed way with sufficient additional information for proving the integrity and provenance of all records to a verifier.

A RecContainer can be used for representing any subset of records of a DataCapsule, or even the entire DataCapsule. A RecContainer targets the amortization of cost across a collection of records, both in terms of byte size as well as computational costs needed for integrity verification. As such, RecContainers are an ideal representation of records in transit or archival storage. Internally, appends, reads, and other operations on a DataCapsule are serialized on the network in the form of RecContainers. Even though RecContainers aren't directly exposed to the users, it is a central concept for the implementation of operations on a DataCapsule.

To create a RecContainer, one first collects the records that should go in the RecContainer. This involves collecting new records in the context of a writer, fetching records from persistent storage for a log server, and so on. After collecting the desired records, some additional information is needed to ensure that the recipient of a RecContainer can prove the integrity of every record contained in the RecContainer. Such information could be in the form of signatures extracted from record heartbeats, or in the form of additional record headers to create a hash-based proof, or both (see below). Finally, all this information is passed through a compression step that involves removing redundant information that can be regenerated on the client-side and then serialized.

How to create proofs? Verifying the correctness of hashes and the signature included in the heartbeat is a direct way of verifying the integrity of a given record. But once a record r has been verified, other records older than r can be verified against r by a hash-based proof. Ignoring the additional 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 3.1(b)). One can use r as the root of a Merkle subtree to generate a proof of integrity by traversing the hash tree to other records. When additional hash-pointers are present, the DataCapsule can be treated as a directed acyclic graph (DAG) composed of nodes (record headers) and links (hash-pointers). To generate an optimized proof for a target record q with respect to a known record r, all one has to do is to find the shortest path from r to q in the graph.

There are some interesting choices for the additional information added to the RecContainer for verification; these choices are guided by the desired optimizations. As an example, one may create a RecContainer by inserting a signature for every single record. But this is quite wasteful if the RecContainer contains a large number of consecutive records; once the most recent record has been verified, all other records can be verified by simply using a hash-based proof.[11] Even if the records aren't consecutive, one could still use a hash-based proof instead of a signature-based proof for all but one record. The decision to use signatures or hashes depends on the desired optimization;

[11] Recall that each record header has a hash-pointer to at least the previous record.

signatures are computationally expensive but using a large number of hashes may be expensive in terms of byte size.[12]

Another choice that affects the behavior of a RecContainer is whether to include an 'absolute' proof or a 'relative' proof. One may create a RecContainer with sufficient number of signatures to create a completely self-sufficient RecContainer such that *anyone* can verify the included records. Or, a RecContainer could be created based on an assumption that the recipient already knows the hash of certain records; such RecContainer may omit signatures altogether and rely completely on a hash-based proof. As an example, when a reader of a DataCapsule requests a log server for certain records of a DataCapsule, it could optionally include information about hashes it knows already (presumably in a previous read request). Such RecContainers are much more efficient since they avoid expensive signature verification altogether, and even allow for amortization of costs even across a number of requests.

Even though the interface to a DataCapsule user only deals with operations at the record level, all the operations are serialized in the form of RecContainers for maximum efficiency on the network; we will discuss this process in the next section.

3.3.3 DataCapsule operations

In this section, we describe the internal details of how various operations on a DataCapsule are materialized. Note that other than DataCapsule creation, clients direct all operations to the DataCapsule by using the 256-bit DataCapsule name as the destination of their requests. The GDP network—the underlying routing network—delivers such requests to a DataCapsule replica hosted on a log server, which responds to the request on behalf of the DataCapsule.

A Creation

The process of creating a new DataCapsule involves sending a 'create' request to a log server, which contains two items: the signed metadata and an 'advertisement certificate' (AdCert).

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 DataCapsule creator and B is a log server.[13] 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,

[12] In our experience, verifying a signature is roughly 3 orders of magnitude expensive than computing a hash.

[13] If a DataCapsule is replicated on multiple log servers, each log server is issued a separate $AdCert$.

the creator acts as a Certificate Authority issuing a certificate to log servers or service providers to serve requests on behalf of the DataCapsule.[14]

Why delegate a specific organization or host for a DataCapsule?

First and foremost, explicit delegation codifies the service-provider model that can be understood and verified by computer code. In alternate models of ubiquitous caching and no explicit delegation (as popularized by various peer-to-peer networks), a reader does not have a remedy for degraded availability or quality of service for a DataCapsule. When a creator explicitly delegates the hosting responsibilities to an organization, it presumably also enters into an implicit or explicit economic agreement for a given level of service that can be enforced by legal contracts.

Second, explicit delegation allows the GDP network—the underlying routing network that we describe in a later chapter—to provide a baseline level of routing security. By verifying that a log server (or organization) that claims to host a specific DataCapsule is, in fact, authorized to advertise the DataCapsule name into the network, the GDP network can ensure that malicious adversaries cannot launch man-in-the-middle attacks or routing black-holes by claiming to own names they don't have authorization for.

Finally, using a cryptographic delegation enables transport-layer security. As we will describe in detail in the design of the GDP protocol (chapter 4), any acknowledgments or responses from a log server must be secured at the transport level. Otherwise, an active man-in-the-middle can, for example, simply drop any append operations and send a spoofed acknowledgment to the writer, thus framing an honest log server. 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.[15] However, note that the $AdCerts$ can be short lived allowing for a DataCapsule 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, all the operations from clients are addressed to the DataCapsule instead of a physical server.

B Append

Recall that a DataCapsule is a single-writer data structure. The single writer holds the private part of the writer-key and maintains some local state preferably in non-volatile memory.

To append data to a DataCapsule in the simplest case of a single log server (without replication), the writer creates the appropriate record structure with a header and body.[16] The writer is free to include *hash-pointers* to any older records as it seems fit.[17] The writer signs the record by creating

[14] Note that in case the DataCapsule 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.

[15] 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 DataCapsule to a particular log server for eternity.

[16] We describe the append operation with replication later in this chapter.

[17] The exact nature of the linking structure created by writer has a number of performance and durability implications, which we will discuss in chapter 4.

a heartbeat, it updates its local state in a non-volatile memory, and then sends the append request along with the signature included in heartbeat to the DataCapsule. The serialized append request on the network is essentially a RecContainer created by the single writer with new records that are now a part of the DataCapsule.

Once the writer sends the append request, 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 DataCapsule. On receiving the RecContainer as part of an append request, a log server uses the common RecContainer API to extract verified records from this RecContainer and store them on persistent storage. The log server sends a secure acknowledgment back to the writer, and notifies any subscribers to the DataCapsule by simply forwarding the RecContainer received from the writer.

The 'single record append request' can easily be generalized to a 'multiple record append request': instead of sending a single record at a time for append, the writer may send a number of records—all linked together appropriately—in a single RecContainer. Regardless of the number of records, any append request needs exactly one signature in the form of a heartbeat for the most recent record, as long as the previous records are reachable from the most recent record by following a sequence of hash-pointers. Using multiple records in a single append request not only has the benefit of amortization of signature generation cost, but also enables 'atomic transaction'-like semantics for DataCapsule where an application would like to spread a number of writes over multiple records (to preserve some application-level logic), but still make sure that the readers either see all the records included in the append request, or none of them.[18]

The role of single writer: The single-writer append-only design enables the single-writer to be the point of serialization—a very useful property that allows us to do perform a *leaderless replication*.[19] However, this benefit comes with two additional costs. First, we use signatures with the writer's private key to maintain write access control and provide data-provenance 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 DataCapsule 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.

Writer's responsibilities: First, 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. It is also

[18] Readers must get an integrity proof from the log server either in the form of hash-pointers or a signature, as we will describe next. For multiple records in a single append request, the only available proof for records other than the most recent records is via a path of hash-pointers starting from the most recent record. Thus, it is impossible for a reader to get an integrity proof for an older record in a multi-record append request without knowing about the most recent record of the said request.

[19] Arguably, the single writer is a leader that drives the replication process. However, the term *leaderless* is in contrast with the typical distributed algorithms that use an explicit leader election process.

the responsibility of the writer to create record headers carefully by (1) picking a good set of hash-pointers that minimizes proof sizes, and (2) ensuring the monotonicity of `seqno`.

The monotonically increasing nature of `seqno` is part of the contract that a DataCapsule writer provides to the readers and subscribers.[20] Log servers do minimal verification for records and merely ensure that records meet the admission criteria, i.e. valid hash-pointers and a valid signature. Because of the monotonic nature of `seqno`, a reader can *efficiently* get the records in the order they were generated and subscribers can reason about the order of heartbeats. While it is certainly possible to come up with a design that does not use `seqno`, the practical challenges of using only a hash to identify a record outweigh the costs of additional responsibility for a writer. Further, using a `seqno` allows us to use the CTR mode of encryption.[21]

C Read

Readers can query old records either by their `headerHash` or `seqno`. When querying by `seqno`, a reader can optionally specify a more recent record r that the reader knows of. The log server replies back with a RecContainer which contains the requested records and a proof based on the information supplied by the reader. If the reader didn't supply any information other than the `seqno` of requested records, then the log server generates a self-sufficient RecContainer with signatures. Otherwise, the log server uses hash-based proofs.

Instead of reading one record at a time and seeking individual proofs, a reader can also query for a range of records by specifying a `headerHash` or a `seqno`, and the number of records before the specified record. Just like multi-record append requests, reading multiple records has the benefit of amortized cost of proofs.

An important pattern that comes up in many applications is querying for the most recent record. DataCapsules do not support the operation of querying for the most recent record directly. Instead, a reader can request for the most recent record heartbeat for a given DataCapsule, which provides the reader with both the `seqno` as well as `headerHash` for the most recent record that a log server is aware of. If the result of a heartbeat query return a heartbeat older than what a client already knows of, which could happen when there are multiple replicas slightly out of sync, a client simply uses the more recent heartbeat it already knows of. Once the reader has acquired the most recent heartbeat, it can then query the corresponding record directly by using the `headerHash`.

An alternative design where the reader can directly ask a log server for the most recent record has the downside of being slightly more difficult to program in the presence of replication. Explicitly separating heartbeat querying from the actual reading allows a reader to get more visibility into the operation without necessarily exposing the client to the complexities of replication. We discuss more details of the consistency semantics that a client can expect in the next section.

[20] One way to look at the argument is: the writer is responsible for ensuring that the information it puts in the DataCapsule is useful to readers. In the same vein, we argue that crafting an appropriate record header (with monotonic `seqno`) is not necessarily changing the requirement that readers trust the writer for meeting certain expectations.

[21] We discuss the case when a writer fails to fulfill this contract of monotonically increasing *seqno*—say because of a crash—in section 3.4.

The question of freshness: A reader trusts the log servers to not provide stale data, for example, when querying for the most recent heartbeat. A fundamental design decision of the GDP and DataCapsules is a service provider model for making the data durable and available; log servers operated by service providers are inherently trusted for keeping the DataCapsule replicas up-to-date. A malicious log server *can* provide stale data to a reader, however this is a violation of the contract that a DataCapsule owner has with the service providers (or log servers) that it delegated the hosting responsibilities to, and can be enforced out-of-band. To detect stale data for mission-critical applications, the writer can append empty records (with null body) at a set interval to force periodic heartbeats (and indicate the period in the metadata); readers that do not get the periodic heartbeats can at least detect data staleness.

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.[22] Because of this heavy cost, a log server uses hashes as much as it can for 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 does the use of hashes reduce the computation cost, it also frees the log server from the burden of storing signatures. To reduce storage costs, the log server can eliminate all signatures except the most recent signature. As a further optimization, a log server can even get rid of all the hashes and regenerate them from the actual data when needed.[23]

D Subscribe

Subscriptions are a client requesting to be notified of new data as it is generated by the single writer. In the simplest model, a client requests that it be notified for a given number of future records, say n. A log server, when it receives an append request from the writer, forwards the RecContainer from the append request to the subscribers. These subscribers perform validation of record included in the RecContainer in the same way as a log server.

Service guarantees on subscription: Subscription, at the very core, is a best-effort service in our current design, which provides a better alternative than polling for fresh data. However, since the GDP is a distributed system, there are cases of network partition or other transient failures where a subscriber may not get notified of new data. A particularly interesting example is that of a DataCapsule with multiple replicas hosted on different log servers; a subscriber maybe interacting with a different log server than the one that the single writer is interacting with. In case various replicas are wildly out of sync for any reason, the subscriber may not receive updates in real-time. As such, for applications that depend on a real-time delivery of updates *must* devise application-

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

[23] The storage cost of signature should be evaluated 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.

level construct to detect such failures. A simple example of an application-level workaround is periodic no-op appends by the single writer at some mutually agreed upon frequency.

In-network multicast vs log server mediated subscription? In our current design, subscription is essentially a generalized form of a read request that is mediated by a log server. An alternate design choice could have been to use network-level multicast where subscribers get the append request (and the associated RecContainer) directly from the single writer. While such a design is a better design for a number of use-cases (e.g. tight control loops where adding a log server adds to the latency, or supporting an extremely large number of subscribers), a log server mediated subscription keeps the design of DataCapsule operations simple. We consider a multicast based approach as a direction for future research.

3.4 Distributed operation: Replication

As with any other distributed system, replication is desired for a number of reasons: durability of information in case of failures, enhanced service availability by avoiding single points of failure, scalability to handle a large number of queries, etc. Recall that a DataCapsule represents a virtual object spread over a number of servers, and not a single replica. Depending on application requirements, the GDP and DataCapsules provide a set of options to achieve a desired subset of properties.

We broadly divide the discussion about replication into durability properties and consistency semantics. Note that while both durability and consistency are results of replicated operation and are deeply intertwined, they can be decoupled and be treated somewhat independently. It is possible to achieve one without the other, and we argue that such a separation is useful to applications. Let's see how.

Consider a number of clients that read from multiple replicas. If all clients agree on the order of updates regardless of the replica they read from, but certain data is missing from all the replicas, that would be no durability but consistency. Such a situation is still useful for certain class of applications, for example a video stream with known missing data is still useful as long as video frames are in known order. On the other hand, a certain information object may be sufficiently durable, but multiple clients that read from different replicas may not agree on what the order of updates was, that would be durability but no consistency. As an example application that could still use benefit from such a situation, consider a shopping cart with unknown order of operations. It doesn't matter what order the items were added to the shopping cart as long as the information is sufficiently durable.

With the GDP, the same underlying infrastructure can support a number of durability and consistency semantics based on how writers and readers interact with a given DataCapsule. The desired durability and consistency modes can be included in the DataCapsule metadata; which allows for the infrastructure and the clients to adjust the interactions with the DataCapsule to ensure standardized expectations.

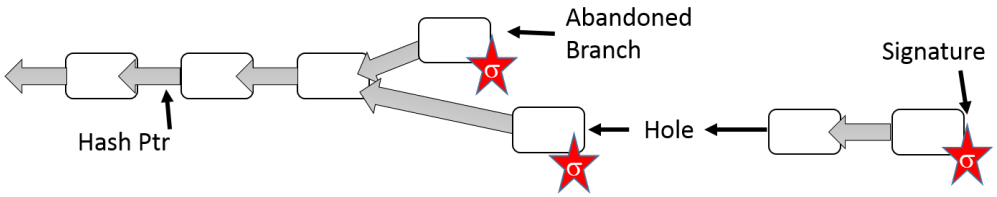


Figure 3.2: A DataCapsule with holes and branches. See subsection 3.4.2.

3.4.1 Durability and consistency in normal operation: No failures

In this section, we consider the durability and consistency that readers can expect when there are no failures. In the next section, we describe how infrastructure failures lead to reduced durability, and discuss the weaker consistency semantics for applications where a writer is allowed to fail and lose its local state.

A Durability Model

DataCapsule design allows durability decisions at a record-level; an application can decide that a specific record, say a record representing a file system snapshot, should be more durable than other records. Depending on application requirements and the support from underlying system, the replication can be performed as either in a *writer-driven* mode or in a *server-driven* mode.

A *Server-driven* mode of durability is essentially optimistic replication: a writer send append request to the DataCapsule, which is delivered by the underlying infrastructure to the closest replica of the DataCapsule.[24] The writer considers 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 DataCapsule. In such a mode of operation, the writer trusts the server to make the data durable by replicating it to other log servers. 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.

In a *Writer-driven* mode of durability, the writer considers an append to be complete only when it receives a desired number of acknowledgments; i.e. the writer directly ensures that the desired number of replicas of a record have been created. There are two potential schemes: (1) 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 in a single message; or alternatively, (2) the writer uses *multicast* for append and receives acknowledgments from log servers directly.

In the absence of failures, even though both modes seem equivalent, there are subtle differences. In a simple non-replicated model of a DataCapsule, an append request is directed to the DataCapsule name and not to a specific log server. However, to perform replication effectively,

[24] This is equivalent to *anycast* in networking terminology.

it is important to be able to refer to specific replicas by directly addressing the log server. In a server-driven mode, the writer can still be shielded from this additional responsibility, which is why we prefer the server-driven mode; log servers must still keep track of other log servers that host a replica of a given DataCapsule. For our prototype system, we currently use a server-driven mode. Nonetheless, discussing the writer-driven mode is certainly useful as it opens up a number of different consistency/durability modes.

B Consistency model: Strictly ordered updates

The simplest mode for a DataCapsule is that of a strict single writer; this single writer is the point of serialization. This serialization is manifested in the form of hash pointers and `seqno` inserted in the record header by the writer. To create record header correctly, we require that the writer remembers what it wrote last, potentially in a non-volatile local memory. When the writer comes back up after a while (say, a reboot), it can use the non-volatile memory to recover its previous state. Thus, all updates are linked together in a linear order, resulting in a 'strict order' of updates as defined by the hash-chain ordering.[25] We call this the Strict Single Writer (SSW) mode.

As an example of SSW mode in practice, consider an IoT device (a temperature sensor, a video camera, etc.) generating data and committing to a DataCapsule that is tied to the device's identity. If the device fails, there's nobody else to add new information to the DataCapsule.

In SSW mode, readers can contact any replica. Some replicas may run behind and provide stale information. A reader must keep track of the most *recent* record that it has read at any given time. On any subsequent request from a different log server, a reader can detect data older than the most recent record that the reader has seen by simply looking at the response. Because a reader can ignore stale information than what it has seen already in the past, it can always order the updates in the order as performed by the single writer. However, because the reader only contacts a subset of replicas (typically only one), different readers may get different values. Such behavior is equivalent to a sequential consistency model.

Note that a strict consistency model can also be achieved by using a smarter reader; the reader contacts *all* the servers and picks the most recent state. A reader desiring strict consistency is prone to a reduced availability in case of network partitions or server failures.[26]

3.4.2 Handling failures: Holes and branches

In the absence of any failures, the replication process we described above goes smoothly and readers see strictly ordered updates. DataCapsules provide a useful storage interface even in the case of failures. Consider an example of a DataCapsule for a home security video archive with one replica inside a household and another replica in a cloud data center. In case of network

[25] Strict order: `http://mathworld.wolfram.com/StrictOrder.html`

[26] Because of the condition that a write must never be rejected, there are no roll-backs; as long as even a single server receives a write, it must necessarily be a part of the DataCapsule. In the classic equation $W + R > N$ for strong consistency, this results in a situation of W, thus R must necessarily be equal to N.

partitions, the security camera can still continue to generate data and write them only to a local replica with reduced durability. During such network partitions, applications that can work with reduced consistency guarantees (i.e. stale data) can keep operating. However, applications that desire a strict consistency must block unless network partition is repaired.

We consider two main categories of failures: failures on infrastructure side (i.e. when not all designated log servers hosting a replica of a DataCapsule are available), and failures on the writer side (i.e. the writer does not maintain local state, or strict single writer model is violated). Recall that a properly signed append request can never be rejected permanently by the infrastructure; the infrastructure may not be able to serve the append request for various reasons, but the ultimate decision of what goes in a DataCapsule is made by the writer.[27]

A Infrastructure failures: Reduced durability

In a situation of server/network failures where a writer does not receive enough acknowledgments, a decision is to be made by the writer: the writer can either block (potentially indefinitely) or it can continue at the chance of a potential loss of records which might create a *hole* (see Figure 3.2). A *hole* represents a sequence of records that are permanently lost.[28] Whether holes are acceptable or not, i.e. the writer's behavior during infrastructure failure, is an application-level decision and depends on the type of data that a DataCapsule holds; such policies are ingrained in the metadata of a DataCapsule, and writer must adhere to such policies as part of the contract between the writer and readers.

Note that a writer's decision to block is merely to satisfy durability guarantees, and not necessarily for consistency guarantees; we will discuss consistency in the next subsection. Even in the presence of holes, it is still possible to achieve the desired integrity verification because of the additional hash-links. In fact, if a writer detects reduced durability, it can even start adding such additional hash-links to ensure that hash-based proofs can still be made to work. We discuss how these hash-pointers work in chapter 4.

When are holes okay? Whether holes can be tolerated is application dependent. 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 filesystem on top of a DataCapsule without any checkpoints.

The decision of whether holes are acceptable or not also has consequences on the state that the writer must maintain. 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

[27] In other words, properly formed update requests may only fail with a 'Service Unavailable' or similar status.

[28] Note that not receiving sufficient acknowledgments is not necessarily indicative of a permanent data loss. In addition to online replication, log servers also perform offline replication to fill any missing data. We describe this later in this section.

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.

Further, whether holes are acceptable or not also drives the decision for the replication mode chosen by the writer. While server-driven durability mode is typically higher performance than client-driven mode, there is a potential for holes and permanent data loss. Such permanent loss can occur if the 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. Thus, applications that cannot tolerate holes should use writer-driven replication.

Note that the interface to the writer (append) does not change in the presence of holes. A reader does need to know that in the presence of holes, it may only get partial results for its read query. A log server marks the non-availability of records as such. Log servers may indicate that they have finished offline synchronization and even then, they were not able to get the desired records. Subscription is essentially a best-effort service and a reader may stop getting records temporarily if the log server dies. However, because a subscriber must renew subscriptions periodically, any subsequent subscription renewals get redirected to an active log server by the underlying GDP network. Thus, in case of failure of a single log server, subscribers see a delay in delivery of a set of records.

B Writer failures: Weaker consistency

So far, we have discussed a more powerful single writer that maintains local state in a non volatile memory. When this condition is not met, the DataCapsule may end up with *branches*. Branches in a DataCapsule occur when two or more records exist that have hash pointers pointing to the same record (see Figure 3.2).[29] Branches result in a 'partial order' of records, since it is not always possible to directly compare the ordering of any given two records.[30]

In case the 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 the presence of accidental or malicious stale servers, the writer may not receive the correct state of the DataCapsule; the writer may start 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 DataCapsule metadata). Nonetheless, the single-writer DataCapsule primitive works well even in case of such failures.

Relaxing single writer mode a little bit, we define a Quasi Single Writer (QSW) mode. In QSW mode, there can be, in fact, multiple writers, but under normal circumstances they do not all write at the same time. Decisions about 'who gets to write in a given window of time' is either done via out-of-band mechanisms, or implicitly imposed by the application constraints.[31] There is no single point of serialization, and each individual writer must first synchronize its local state with the

[29] This definition of a branch is ignoring the *additional* hash pointers.

[30] Partial order: `http://mathworld.wolfram.com/PartialOrder.html`

[31] Note there are no server-side locks.

infrastructure at the startup time. There are chances that either the 'one writer at a time' property or the 'synchronization at startup' doesn't really work, which could result in *branches*.[32]

In QSW mode, the consistency model is that of strong eventual consistency (SEC). Even in the presence of branches, asynchronous replication between replicas will eventually converge. The 'strong'-ness is implied by the fact that a DataCapsule is a CRDT; the ordering in which updates are applied does not matter, and that the asynchronous replication is merely a union of all the records. Once again, the DataCapsule metadata marks the mode of operation of the writer and a reader must take into account the consequences of such branches.

Two examples of the QSW mode in practice: (1) A personal filesystem maintained by a user, but mounted on a number of devices. If the user only works with any one device at a given time, multiple writers at the same time are rare (but still possible). (2) A computer process directly writing to a DataCapsule (as opposed to an IoT device). Because of application-level failures and network partitions, it is possible that there are multiple instances of the process all writing at the same time.

We don't discuss in detail the situation of multiple writers all writing directly to a DataCapsule at the same time. It is certainly possible to do, but results in extensive branching. Even though the consistency semantics are the same as QSW mode (i.e. SEC), it is difficult for an application to deal with such extensive branching. Instead, we recommend that users run a service that collects the writes from multiple writers, serializes them in some application specific order, and then acts as the single writer of the DataCapsule.

Note that the interface to the writer (append) does not change in the presence of branches. The interface to the reader does change a bit. In presence of branches, reading old records must be done carefully. Fetching a heartbeat may result in multiple heartbeats that represent different branches of a DataCapsule. A log server only returns a set of consecutive records at a time as the result of a read operation, but the reader must decide which heartbeat(s) to use for a read request. Subscriptions, on the other hand, are best effort. The log server a subscriber is talking to may be running behind and may get updated out of order. As such, the behavior for subscribers is undefined in the case of branches. Nonetheless, the allowed deviations from a single writer with persistent state can be indicated in the DataCapsule metadata, allowing readers to set the expectations correctly.

C Offline synchronization process

In addition to online replication, log servers also perform periodic background synchronization for partially out-of-date replicas by filling temporary holes or synchronizing branches. Recall that DataCapsules are hosted by one or more service providers chosen by the DataCapsule-owner based on some economic model. The infrastructure is not trusted with the contents of a DataCapsule, and this should be taken into account even during the offline replication especially when performing synchronization between log servers belonging to different service providers. Because of such distrust, a log server must verify any information received from other log servers during

[32]The semantics of subscriptions are not well defined in case of branches.

synchronization process in very much the same way as if it received an append request from the DataCapsule writer.

The append-only design with a writer-driven serialization, coupled with the fact that a well-formed append may never be *rejected*, makes a DataCapsule a Conflict-Free (Commutative) Replicated Data Type (CRDT) [45]. Since records are immutable, synchronization is essentially a union of verified records between active replicas. Thus, it is possible to use a 'leaderless' replication among replicas of a DataCapsule by using an anti-entropy background synchronization protocol.

At the core of any replication algorithm is a compact representation of the local DataCapsule state; such state is defined by the records of DataCapsule that a log server possesses. Using a compact representation allows for efficiently carrying out state reconciliation using anti-entropy gossip protocols. We discuss two algorithms below for such state reconciliation that primarily differ in the representation used. In our current prototype, we use the first algorithm for its simplicity.

The first algorithm is relatively simple and treats the records in a DataCapsule as a bag of items, each identified by the `headerHash` of the record header. A log server sorts the records by the `headerHash`, and creates an in-memory Merkle tree composed of the sorted hashes.[33,34] All replicas of a DataCapsule maintain such Merkle trees with their individual local states. A log server can then start syncing with a remote log server by traversing this Merkle tree—starting from the root and then only selecting sub-trees for which the top-hash differs from the remote hash. After a number of back-and-forth steps of exchanging hashes of sub-trees, the initiating log server can ascertain the additional records that it can retrieve from the remote log server. The log server then requests records from the remote log server, along with appropriate proofs, which the remote log server returns in the form of a RecContainer.

The second algorithm uses the relationships among records, as defined by the hash-pointers. The local state of the DataCapsule, that may be full of branches and/or temporary holes, can be efficiently and uniquely described in the form of a set of hash pairs (see Figure 3.3). The first hash in a hash pair represents a record that either has zero or more than one (but not exactly one) successors as present in the local state; the second hash is a record that can be reached to by following the `prevHash` pointers back from the first hash and is a candidate for being the first hash of another hash pair.

The algorithm itself has two phases: the first phase attempts to fill the temporary holes, and the second phase attempts to synchronize branches. At the beginning of first phase, the log servers send each other the unique representation of their local replica of a DataCapsule. After the exchange of the representation, each log server knows the sequence of contiguous records that it needs from the other log server to fulfill its own holes. After this partial synchronization is done, the log servers then update their own unique representation and initiate the second phase, where they do a similar information exchange.

[33] Such a Merkle tree for sorted hashes can also be viewed as a prefix-based tree, where items in a sub-tree share a common prefix. A representation using prefixes is a lot simpler in terms of implementation.

[34] Note that this is not to be confused by the DataCapsule hash-pointer structure that may look like a Merkle tree in certain cases.

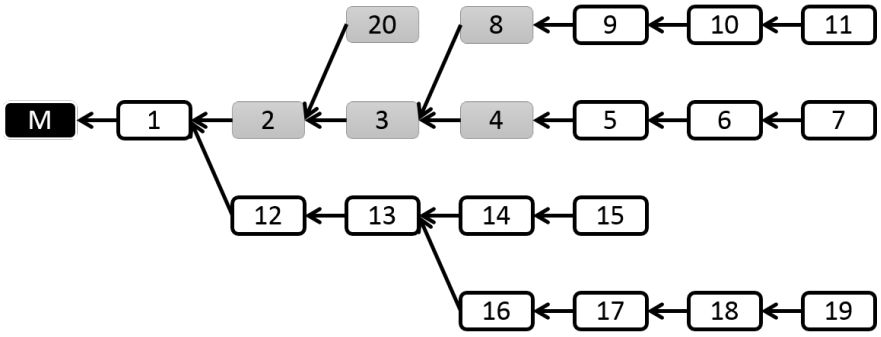


Figure 3.3: An example of a very branched DataCapsule; grayed records are the records that are not present in the local state. A unique representation of the local state (accounting for missing records) would be: $sorted((19,13);(15,13);(13,M);(7,4);(11,8))$. Note that even though, for example, record 4 is missing from the local state, a log server knows about the existence of such record including its `headerHash`. Further, the pair $(13, M)$ exists in the state because based on the locally available information, the log server doesn't know that a record like 2 exists.

The algorithm requires two phases because the local replica of a DataCapsule, when represented as a directed graph, may have more than one weakly connected components.[35] For example, in Figure 3.3, the local state will have three weakly connected components. As such, there maybe branches that the log server may not even know about after the first phase.[36] The algorithm takes a conservative approach to minimize extra data transfer between the log servers, and waits for the second round to infer information about the relative placement of missing records. After the first phase finishes, the two log servers synchronizing with each other have the same number of weakly connected components.

[35] `http://mathworld.wolfram.com/WeaklyConnectedComponent.html`

[36] For instance, in Figure 3.3, the log server cannot infer any information about the placement of a contiguous sequence of records represented by the hash-pair $(20, 2)$ that it may receive from a remote log server.

Chapter 4

Making DataCapsules Practical: The Engineering

In the previous chapter, we outlined the core design of DataCapsules. However, a number of issues need to be addressed in order to make DataCapsule vision a reality. For example, clients in real world have limited local state and they cannot work with a proof that requires sending an unbounded number of hashes over the network. Further, application developers desire richer interfaces than the native DataCapsule interface. And finally, the operations on a DataCapsule ought to be carried out over an insecure network and it is important to address a number of subtle security issues. In this chapter, we address these practical concerns. Note that these are more than mere optimizations and essential to making DataCapsules practical and integrate well with the GDP.

4.1 Managing application-specific requirements: Beyond a hash chain

The structure of the DataCapsules, as we have described thus far, is a very skewed Merkle tree and is essentially a hash-chain (see Figure 3.1b). 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 additional hash-pointers in the record header that can point to any arbitrary record older than a given record. Using such additional 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 DataCapsule. This kind of configurability allows for aligning proof generation with application specific access patterns.

Using hash-pointers has a number of implications: (1) 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. (2) For a reader reading old data, the overhead of verification is dependent both on the linking structure and the access pattern. (3) 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. 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 filesystem checkpoints; (2) a Merkle tree does not work very well with DataCapsule-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, such as streaming video stored in a DataCapsule; (4) Merkle trees require relatively larger state to be maintained on writer, which may not be available on all devices.[1]

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', i.e. tuning the work needed on writer vs reader.

For formulaic strategies, a writer can calculate in advance how long a specific `headerHash` will be needed.[2] With the constraint that a writer can only store limited number of hashes in its local state, we propose three generic strategies that can be used by applications as a starter template (see Figure 4.1).

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 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

[1] 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.

[2] In our implementation, a writer maintains a small state machine that dictates when a specific `headerHash` is no longer needed.

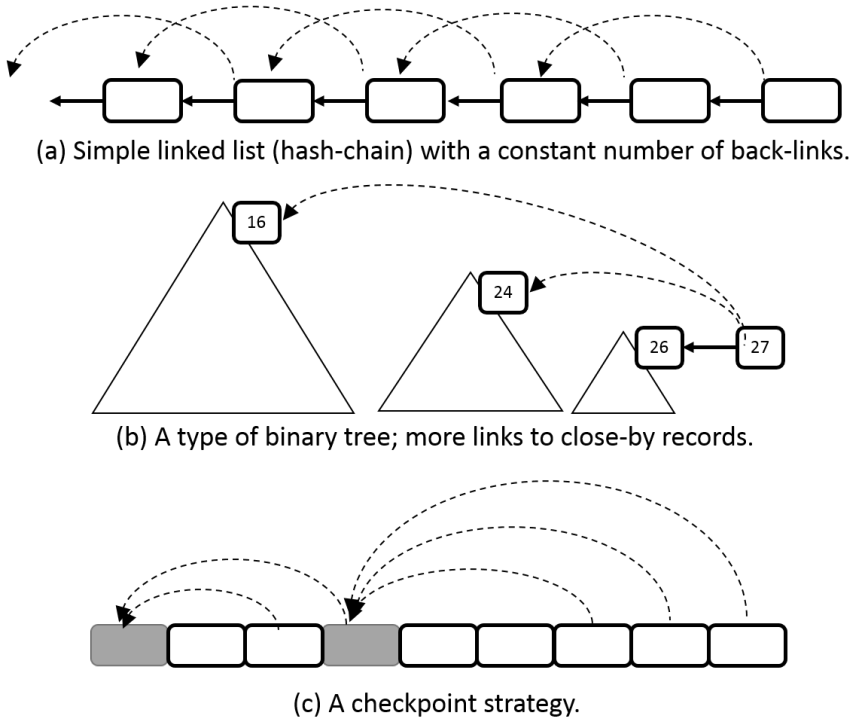


Figure 4.1: Generic strategies for creating additional hash-pointers.

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. While this strategy requires more bookkeeping on the writer side than a simple linked list, proof sizes are logarithmic sized. Such a design is a good fit for readers that perform random reads.

3. A checkpoint strategy. Consider an application that performs checkpoints every n records by dumping its local state to the DataCapsule; 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 key-value store CAAPI, that we will describe in next section, 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 DataCapsule.

4.2 Building a CAAPI

In this section, we describe our experiences developing two real-world CAAPIs. Recall from subsection 2.4.2 that a CAAPI (Common Access API) is a library that provides a more familiar interface but uses DataCapsules underneath for persistent storage.

4.2.1 A key-value store with history and snapshots

As part of TerraSwarm (recall from subsection 2.5.1), we observed a specific usage pattern. A significant number of applications used time-series data with a small JSON string encoding a dictionary that contained key-value pairs. Especially small battery powered environmental sensors generated data of the following type: `{ "timestamp": "2019090307233404", "temperature": "29.0", "humidity": "65", ...}`. Such data was then parsed and store in a database for retrieval at a later time.

To address such a usage pattern, we designed a key-value store CAAPI that stored the underlying data in a DataCapsule as it arrived. Our CAAPI is mostly targeted at time-series dictionaries with a few keys and small data values, but the values changes often. The CAAPI in the form of a Python module that a user can import. The module provides a dictionary-like interface that resembles Python's built-in `dict` class with two additional features: (1) the key-value store is backed by a DataCapsule and thus persistent across program restarts, and (2) in addition to providing the current value of a key, it also supports asking historical values of a key.

Each update to the key-value store is internally mapped to a new record in the DataCapsule. Additionally, the CAAPI maintains periodic checkpoints with a varying granularity similar to the `dump` utility in Linux. Tunable granularity of checkpoints allow for achieving an acceptable balance between checkpointing often vs minimizing storage overhead.

A future enhancement to the CAAPI would be to separate the checkpointing in a different DataCapsule. This allows for adjusting the checkpointing needs based on access patterns and application specific needs. This CAAPI was developed before we had support for more extensive linking with additional hash-pointers. However, a checkpoint-style linking strategy, as described in the previous section, would have been extremely useful to guide the proof-generation with application-specific access-pattern.

4.2.2 A filesystem for machine learning

To help with the use of DataCapsules and the GDP for fog robotics (recall from subsection 2.5.2) and machine learning applications in general, we developed a CAAPI for use with TensorFlow [36]. Our TensorFlow CAAPI is in the form of a C++ library that can be loaded at run time and works with existing TensorFlow code. The CAAPI allows a user to utilize the GDP and DataCapsules for all file system access by merely specifying a GDP path instead of a local file system path.

TensorFlow has support for custom file system plugins. Such custom file system plugins are required to implement a narrow interface to a file: given a path, the plugin should be able to return

a file-handle in the specified mode (read-only vs read-write); writes are append-only and reads can be random access. Since the only required mode of writing is an append, it is quite straightforward to map the operations on a file to a DataCapsule. We must reemphasize that the append-only nature of a DataCapsule does not prevent us from creating an arbitrary mutable file interface.

For our specific implementation, we map regular files to DataCapsules in a one-to-one mapping. We maintain the directory structure for the entire file system in a root DataCapsule that identifies a mapping of full path names to DataCapsule names. Sub-directories are merely just an entry in the root DataCapsule. For TensorFlow code, users specify the file paths as `gdp://root-DataCapsule-name/path/to/file`, where the `root-DataCapsule-name` is a hexadecimal encoding of the DataCapsule that represents the root directory structure. Such paths can be used for the entire life-cycle of a machine learning application; some examples are training data, logs for model-training progress, checkpoints for trained models, and actual application data that the trained model operates upon.

Our current design, while simple, has a limitation that the consistency guarantees are limited to single files; there are no guarantees of consistency across files. Such behavior may not be suitable for arbitrary applications, but this works just fine for machine-learning applications. While an alternate design of laying out the entire file system in a single DataCapsule would have allowed us to get stronger guarantees across files, our current design is simple and effective. Further, it allows detaching individual files off of one file system and attaching them elsewhere with minimal data copying.

4.3 Securing network operations: The GDP protocol

Even though DataCapsules provide an intrinsic security for information, the actual communication must still be translated to fit the underlying transport provided by the GDP network.[3] The GDP protocol is the materialization of operations on DataCapsules to GDP PDUs that the GDP network can move around.

The GDP protocol is responsible for not just transferring parts of a DataCapsule around, but also a number of status message such as acknowledgments. While the security of datagrams containing DataCapsule fragments can be reasoned about by using DataCapsule primitives, various status messages and acknowledgments do need some mechanisms to provide an end-to-end guarantee.

To illustrate why the security of status messages and acknowledgments is necessary, consider the case of an append operations; the writer must be able to securely assert that an append operation actually reached a log server designated by the DataCapsule owner. A malicious on-path adversary should not be able to silently drop such append requests and send an acknowledgment to the writer pretending to be the correct log server (and framing an honest service provider in the process).

[3] Recall that the GDP network provides a UDP-like datagram communication for applications.

There are a number of communication protocols and 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). Using existing tools without modification does not work for at least two reasons:

First, existing secure channel protocols are designed for host-to-host communication. The GDP network is an information-centric network where one communicates directly with the appropriate DataCapsule and not a physical host.[4] While it is true that a DataCapsule is eventually hosted on a physical host that is replying on behalf of the DataCapsule, a user does not necessarily know of the host's identity in advance. A client could exchange some information with the underlying host, but that brings us to the second challenge.

The second challenge stems from the fact that DataCapsules are replicated across the network. A client's communication with a DataCapsule is inherently datagram based *anycast*. 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 DataCapsule, the client must pause and do explicit state negotiation with the new server 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.[5]

With this background, we needed to devise transport layer security mechanisms in the GDP protocol. We present a high level overview next.

4.3.1 GDP protocol design principles

Before discussing details of the protocol, we need to understand the type of messages for DataCapsule operations. Viewing the DataCapsule as a storage interface, various DataCapsule 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 DataCapsules, examples include DataCapsule-creation, appends, messages for synchronization among log servers, etc.
2. Reads, or operations that query existing state. DataCapsule operations that fall in this category are: reads, subscriptions, queries for metadata, etc.

For any DataCapsule 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

[4] Recall that the DataCapsule operations from a writer or a reader are directly addressed to the DataCapsule and not a host; the network finds the closest physical node advertising for the given DataCapsule and delivers the request to such node.

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

requests or responses can last an arbitrary amount of time and can span multiple message exchanges. In the most generalized sense, we can designate entities initiating requests as clients, and those that respond as servers.[6]

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 authenticity proof for the server to verify the identity of request creator and the contents of the request. Such requests can be forwarded by arbitrary entities without tampering, and everything should still work.[7]
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 because a log 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 DataCapsule should not be affected.

4.3.2 The GDP Protocol

With the above design principles and the fact that a DataCapsule is an authenticated data structure with any state update operation designed to be idempotent, the key task of the GDP protocol is to ensure *secure acknowledgments* and facilitate 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 DataCapsule's name and a log server is merely serving on behalf of the DataCapsule. The only private key that a log server has is its own private key.[8] Thus, if the log server signs the

[6]This terminology may create some confusion with GDP clients and log servers, but we hope this will be clear from the context. Note that a GDP client is always a client, but a 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.

[7]For example, in our design, a log server forwards an append request created by the writer to other log servers during replication and to subscribers subscribing to the DataCapsule. Even though replica log servers and subscribers receive the 'state update' from the original log server acting as a proxy, they should still be able to verify the authenticity of the append request.

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

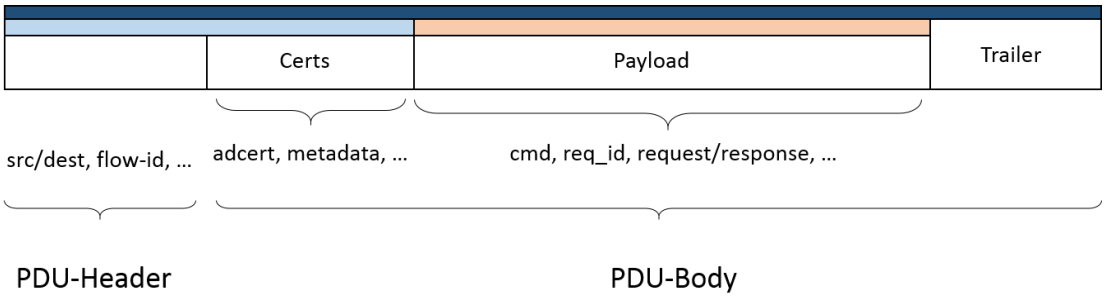


Figure 4.2: The GDP protocol PDU. PDUs are split in two parts: a *PDU header* and a *PDU body*. The PDU Header primarily contains 'source' and 'destination' information and is used by the GDP network to route messages. The PDU body is split into three parts: *certs*, *payload* and *trailer*. *Certs* contains certificates/delegations and metadata(s) for key-exchange; *payload* contains the actual request/response to/from a log server; *trailer* contains an HMAC (or a signature) from the log server.

acknowledgments 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 DataCapsule in question.[9]

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. This is possible because a client's message has another layer of security already built-in because of the DataCapsule construction, and GDP protocol only needs to protect the responses that come back from a log server.

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 on secure acknowledgments, the client also includes its own metadata along with a request; the client and the server 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.[10,11]

[9]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 the 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 cannot be replayed as a response to another request. Note that this requestID is not to be confused with a `seqno` used in DataCapsule records.

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

[11]In actual practice, only the first request from a client includes client's metadata. If a subsequent request gets redirected by the GDP network to a different log server, such a log server falls back to using a signature but indicates in the response that the client should send its metadata again.

Figure 4.2 shows the GDP PDU format. The PDUs defined by the GDP protocol 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 server; *trailer* contains the HMAC/signature from the log server.

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.

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: The GDP payload contains compressed, network efficient versions of 'DataCapsule-create', 'append', 'read', 'subscribe' and their corresponding responses. An essential element of this compression is to avoid transmitting information that can be securely generated on the other side.[12] 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. Note that any signatures/hashes are over a serialized version of the in-memory representation, and not the on-the-wire bytes. This ensures that any accidental/intentional incorrect assumption of shared state does not compromise security.[13]

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, 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.

[12]Note that this payload compression is actually implemented as part of the general RecContainer compression that we discussed in the previous chapter (subsection 3.3.2).

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

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. While these ideas aren't strictly necessary for our initial prototype, these need to be addressed for a practical adoption of DataCapsules.

Secure (private) 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 DataCapsule-writer.[14] 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 DataCapsules than purely communication based streams, because the data storage aspect of DataCapsules 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 [54].

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 DataCapsule will be absolutely secure and trustworthy to protect the decryption keys. 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 DataCapsule-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.

DataCapsule attach request

DataCapsules, 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 *DataCapsule attach* request. A DataCapsule attach request is a special type of request addressed to a principal that's not a DataCapsule (a service, application, user); it specifies a number of input and/or output DataCapsules that the principal should read from/write to. Such a request acts as a glue to enable composability of DataCapsules without compromising on security.

A DataCapsule 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

[14] The issue of absolute correctness of time on the DataCapsule-writer is out-of-scope of the current work.

of just calculating an HMAC. In addition to providing a way to compose services, a DataCapsule attach request also provides for a way to perform in-band exchange of decryption keys that are required to access the DataCapsules included in the request.

DataCapsule truncation

DataCapsule truncation is the marking of old data 'safe to delete'. The notion of 'old' can be either described with respect to time or number of records. Since 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. DataCapsule 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.

DataCapsule truncation is an essential element of the service provider model for DataCapsule hosting. In fact, DataCapsule owners may base their decision process of choosing a service provider based on contractual guarantees that truncated portions of a DataCapsule are actually deleted from all persistent state. Such guarantees are essential for a widespread adoption of DataCapsules by enterprises and companies bound by legal requirements on data retention.

Transient DataCapsules

Taking DataCapsule truncation to an extreme level, a creator can designate a DataCapsule to only require a small number of records. Such a DataCapsule 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 and extends the utility and flexibility of DataCapsules.

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 DataCapsules 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 an 'append' and sends a secure acknowledgment as usual. However, if the writer cannot 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.

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 DataCapsules could be limited to *private* routing domains that require a cryptographic authorization to join. The GDP network, as we will describe in next few chapters, does provide such support.

Preventing side-channels at DataCapsule level

The information-leakage problem is especially challenging for public DataCapsules because a third party can monitor communication in real-time by just subscribing to the appropriate DataCapsule; 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 DataCapsule also makes things easier for an adversary, since 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.

A privacy centric writer can partially alleviate this side-channel leak by using 'pointer-records' to overlay a single DataCapsule ('primary' DataCapsule) over a collection of 'secondary' DataCapsules 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 DataCapsules). In this scheme, a pointer record contains N, s and DataCapsule names for each of the m DataCapsules, with all of this information encrypted using the encryption key that the writer would otherwise use to encrypt records in a normal DataCapsule.

Only a reader with the appropriate decryption key can decrypt the information in pointer-records, and follow the pointers to the secondary DataCapsule to retrieve the real payload. An adversary subscribed to just the primary DataCapsule 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 DataCapsule 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 DataCapsules can perform some correlation of individual DataCapsules as constituent DataCapsule of an overlay DataCapsule, however it can never be sure of the missing information.

Chapter 5

The GDP network: Delegated Secure Flat Routing

In this chapter, we present the design of a secure and scalable routing network with flat cryptographic names, called the GDP network. Recall from chapter 2 that the GDP network provides a routing fabric for enabling the DataCapsule infrastructure. It is also available for direct use by end-users if they so desire.

The GDP network is an information-centric network: instead of routing messages between hosts identified by structured IP addresses, it routes information between cryptographic principals named by flat 256-bit long names (GDP names). Recall that GDP names can represent objects and hosts alike; objects could be services or information objects like DataCapsules. At an operational level, the GDP network is composed of many *routing domains* that do not necessarily trust each other, yet coexist together. Such coexistence occurs despite the lack of a single administrative entity performing namespace allocation as in IP-based networks.

A key aspect of the GDP design is the service provider model, which translates to the following in the GDP network context: users maintain ownership of their names, but can *securely delegate* hosting of content and services to infrastructure owners/operators. Using flat GDP names compliments this design philosophy by providing location independence for delegated names.

From a security perspective, flat namespace routing by itself has traditionally been considered a challenging security problem. Thus, allowing such delegation merely seems to add to the problem. The security challenge primarily arises from the fact that users pick their own names, which makes it very easy for malicious actors to make spurious claims to arbitrary names or lure network traffic via themselves by falsely claiming to offer a good route to an arbitrary name (see Figure 5.1). Even with hierarchical names assigned by official administrative entities, such as in the IP network, routing security can be compromised if malicious entities get access to the core of the network [55], [56]. IP network administrators maintain day-to-day security of their networks by filtering abnormal BGP announcements and by an elaborate set of manual checks and balances. Such mechanisms do not work for flat namespace networks, because these networks, by their very nature, expose the

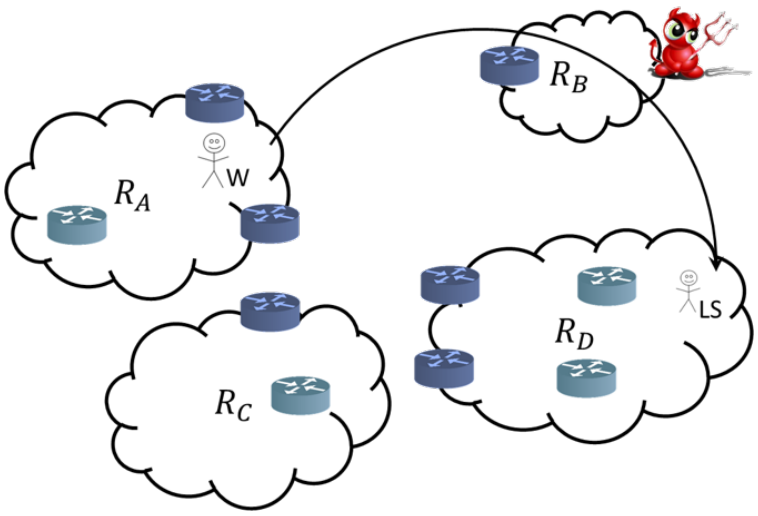


Figure 5.1: Adversarial infiltration in a network with routing domains R_A, R_B, R_C, R_D that do not have any incentive to trust each other. A client W does not even know whether a log server named LS is legitimately placed in a R_B or R_B is just pretending to have a close-by copy.

routing protocols to potentially malicious network participants. We will discuss previous attempts to address this problem in more detail in the next section.

The security guarantee that the GDP network provides is: given a GDP PDU with a specific (src, dst) pair comprised of flat 256-bit addresses, the GDP network will attempt to efficiently and securely route the PDU from src to dst—even in the presence of adversarial entities who may claim to possess any given name and control parts of the network. Note that src and dst are bound to hardware positions within the network by a process of secure advertisement (see a formal threat model in section 5.2). In addition to protecting communication paths, the GDP network also enables native support for controlling the *scope* of a names: users can specify policy on whether a given name should be visible globally or just within a routing domain, and the GDP network enforces such policies. Such a feature, for example, enables DataCapsule owners to lock down the information to private domains.[1]

We discuss the design and implementation of the GDP network as an overlay network in this chapter, however, it should be viewed as a tunneling protocol inside the conventional IP networks for ease of deployment. In fact, individual routing domains can choose to run the GDP network as a native protocol and still be compatible with the rest of the system.

[1]One can view this as a native GDP network-firewall, although we avoid such terminology because of subtle implications that the term 'firewall' has.

5.1 Background and related work

The GDP network builds upon almost two decades of distributed systems knowledge, and almost every individual component of the architecture has been studied elsewhere in the past. The GDP network merely fills the gap between existing systems and what is needed to achieve the high-level goal of providing a scalable and secure routing between flat names that support a service-provider oriented, multi-stakeholder view of computing landscape, such as those prevalent in edge computing.

5.1.1 Relevance to current Internet infrastructure

The organizational structure of the GDP network is inspired from ISPs and ASes (Autonomous Systems) from IP routing. The organizational hierarchy is intentionally designed to mimic the physical infrastructure to ensure ease of deployment. It also supports well understood economics models with real-world entities. However, the GDP network differs from the current Internet in that it enables anyone to start a new autonomous organization (called a *routing domain* in the GDP network) as opposed to authoritative, bureaucratic, and legislative structures of the current Internet [57].

Routing security is a challenging task even for IP networks, mostly due to BGP [55], [56], [58]. The most effective solution in IP routing is to disallow an adversary to join the network as a route-advertising entity (AS) in the first place. Such a solution is inherently limiting in our use case, since even normal users *must* advertise the names they claim to own in flat namespace networks. A number of past systems have used *network assigned* names even for flat namespace routing, however such solutions restrict the autonomy of individuals especially in global systems such as ours [59]. Recent work such as Secure BGP [60] aims to alleviate IP's routing security challenges; we use a similar principle of using cryptographic tools to control who can advertise for a given address.

In the IP world, the prime method of delegation of services is a redirection via DNS to an appropriate hosts managed by a service provider. Using DNS as a redirection mechanisms is useful, however it remains a coarse grained mechanism for providing geographically relevant services [61]. Further, DNS is an imperfect mechanism for asserting name ownership and is not impervious to geopolitical disputes, litigation, censorship, or adversaries. Of course, we do not deny the usefulness of human readable names, but our goal is to enable security at the network level itself.

A relevant IP technology worth mentioning is Mobile IP [62]. Mobile IP allows users to keep a fixed network identity even when moving around, but it has seen rather slow adoption rate [63]. Another such technology is IP anycast; IP anycast allows for multiplicity of service providers, however it is limited to a small fraction of the namespace; in practice, IP anycast is mostly limited to large enterprises with substantial resources and influence [64].

5.1.2 Previous academic research

The design of the GDP network is influenced by a number of previous academic systems spanning a wide range of goals and functionalities. We discuss a few noteworthy categories below.

A Naming and ICNs

Many previous distributed storage systems (SFS [65], OceanStore [17], etc.) have used cryptographically derived names for their *self-certifying* property to achieve integrity verification of the information objects. The concept of using such an identity as the network address and making information objects first class network entities is at the core of an entire class of networks called Information Centric Networks (ICN). The GDP network allows for direct addressing of both objects (such as DataCapsules) and hosts, and thus fits the description of an ICN.

A high level summary of the commonalities and differences among various well known ICNs is provided by Ghodsi *et al.* [34]. A few notable systems are CCN [66], NDN [67], DONA [68], PRISP/PURSUIT [69], NetInf [70], TRIAD [71]. A number of Future Internet Architecture (FIA) projects such as NDN [67], MobilityFirst [72] and XIA [73] also propose similar designs of higher level abstractions than IP. Research on Distributed Hash Tables from the early 2000's also fueled the development of many of these ICNs [74]–[77]. Our approach of making named objects as first-class citizens on a network is certainly inspired by these past systems.

A number of general ICN challenges [78] and those specific to ICN security [79] have been identified in the form of RFCs by ICNRG working group established by IRTF. As echoed by these RFCs and highlighted by Ghodsi *et al.* [34], control over information and privacy of access remain important research problems. In order to provide such control, the GDP network requires explicit authorization in form of cryptographic delegations before a host can even advertise a given named content; this is in contrast with the traditional ICN design with *pervasive caching* [34].

B Flat address space routing

Flat address-space routing has been extensively studied in the past, and a number of distributed storage systems use structured peer-to-peer networks to implement such routing [74], [75]. Early flat namespace routing systems focused on scalable, self-organizing routing schemes that work without the route aggregation property used by IP networks. A number of these routing schemes follow a variation of the following algorithm.

To begin with, each node is assigned a name from the flat namespace. When a new node joins the network, it follows a joining algorithm which results in this new node establishing connections to a number of other existing nodes in the network. Each node maintains a list of directly connected nodes in a local 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 systems that use the scheme. A common distance metric, for example, is based on a combination of common prefix length (between destination address and neighbor IDs) and ping latency.

These early systems didn't necessarily have suitable protection from adversaries and had limited deployment potential in the open Internet. A survey by Urdaneta *et al.* [14] on DHT security gives an excellent overview of techniques used by various DHTs, and they also conclude that securing decentralized systems is not an easy task.

In the GDP network context, the biggest challenge posed by such security issues is that an adversary can advertise for a name of an arbitrary victim, and then 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 federation goal of the GDP 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 and perform sophisticated passive network traffic analysis without detection.

The GDP network is designed with two security goals: routing security and control over *scope* of routing. While we don't claim to have discovered a solution to generalized problems, we rephrase the problem definition itself to use explicit delegations and client-side policy specification, thereby providing the security properties described above.

C Future Internet architectures

A number of projects targeting Internet architectures of the future target similar goals as the GDP network. Named Data Networking (NDN) [67] is a prominent ICN that takes a very different approach of using human readable hierarchical names instead of the flat names as used by the GDP network. We chose a different approach than NDN because cryptographic names enable an integrated key-management solution, where the trust anchor is the name itself.

There are some broad architectural similarities between the GDP network and MobilityFirst [72] in terms of relying on a global lookup service. MobilityFirst uses GUIDs that resemble GDP names. However, the names are essentially assigned by the network which does not fit our goal of complete autonomy without depending on a trusted centralized authority. The GDP network generalizes the naming to include hosts, services, etc. in a manner similar to XIA [73]. However, while both XIA and MobilityFirst aim to target routing security, they have limited support for some of the other goals important to us, such as delegation to service providers, organizational structure, and network isolation.

SCION [80] is an attempt to fix the problems of the current Internet and incorporate important properties such as routing security and network isolation into the current routing architecture. While impressive in their design and goals, SCION is fundamentally different from the GDP network; the

GDP network is primarily an overlay network relying on cryptographic security provided by flat names in combination with the organizational structure provided by trust domains.

Probably DONA [68] is one of the closest existing systems to the GDP network in terms of architecture. The GDP network is, however, more explicit about the organizational structure, and provides more flexibility and autonomy for administrative domains. For intra-domain routing in the GDP network, verifiable routing information is acquired on-demand from a distributed database which is similar to *resolution handlers* (RHs) in DONA. Note that such a combination of on-demand routing lookup and a separation of routing from actual forwarding has been used to allow for flexibility of routing and simplification even in IP routing [81], [82].

Finally, note that most of these Future Internet architectures by themselves are only a partial solution to the problem of secure ubiquitous storage platform that we are attempting to address in the GDP, primarly because they do not provide any semantics for information update.

5.2 The GDP network threat model

In this section, we highlight aspects of the general threat model discussed in chapter 2 specific to the GDP network. We assume typical security properties of cryptographic primitives (signatures, hashes, etc.), and that the clocks are synchronized to second-level precision to allow for short lifespans of our cryptographic delegations. We also assume that the private keys are protected adequately by the appropriate entities.

A An open federated network

The GDP network allows anyone to participate in the infrastructure. This includes malicious entities that may introduce their own routing domains and/or GDP routers, or simply join open routing domains. An adversary can create an arbitrarily large number of their own objects and advertise for them. But under no circumstances can such an adversary (1) insert itself in the path of other people's communication *at will*, and (2) pretend to have a name that it doesn't own. When the GDP network is operating as an overlay, we generally assume a baseline level security for the underlying protocols such as TCP/IP (i.e. no IP-hijacking or BGP routing attacks).[2]

Clients that do not want to setup their own routing infrastructure use services provided by a *trusted* routing domain. The trust implies that the routing domain (and the associated infrastructure) will follow the correct protocol, i.e. the routing domain will not snoop on data to perform side-channel attacks, and will respect the policies embedded in cryptographic delegations (e.g. ensure the advertisements don't leak outside the desired boundaries if dictated by the policies).

The onus of appropriate delegations and policy specification for a given object (such as a DataCapsule) is on the object owner. Further, the owner who delegates hosting of an object to a

[2]Note that existing mechanisms to secure TCP/IP traffic are complimentary to the security guarantees of the GDP network. As an example, a routing domain with infrastructure spread over data centers around the world can (and should) use technologies such as VPN for securing communication through the public Internet.

server also must trust the routing domain that the server connects with.[3] We assume that the object owner makes the 'correct' decisions; i.e. takes into account what routing domain does the server connect to, and such.[4] Similarly a client trying to reach a certain object in a foreign routing domain has to trust the owner of the object to pick a trustworthy service providers; e.g. in Figure 5.1, the client W has to trust both R_A and R_D, but not third-parties such as R_C or R_B.[5]

B Adversarial infiltration in a routing domain

Both security guarantees of the GDP network (routing security and network isolation) hold as long as the source and destination routing domains are trusted. But these guarantees are diminished slightly when an adversary infiltrates inside a routing domain.[6]

When an adversary infiltrates a routing domain by compromising a GDP router, the routing security properties still hold true to some extent; the adversary can only affect traffic flows that pass through the compromised GDP router (e.g. for all directly connected clients), but cannot alter existing routes at arbitrary places in the routing domain.[7] In addition to observing traffic on these flows (as a man-in-the-middle), the adversary could further try to subvert the network isolation properties by ignoring any policy specification and directing the traffic elsewhere in the system. The GDP network mechanisms, as we will discuss later in the dissertation, provide remedies to ensure that such traffic redirection cannot be done in-band via the GDP network; an adversary can certainly create covert communication channels out-of-band (such as by opening a direct TCP channel to a server elsewhere, or by tunneling it inside other GDP network traffic).[8]

A slightly different type of adversarial infiltration is by compromising an existing entity (e.g. a client) that has valid authorization to join a private routing domain. Once again, such compromised entity cannot alter routing paths. Similar to the situation with a compromised GDP router, an adversary *can* subvert network isolation properties by actively probing private objects and create a covert channel to the outside world, but cannot use in-band GDP network routing for such leakage.

Thus, the goal of the GDP network is fairly modest in case of breach inside routing domains, i.e. ensure that a single point of entry for an adversary in a large routing domain can do limited damage. The GDP network does not *magically* protect against a malicious client trying to *bridge* two disjoint

[3] Specifically for the case of DataCapsules, this means that the DataCapsule owner must trust both the log server *and* the routing domain that the log server connects with. Once again, this trust is only limited to service providers (log servers and routing domains) providing appropriate services without performing sophisticated side-channel attacks.

[4] This is a form of transitive trust where the object owner trusts the server owner for its choice of routing domain.

[5] This requirement of trust from the owner is consistent with our overall GDP threat model, where readers trust the DataCapsule owner for correct policy specification.

[6] Recall from chapter 2 that routing domains could be either public or private, i.e. require a cryptographic authorization to join the domain.

[7] In case of routing domains with a custom network topology hand-crafted by the routing domain administrator, this may not be true and a compromised GDP router may actually be able to subvert the routing state of the network. We will discuss such special case in the next chapter along with the discussion on custom routing topology.

[8] Note that such out-of-band traffic can be suppressed with cooperation of underlying routing/switching infrastructure.

private routing domains for subverting network isolation by blocking all information channels,[9] but it guarantees that the malicious client cannot publish objects under the same name elsewhere in the system in order to attract traffic from benign users, or use GDP network mechanisms against itself for that matter.

C Miscellaneous mischievous behavior

The GDP network does not provide mechanisms for access control on objects at the server level; any filtering for requests based on some access-control list on servers is certainly possible but out of scope of this dissertation. Similarly, any guarantee on the content/correctness of datagrams delivered by the GDP network is out of scope of our current work. While the GDP network exposes $AdCerts$ to applications, these $AdCerts$ are intended to assist application writers with identity verification and do not provide any security guarantee on the content of datagrams.

The GDP network architecture provides only limited defense from a DoS (Denial of Service) attack. There are a number of ways an adversary can launch DoS attacks: compromise a GDP router and stop forwarding any traffic, compromise a GLookupService and prevent lookup of routes, create unbounded new names to cripple the global GLookupService, send requests to a large number of destinations to keep GDP routers busy with fetching new routes, or traffic flooding to specific destinations, just to name a few. As demonstrated by current Internet, rate limiting by economic means is a way to curb such behavior.

5.3 The GDP network overview and architecture

The routing fabric provided by the GDP network is comprised of *GDP routers* and overlay links between them. GDP routers route datagrams with 256-bit long flat source and destination addresses. The routing infrastructure is partitioned into administrative entities called *routing domains*. End-users connect into the infrastructure by finding a routing domain that they trust and connecting to a GDP router of that routing domain. They may then advertise names that they either *own* or are delegated to advertise.[10]

The interface that the GDP network provides is that of secure datagram delivery between endpoints. In many ways, this is similar to the guarantees provided by UDP, except for one big exception: it is non-trivial for a user to spoof an arbitrary source address. Similar to UDP, datagrams may be dropped, reordered, or re-transmitted. With appropriate application level constructs, it is certainly possible to devise byte-stream interfaces like TCP, however we consider that as out of scope of the current work.

[9] This, in some sense, is a fundamental challenge with any *perimeter defense* and is not exclusive to the GDP network.

[10] We use the terms 'connection' to refer to a generic protocol that provides the notion of a 'channel', e.g. TCP, SCTP, dTLS. Our definition of 'connection', thus, means establishing such a channel followed by some GDP network specific handshake—we describe such handshake mechanisms later.

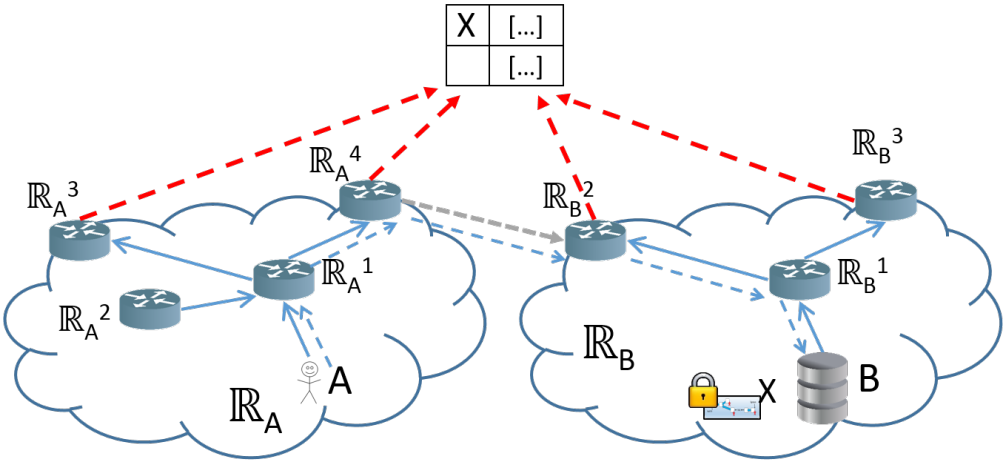


Figure 5.2: Two administrative (routing) domains, $\mathbb{R}_A$ and $\mathbb{R}_B$ in the GDP network. A client A in $\mathbb{R}_A$ attempts to reach object X hosted on a server B. The GDP network ensures that (1) an adversary cannot claim to have a nearby copy of X and influence the routing state; (2) X is not made available outside of $\mathbb{R}_B$ if the policies dictate so; and (3) the infrastructure can find a route to X if X is indeed publicly available.

For those users that are interested in restricting the visibility of names to specific parts of the network (e.g. a routing domain), the GDP network also exposes the boundaries of routing domains to end-users so that they can specify appropriate policies.[11] Such explicit availability of infrastructure ownership information distinguishes the GDP network from traditional routing, where end-users are rarely made aware of who owns the infrastructure. GDP network users can, however, ignore this additional information in the simplest of cases.

Note that while we discuss multiple distinct stakeholder roles in the infrastructure, the same real world entity might be playing multiple of these roles at the same time. As an example, a user who does *not* want to outsource object hosting to a third party can run his or her own servers and a routing domain with GDP routers. In fact, we envision that users with stricter QoS requirements and heightened privacy needs will, in fact, run their own private infrastructure that is still connected to the GDP network in a way similar to how private IP networks complement the global IP network.

5.3.1 Concepts: Names, owners, delegations, and advertisements

We first review a few key concepts. Some of these were introduced earlier in dissertation, but we generalize those concepts in context of the GDP network. Recall that every nameable entity in the GDP has a GDP name, which is a flat 256-bit name. In the GDP network, this same GDP name

[11] We don't currently provide a formal mechanism for acquiring such information on infrastructure ownership. At present, we assume that users acquire such information out of band, but they do have access to the said information in some format that utilizes the *OwnCert* mechanism.

serves as the network *address* for the object. Also recall that GDP names are derived from the *metadata* that contains a public signature key.

The key abstraction in the GDP is a DataCapsule, but we generalize the concept to an *object* in the GDP network; an object could either be a named information chunk (DataCapsule, or even a partial replica) or a named service instance. Any object on top of the GDP network defines an API defined as a set of RPCs encapsulated within GDP network datagrams. This API could be as simple as `GET` and `PUT` for interacting with small information objects. It could be a bit more involved for partial updates to information objects (e.g. a GDP protocol to interact with DataCapsules). It could be even more complicated, providing a set of operations for accessing service instances. The GDP network merely involves delivery of datagrams and is not concerned with the nature of the RPC.

Each object has a specific *owner*. The owner of an object is typically a human who has policy-making authority on the object and is ultimately responsible for the life-cycle of the object. The GDP network view assumes that users generally outsource the hosting responsibilities for objects to others. The owner is the only entity in possession of the private signature key associated with the object, which is how the owner exercises its policy making authority.

Servers host objects. The owner of a specific object explicitly delegates the hosting responsibility to one or more servers and usually maintains either implicit or explicit economic agreements with such servers. In a more generalized model, servers are owned and operated by service providers. Object owners delegate the responsibilities to service providers who then delegate it to a subset of servers they operate. Depending on the exact semantics of the objects being hosted, these servers may coordinate among themselves to maintain the desired consistency semantics for objects. From a user's perspective, all replicas are *roughly* equal and the GDP network does not make specific guarantees as to which replica will a user be directed to. The only guarantees are that a replica within the same routing domain will be reached before any replica outside the routing domain.[12]

From an object life-cycle viewpoint, a *client* is a process running on behalf of an end-user (human or even an application) that cares about generating and consuming information objects, or generating requests to service instances and processing responses. Clients interact with such objects by making requests directly to the name of the object (and not to the server that hosts the objects).

Note that in general, *client* is a rather context dependent term. From the perspective of the GDP network as a whole, a client is any active computer process that connects to the routing fabric. Even servers are a type of client from this viewpoint. From the perspective of an individual GDP router, anyone who connects and advertises one or more names is a client. Even GDP routers can be considered clients when they connect to other GDP routers and advertise for all the names that they know of. Among other things, when a GDP router connects to another GDP router, it acts like a client and follows the same secure advertisement process as other network entities to advertise its presence.

[12]In case of multiple replicas of a DataCapsules, log servers hosting these replicas may engage in a replication protocol to ensure that replicas are not running behind. Additionally, for a log server to synchronize its copy of DataCapsule with a specific replica elsewhere, it must use the name of the log server that hosts such a replica.

Cryptographic delegations: As mentioned above, object owners delegate the responsibility of hosting objects to service providers by generating cryptographic delegations called *AdCerts*. An *AdCert*, such as $(X \rightarrow_A Y)$, is a signed statement by the owner of name X allowing a server (or a service provider in the general case) with name Y to host the object X. This delegation is done out-of-band, and can include policies such as "X should be visible only within certain routing domain $\mathbb{R}$".[13]

Making names available to the network: For a name to be available in the network, one must *advertise* the name. A client or server connects to a GDP router belonging to a specific routing domain over a TCP channel and *advertises* its own 256-bit name into the routing fabric by completing a cryptographic challenge/response to prove the possession of the secret key corresponding to the name being advertised. Followed by advertisement of their own name, servers advertise names of objects that they are delegated to serve by presenting the corresponding *AdCerts*. The result of this *secure advertisement* protocol is another type of cryptographic delegation called *RtCert*, which allows a GDP router to route traffic on behalf of the advertised names; we discuss the details in a later section.

5.3.2 Organizing infrastructure into routing domains

As mentioned earlier, the key organizing principle for infrastructure in the GDP network is that of a *routing domain*.

A Routing domain: Overview

Routing domains are essentially administrative domains representing resource ownership for routing infrastructure. The key infrastructure component maintained by a routing domain are GDP routers. GDP routers are the IP-router equivalent for the GDP network; these are the nodes that do the routing and forwarding between communicating entities. Thus, a routing domain is essentially a set of GDP routers owned by the same administrative entity that are connected together as a graph.[14]

Routing domains are associated with flat GDP names and corresponding private keys owned by the domain administrators.[15] The resource ownership claimed by a routing domain is cryptographically asserted by using *OwnCerts* signed with the routing domain's private key; an *OwnCert* $(\mathbb{R} \rightarrow_O R)$ signifies that a GDP router R is owned by a routing domain $\mathbb{R}$, and that $\mathbb{R}$ allows R to act as its agent.

A routing domain has well specified entry-points for outsiders that are not part of the routing domain; these are called *border routers* and are specifically tagged as such. Other routers are called *internal routers* (See Figure 5.3).[16] The task of border routers is strictly to facilitate inter-domain

[13] We only discuss policy specification as an abstract concept. Specifically, we don't specify a formal language for policy specification. We consider a policy specification language as a direction for future work.

[14] In some sense, routing domains are the equivalent of Autonomous Systems in the traditional IP networks.

[15] The GDP name and the key are linked to each other by the same metadata mechanism as used for DataCapsules.

[16] The determination of whether a GDP router is border router or internal router is accomplished by putting such information in the immutable metadata from which the name of the GDP router is derived.

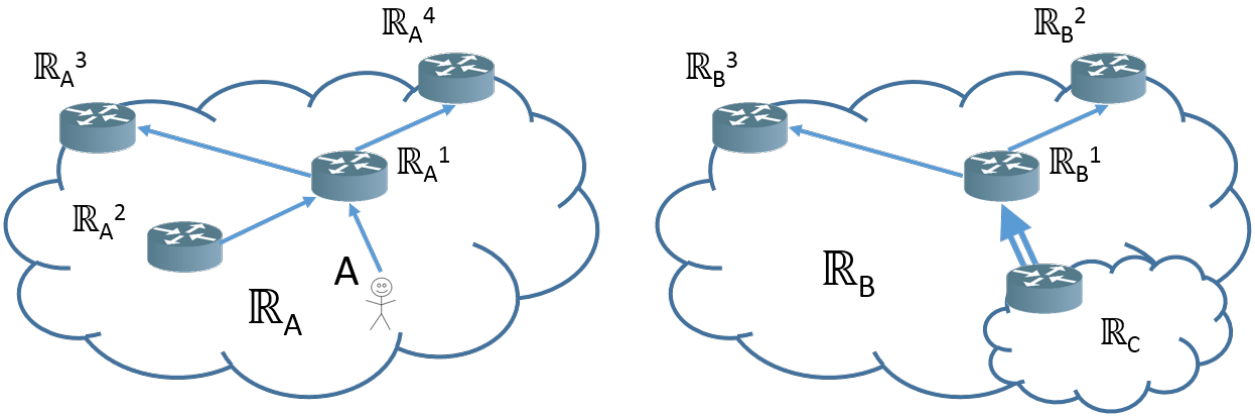


Figure 5.3: *Left:* Border (R_A^3, R_A^4) vs internal (R_A^1, R_A^2) GDP routers. If $\mathbb{R}_A$ is private, A must connect to an internal GDP router and provide a $JoinCert$ issued by $\mathbb{R}_A$. *Right:* Subdomains. $\mathbb{R}_C$ as a sub-domain of $\mathbb{R}_B$; a border GDP router for $\mathbb{R}_C$ connects to an internal GDP router of $\mathbb{R}_B$ as if it were just a client; such border router must have an $OwnCert$ issued by $\mathbb{R}_C$, which in turn has an $OwnCert$ issued by $\mathbb{R}_B$. Note that an $OwnCert$ implies a $JoinCert$.

communication; clients and servers that are part of a routing domain typically connect to the internal routers of such a domain.

Private vs public domains: Routing domains could be either *open* (for anyone to join), or *private* ()requiring further authorization). This property of the routing domain is marked as part of the routing domain metadata. The authorization for joining a private domain is maintained cryptographically via a $JoinCert$. A $JoinCert$ $(\mathbb{R} \rightarrow_J X)$ allows a client or server X to join a routing domain $\mathbb{R}$, and is signed with the private key of $\mathbb{R}$. For private domains, clients joining the domain must necessarily connect to internal routers and provide a $JoinCert$ at the time of connection. Only border routers accept connections that come without a $JoinCert$, and filter out attempts to reach information that should not be visible outside the domain. Note that an $OwnCert$ *implies* a $JoinCert$. As such, all GDP routers for a given private routing domain can join by presenting their corresponding $OwnCert$.

Subdomains: Routing domains could be further divided into *sub-domains* for controlling granularity of administration (see Figure 5.3). Such parent-child relationship between routing domains and sub-domains is also encoded using $OwnCerts$. For example, $(\mathbb{R}_1 \rightarrow_O \mathbb{R}_2)$ represents the domain $\mathbb{R}_2$ as a sub-domain of $\mathbb{R}_1$. The ownership hierarchy is a strict tree: at the leaves are GDP routers. A standalone GDP router is a routing domain by default.

Routing domains represent the boundaries for the network isolation property; in some way, the boundaries of a routing domain represent *firewalls* for specific objects. For routing domains with sub-domains, these sub-domains allow for more fine-grained control over scope of information. In order to be able to use such sub-domains for fine-grained placement and control, it is essential that the organizational structure of the routing domain be visible to users, so that they can create appropriate policies.

B Inter-domain interaction

In the simplest GDP network model, all routing domains are completely autonomous, and they use the GDP network as on overlay on top of the public IP network for all inter-domain traffic. A global GLookupService (see below) serves as a globally distributed repository associating GDP names to routing domains in a verifiable way. Traffic from a client sending datagrams to a destination in a different routing domain eventually reaches a border GDP router of the routing domain; such a border GDP router, when encountered with a destination that it doesn't know the path to, can lookup information for the GDP name in the global GLookupService, and then create an on-demand connection to the border GDP router of a remote routing domain that is delegated to route traffic for the destination GDP name (see Figure 5.2).

With this simple GDP network model, all routing domains form a *pseudo* fully-connected topology. What this means is that each routing domain *can be* directly connected to all other routing domains, but only if the need arises. A given pair of routing domains don't have to be connected if there is no inter-domain communication between such domains. Moreover, such connections are transient: they are established as needed and cleaned up when inactive. Using direct connections between routing domains, as opposed to using an intermediate domain as a transit, enables simplification of reasoning about the security of communication paths.

An architecture with such direct inter-domain connections may not seem scalable at first because of the possible quadratic number of inter-domain connections. However, there are two important points: (1) The number of *active* inter-domain communications is far smaller in practice. For example, even in the traditional IP network, there are only a few popular autonomous systems (ASes) that have active communication paths with virtually every other AS, but most ASes only have a rather small number of active inter-AS communications.[17] (2) A single connection between two given routing domains can, in theory, multiplex all possible user-to-user communications between the given domains.[18]

To allow for more flexible approaches to inter-domain routing, the simple GDP network model can be extended beyond direct connections between routing domains. We discuss such an extension with the help of an example in the next chapter where routing domains use a *transit* domain.

C Global GLookupService: Enabling sharing of state

A key component of the infrastructure, a global *GLookupService*, lives in the core of the GDP network and enables inter-domain routing. The global GLookupService is essentially an untrusted key-value store that contains verifiable routing and delegation information which can be queried by anyone. Communication to the GLookupService is done out-of-band and not through the GDP

[17] Imagine, for example, the likelihood of an active communication path between a small rural ISP in North America and a small university in Asia.

[18] An excellent example of an existing system that follows similar architecture of direct inter-domain connections is the email infrastructure; users of a given email service all connect to the mail server of the domain, which then acts as a relay to all other domains in the world. On the receiving side, a similar process is carried out in reverse: the recipient mail server collects (and often holds) email for its users, who are then delivered the message.

network.[19] The keys in the GLookupService are flat names, and the values are the cryptographic delegations that guide how to reach the name in a verifiable form. Large routing domains can have a local GLookupService inside the domain to facilitate intra-domain communication.

In addition to normal GDP names, the global GLookupService also contains routing domains names as keys; for such keys, the values are the corresponding border GDP routers of the routing domain. This enables anyone to reach the names that have been delegated to a given routing domain by first talking to the border GDP routers of the said domain.

Note that for a given key, there can be more than one value. As such, the values in the GLookupService are in fact, *sets* of possible ways to get to the corresponding keys. As an example, a routing domain $\mathbb{R}$ with two border GDP routers R_1 and R_2 will be represented in the GLookupService as $[key : \mathbb{R}, value : set(\ (\mathbb{R} \rightarrow_R R_1), (\mathbb{R} \rightarrow_R R_2)\ )]$.

Viewing the GLookupService as a single component is merely a conceptual representation. In practice, the global GLookupService is a highly distributed service with globally distributed caches. Designing a scalable GLookupService can be achieved with existing systems. For example, a fast, multi-level lookup service similar to DNS can serve the purpose of the GLookupService. Existing locality aware DHTs [75] can also be used as a potential implementation of such distributed key-value store.

Note that using DHTs—with their potential security issues that we were trying to avoid in the first place—for implementing a global GLookupService does not necessarily break our security model, because such a DHT is used only for lookup and not for actual routing of information. As we discussed in the threat model, a compromised GLookupService could result in a denial of service, but not a man-in-the-middle attack.[20]

Does the global GLookupService represent a repository of *active* paths? In the most general scenario when the GDP network is deployed as an overlay network, the answer is no. The global GLookupService merely represents a repository of publicly accessible delegations (i.e. what names are authorized to advertise for what other names). In the general overlay scenario, there may or may not exist an active network connection for every delegation. However, a delegation in the GLookupService does indicate that an active network connection *can be* established if need be.[21]

5.4 Routing workflow: A bottom-up view

To enable the routing security properties of the GDP network as a whole, each GDP router individually follows a principle of *verify before accept* (i.e. verify a sender's legitimacy for a given

[19] This is a conscious design choice that allows us to have a baseline access to a *routing oracle* to bootstrap the overlay GDP network. In our current design, the GLookupService operates on top of UDP.

[20] Note that a compromised GLookupService that results in a denial of service is not contradictory to our threat model. Our threat model considers adversaries that would like to insert themselves in arbitrary communication paths, and perform either active or passive man-in-the middle attacks. A denial of service attack by an active man-in-the-middle that drops all messages is merely the consequence of a broader attack.

[21] In the next chapter, where we discuss GDP network deployment with custom topologies inside routing domains, a GLookupService does in fact include a number of active paths between GDP routers in addition to potential paths.

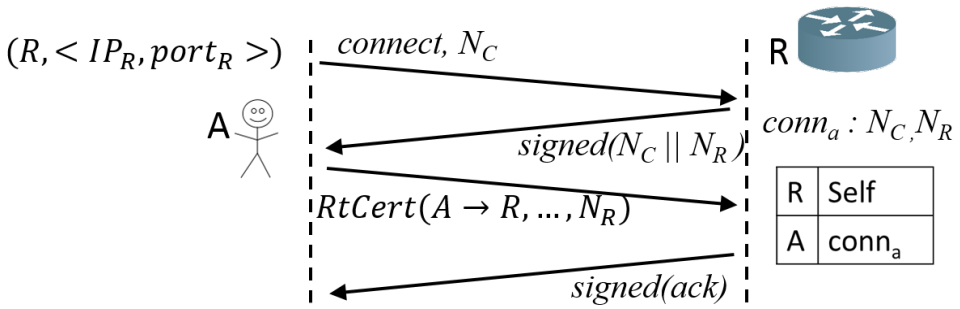


Figure 5.4: Simple secure advertisement. *A* needs to know the GDP name and the IP/port of a GDP router *R* that it wants to connect to. *A* then advertises its name to the GDP router *R* by completing a challenge-response process. The challenge-response process includes four messages: (1) The client sends a nonce N_C to the GDP router. (2) The GDP router generates a nonce N_R and sends a message $N_C||N_R$ signed with its private key. (3) The client generates an $RtCert$ that includes N_R and is signed with the client's private key. (4) The GDP router sends back an acknowledgment signed with its private key. At the end of this message exchange, *R* has verified that it is talking to the correct *A*, and populates its forwarding table. This is a simplified version of a secure advertisement, see Figure 5.5 for how the process is extended when *R* is a part of private routing domain that requires a $JoinCert$, and where *A* may have other names to advertise in addition to its own.

source address before accepting traffic) and *verify before send* (i.e. only send messages to those who are allowed to receive traffic for a given destination address).

In the next few sections, we describe some key mechanisms and the decision flow for an individual GDP router that enables these properties. We then move on to how a collection of GDP routers enables intra-domain routing. And finally, we extend the same basic principals for inter-domain routing.

5.4.1 Connecting to the GDP network: Secure advertisements

Other than simply forwarding GDP PDUs, GDP routers collectively perform the important task of verifying users' claims for names when they connect into the GDP network. The essential mechanism is a *secure advertisement* of names. At the core of this *secure advertisement* process is a challenge-response mechanism and exchange of cryptographic delegations. GDP routers enforce such secure advertisement process on anyone that wishes to advertise GDP names to them, whether that is a regular client or another GDP router, and ensure that the policies encoded in the chain of delegations are followed appropriately.[22]

The process works as follows. To start off, a client joins the GDP network by joining a trustworthy routing domain. The client needs to know the name of the routing domain and the network address of an internal GDP router for the routing domain. Upon connecting to the GDP

[22]In contrast, typical flat namespace networks simply allow clients to advertise any names that they wish.

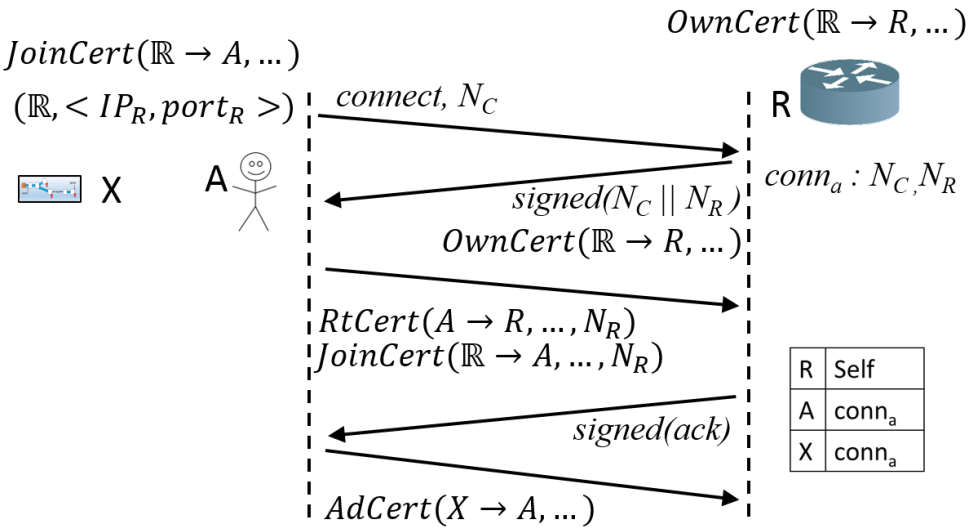


Figure 5.5: A more complex process than in Figure 5.4; R is owned by a routing domain $\mathbb{R}$ that requires a $JoinCert$. A acquires $(\mathbb{R} \rightarrow_J A)$ out of band and connects to R. B advertises its name to R by completing a challenge-response, and then advertises an information object X that it has an $AdCert$ for. At the end of the process, R can populate its forwarding tables for A and X.

router, the client must complete a challenge-response protocol with the GDP router for a name it claims to own (in this case its own name, see Figure 5.4.) At the end of this challenge-response protocol, both the client and the server can verify that they are talking to who they think are talking to. A GDP router populates its local forwarding table and forwards the information appropriately for the names to be available elsewhere in the network. For handling non-local destinations, the GDP router can be configured in a number of different ways to reason about the next node that the GDP router should forward the PDU to.

However, a simple challenge-response, while sufficient to demonstrate the ownership of a server's or client's name to a single GDP router, requires more work to be a usable mechanism for overall routing security of the GDP network. Specifically, there are three challenges:

(1) How to delegate and advertise *objects*?

Recall that the servers do not have access to the private key for objects they host, and thus, they cannot participate in challenge-response on behalf of such objects. To address this challenge, the object owner *delegates* the authority to advertise for the names to appropriate servers along with policies on scope and visibility. This is done by an out-of-band cryptographic delegation with a limited lifespan called $AdCert$. An $AdCert$, $(X \rightarrow_A Y)$ is a statement signed by the private key for X (controlled by X's owner) that allows a server Y to host and advertise for object X. The server Y, in turn, advertises its own name Y to a GDP router via challenge-response, followed by presenting $(X \rightarrow_A Y)$. (see Figure 5.5). We call names advertised in this manner as *delegated* names. Such advertisement binds names to particular locations in the physical network.

(2) How can a GDP router verify that a client multiple hops away has completed a 'secure advertisement' with some other GDP router?

An underlying security principle in the GDP network is that a GDP router shouldn't just take for granted that some other GDP router has performed its job well. Instead, we use a transitive proof of verification in the form of another type of cryptographic delegation called $RtCert$. Recall that a GDP router itself has a cryptographic name. The final result of the 'secure advertisement' process is that the client issues an $RtCert$ to the immediate GDP router. An $RtCert$ $(Y \rightarrow_R R)$ is a limited lifespan delegation of routing responsibilities for name Y to a specific GDP router named R. The GDP router R acts as a client to other upstream GDP routers (see Figure 5.7): it first advertises its own name, followed by all the appropriate $RtCert$'s and $AdCert$'s that it has received from all the clients. If an upstream router R_u sees a sequence $(X \rightarrow_A Y)$; $(Y \rightarrow_R R)$ from a router R, R_u can reason that it is safe to send/receive traffic for both X and Y from R.[23]

(3) How to enforce network isolation and restrict access to private domains without trusting *all* GDP routers?

As mentioned earlier, routing domains could be *private* and may require authorization to join in the form of a $JoinCert$. Similar to verifying the advertising delegations from another GDP router multiple hops away, a GDP router must also verify whether a client multiple hops away has, in fact, authorization to join the given routing domain. Based on its own verification, a GDP router may block incoming traffic from unauthorized senders. As a result, the evaluation criteria for a chain of delegations becomes slightly more involved when considering $JoinCerts$. We discuss such generalized evaluation in the next chapter.

Only after a client completes the secure advertisement process that involves the challenge-response phase *and* issuance of an $RtCert$, will a GDP router allow the client to use the given name, populate its local forwarding tables, and forward the information to appropriate upstream infrastructure.

Before moving further, let's summarize the types of delegations: $AdCert$ allows for delegation of storage responsibility to a host; $RtCert$ allows for delegation of routing responsibilities to a GDP router (or a routing domain); $JoinCert$ allows creation of a distributed access-control list for who is allowed to join a routing domain; $OwnCert$ allows asserting the resource ownership hierarchy in a provable manner. In our implementation, the required metadata for verification is included along with the delegations. We skip the details for simplicity of explanation. Figure 5.5 shows how these various cryptographic delegations fit in the 'secure advertisement' protocol. We discuss the generalized evaluation of chains of delegations in the next chapter.

5.4.2 Intra/inter domain advertisement flow and routing

Once a GDP router receives a message to be forwarded, it verifies that the source address has been advertised on the particular connection by looking at the local forwarding table; if not, the message

[23] We generalize the chaining of $AdCerts$ and $RtCerts$ in a manner somewhat similar to X.509 certificates; we discuss the chaining modes later.

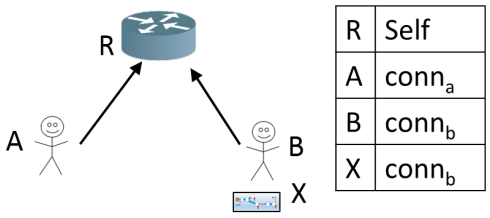


Figure 5.6: Forwarding table of a GDP router for simple local routing. Both A and B finish a challenge-response protocol with R. Additionally, B advertises for content X by means of an $AdCert$ ($X \rightarrow_A B$).

is simply dropped by treating it as unverified incoming traffic (*verify before accept*). To verify destination routes (i.e. *verify before send*), the process is very simple for local destinations: the GDP router looks up the destination in its local forwarding table. The forwarding table is populated only if the GDP router has finished the secure advertisement process for the given destination address. If a local entry is found, the GDP router performs local forwarding (see Figure 5.6). If, however, there is no local entry for the destination, then the next step in forwarding process depends on the configuration of GDP router and the network topology in the routing domain. For example, the GDP router may forward the message to a *default route*, or lookup information in a GLookupService. We discuss the various strategies below.

A Intra-domain routing

A single routing domain could have many GDP routers organized into sub-domains and such. The exact internal topology and routing strategy is left to the routing domain's discretion to allow for a wide range of setups from very small single-router domains to huge corporate domains. As a consequence, the configuration of GDP routers and intra-domain advertising propagation varies as well.[24]

Single GDP router deployment: For very small deployments, such as a small house, a single physical node can serve as both internal router as well as a border router, but it still should be considered as two logical GDP routers (see Figure 5.6). The single GDP router can forward datagrams for all intra-domain communications, and act as a border GDP router for inter-domain communication as well.

Multiple GDP routers in a tree topology: For slightly larger routing domains with a handful of GDP routers, a domain administrator might choose, for example, a tree topology enabled by 'default routes' (see Figure 5.7). In this topology, a GDP router uses its parent as a default route; if no matching destination can be found in the local forwarding table, then a GDP router forwards

[24] Note that for the purpose of discussing intra-domain routing, any communication from inside a routing domain to destinations outside, all that an internal GDP router has to do is forward the datagrams to a border GDP router. We discuss the operational details for a border GDP router when we discuss inter-domain routing next. Conversely, for communication that originates from outside and is directed to a destination inside the routing domain, the border router simply forward the communication to the appropriate internal GDP router by following intra-domain routing process.

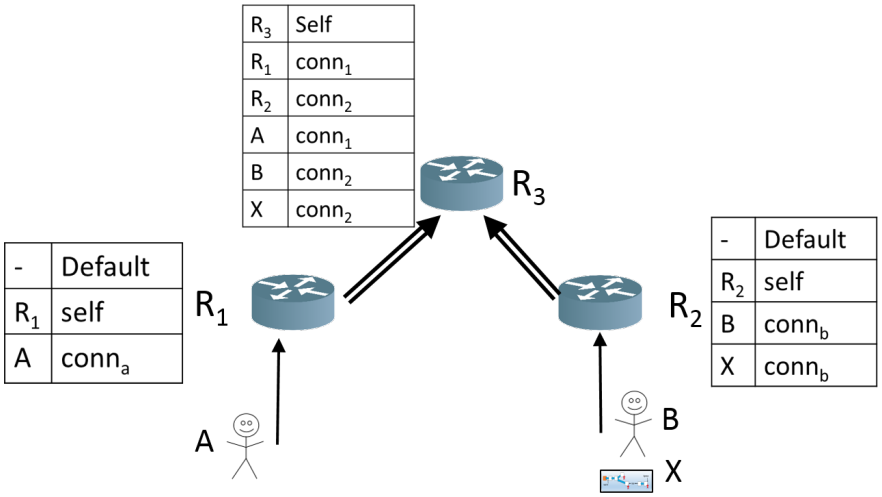


Figure 5.7: A tree topology with default routes. R_3's forwarding tables contains information from both R_1 and R_2 (assuming no policy restrictions).

messages to its default, and so on. The root of the tree acts as a border GDP router; destinations that the root doesn't know about are handled via inter-domain routing process.

In such a tree topology, each GDP router does perform source address verification on all incoming traffic, except for traffic coming from a parent on the default route.[25] In such a tree topology, a child GDP router is merely a client from the perspective of the parent GDP router, and as such, it needs to complete the secure advertisement process *and* send all the advertisements to the parent GDP router along with the $RtCerts$ that it receives from clients that connected to this child GDP router. A GDP router anywhere in the tree tracks the entire sub-tree's advertisements, and thus a simple tree is a good candidate for geographically separated routing domains with relatively smaller set of names.

Large domains with a local GLookupService: For larger domains with many GDP routers and many GDP names (such as a university network, a corporate network, etc.), a domain administrator may desire scalability and fault-tolerance to handle temporary failures of GDP routers.[26] In such large domains, the domain administrator may choose to go with an approach similar to inter-domain routing by setting up a local GLookupService within the domain. The GDP routers in the domain are then configured with the local GLookupService enabling them to connect to other GDP routers

[25]Not doing source address verification for the traffic forwarded by a parent to its child does not violate the security guarantees of the GDP network. Based on our threat model: since the parent GDP router is the only way for the children to reach to the rest of the world, it *has* to be in the path of all such communication. Recall that the GDP network does not provide any security from on-path adversaries.

[26]For example, in the tree topology above, any GDP router failure not only affects directly connected clients to that GDP router but also the *entire* sub-tree under that GDP router; recovery from such failures is quite expensive.

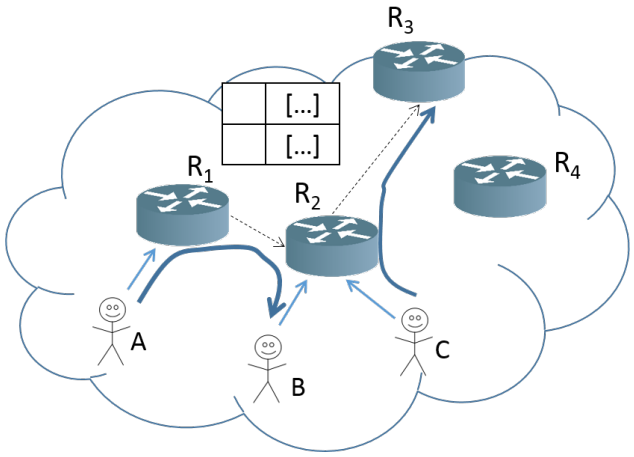


Figure 5.8: A pseudo fully-connected topology inside a routing domain, where GDP routers connect to each other directly when a communication path needs to be established (see the mechanism in Figure 5.9). R_1 and R_2 are connected because of an active communication between A and B. Similarly R_2 and R_3 (a border GDP router) are connected because of a communication between C and some external party. R_4 is not connected to anyone because there are no active communications at the moment. Note that all these connections are created after looking up in a local GLookupService.

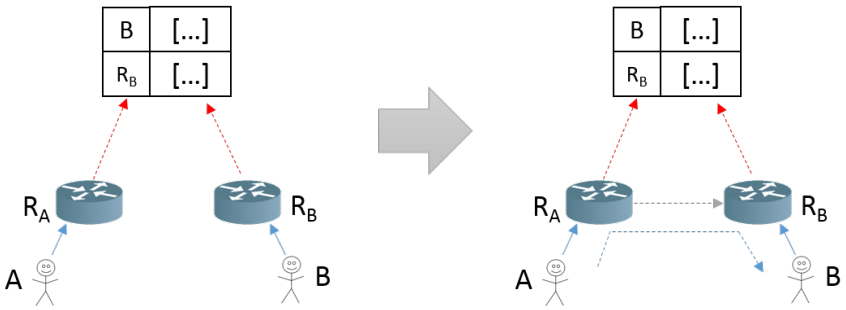


Figure 5.9: On-the-fly connection created by looking up in a GLookupService. For the sake of clarity, the GLookupService only shows information that R_B put; the information put by R_A is not shown. Following this mechanism, large routing domains may come up with a pseudo fully-connected topology as shown in Figure 5.8

directly when needed.[27] The local GLookupService serves as a repository of delegations from which GDP router can lookup information needed to route to destinations not present in the local forwarding table. See Figure 5.8.

[27]In the basic GDP network setup operating strictly as an overlay network, the links between GDP routers (and even routing domains) are TCP connections that can be created on demand during the routing process. A non-overlay network, e.g. IP networks, does not have such flexibility, and are limited by what physical links exist. In the next chapter, we generalize the GDP network beyond a strict overlay and consider the case where a routing domain may want to have a custom topology other than a pseudo fully-connected topology.

Let's see how the routing process works in such a case. Recall that a GLookupService is essentially a key-value store, where keys are GDP names and values are verifiable information on how to reach the specific key. The information that goes in a GLookupService is similar to that in the forwarding table of a GDP router. To begin with, when a GDP router R_A starts up, it populates the reachability information in the form of an $RtCert$ $(R_A \rightarrow_R (IP_A : port_A))$ signed by the private key of R_A. After startup, the GDP router keeps forwarding all $RtCerts$ or $AdCerts$ that it receives as part of secure advertisement process to the local GLookupService.

When a client A connected to R_A wants to reach a name B not present in the local forwarding table of R_A, R_A recursively queries the local GLookupService for B (see Figure 5.9). On querying for B, the response from local GLookupService is $(B \rightarrow_R R_B)$; something that presumably resulted as a part of secure advertisement of B to R_B. R_A can locally verify the validity of this $RtCert$. Since R_A doesn't know about the GDP name R_B in its local forwarding table, it then queries the local GLookupService again for R_B, when it receives $(R_B \rightarrow_R (IP_B : port_B))$: an $RtCert$ that R_B put in the local GLookupService at startup time just like R_A did. R_A can then initiate a connection to R_B as a client, complete a secure advertisement process issuing an $RtCert$ $(R_A \rightarrow_R R_B)$,[28] advertise for A to R_B,[29] and start forwarding traffic.

While we showed how a GDP router can query the local GLookupService in one specific example shown in Figure 5.9, this can be generalized to three cases. A GDP router queries the local GLookupService until either (1) R_A finds some destination it already knows, or (2) if it finds an $(IP : port)$ pair that it can create a new connection to as in the example above, or (3) there's nothing else to query. As an example of situation (1): for another communication initiated by A to a client B' connected to R_B, R_A can stop the recursive query at R_B because it presumably knows how to get to R_B as a result of a previous query. For destinations that are not present in the local GLookupService (and in the local routing domain), the query fails. In such a case, the local GLookupService can return the information about a border GDP router, thus kicking off the inter-domain routing process.

Note that the above strategies (i.e. a tree topology, or a setting up a local GLookupService) can be mixed and matched as desired. For example, a university campus with multiple buildings may choose to create a tree topology for GDP routers within a building, and the root of all such trees could be configured to use a domain GLookupService to create on-demand connections as need be. Further, the domain administrator may choose to declare individual buildings as sub-domains and delegate the administration of such sub-domains to others (e.g. building managers).

While such combinations provide quite a bit of flexibility in intra-domain organization of resources, some domain administrators may like to adopt a more traditional routing strategy, e.g.

[28] When R_A connects to R_B and issues an $RtCert$ $(R_A \rightarrow_R R_B)$, such an $RtCert$ should be a *limited RtCert* that allows R_B to locally validate that it is talking to the correct R_A, but does not allow R_B to present the $RtCert$ to others and claim that it has a path to either R_A or A. All cryptographic delegations can include such user-specified restrictions, which we will discuss in the next chapter. The reason for such a *limited RtCert* is that R_B should not be able to claim to other GDP routers that it can route traffic on behalf of R_A or a different client A' connected to R_A.

[29] Advertising A to R_B is to ensure that R_B does not reject incoming traffic from R_A based on its incoming traffic checks.

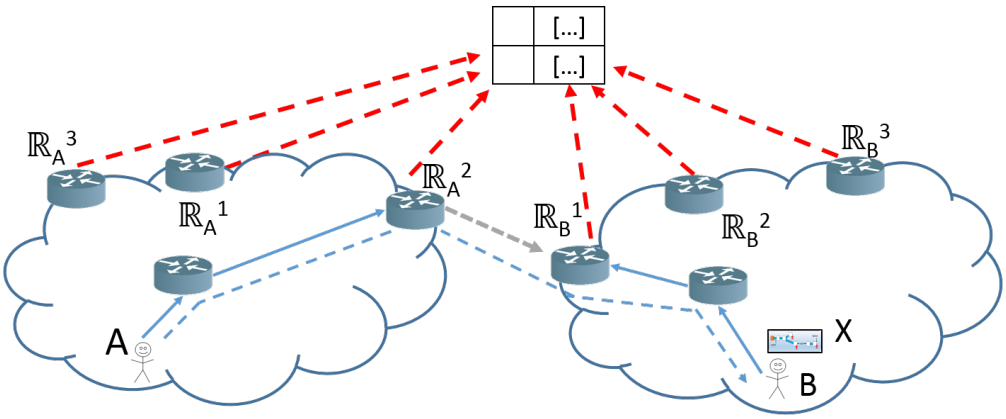


Figure 5.10: Inter domain routing; A is part of $\mathbb{R}_A$ and wishes to reach X. Border GDP routers of $\mathbb{R}_A$ reach out to the global GLookupService, find that X is stored on B which is connected to $\mathbb{R}_B$. To reach $\mathbb{R}_B$, an on-demand connection is made to border router $\mathbb{R}_B^1$. Since $\mathbb{R}_A^2$ initiates the connection, it first advertises the presence of A (and the associated chains).

routing based on physical topology as done in IP-routing, where GDP routers should not connect to each other directly but use other GDP routers as intermediate routers. In such cases, the domain administrator can achieve their goals by deploying the GDP network with a custom topology for parts of the routing domain. We discuss these details in the next chapter.

B Inter-domain routing

Inter domain routing is facilitated by a scalable global GLookupService. Such inter-domain routing is very similar to the intra-domain routing in the case of a local GLookupService as we just described. Each routing domain populates its own reachability information in the global GLookupService, typically in the form of key-value pairs $\mathbb{R} : (\mathbb{R} \rightarrow_R R_i)$ and $R_i : (R_i \rightarrow_R (IP : port))$, where R_i are the border GDP routers for the routing domain $\mathbb{R}$. Records that map flat names to signed IP:port pairs (e.g. $X : (X \rightarrow_R IP : port)$) facilitate the creation of new on-demand connections, and ensure that the GDP network plays well as an overlay on top of an existing IP network across various domains.

If an advertisement for a name X by a client connected to a routing domain $\mathbb{R}$ has a global scope, then depending on the configuration of $\mathbb{R}$, this information is passed on to the global GLookupService by either the border routers or the local GLookupService of $\mathbb{R}$ (see Figure 5.10). When included in the global GLookupService, this information typically looks like $X : (X \rightarrow_R \mathbb{R})$. Regardless of who propagates the advertisements, the global GLookupService (and anyone who queries it) can verify the correctness of the advertisement.

For a client that wishes to reach an information object not in the current routing domain, such request eventually reaches a border GDP router (see Figure 5.10). The border GDP router then looks up for the destination from the global GLookupService. Much like the query pattern in case

of intra-domain routing with a local GLookupService, the border GDP router keeps querying unless either it reaches either (1) an $IP : port$ pair of a border GDP router of another routing domain, or (2) an address already in the forwarding table of the border GDP router, or (3) the lookup fails.

In case (1), the two border routers from different domains establish a new connection. In Figure 5.10 for instance, the GDP router $\mathbb{R}_A^2$ from the initiating domain $\mathbb{R}_A$ presumably acquired a full chain of delegations from the global GLookupService that enable it to verify that X has been delegated to $\mathbb{R}_B$, and that $\mathbb{R}_B$ has delegated a border router $\mathbb{R}_B^1$ as its entry-point. After initiating a TCP connection to $\mathbb{R}_B^1$, $\mathbb{R}_A^2$ first advertises for the source address A so that $\mathbb{R}_B^1$ does not block traffic originating from A by marking it as unverified incoming traffic. During this advertisement, $\mathbb{R}_A^2$ must also present a *limited RtCert* ($\mathbb{R}_A^2 \rightarrow_R \mathbb{R}_B^1$) valid only for the specific use of inter-domain routing between $\mathbb{R}_A$ and $\mathbb{R}_B$.[30]

Case (2) is a result of an additional flow on existing connections. For example in Figure 5.10, if $\mathbb{R}_A^2$ and $\mathbb{R}_B^1$ already had an established connection for some preexisting pair of addresses, they can reuse the existing connection by simply adding new advertisements.

An important question is: what if there are multiple replicas of a given destination resource in multiple routing domains? In the general case, when performing a recursive lookup from the GLookupService for a given name, a GDP router might get multiple responses for the same `GET` operation, e.g. in Figure 5.10, $\mathbb{R}_B$ can be reached by either of $\mathbb{R}_B^1$, $\mathbb{R}_B^2$, or $\mathbb{R}_B^3$. The GDP network must perform a type of "anycast" in this case, choosing one of the replicas as a destination. While the GDP network does not put any restrictions on how a GDP router should decide between replicas in such a case, policies by a GDP router to favor *close-by* entry-points (by some definition of close-by), or additional hints for preferred domain put by object owners can guide how to handle such cases. In fact, information for popular content might even be pre-fetched and connections established in-advance to enable a quicker establishment of the flow.

5.5 Conclusion

Routing in flat namespace, especially in an open federated network, is a challenging security problem. However, flat names allow us to place a trust anchor in the name itself. With the help of a secure delegation mechanism and the secure advertisement procedure, the GDP network enables *every* GDP router to independently verify the following two facts:

1. Any incoming GDP datagram that a GDP router receives originates from either the source specified in the GDP datagram itself, or an entity delegated to operate on behalf of the source, or (in few cases) another GDP router that the recipient GDP router is explicitly configured to trust.
2. Before forwarding a GDP datagram to a peer (which could be another GDP router or a client), the GDP router can know for sure that the peer is authorized to receive GDP datagrams for the specified destinations.

[30] Once again, it is important to put such restrictions to ensure that $\mathbb{R}_B^1$ cannot misuse this *RtCert* to fool an unsuspecting user by claiming that A has directly delegated routing responsibilities to $\mathbb{R}_B^1$.

In this chapter, we discussed the GDP network exclusively as an overlay network. In the next chapter, we discuss (1) how to extend the GDP network into domains in which it is *not* operating as an overlay network, (2) how to enable a global deployment of the GDP network, and (3) how to generalize the scoping guarantees.

Chapter 6

Making the GDP network Real: The Engineering

In the previous chapter, we discussed the core mechanisms of secure routing. In this chapter, we discuss the applications of those mechanisms to real world scenarios.

6.1 Deployment in real networks: Beyond an overlay

Real networks are rather complicated. Our simple scenarios we discussed in the previous chapter require further discussion to be useful in practice. While the pseudo fully-connected topology operating as an overly is a good start, large domains may desire custom intra-domain routing topology.[1] For better control over latency and QoS, domain administrators may want to create GDP communication paths that match the underlying physical topology. Such topologies may require support for multiple hops, which needs additional mechanisms than a pseudo fully-connected topology.

Both these requirements put an interesting twist to our simple model of the GDP network because GDP routers may not be able to establish new connections as needed. We translate the problem to: rather than establishing a new connection between GDP routers R_A and R_B, how can one use an optimally located GDP router R_C to which both R_A and R_B are already connected to (see Figure 6.1).

Note that this same idea can be extended to inter-domain routing to enable *transit* domains that provide routing services to other domains, however we believe that such kind of inter-domain routing will be done on a case-by-case basis.

[1] Recall that we defined a pseudo full-connected topology as a topology where each GDP router can be connected directly to all other GDP routers, but only if the need arises.

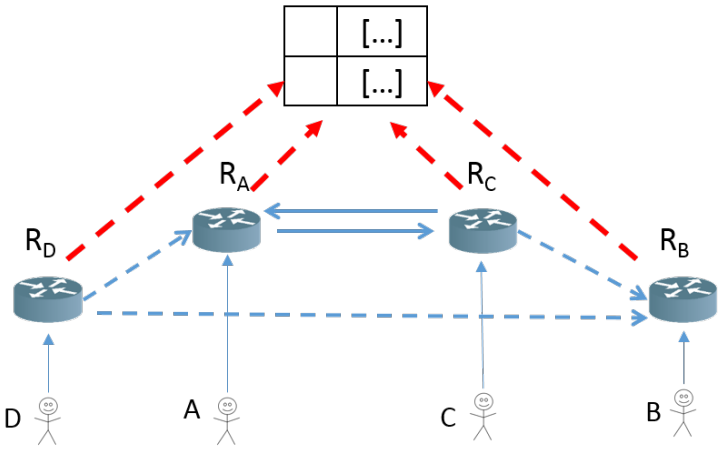


Figure 6.1: An example where R_A and R_C are configured by a domain administrator to connect with each other and issue each other unrestricted $RtCerts$. Other GDP routers, R_D and R_B, operate as normal by connecting to a local GLookupService. All GDP routers forward advertisements to the local GLookupService as usual. In this example, R_D can directly connect to R_B if D wants to communicate to B. On the other hand, because R_A and R_C are connected to each other with unrestricted $RtCerts$ that them to route traffic for each other, D's communication to C can be routed via $R_D \Rightarrow R_A \Rightarrow R_C$.

6.1.1 Extending the GDP network beyond a pseudo full-connected topology

To create specific routing topology, we first discuss how would a domain administrator configure a single custom link between one given pair of GDP routers. At boot time, the GDP routers in question connect to each other and issue the other GDP router an $RtCert$ in the same way as if they were connecting to the other GDP router as a client. The final topology is identical to as if the GDP routers had connected to each other with an on-demand connection, except that the $RtCert$ is not a limited $RtCert$. These $RtCerts$ are also populated in the local GLookupService.

The routing process for other GDP routers in the domain stays the same (see Figure 6.1.). If there is a local GLookupService in the domain, other GDP routers can simply query the local GLookupService for routes as usual.[2] For example in Figure 6.1, if D wants to communicate to C, then R_D acting on behalf of D will lookup the local GLookupService for C. From the initial query, R_D learns that C can be reached via R_C. R_D continues the recursive query and looks for the reachability information for R_C, where it learns two ways to reach R_C: one via a new direct connection between R_D and R_C, and the other via R_A.[3] Since R_D already has a connection to R_A

[2]Even if there is no local GLookupService, there is no substantial change to the routing process for rest of the GDP routers in the routing domain. We just choose an example with a GLookupService because it is more interesting.

[3]The exact entry in the local GLookupService, in this case, would be $R_C \Rightarrow set(\ (R_C \rightarrow_R (IP_C : port_C)), (R_C \rightarrow_R R_A)\ )$. The first $RtCert$ is present only if the routing domain administrator configures R_A and R_C like all other GDP routers, where they populate their *direct* reachability information via an overlay link. If desired, this direct connection functionality can be turned off.

in this example, the recursive process can be terminated and R_D can populate its local forwarding table accordingly.

With this simple mechanism, a domain administrator can create any arbitrary intra-domain routing topology as they desire. Note that for routing domains operating with a completely custom topology, the local GLookupService does not contain any keys for which the value is an $(IP : port)$ pair. While this simple mechanism allows for GDP network deployments with custom topologies, this kind of *forced* creation of a permanent connection comes at two costs:

- The security guarantees are to be reasoned about carefully, because such a situation explicitly grants permission for each GDP router to route *all* traffic on behalf of the other GDP router.[4] In the example scenario above, for every communication path to C that involves a direct connection to R_C, there is another valid path that involves routing via R_A, which may not always be the most optimum path, but allows an adversary to compromise R_A in order to affect traffic intended for R_C and its clients.[5]
- With multiple possibles paths, the path discovery mechanism needs to be more clever. Adding a large number of custom paths adds a number of choices for the path between a given pair of source and destination, making it harder for a GDP router to pick the next destination in the forwarding process.[6] As such, discovery of the most optimum path requires some global knowledge. We believe a local GLookupService could help to address this problem by computing paths as an alternate to recursive querying, however, we consider that as a future enhancement to the GDP network and out of scope of the current dissertation.

Overall, while it is possible to create a GDP network deployment with a completely custom topology within a domain, this requires putting more trust in individual GDP routers. A single compromised GDP router in a large domain *can* launch a similar kind of man-in-the-middle attack against arbitrary communication paths in the domain that the GDP network aimed to solve in the first place. However, such issues are contained within the specific routing domain. Even though the diminished security guarantees may seem quite problematic, the status quo for the current Internet infrastructure is, in fact, a domain administrator trusting the infrastructure they operate. Thus, even if a domain administrator runs a GDP network deployment with a custom topology, the security issues are no worse than the conventional IP networks. Just the fact that a domain is using the GDP network does provide them the benefits of being able to participate in the GDP, and allows them to reap the benefits of DataCapsules.

[4]The limited *RtCert* in case of an on-demand connection, as we discussed in the previous chapter, specifically is an attempt to contain such damage.

[5]For instance, if the relative placement of GDP routers in Figure 6.1 represents actual physical placement and routing cost, then a path like $D \Rightarrow R_D \Rightarrow R_A \Rightarrow R_C \Rightarrow C$ is not too far from optimum. On the other hand, a path $B \Rightarrow R_B \Rightarrow R_A \Rightarrow R_C \Rightarrow C$ certainly is quite bad but it is a valid path.

[6]This issue manifests itself as follows: when a GDP router queries from a local GLookupService for a given GDP name, it receives back a *set* of values representing the nodes that can route on behalf of the queried GDP name. If the GDP router does not have *any* of those new set of nodes in its local forwarding table, then it must continue the recursive query. The question is: which value from the set should the GDP router use for the next query to the GLookupService, or should it query for *all* of these values? This results in the GDP router essentially doing a graph traversal in the most general case, which could become quite expensive if the graphs are large.

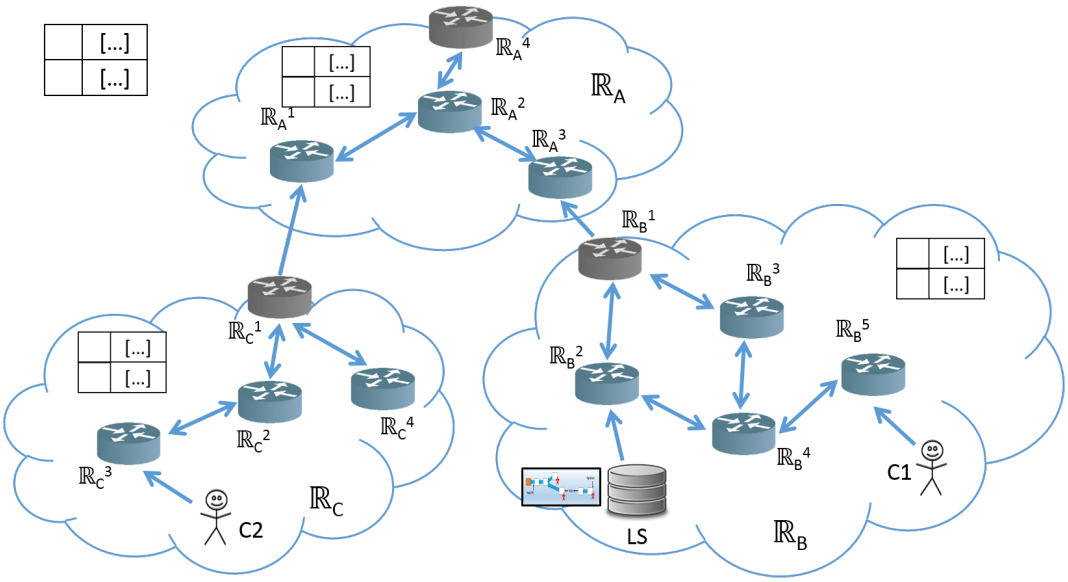


Figure 6.2: An example scenario where we have three routing domains: $\mathbb{R}_A$, $\mathbb{R}_B$, and $\mathbb{R}_C$. Similar to transit networks in the Internet routing, the domain $\mathbb{R}_A$ serves as a transit domain that connects the other two routing domains. Each of these routing domains have their own internal GLookupService, as well as a global GLookupService. Each routing domain has border GDP routers (gray) as well as internal GDP routers (blue).

6.1.2 Example deployment scenario

Using the first principals described in the previous chapter and the extension to custom topologies, we discuss a somewhat more realistic deployment scenario with an example topology shown in Figure 6.2 including a transit network.[7] The discussion serves at least two purposes: (1) reinforce the principals with a slightly more complicated example, and (2) demonstrate that the GDP network is, in fact, applicable to real world situations by providing sufficient control (and policy specification) to routing domains to manage the path taken by messages.

Figure 6.2 shows the final topology that various GDP routers are connected in. We assume that all domains in Figure 6.2 are running the GDP network with custom topologies as shown in the figure. We also assume that the local GLookupService for each of $\mathbb{R}_B$ and $\mathbb{R}_C$ push the appropriate information to the upstream network $\mathbb{R}_A$, which in turn populates the global GLookupService; all the propagation of information through GLookupService's takes into account the *scope* related policies as ingrained in the delegations included in the corresponding GLookupService. There are a couple questions and concerns we would like to answers to.

How does a routing domain set up a custom topology inside the domain? Domain administrators for each domain can configure their GDP routers using the *forced* connection between each pair of GDP routers that they want to connect.

[7] Transit network: `https://www.thousandeyes.com/learning/techtorials/transit-provider`

The local GLookupService for $\mathbb{R}_C$, for example, looks like the following before $\mathbb{R}_C$ is connected to the upstream domain.

- $\mathbb{R}_C^1 \Rightarrow (\mathbb{R}_C^1 \rightarrow_R \mathbb{R}_C^2)$.
- $\mathbb{R}_C^2 \Rightarrow (\mathbb{R}_C^2 \rightarrow_R \mathbb{R}_C^1), (\mathbb{R}_C^2 \rightarrow_R \mathbb{R}_C^3)$.
- $\mathbb{R}_C^3 \Rightarrow (\mathbb{R}_C^3 \rightarrow_R \mathbb{R}_C^2)$.
- $\mathbb{R}_C^4 \Rightarrow (\mathbb{R}_C^4 \rightarrow_R \mathbb{R}_C^1)$.
- $C2 \Rightarrow (C2 \rightarrow_R \mathbb{R}_C^3)$.

Note that the GDP routers pass the advertisements they receive to the GLookupService, as in the case of client $C2$. Also, note that there are no $(IP : port)$ pairs; this is because the GDP network is running with a custom topology in our example. Further, note that $C2$ is a normal client, which isn't visible outside the routing domain. Same goes for the client $C1$. However, log server LS in $\mathbb{R}_B$ is visible publicly to allow for outside clients to connect. As such, when LS connects to $\mathbb{R}_B^2$, it issues two $RtCerts$: $(LS \rightarrow_R \mathbb{R}_B^2)$ and $(LS \rightarrow_R \mathbb{R}_B)$. The first $RtCert$ is what is used for intra-domain routing and the second used for making LS visible outside $\mathbb{R}_B$.

How do routing domains set up the transit network connections, and how is it enforced? When the domain administrator of, say, $\mathbb{R}_B$ enters into a transit agreement with $\mathbb{R}_A$, $\mathbb{R}_A$'s domain administrator issues a $JoinCert$ and gives it to the domain administrator of $\mathbb{R}_B$, which essentially authorizes infrastructure from $\mathbb{R}_B$ to connect to an internal GDP router of $\mathbb{R}_A$. The domain administrator of $\mathbb{R}_B$ issues an $RtCert$ that looks like $(\mathbb{R}_B \rightarrow_R \mathbb{R}_A)$; $\mathbb{R}_A$ can then populate the global GLookupService with this $RtCert$ allowing anyone else in the world to validate that $\mathbb{R}_A$ is delegated to route traffic for $\mathbb{R}_B$. Separately, the border GDP router $\mathbb{R}_B^1$ connects to an internal GDP router $\mathbb{R}_A^3$ of $\mathbb{R}_A$ by presenting a $JoinCert$ $(\mathbb{R}_A \rightarrow_J \mathbb{R}_B^1)$.[8] On connection, $\mathbb{R}_B^1$ issues an $RtCert$ $(\mathbb{R}_B^1 \rightarrow_R \mathbb{R}_A^3)$ and treats it as a default route.[9] Because $\mathbb{R}_B^1$ is a border GDP router, it already presumably has an $RtCert$ $(\mathbb{R}_B \rightarrow_R \mathbb{R}_B^1)$, which it then also sends to $\mathbb{R}_A^3$. $\mathbb{R}_A^3$ forwards all these $RtCerts$ to the local GLookupService of $\mathbb{R}_A$.

To ensure that $\mathbb{R}_B$ and $\mathbb{R}_C$ actually use $\mathbb{R}_A$ as the transit network, they don't advertise their availability in the global GLookupService themselves. Instead, they delegate this responsibility to $\mathbb{R}_A$, which $\mathbb{R}_A$ can fulfill by using the domain level $RtCerts$ that it receives at the time of transit negotiations. Also, note that border GDP routers of $\mathbb{R}_B$ and $\mathbb{R}_C$ connect to an internal GDP router of $\mathbb{R}_A$ and not a border GDP router. This is because $\mathbb{R}_B$ and $\mathbb{R}_C$ are (hopefully) paying customers from the perspective of $\mathbb{R}_A$ that are buying a transit service. This exact process is used when a single client seeks routing services from a routing domain (i.e. the Internet Service Provider equivalent

[8] Such information about which border GDP router from $\mathbb{R}_B$ will connect to $\mathbb{R}_A$ is conveyed to $\mathbb{R}_A$ at the time of transit negotiation, so that it can issue appropriate $JoinCerts$.

[9] $\mathbb{R}_B^1$ can simply forward datagrams to unknown destinations to such default route, but it must carefully filter any incoming traffic on such default path and discard any traffic that attempts to access private GDP names that should not be made available outside the routing domain.

in the GDP network); the routing domain issues a $JoinCert$ to the client authorizing it to connect to the given routing domain, and the client issues appropriate $RtCerts$ to the domain.[10]

What happens when a log server LS joins $\mathbb{R}_B$, and how do advertisements propagate? When a log server LS joins $\mathbb{R}_B$, it performs a secure advertisement process with an internal GDP router $\mathbb{R}_B^2$ which results in an $RtCert$ ($LS \rightarrow_R \mathbb{R}_B^2$). This $RtCert$ allows $\mathbb{R}_B^2$ to route traffic on behalf of LS. After issuing $RtCert$, the log server sends all the $AdCerts$ that it is granted by the DataCapsules it hosts. $\mathbb{R}_B^2$ forwards the $RtCert$ and all the $AdCerts$ to the local GLookupService, which makes the information available to any other internal GDP router that asks.

For LS (and any DataCapsules it hosts) to be visible outside $\mathbb{R}_B$, LS must issue another $RtCert$ ($LS \rightarrow_R \mathbb{R}_B$); this $RtCert$ is issued to the routing domain itself, which is forwarded by the local GLookupService to upstream GLookupService. In case an external entity wishes to reach LS, it can acquire this $RtCert$ ($LS \rightarrow_R \mathbb{R}_B$) from appropriate sources. Such an external entity can then recursively query for $\mathbb{R}_B$, and then for the border GDP routers, and eventually reach LS.

Client $C1$ wants to talk to the log server LS in the same routing domain. What happens? Assuming that $C1$ has finished connecting, it sends a datagram with the destination LS. The GDP router $\mathbb{R}_B^5$ looks for LS in its local forwarding table. If there is no entry for LS, $\mathbb{R}_B^5$ recursively queries the local GLookupService. Such recursive queries to the local GLookupService result in $\mathbb{R}_B^5$ performing a graph traversal starting from LS. At each step of the recursive query, $\mathbb{R}_B^5$ either (1) reaches a node it knows how to talk to, or (2) finds a cycle in the graph, where it terminates the branch it is following, or (3) exhausts all possible branches and realizes that it does not have a path to LS. For the case where there is a path, it just picks that path and sends the datagram to the appropriate next hop.

Note that in our actual implementation, the next hop GDP router follows the same process. However, there is no reason that the initial GDP router sends the information it queried to the next hop, since such information is in the form of verifiable $RtCerts$ and $AdCerts$. Further, for intra-domain lookup, a local GLookupService can perform this path computation and simply return the verified results to the client.

Client $C2$ wants to talk to a log server LS in a different routing domain. What happens? If a client $C2$ in a different routing domain $\mathbb{R}_C$ wants to talk to the log server LS, then it follows the same process as client $C1$. However, because there is no local LS in the routing domain $\mathbb{R}_C$, the GDP router $\mathbb{R}_C^3$ forwards the datagrams to a border GDP router, which in turn sends them to the internal GDP router $\mathbb{R}_A^1$ based on the default path strategy.[11] $\mathbb{R}_A^1$ can then query its local GLookupService to follow the following chain of delegations: $LS \Rightarrow \mathbb{R}_B \Rightarrow \mathbb{R}_B^1 \Rightarrow \mathbb{R}_A^3 \Rightarrow \mathbb{R}_A^2$. Once the message reaches $\mathbb{R}_B^1$, it can find LS using the intra-domain routing process as in the case of $C1$.

[10] If the client desires to be publicly available allowing people to route to it via the parent routing domain, the client must issue an $RtCert$ to the domain, in addition to the $RtCert$ that it issues to the internal GDP router. If, on the other hand, the client wishes to only access other services but not be visible to outsiders, then it only needs to issue only the latter $RtCert$.

[11] Note that if $C2$ has not been advertised to $\mathbb{R}_A^1$ in the past, then $\mathbb{R}_A^1$ will reject the incoming traffic from an unknown source. This can be easily fixed with first sending a chain of $RtCerts$ that allows $\mathbb{R}_A^1$ to verify the $\mathbb{R}_C^1$ can route traffic on $C2$'s behalf.

6.2 Generalized evaluation of certificates/delegations

Recall that $RtCerts$ and $AdCerts$ can be chained; e.g. $(X \rightarrow_A Y)$; $(Y \rightarrow_R R)$ implies that 'R can route traffic for X'. However, with the introduction of other type of cryptographic delegations ($JoinCerts$ and $OwnCerts$), the cryptographic delegations form a directed graph (see Figure 6.3 for an example). Given a set of delegations (i.e. a subset of the graph), a GDP router can use the certificate chain evaluation criteria to assert the validity of advertisement for a given name. With a few different type of delegations, validating a number of certificates together becomes non-trivial. In this section, we present a number of rules that a GDP router (or anyone else) can use to validate whether a given set of delegations allow a specific entity to advertise for a given name.

Let's first summarize the four types of cryptographic delegations that we have discussed so far:

- $AdCert$: A cryptographic delegation issued by object owners to servers for delegating hosting of such objects (e.g. DataCapsules). An $AdCert$ is presented to a GDP router by the server *after* it has completed the challenge-response phase of the secure advertisement process. $AdCerts$ are visible to applications for enabling end-to-end security properties.
- $RtCert$: A cryptographic delegation issued for delegating routing functionality (i.e. send/receive messages for the given name). Such delegations are quite flexible in terms of issuer/issuee. For example, the issuer could be a client, a GDP router, or even a routing domain. Issuees, for example, could be GDP router, an $(IP : port)$ pair, or even routing domains.
- $JoinCert$: A cryptographic authorization issued by domain administrator of private domains to clients, log servers, etc. A $JoinCert$ is presented by the client to an internal GDP router during the secure advertisement process.
- $OwnCert$: A cryptographic delegations that represents resource ownership and allows a given entity (GDP router, log server, etc.) to act as an agent of the issuer. An $OwnCert$ is issued by domain administrator to sub-domains, GDP routers, etc.

While we will not go in detail, these delegations can also include policy specifications and chaining restrictions to provide even more fine-grained control on under what condition is a specific chain valid, and where the advertisement can be sent; such policies are also to be accounted for during validity evaluation of a specific claim for a name advertisement. As an example, the owner of a DataCapsule can include a white-list of allowed routing domains, or an $RtCert$ issued during a on-demand connection between two GDP routers can have a chaining restriction that prevents any further delegations to be appended to the chain delegations.

In addition to making a *valid* or *invalid* decision on a given chain, each GDP router independently analyzes the 'scope' of advertisement for a given name and decides whether it should be sent elsewhere. For example, a border GDP router (or a local GLookupService, depending on how a routing domain is configured) filters all name advertisements that do not meet the global publication criteria. An advertisement for a specific name could be kept strictly local to the GDP router, within the routing domain, to certain levels of hierarchy involving parent domains, or be made available globally.

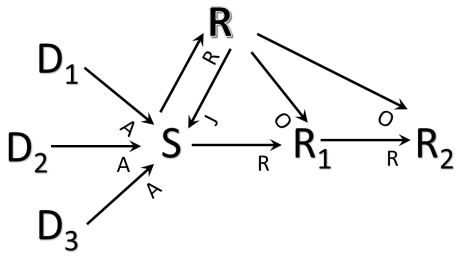


Figure 6.3: A visual representation of delegations in practice. D_i are DataCapsules delegated to log server S by their respective owners. Individual $(D_i \rightarrow_A S)$ can have policy specification such as whether a given D_i is public or restricted in scope. S joins the routing domain $\mathbb{R}$ by connecting to a GDP router R_1; R_i are owned by $\mathbb{R}$. S is authorized to join $\mathbb{R}$ because it has been granted a $JoinCert$ by $\mathbb{R}$. S grants $(S \rightarrow_R R_1)$ as well as $(S \rightarrow_R \mathbb{R})$; the latter is only necessary if either S or any of the D_i's are exposed to the world outside $\mathbb{R}$ and the domain administrator does not want to expose the internal topology to the outside world. If, for example, D_1 is global but D_2 and D_3 are restricted to within $\mathbb{R}$ as specified in the corresponding $AdCerts$, $(D_1 \rightarrow_A S)$; $(S \rightarrow_R \mathbb{R})$ will be a valid chain but $(D_2 \rightarrow_A S)$; $(S \rightarrow_R \mathbb{R})$ will not be valid.

6.2.1 Evaluation rules: An algebra for secure routing

There are a number of self-consistent rules for such validity evaluation that allow the GDP network to achieve the desired security properties under the threat model discussed in the previous chapter. Let's see what the general rules are:

1. Each individual link in the chain can be generated completely independently with a different lifespan.[12]
2. For a chain to be valid at a given time, each individual link in the chain should be valid. For example, if in a chain $(X \rightarrow_A Y)$; $(Y \rightarrow_R R)$, the $AdCert$ expires at time t_0 and $RtCert$ expires at a later time t_1, then R can only claim to route traffic for X before time t_0. It can still, however, route traffic for Y till time t_1.
3. The order of validation follows the direction of the graph, e.g. for $A \rightarrow B$; $B \rightarrow C$, validation of $A \rightarrow B$ must be done before $B \rightarrow C$.

Now, let's see the rules that are specific to each type of delegation.

- $AdCert$: An $AdCert$ authorizes the issuee to advertise for the name of the issuer.[13] For example, a DataCapsule X can be delegated to a log server Y with an $(X \rightarrow_A Y)$.
  - In any certificate chain, there can be at most one $AdCert$; such $AdCert$, if present, is always at the beginning. For example, $(X \rightarrow_A Y)$; $(Y \rightarrow_A Z)$ is not a valid chain.[14]

[12]Note that for conciseness, our representation omits expiration time.

[13]In the GDP, we generalize this to a conglomerate of storage servers.

[14]In the current GDP design, this is to ensure that service providers can't simply outsource servicing of names to others without the knowledge of end-users.

  - An $AdCert$ can have chaining restrictions on the chain as put by the $AdCert$ issuer. A chaining restriction in an $AdCert$ represents what this $AdCert$ can be chained together with. For example, an $AdCert$ can include a whitelist of routing domains that must be used for routing.[15]
  - $AdCerts$ are exposed to the applications, and they can be used for any application-level end-to-end verification.[16]
- $RtCert$: An $RtCert$ grants delegation of routing responsibilities to a GDP router or a routing domain.
  - Just like $AdCerts$, $RtCerts$ can also include chaining restrictions.
  - $RtCerts$ can be arbitrarily chained, except for restrictions placed earlier in the chain.
- $OwnCert$: An $OwnCert$ codifies resource ownership.
  - An $OwnCert$ is issued by an *organization*, such a routing domain.
  - An $OwnCert$ implies transitivity, i.e. $(X \rightarrow_O Y)$ and $(Y \rightarrow_O Z)$ is equivalent to $(X \rightarrow_O Z)$.
  - Any part of the infrastructure can have at most one owner. Thus, a hierarchy of $OwnCerts$, when represented as a graph, is a strict tree with each child having at most one parent.
  - An $OwnCert$ implies $JoinCert$ (see $JoinCert$ discussion below).
  - An $OwnCert$ also codifies whether a $JoinCert$ is required for a sub-tree of the organizational hierarchy or not. Any $JoinCert$ requirements are enforced for the entire sub-tree. For example, if X requires a $JoinCert$, and $(X \rightarrow_O Y)$; $(Y \rightarrow_O Z)$, then both Y and Z must require a $JoinCert$.
- $JoinCert$: $JoinCert$ represents authorization to join a subtree of the organizational hierarchy represented by $OwnCert$ and is always granted to an active host, i.e. in $(R \rightarrow_O X)$, R is either a sub-domain or a GDP router, and X is an active host (client or log server).
  - $JoinCerts$ are for use only within the issuing routing domain. They don't need to be passed to, for example, a global GLookupService.
  - Border GDP routers do not require a $JoinCert$ from those connecting; their job is to facilitate communication from outsiders. Instead, they must perform filtering of routes. Only internal GDP routers require a $JoinCert$.
  - An $OwnCert$ implies a $JoinCert$.
  - A $JoinCert$ created by a node in the organizational hierarchy (as defined by $OwnCerts$) implies a valid $JoinCert$ for the entire sub-tree rooted at the issuing node.
  - If a $JoinCert$ is needed for a given domain, all sub-domains must necessarily require $JoinCert$.
  - For an organizational component that requires a $JoinCert$, a certificate chain is valid only if every single node in the chain (except for content that starts $AdCert$) has a valid

[15] A domain-specific language can be used for such policy specification. However, we consider the details as out of scope the dissertation.

[16] See GDP-protocol in chapter 4 for an example.

JoinCert. The only exceptions are for default paths, or for the part of the chain before a border GDP router.
- *JoinCerts* are issued only to active entities that must go through a challenge-response phase.
- *JoinCerts* are not needed for names advertised via *AdCert* delegation. The entity delegated to advertise names via *AdCerts*, however, must present a *JoinCert* if the domain requires it.

See Figure 6.3 for an example of some of these rules in practice.

6.2.2 Security analysis of delegations

While a more detailed formal analysis of the security scheme is out of scope of this dissertation, we provide intuition for why the scheme works. Probably the weakest point of the security scheme remains regarding the lifespan of delegations and a lack of a revocation scheme; a GDP router is able to receive traffic on behalf of a client for as long as the entire certificate chain is valid even after the client goes away. Thus, one should be cautious in creating long-lived delegations to limit the window of attack.

At a high level, only the owner of a given name can sign cryptographic statements and generate delegations. Using the flat 256-bit name as the trust anchor ensures the security of verifying the delegations themselves, and one can be reasonably certain that a given delegation wasn't fraudulently signed by an adversary.

The 'secure advertisement' protocol is prone to an on-path adversary in limited form: an active man-in-the-middle can simply hijack an existing TCP connection, kick out a legitimate client immediately after advertisement, and start sending/receiving traffic on behalf of advertised names. However, this is broadly consistent with our threat model that we do not protect against on-path adversaries. Using appropriate nonces during the challenge-response part of the 'secure advertisement' protocol prevents replay attacks.

The certificate chains, in a way, represent the transitive trust graph. Any attacks on the certificate chain by insertion of new entries inherently implies that the transitive trust is broken, e.g. a malicious actor M in a certificate chain $(X \rightarrow_A Y)$; $(Y \rightarrow_R \mathbb{R})$; $(\mathbb{R} \rightarrow_R M)$ simply implies that both X and Y chose their routing domain $\mathbb{R}$ poorly because M can now receive traffic for X and Y and not forward it. Such attacks are out of scope of our threat model. However, note that M cannot forge $(\mathbb{R} \rightarrow_R M)$.

While an *RtCert* and *AdCert* are similar in many respects; an *AdCert* is passed to any RPC-application running on top of the GDP network and allows verification of the identity of a remote entity. Further, an *AdCert* acts as a special marker that ends a certificate chain and ensures that there are no infinite loops during evaluation.

A more interesting situation is that of attempts to undermine network isolation. Can an adversary join a routing domain when not allowed to? What about an adversary that has already broken into

a routing domain? As discussed in the threat model, unless an adversary compromises a router that's on the communication path anyway, they cannot affect routing state of other flows.

6.3 Storage organizations and the GDP network

An interesting case arises when we consider delegation of a DataCapsule to a 'storage organization' instead of a single log server.[17] In our discussion of all the mechanisms thus far, we claimed that for *AdCerts* that delegate hosting of a DataCapsule to a log server, the log server can be generalized to a storage organization. The storage organization can further delegate the DataCapsule hosting to one of the log servers that it owns. From the perspective of the GDP network verifying a log server's claim to host a specific DataCapsule, this generalization works well. If a DataCapsule D is delegated to a storage organization $\mathbb{S}$, then a log server S owned by $\mathbb{S}$ can present an *AdCert* ($D \rightarrow_A \mathbb{S}$) together with the *OwnCert* ($\mathbb{S} \rightarrow_O S$). With a combination of these two delegations, S can convince the GDP network (and other clients) that it can rightfully advertise for D.

However, there are two problems. First, from a system design perspective, this is slightly problematic for the following reason. Because the storage organization $\mathbb{S}$ could potentially be operating many log servers S_1, S_2, etc., not all log servers may have a copy of a given DataCapsule D. A GDP router R trying to reach D (on behalf of a client) can find all the log servers S_1, S_2, etc. by looking up in the global GLookupService, but it has no way of knowing the correct subset of log servers that host a replica of D. Second problem comes from a security perspective. If one of the log servers S belonging to $\mathbb{S}$ is compromised, it can rightfully claim to advertise *all* of the DataCapsules delegated to $\mathbb{S}$. Even further, if S itself is not compromised but an adversary can somehow observe traffic to S, it can influence the routing state of the GDP network by using publicly available delegations ($D \rightarrow_A \mathbb{S}$) and ($\mathbb{S} \rightarrow_O S$), and force traffic to any given DataCapsule D delegated to $\mathbb{S}$ towards S.

There are a few mechanisms and strategies to address these challenge, each of which could be applicable in a given context. We discuss a few of them below.

The first strategy is to explicitly allow all log servers belonging to a storage organization to receive traffic for any DataCapsule delegated to the storage organization. This strategy works well in the case of collaborative log servers. There are two scenario where this strategy is quite appropriate:

1. A small storage organization that runs a few log servers where each log server, in fact, hosts a replica of all DataCapsules delegated to the storage organization.
2. A very large storage organization with globally distributed infrastructure, where not all log servers maintain a replica of all the DataCapsules delegated to the storage organization but they can *proxy* requests to the correct log server. For this scenario, log servers need an out-of-band mechanism to find what other log servers in the organization host a replica of a given DataCapsule. When a log server receives requests for a DataCapsule that it does not

[17] Recall from chapter 2 that 'storage organizations' are organizations that primarily operate log servers and provide persistent storage for DataCapsules as a service.

host locally, it can *proxy* the requests to an appropriate log server (by using the log server's name).[18] Further, depending on the frequency of access of a given DataCapsules across geographic regions, the storage organization may migrate DataCapsules to provide the best quality of service to the clients.

A second strategy is to expose the internal details of a storage organization in a limited form to DataCapsule owners, and ensure that a specific DataCapsule is delegated not to the organization as a whole but to a specific subset of infrastructure. This strategy does not require *all* log servers to collaborate with every other log server in the organization. There are two mechanisms to implement this strategy:

1. An organization could expose the geographic placement of its resources to DataCapsule owners by creating sub-organizations, and DataCapsule owners can delegate DataCapsule hosting to a specific sub-organization instead. Within a sub-organization, any log server should be able to serve requests for a DataCapsule delegated to that sub-organization using either of the approaches discussed in the first strategy.[19]
2. Instead of exposing details of the geographic placement of infrastructure, a storage organization could work at a more abstract notion of a *replica group*. As the name suggests, a *replica group* is a group of log servers that are presumably geographically separated and maintain replicas of a DataCapsule. The workflow and mechanisms are similar to the sub-organization discussion above: the storage organization partitions log servers in *replica groups* (as identified by the $OwnCerts$ it issues), and DataCapsule owners pick a *replica group* to place their DataCapsule in by including the replica group in the $AdCerts$ that they issue. However, for a chain of delegations $(D \rightarrow_A \mathbb{S})$, $(\mathbb{S} \rightarrow_O S)$ to be valid, the included *replica group* in both delegations should match.[20]

6.4 A scalable GLookupService

A core component to enable the GDP network is a scalable GLookupService. In the previous chapter, we merely assumed the existence of a globally distributed key-value store. In terms of actually making this work, we reuse an existing system called Redis [83], which is a scalable key-value store with support for replication.

In terms of the interface that GLookupService needs to support, there are two key operations: a `get` operation to lookup the routing information associated with a GDP name and a `put` operations

[18] For example, AWS S3 buckets follow this model: S3 buckets are located in a specific AWS region as specified by the bucket owner. If a client accesses the bucket by connecting to a different AWS region, the requests are proxied by the AWS infrastructure without the user noticing.

[19] An example is AWS EC2 infrastructure, where a user picks the parts of the EC2 infrastructure to host services in at the granularity of regions (i.e. data centers). Users connect directly to the designated parts of the infrastructure rather than being proxied (as in the case of S3 buckets).

[20] Note that implementing *replica groups* requires local filtering of delegations by a GDP router based on *replica groups*. Alternatively, the GLookupService interface could be amended to include support for optional fields. We consider these as directions for future research.

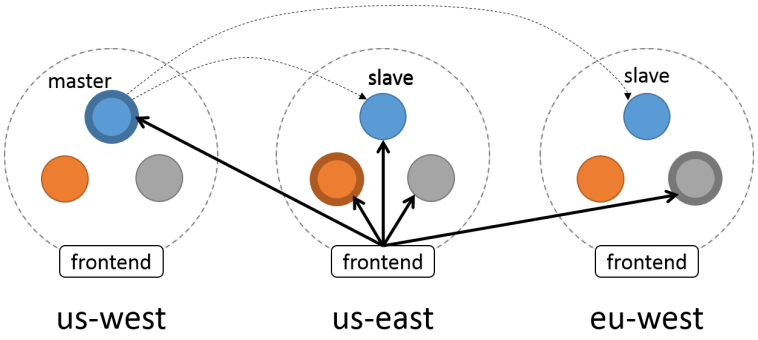


Figure 6.4: A reference deployment of global GLookupService using Redis as the backend. We deployed it at three different Amazon EC2 data centers in the world (called a *site*). We also divide the keyspace in three roughly equal partitions (marked with different colors), and ensure that the masters are spread across the sites. Each site also has a UDP front-end exposed to GDP routers; this front-end acts as a Redis client and uses local nodes for reads, but sends updates to the master (as dictated by Redis architecture).

to add new *RtCerts*, etc. As for performance, it is extremely desirable to be able to quickly lookup names regardless of location; a GDP network where everyone in the world needs to request route information from a GLookupService in a single *centralized* location wouldn't perform well. Thus, we must have a way to replicate the same information at multiple places and keep it reasonably up to date. Quick reads by widely replicating information comes with the trade off that any updates may not be immediately visible; this is because of the delays in information propagation. However, this is an acceptable trade off in the context of the GDP network.

With these design goals and constraints, we chose to use Redis [83] for a prototype deployment of the global GLookupService. Redis is an open-source in-memory key-value store that supports replication. Values in Redis can be just plain strings, or more sophisticated data structures such as lists, sets, etc. The entire key-space is divided into 16384 key-slots, and each key is deterministically mapped to a key-slot based on the CRC.[21] The entire data is sharded based on the key. Each server is responsible for a specific set of key-slots, and could act as either a master node or a slave node. Writes only happen on master, but reads can be performed from any node. If the master goes down, a slave is automatically elected master.

For our global GLookupService based on Redis, the keys are the GDP name and the values are a `set` of all the valid *AdCerts* and *RtCerts* associated with the GDP name. We maintain a front-end that serves the GDP network requests for update and lookup via UDP on one end, and acts as a Redis client to fetch the requests on the other end. This front-end also lazily cleans up any *AdCerts* or *RtCerts* that are expired. This front-end is a fairly lightweight process that maintains persistent connections to a number of Redis servers (see Figure 6.4).

[21] Cylic Redundancy Check.

The actual data is replicated and stored at a number of data centers in the world (see Figure 6.4); we perform a geo-replication by using a smart front-end coupled with clever placement of replicas.[22] We try to replicate *all* data in each data-center in a way that the masters are spread over various data centers. Each data center has a single front-end that GDP routers talk to. For read requests, this front-end uses the closest replica (presumably in the same data center). Updates must go to the designated master for the key-slot that the GDP name belongs to; such master could be in a far away data center.

While such an architecture seems to work for the moment, this is merely a proof-of-concept using existing tools and systems as much as possible. This prototype global GLookupService also demonstrates an example deployment strategy in a federated way: replica data-centers could be operated by real-world organizations. While updates are targeted to a specific node in our proof-of-concept, such design is purely because of how Redis operates. In a real federated deployment, a non-leader based approach could be used where updates can be applied to *any* node; these updates can then be propagated to other nodes and merely merged in the form of a set union of unexpired $RtCerts$ and $AdCerts$ (with an additional verification step to prevent garbage). In such a model, organizations do not trust each other for correctness but merely completeness of information, which matches our service provider oriented threat model.

[22] A form of geo-replication is supported in the Redis proprietary enterprise version.

Part III

The Results

Chapter 7

Implementation and Evaluation

In the first part of the chapter, we briefly describe two software implementations of the GDP. The first implementation is a research prototype written purely in Python. The second implementation is developed by a group of collaborators in C/C++, and is intended to be a production-level system. The former is what we claim to be a contribution of this dissertation, and we hope the latter to be the future of the GDP. For this reason, we focus more on the research prototype in this dissertation.

In the later part of this chapter, we show empirical evaluation of the GDP design and architecture with the help of the prototype implementations.

7.1 The GDP research prototype

The purpose of this implementation is threefold: first, it provides a proof of concept to demonstrate that the proposed design of the GDP can actually be implemented; second, it provides a reference implementation that can be used as a starting point for a more refined system suitable for production use; third, it serves as a platform for experimentation and refinement of the GDP design itself.

With these goals in mind, we chose to develop this prototype in Python. Even though Python is an interpreted language with significant overhead in runtime performance, it allows us to do quick prototyping and move rapidly. Further, with Python's support for extensions written in C or C++, this prototype can be extended to include specific functionality written in low-level languages. The research prototype is currently over 6k lines of code (SLOC), where a line of code is any "non-blank, non-comment line".[1] The total number of lines, including blank and comment lines, is over 12.5k. The code is available in a `git` repository hosted at `https://repo.eecs.berkeley.edu/git/projects/swarmlab/gdp-v1.git`. In the following discussion, we will refer to the research prototype as simply "the prototype".

The prototype heavily uses existing Python libraries and frameworks. Most notably, it uses 'Python Twisted' as an event-driven programming framework for handling almost all of the network

[1] Measured using David A. Wheeler's 'SLOCCount'.

I/O.[2] All network messages and many data structures are specified via Protocol Buffers, which allows for extensibility and language agnostic operations.[3] All persistent data is maintained in SQLite, which provides a lightweight database engine that does not require a standalone database process.[4] For cryptographic operations, the prototype can use either `pyca/cryptography` or 'M2Crypto' as a backend.[5] For a number of operations that can be expressed in the form of well-known graph algorithms, we use 'NetworkX' as a graphing library.[6] Finally, we use 'PyTest' for testing and validating the software.[7]

7.1.1 GDP features implemented by the prototype

The prototype is a work in progress and implements most, but not all, of the GDP design that we described. Most of the security features are implemented; here is a small, non-exhaustive list of such features:

- All GDP names derived from the hash of metadata, thus enabling the name as a trust anchor.
- For a DataCapsule, records are linked using appropriate hash pointers. This property, together with GDP names generated from hash of metadata, allows for creation of verifiable `RecContainers`.
- `RecContainers` (subsection 3.3.2) are implemented and can be used as a secure transport for all DataCapsule related operations, such as *append*, *read*, *subscribe*, and both online and offline replication.
- Two type of delegations—*AdCert* and *RtCert*—that use the GDP name as the trust anchor have been implemented.
- Support for secure advertisement via a challenge response between a client connecting to the routing fabric and the GDP router that it is connecting to.
- Ingress filtering of all GDP-PDUs by a GDP router to prevent blatant source address spoofing: a GDP router ensures that the remote side (whether it is another GDP router, a client, or a log server) has provided sufficient proof that it can send datagrams with a given source address.
- Egress verification of routes by a GDP router: before forwarding a GDP-PDU to a remote party, a GDP router verifies that the remote side is, in fact, authorized to receive GDP-PDUs with the given destination address.
- Support for secure acknowledgments from a log server to clients (and to other log servers during synchronization). Provides end-to-end transport level security against tampering of information by intermediaries.
- Verifiable information in GLookupService, that ensures that a GLookupService needn't be trusted for correctness of information.

[2] `https://twistedmatrix.com/trac/`
[3] `https://developers.google.com/protocol-buffers/`
[4] `https://www.sqlite.org/index.html`
[5] `https://cryptography.io/en/latest/` and `https://pypi.org/project/M2Crypto/`.
[6] `https://networkx.github.io/`
[7] `https://docs.pytest.org/en/latest/`

7.1.2 Software components and code organization

The software closely follows the design described in earlier chapters. There are, however, a few components that are grouped together for minimizing duplication of effort. The whole code base is organized as a Python package called `gdp` (see below), with a number of sub-packages that implement the client, log server, routing infrastructure, etc. We only provide a brief overview of our research prototype in this dissertation, but we encourage an interested reader to explore the code.

```
gdp
|-- client          # client side interaction with infrastructure
|-- ds              # common data structures as Protocol Buffers
|-- gdprpc          # common RPC library
|-- __init__.py
|-- routing         # routing components
|-- server          # log server for persistent storage
‘-- utils           # common utility functions
```

A Common features and code

At the heart of this research prototype is a common RPC library (`gdp.gdprpc`) that handles requests and responses at the GDP PDU level, and can be reused for various GDP components that connect to the routing fabric such as clients and log servers. Even the GDP routers include this same RPC library to handle advertisement related requests and responses.

This RPC library provides a base class called `Agent` that other components in the GDP can inherit from. An `Agent` represents a very simple active entity that knows how to (1) handle (and maintain) *AdCerts* and perform secure advertisements for a number of names in the network, and (2) establish a secure channel between a client and a DataCapsule (or any other service with a number of replicas), which allows for a client to reason about the end-to-end transport level security properties.

Other than the RPC interface, a number of common utility functions are also grouped together in the form of a `gdp.utils` sub-package. This includes logging, common cryptographic operations, and handling records and RecContainers.[8] For our cryptographic operations, we use the Python `cryptography` package. We use SHA256 as our hash algorithm and ECDSA for signatures with NIST curve `sect283r1`. The choice of ECDSA is to keep key and signature sizes small (78 bytes).

Finally, various Protocol Buffer messages and data structures are separated into a sub-package called `gdp.ds`. Most importantly, the Protocol Buffer definitions allow for an easy migration path to alternate implementations, one networked component at a time.

[8]Recall that a RecContainer abstraction for enabling an efficient on-the-wire and archival storage format. RecContainer targets an amortization of computational and storage costs over a number of records, while enabling a self-sufficient collection of records.

B Routing infrastructure

The routing infrastructure, included in sub-package `gdp.routing`, implements two key components: the GDP router and the GLookupService.

The GDP router implementation can be initialized/configured in a number of ways to accommodate various scenarios: (1) a standalone mode useful for testing, (2) a mode with a specified 'default route' for creating arbitrary hand-crafted topology, and (3) a fully functional mode with a configured GLookupService that can be used for sharing delegation state across GDP routers. The GDP router is internally split into two components: a routing fabric and a routing-agent. The routing-agent handles the secure advertisement process and inserts verified state in an internal database called `RouterDB` (essentially a forwarding table). The routing fabric performs the actual switching of incoming PDUs by looking up verified routes in the forwarding table.

Additionally, we introduce the concept of a CertGraph that can handle a complex graph of delegations (see chapter 6).[9] A CertGraph provides a simple interface with two operations, `insert-cert` and `query-path`, and allows for a modular way of handling arbitrary complex routing state.

Our prototype also provides two implementations of GLookupService with different back-ends: (1) a simple standalone implementation where the in-memory state is stored in a Python dictionary, which is useful for testing, and (2) a distributed back-end for the state that is maintained in an off-the-shelf Redis cluster. Both implementations respond to *GET* and *PUT* queries over UDP.

Our Redis based GLookupService is deployed in three geographical locations worldwide in different Amazon EC2 regions: US west coast, US east coast, and Ireland. Using Redis allowed us to quickly prototype our GLookupService without needing to worry about replication and fault-tolerance. Our deployed Redis cluster consists of a total of 9 nodes. Among these 9 nodes, there are a total of three masters—all in different zones. Each master has two slave in the other two regions that replicate data on the master. Thus, each key is replicated in each of the three region. In each of the three regions, we also run a GLookupService front-end that enables GDP routers to issue `GET/PUT` commands. Internally, this front-end sends write requests to the master node for the given key, but serves read queries from the closest node.

C Log servers and persistent storage

Log servers in our prototype are implemented as a sub-package `gdp.server`. The log server is a subclass of `Agent` that maintains a list of DataCapsules that it has been delegated to serve. Additionally, the log server maintains $AdCerts$ for these DataCapsules, which enables the log server to advertise for these DataCapsules to the GDP network.

To store persistent data, the log server internally uses a separate SQLite database for each DataCapsule, which allows for quick lookup and response to client queries based on database indices. The log server can serve the following requests to clients: `CreateReq, MetadataReq,`

[9] The CertGraph implementation currently supports $AdCert$ and $RtCert$, but can be extended to handle $JoinCert$ and $OwnCert$ as well.

`AppendReq, ReadReq, HeartbeatReq, SubReq`.[10] The log server also maintains an in-memory copy of active DataCapsules. This in-memory copy is represented as a graph of nodes connected with hash pointers, which allows for quick construction of proofs for read requests.[11]

The log server also implements both online and offline synchronization. Online synchronization is implemented via a `FwdAppend` request, which forwards appends sent by a writer to all known replicas. Offline synchronization for any missing state is implemented via an anti-entropy syncing algorithm.

D Clients

In our prototype, a client is a subclass of `Agent` and provides some basic initialization of DataCapsules. The 'Client' class lives under the sub-package `gdp.client` and enables creating new DataCapsules, or opening existing DataCapsules in either read only mode or a single writer mode. The result of these operations is a DataCapsule object, which further enables reads, appends, and subscriptions. Appends and reads can be performed in a variety of modes (e.g. single vs. multiple records at a time, synchronous vs. asynchronous modes, etc.). Any read operations automatically trigger the verification of data integrity by validating proofs provided by the log server.

While the actual operations are performed using the DataCapsule object, it is the 'Client' that maintains appropriate local state. For example, a writer client maintains the required state in a non-volatile store maintained in a disk-backed SQLite database. The writer client also implements the logic necessary for calculating if hashes should be kept around for constructing hash-pointers in future records. Further, our current prototype supports any arbitrary hash-pointer linking strategy that can be described as a Python generator function.

7.2 The GDP production system

In addition to the research prototype, there also exists another prototype of the GDP developed by a group of researchers.[12] It is mostly written in low-level languages (C and C++). We call this the production system because there are active users for this system and we maintain a basic server-side infrastructure in Berkeley. This infrastructure primarily includes four log servers and four GDP routers in two different physical locations, and a few other key services to improve the usability of the system. The production system is organized in the following architectural components.

- The GDP library written in C, called `libgdp`. This library includes the functionality for: (1) interacting with records (creation of new records, verifying existing records, making records available to the applications, etc.), (2) an RPC layer that keeps track of requests and responses

[10] Recall that in our current design, subscriptions are maintained by the log server.

[11] Proof construction is equivalent to finding the shortest path in this in-memory graph.

[12] The author of this dissertation had relatively minor role in the software engineering effort for this prototype. We include the prototype in the dissertation for two reasons: (1) to serve as a reference for the current state of the GDP, and (2) for the sake of completeness; we use this prototype for a few benchmarks in the later parts of this chapter.

to a remote entity (including retransmissions if necessary), (3) serializing/deserializing requests and responses to the GDP PDU format, (4) interacting with the GDP network (e.g. advertising names). Readers and writers link against `libgdp` to interact with DataCapsules.

- The log server implementation, called `gdplogd`. `gdplogd` also links against `libgdp`. `gdplogd` provides persistent storage for DataCapsules by keeping on-disk state in SQLite databases.[13]
- The GDP router implementation, which is written as a Click [84] module. The Click GDP router implementation uses a separate router-to-router protocol that supports fragmentation and reassembly of GDP PDUs into UDP datagrams for the most optimal use of network bandwidth. We call this router-to-router protocol as GDP-in-UDP tunneling. This GDP-in-UDP protocol also handles packet loss by using a NAK-based strategy (i.e. the recipient requests for UDP fragments that it didn't receive).
- The GLookupService implementation, called `gdpribd`. The `gdpribd` keeps state in a MySQL/MariaDB database, and uses a custom front-end to serve requests from the Click GDP router. Internally, `gdpribd` uses a third party graph engine to keep state of the network graph and calculate shortest path between a pair of endpoints. Note that in the current design of the Click GDP router and `gdpribd`, there is no support for $AdCert$ or $RtCert$ delegations. The `gdpribd` serves as a trusted oracle with a global view of the network state.
- A number of language bindings on top of `libgdp`, which allow users to write applications in high-level languages. We currently support language bindings for Python and Java. There is partial support for JavaScript as well.
- A number of applications and services on top of this production system. Notable examples include a number of protocol translators (MQTT, CoAP, HTTP via RESTful gateway), in-browser data visualization services, a TensorFlow CAAPI, audio/video storage in DataCapsules.

While this production system does not support all the features implemented by the research prototype, it is significantly more mature in terms of software quality and usability. In the current state, the two prototypes are not compatible with each other either. We hope that the features implemented by the research prototype will eventually make their way into the production system, and that the two versions will be compatible with each other.

7.3 Evaluation of the GDP design

In this section, we show how the system performs in practice. We first attempt to answer the broader performance related questions with the help of some macro benchmarks where we compare the performance of the GDP and DataCapsules with existing systems. Then, we go deeper to discuss performance of some of the GDP specific operations with help of some micro benchmarks.

[13] In the current implementation, each DataCapsule is stored in a separate SQLite database.

7.3.1 Macro benchmarks

The GDP and DataCapsules target an ambitious goal of providing secure ubiquitous storage and a native routing fabric for applications. In this section, we look at how well the GDP performs as a system regardless of the security goals. We first look at the overall system performance with the help of a Tensorflow CAAPI (recall from subsection 4.2.2). Then, we look at how the GDP network compares with existing IP infrastructure for throughput and latency. Finally, we consider the storage performance of DataCapsules with NFS in terms of throughput and IOPS (I/O operations/sec).

Note that some of the following benchmarks are done on a public cloud (Amazon EC2). Using a cloud infrastructure for performance measurement has the benefit of standardization of underlying infrastructure and ability to scale in terms of resources. However, a downside of a public cloud is potentially erroneous measurements because cloud resources are inherently shared. To account for spatial and temporal variability, we repeat the measurement a number of times and ignore the outliers on the lower-end of performance.

Also, note that the evaluation is primarily performed using the research prototype, unless otherwise noted. We must reemphasize that this is merely a prototype in a high level language. The observed performance can be much better if software engineering were the focus of the dissertation (which it is not). Hence, absolute numbers are less important than the relative performance of various operations.

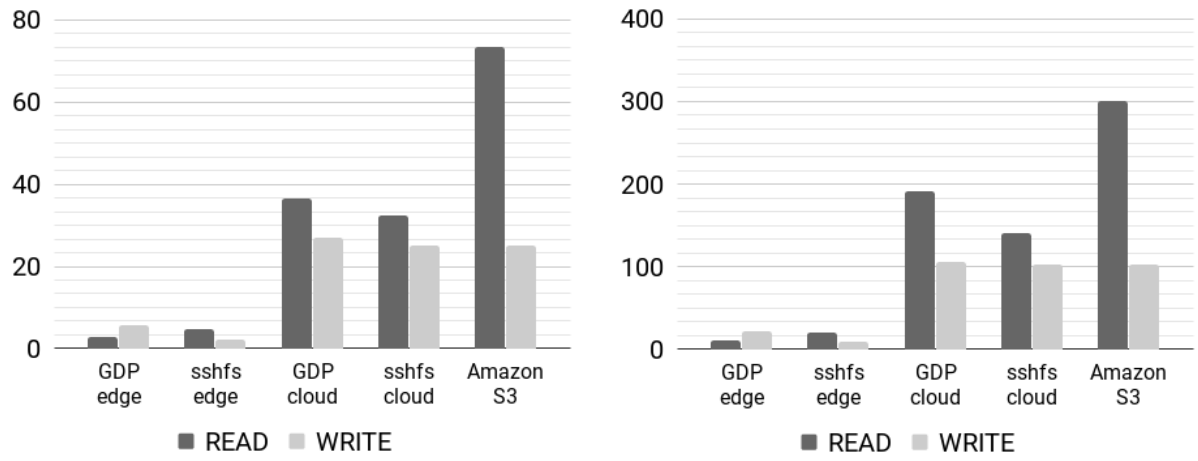
A Tensorflow CAAPI: An application level benchmark for the GDP

In this section, we illustrate the real world usage of the GDP and DataCapsules with the help of a case study on machine learning for robotics applications at the edge (see subsection 2.5.2). Recall from subsection 4.2.2 that we developed a CAAPI for TensorFlow to help with our Secure Fog Robotics initiative (see subsection 2.5.2).

TensorFlow supports custom filesystem plugins that allows an arbitrary system to be used for maintaining and interacting with all persistent state (e.g. training/test data, training progress, models, etc.). By default, TensorFlow ships with a number of filesystem plugins to use cloud storage such as Amazon S3 or Google Cloud Platform storage. Using the same plugin functionality, our CAAPI enables existing TensorFlow code to use DataCapsules for maintaining persistent state. Note that this CAAPI was developed to work with the production system, and not the research prototype.[14]

For the purpose of benchmark, we are interested in the overall time it takes to load or store data using our CAAPI. Specifically, we use two publicly available machine learning models as the sample data for our experiments. The first model is a small model of size 28 MB

[14]Note that this CAAPI does not support encryption or signatures on records. As such, the time measurements reported in Figure 7.1 do not include time needed for signatures or encryption/decryption. This limitation does not apply to the other experiments reported in this chapter.



(a) Model `ssd_mobilenet_v1_coco`.

(b) Model `faster_rcnn_resnet50_coco`.

Figure 7.1: Read/write times (seconds) for Tensorflow CAAPI comparing the GDP to other options. We report the time taken for reading/writing two different pre-trained models (averaged over 5 runs). The left model has size 28 MB, and the right model is 115 MB in size. Smaller is better.

(`ssd_mobilenet_v1_coco`), and the second model is a much larger model with size 115 MB (`faster_rcnn_resnet50_coco`).[15]

For our experiments, we place a client that runs the CAAPI in a residential network with the Internet bandwidth capped to 100/10 Mbps (upload/download).[16] We run two sets of experiments: first using cloud infrastructure using Amazon EC2 and S3, and then using edge infrastructure placed in the same residential network. Note that the Internet bandwidth cap only applies to the first set of experiments.

For our first set of experiments, we measure time taken for storing or loading models from an Amazon S3 bucket in the closest S3 region. We then run the GDP infrastructure in Amazon EC2 in the same region as used for the S3 bucket. We also compare the performance against SSHFS [86] on the same host as our GDP infrastructure. This is accomplished by mounting a remote directory on local machine using SSHFS, and treating the mount as a local file.[17] For our second set of experiments, we use the GDP infrastructure in local environment using on-premise edge resources. Once again, we run SSHFS for comparison. The results for loading/storing the model across all these set of infrastructures are presented in Figure 7.1.

A few observations from these experiments: First, the GDP and DataCapsules provides performance somewhere between that of SSHFS and S3 when using the cloud infrastructure. As

[15] TensorFlow detection model zoo: `https://github.com/tensorflow/models/blob/master/research/`

[16] 100/10 Mbps is a good representative of an average household Internet connection in United States [85].

[17] We note that the TensorFlow's S3 implementation for loading data is not particularly efficient, thus the non-standard use of SSHFS with TensorFlow provides a better comparison.

expected, the performance when using edge resources is orders of magnitude better. Second, storing the models in the cloud is constrained by the limited upload bandwidth, which is why the 'write' time is roughly the same for all the cloud experiments for both the models. Third, the read performance is worse than write performance for cloud. This peculiarity is because of the way TensorFlow uses the filesystem: writes are simply handed to the filesystem plugin (which can then be done asynchronously), but TensorFlow reads the models synchronously (e.g. for sequentially reading a file, a read call for '(offset=$n + x$, size=x)' is issued only after the read call for '(offset=n, size=x)' finishes).

Overall, these experiments show that given equivalent infrastructure, the GDP and DataCapsules provide comparable performance to existing cloud systems (S3). Additionally, the GDP and DataCapsules enable the use of close-by infrastructure for better performance, while enabling better end-to-end security and better control on the data ownership and placement. Even further, the federated nature of the GDP implies that power users can set up their own private infrastructure to achieve a given Quality of Service and still enjoy the benefits of a common platform.

B The GDP network performance

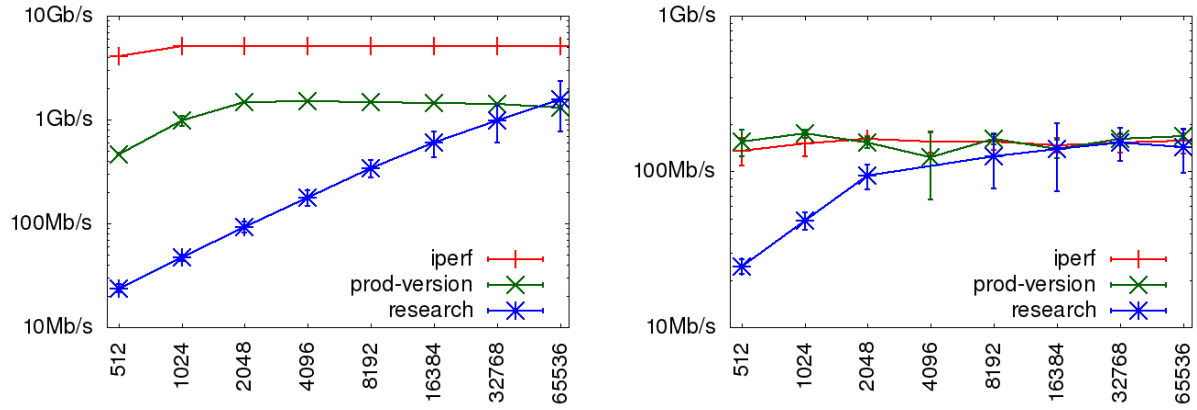
From a user's point of view, the GDP network's main function is to transfer data from a sender to a recipient. In the following set of benchmarks, we compare performance of the GDP network with the underlying TCP/IP infrastructure. Note that because we operate the GDP network as an overlay on top of TCP/IP, these set of experiments essentially measure the overhead introduced by the GDP network. In terms of performance, we are interested in two key metrics: throughput and latency.

What's the sustained throughput that the GDP network can offer?

Throughput offered by a network is a challenging task to measure, especially because there are quite a few parameters that can be tweaked. We base our performance measurements on the well established tool `iperf` [87] that can be used to measure bandwidth between a given pair of hosts. For throughput measurement supported by the GDP network, we developed the GDP equivalent of `iperf`, which we call `traffic-agent`. Both `iperf` and `traffic-agent` can be operated either as a *source* of data or a *sink* of data. We developed this `traffic-agent` for both the research prototype and the production system.[18]

Given a set of endpoints, one of the most significant factor that affects observed bandwidth is PDU size. In most types of networks, there is a fixed processing cost per PDU, because of which smaller PDU sizes result in lower bandwidth. As such, in the following set of experiments, the key parameter that we vary is PDU size. By changing the PDU size, we demonstrate the limits of (1) PDU processing speeds, and (2) sustained bandwidth supported. Specifically, we use PDUs of size 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 bytes. For `iperf`, the equivalent variation is the buffer size on the reader and the writer.

[18] Note that the research prototype implements our principles of *verify before accept* and *verify before send*. However, the production system does not currently support these secure routing principles.



(a) Both sender and recipient in the same network. (b) Sender and recipient on opposite coasts in the US.

Figure 7.2: Sustained throughput as a function of changing PDU size. We measure the raw TCP/IP throughput using `iperf`, and compare this with the two software implementations of the GDP. In these results, `prod-version` refers to the production system (see section 7.2) and `research` refers to the research prototype (see section 7.1).

Note that the following performance measurements are done on the public Amazon EC2 cloud.[19] Specifically, our performance measurement is limited to two regions: (1) USA West coast (`us-west-2` in Oregon), and (2) USA East coast (`us-east-1` in N. Virginia). We use `c5.xlarge` instances with 4-cores, 8 GB of RAM and *up to* 10 Gb/sec of advertised network bandwidth.[20,21]

We perform two sets of experiments by varying the relative location of sender and recipient. In the first set of experiments, all machines are local relative to each other. For this set of experiments, both the sender and the recipient are in the same EC2 region (`us-west-2`). The typical round-trip latency between any given pair of machines is about 100-200 μs in this environment. In the second set of experiments, we measure throughput through the regular Internet. The sender and the recipient are in different regions (`us-west-2` and `us-east-1`) for this set of experiments. The typical round-trip latency between a pair of machines from different regions is about 70-75 ms.

Note that the network characteristics vary quite a bit even during the experiment. Further, TCP slow start introduces a *warm up* period before a steady saturated state is achieved. To account for these variations during each experiment with a given PDU size, we report the average sustained throughput and the standard deviation measured in 1 second intervals only after the throughput becomes stable. For `iperf`, we run the tool for its default 10 seconds reporting window; we repeat

[19] Amazon EC2 uses the concept of a 'region' as a specific geographic area, and 'availability zone' as an isolated location within a region. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`

[20] `https://aws.amazon.com/ec2/instance-types/c5`

[21] As measured by the `iperf` utility, we see the actual bandwidth to be 4.98 Gb/s in most cases, even though the advertised bandwidth is *up to* 10 Gbps.

the experiment 10 times, and report the average and standard deviation. The results for these throughput experiments are presented in Figure 7.2.

A few observations from the first set of experiments (Figure 7.2a): First, `iperf` represents a ceiling of the raw network performance. As a well polished tool that's doing nothing more than generating a stream of bytes and sending it to the recipient, it is hard for the GDP network to compete with `iperf`'s performance when the network bandwidth is plentiful. Second, the research version has a relatively high per PDU processing overhead. The throughput increases with the increasing PDU size, and continues to increase even at large PDUs. On closer inspection, we discovered that the limitation is not in the GDP router but the `traffic-agent` client. Third, the production version GDP router is high performance even at lower PDU sizes. Since it is written in a lower level language, such performance is to be expected. Finally, the fourth observation is that with some tuning of PDU sizes, both versions of GDP routers can achieve throughput beyond 1 Gbps with commodity hardware.

For the second set of experiments (Figure 7.2b), we see a somewhat similar pattern as the first set. However, the available bandwidth quickly becomes the limiting factor, and the performance of all three tools becomes somewhat similar after about 8k bytes/PDU.

Overall, this experiments show that while the end-to-end performance is certainly lower than what one can achieve with raw TCP/IP, there is a lot of room for improvement just by clever software engineering. Further, running GDP routers on commodity hardware is certainly viable for many use cases.

What's the round-trip latency with the GDP network?

To perform the latency experiments, we developed a custom `ping` application that is equivalent to `ping` utility available on various operating systems. We use the same setup as in the throughput experiments above. The only difference is that we perform this experiment only with the research prototype. We measure the average, minimum, and maximum round-trip latency for 100 messages, both in the case of traditional `ping` utility and our custom GDP-`ping`. In case of the GDP, the first PDU incurs an additional latency of lookup that the GDP router may perform. As such, the following results do not account for this first PDU latency. We will discuss the latency of the first PDU in the next section on micro benchmarks.

For the setup where the sender and recipient are in the same region (`us-west-2`), the average round-trip `ping` time is 0.120 ms.[22] For a similar setup, the average round-trip GDP-`ping` latency is 1.521 ms.[23] For the second scenario where the sender and recipient are in different regions, the average round-trip `ping` time is 75.295 ms.[24] For a similar setup, the average round-trip GDP-`ping` time is 76.563 ms.[25]

Based on these latency results, we make the following observations: First, in the local network, while the relative latency for GDP-`ping` is larger than the `ping` latency by an order of magnitude

[22] min: 0.108 ms, max: 0.160 ms.
[23] min: 1.076 ms, max: 18.086 ms.
[24] min: 75.224 ms, max: 76.786 ms.
[25] min: 75.946 ms, max: 93.742 ms.

(1.521 ms vs 0.120 ms), the actual values are rather small. The overhead introduced by the the GDP network is of the order of a millisecond. When considering the second experiment, we can see that the GDP network overhead is almost negligible. Second, while the average latency is good, there is a bit of unpredictability. While the average round-trip GDP network latency is 1.521 ms, the maximum latency goes up to 18 ms. We attribute this unpredictability to our code being in a higher level language, and we consider this as a crucial limitation that future versions of the GDP network must address.

C DataCapsule performance

What is the performance of a DataCapsule when used purely as a remote storage repository? In the following set of benchmarks, we compare DataCapsule performance to a single append-only file mounted from a remote NFS mount.[26] An append-only file is a useful pattern that applies directly to a number of machine-generated data, such as IoT sensors, log files, etc. The two major performance characteristics that we are interested in are: volume throughput and IOPS (Input/Output operations/second).

What's the throughput in/out of a DataCapsule as compared to a remote file?

Just like network throughput is highly dependent on the PDU size, storage throughput is affected heavily by the I/O size for typical storage systems. In general, smaller I/O size allows for a greater IOPS but lower throughput. The opposite is true as well, where larger I/O size provides more throughput but lower IOPS. As such, for our throughput measurement, we primarily vary the I/O size and see how that affects the performance characteristics.

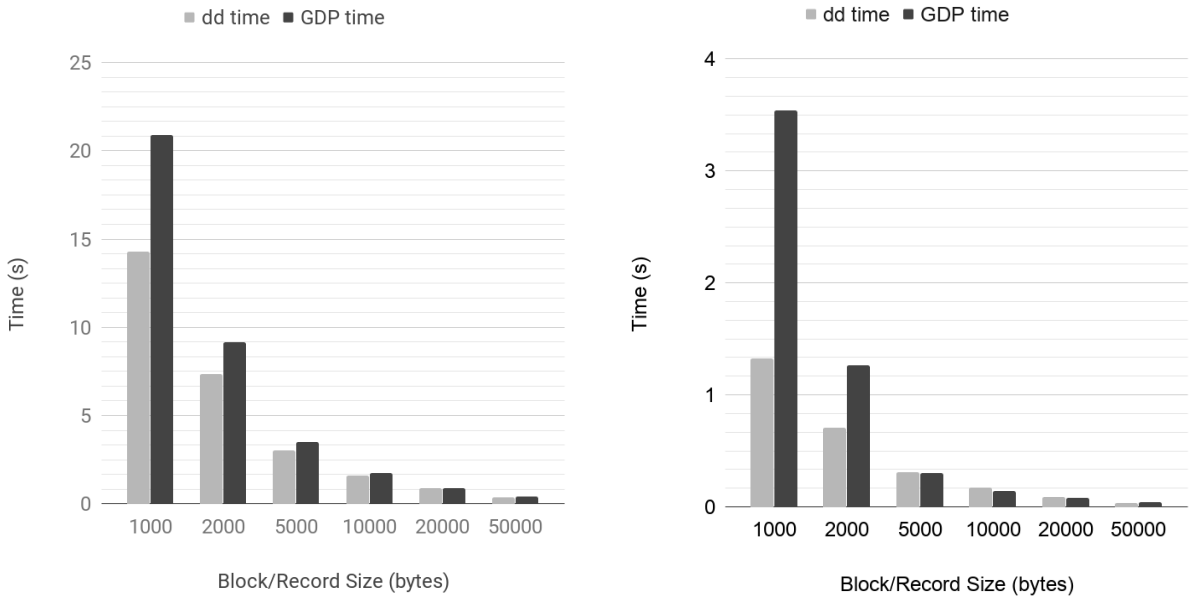
We consider two sets of experiments: first for writes, and second for reads. Both sets of experiments are performed in Amazon EC2 using two `c5.large` instances in the same region, where one machine serves as a client and the other serves as a server. In case of NFS, the server side is simply an NFSv4 server that exports a directory, and the client side mounts this directory at a mount point in the local file system. In case of DataCapsules, the server side runs a GDP router and a log server—both on the same machine, and the client side is a simple GDP client that reads from or writes to a given DataCapsule.

For generating the workload for NFS file, we use the tool `dd`—both in case of reads and writes. `dd` allows us to generate the workloads we desire while providing control on block size (i.e. the size of reads and writes) and the exact behavior of reads/writes. Additionally, we need to tune the NFS mount parameters, especially in the case of reads, to ensure that filesystem optimizations such as prefetching and caching don't make the results unfairly favorable to NFS.[27,28]

[26] This comparison shouldn't be taken as criticism of NFS. NFS is a general purpose artifact designed to handle a variety of filesystem related operations. An append-only file is the most favorable operation for DataCapsules. Instead, such a comparison is merely to provide a reference point for a reader to understand the DataCapsule behavior.

[27] For writes: we mount the NFS with default mount options and use the following invocation for `dd`: `dd conv=fdatasync,fsync,notrunc oflag=dsync,sync,nocache,nonblock [...]`.

[28] For reads, we use the additional options `-o nosharecache,noac,sync,lookupcache=none,rsize=BLOCK_SIZE` for the `mount` command, and use the following invocation for `dd`: `dd conv=fdatasync,fsync iflag=dsync,sync,direct [...]`.



(a) Append.

(b) Sequential read.

Figure 7.3: Comparison of performance for GDP DataCapsule vs. NFS (using the tool dd). We can see the variation in time needed for (a) appending, and (b) sequentially reading 10MB data as the block/record size changes.

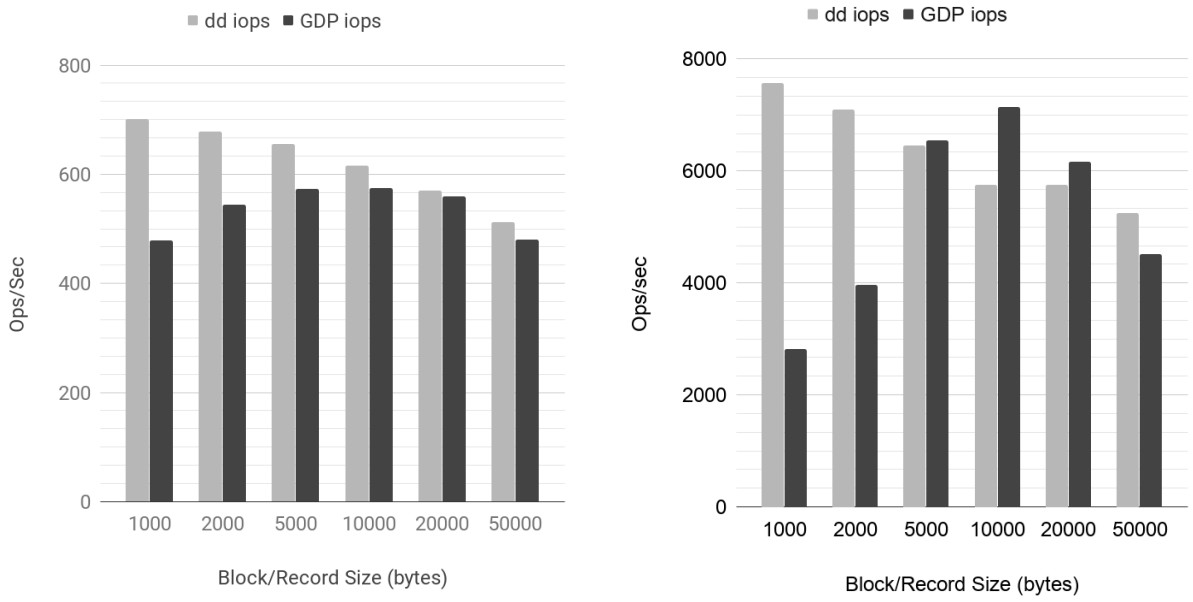
In case of DataCapsules, we use a custom designed command line utility called `log-append` for appending and `log-read` for reading. `log-append` allows user to create a given number of records of a specific size (i.e. the I/O size) and append those to a DataCapsule. We use a single record per append request for DataCapsules.[29] `log-read` simply reads a given number of sequential records from a given DataCapsule; we use a DataCapsule previously populated with `log-append`, which indirectly allows us to control the read size. For the purpose of these benchmarks, we use a DataCapsule with a single replica and a simple linked-list style structure.

For write benchmarks, we write 10MB data (10 MB = 10,000,000 bytes), both for NFS and DataCapsules. I/O size is varied from 1000, 2000, 5000, 10000, 20000, 50000 bytes.[30] For reads, we sequentially read this same file/DataCapsule generated for a given I/O size. We measure the total time taken for these reads/writes, and report the results in Figure 7.3.[31]

[29]For best performance, multiple records should be batched together to create a single append request. However, that would lead to an unfair advantage to DataCapsules.

[30]We avoid records larger than 60k in size in our prototype. While these may seem much smaller for a typical disk access (SSDs for example have a half megabyte block size), using a larger size in the present case will not the results significantly, because after a certain limit, the fragmentation at the network level becomes the limiting factor.

[31]We repeat each experiment 3 times for a given I/O size, and report the 'best of 3'.



(a) Append operations per second.

(b) Read operations per second.

Figure 7.4: Average I/O operations/sec (IOPS) for NFS (using the tool dd) vs GDP. The IOPS are calculated from the data in Figure 7.3.

A few observations from these experiments: first, NFS outperforms DataCapsules in each of the experiments. However, the relative performance of DataCapsules becomes better as the block size is increased. Second, read time for any given I/O size is much smaller than the corresponding write time. This is expected behavior for most storage systems, but we explicitly point it out here because of the seemingly opposite trend observed in Figure 7.1.[32] Third, even though NFS performance is better than DataCapsules, we believe that the GDP and DataCapsules can provide a close competition to NFS, especially when considering the fact that NFS is a highly polished software written in a low-level language with decades of software engineering effort, whereas our current implementation is merely a research prototype.

How many IOPS can a DataCapsule support as compared to a remote file?

This set of throughput experiments above also provides useful insights into the relative performance of DataCapsules with NFS when considering the IOPS (I/O operations/sec). Based on the number of reads and writes and the time taken, we calculate the average IOPS for NFS vs DataCapsules, both for append and sequential read (see Figure 7.4).

For dd and NFS, the number of IOPS decreases gradually as the block size increases. This is quite expected, since each append or read operation takes longer to complete with larger data

[32]Recall that in Figure 7.1, the relative performance of reads is worse than writes because of the peculiar way TensorFlow uses the filesystem.

size. For DataCapsules, however, the pattern is quite revealing: the IOPS rate is surprisingly low for smaller record sizes; it first increases and then starts decreasing after around 10k record size. The reduced IOPS with larger records can be explained in the same way as for NFS. Upon closer inspection, we found that the relatively smaller IOPS at small record sizes is because of a number of sub-optimal code fragments in our research prototype.[33] We believe that a well tuned implementation can easily achieve the same performance curve as for NFS.

While these experiment provide a relative comparison between NFS and DataCapsules, the absolute performance for NFS is quite small compared to real-world deployments. Real world remote filesystems rely very heavily on caching, pre-fetching, potentially unsafe write operations that accept a very small chance of data corruption for large gains in performance, aligning reads/writes to disk sector boundaries, etc. In our simplistic experiment, we switch off most of these optimizations, and we see "comparable" performance for the two systems. We believe that a real filesystem CAAPI can also benefit from all these optimizations, thus leading to much higher absolute performance while providing the security benefits of DataCapsules that go beyond just local environments.

7.3.2 A deeper look at performance and scalability

In our macro benchmarks, we compare the performance of the GDP and DataCapsules with a number of real world systems. However, there are a number of performance questions unique to the GDP for which we cannot draw a comparison with other existing systems. In this section, we dig deeper into GDP specific issues. Note that we exclusively use the research prototype for all micro benchmarks in this section.

A Performance for the GDP network *before* the steady state

In the previous section, we looked at the throughput and latency characteristics of the GDP network in a steady state. But what is the cost of achieving this steady state? Specifically, we are interested in the cost of secure advertisement, the overhead of lookup from a GLookupService, and the round-trip latency for first PDU—something that we omitted from our previous reporting of latency.

What is the cost of secure advertisements?

Advertising GDP names in the network is not free. During the secure advertisement process, a client first needs to establish a TCP connection followed by two round-trips to the GDP router that involve some cryptographic operations such as issuing an $RtCert$. Log servers may continue the advertisement process even further by sending $AdCerts$ for all the DataCapsules they host, which adds to the cost.

To quantify the additional cost of secure advertisements, we do a simple experiment. We run a client and a GDP router on the same commodity laptop so that we can rule out the variability

[33] An example pattern of a sub-optimal code fragment is: using the `in` operator on a `list` of records in a loop, which amounts to an exponential increase in processing time for the specific code fragment. However, because lists are highly optimized data structures in Python, each single code fragment like this adds to the running time only slightly as compared to the overall running time. For this specific example, using a `set` data structure would solve the problem.

introduced by the network. We then measure the total time taken for the client to advertise its own name to the GDP router.[34] Over 100 runs, the average time taken for the secure advertisement process is 25.3 ms.[35] After the initial secure advertisement process is complete, we measure the additional cost for advertising one name by sending an $AdCert$. We measure this cost for 100 names. We find that for advertising each additional name by sending one $AdCert$ at a time, the average time is 11.5 ms.[36,37]

While these numbers seem large, we make two observations. First, the closest real world equivalent to the initial secure advertisement is establishing a TLS session. Creating a TLS (or even HTTPS) session is not free, and the additional latency introduced by HTTPS session establishment can range anywhere from 10s to 100s of ms [88]. Thus, the cost of initial secure session establishment is on par with the TLS/HTTPS session establishment. Second, there is a lot of room for improvement in our prototype in terms of optimizations. We note three specific areas of improvement: (1) our research prototype performs cryptographic operations in Python in a single threaded environment. While Python allowed for a quick prototyping environment, we realized that it is not a good fit for high performance cryptographic operations. As such, operations such as validation of certificate chains, issuance of $RtCerts$, etc. are noticeably slow. (2) Bulk operations, such as sending multiple $AdCerts$ in a single request for subsequent name advertisements, are extremely useful because they can be performed in parallel. (3) Our RPC implementation is less than optimal. This results in extra overhead for each request/response, such as advertising additional names one at a time.

How expensive is a lookup from a GLookupService?

A GLookupService is crucial for enabling dynamic route lookup when a destination can not be found in the local forwarding table of a GDP router. However, a GLookupService is architecturally very simple: the simplest implementation of a GLookupService is dictionary that allows `GET` and `PUT` operations.

To measure the performance of GLookupService, we run a simple standalone GLookupService. We run a client on the same machine to avoid the effects of network latency and perform a number of GET operations. The observed round trip time for a GET request, averaged over 100 requests, is 1.23 ms.[38,39]

To put these numbers in perspective, the closet real-world equivalent of lookup from a GLookupService is a DNS query. The observed latency of a DNS lookup is quite compara-

[34] The time window that we measure starts from the first message from the client initiating a TCP connection and ends when the client receives a final secure acknowledgment from the GDP router. See Figure 5.4.

[35] Statistics (all times in ms): Min: 20.495, Max: 35.030, Avg: 25.281, Std: 2.509.

[36] Statistics (all times in ms): Min: 8.078, Max: 15.561, Avg: 11.461, Std: 1.918.

[37] Note that we measure the time between a client sending an $AdCert$ and receiving an acknowledgment back from the GDP router. This time window includes the time taken by the GDP router to verify the authenticity of the $AdCert$.

[38] Statistics (all times in ms): Min: 0.81, Max: 2.47., Avg: 1.23. 90 percentile: 1.74, 95 percentile: 2.00.

[39] Note that these measurements only represent the time taken to complete a GET request, and do not account for the time that a GDP router must spend in verifying the authenticity of returned results by checking signatures.

ble. As an example: the average DNS query time for `com` from a public DNS server adds roughly 2.1 ms to the round-trip `ping latency`.[40]

What is the time for first PDU?

The separation of active forwarding state in GDP routers from verifiable routing information in a GLookupService has an overhead for lookup of destinations that can not be found locally. In addition to the lookup from a GLookupService, the initiating GDP router must also connect to the destination GDP router and go through a secure advertisement process, which adds to the overhead.

In order to measure the latency introduced by the GDP network architecture for such a case, we preform the following experiment. We run five processes: a stand alone GLookupService, two GDP routers configured to query this GLookupService, and one client process attached to each of the GDP routers. We run all these processes on the same machine; this allows us to bypass the variability introduced by network. After both the clients have advertised their names, we initiate a single GDP-`ping` from one client to the other. This triggers a GLookupService lookup by the GDP router, followed by an on-the-fly connection creation. The GDP-`ping` message is delivered to the destination, and it responds back with a GDP-`ping` response. Note that this response does not trigger an additional GLookupService lookup or connection creation. We measure the total time that the originating client has to wait before receiving this GDP-`ping` response.

The observed round-trip time for this first GDP-`ping`, averaged over 10 runs, is 29.6 ms.[41] Note that we restart the entire infrastructure after every run, so that we can measure the round-trip time for the first message specifically when it requires a GLookupService lookup.

With these results, we make the following observations: First, the results are quite expected. The cost of first message is dominated by the cost of secure advertisement that the initiating GDP router must perform. Second, while this is a relatively large overhead, improvements in the cryptographic performance of our implementation can significantly reduce this delay. Third, pre-fetching popular routes and pre-establishing connections to the corresponding GDP routers can greatly minimize these delays.

B Scalability of the GDP network

The scalability of traditional IP routing comes from the fact that IP addresses are hierarchical. This hierarchy allows IP routers to use prefix aggregation and exchange routing state in a rather compressed form. The GDP network differs from the IP design pattern in a fairly significant way by adopting location independent names as a fundamental concept. Further, the GDP network also departs from previous designs for flat namespace networks that use DHT routing for achieving scalability in terms of number of names. The GDP network, instead, adopts a distributed global GLookupService as a shared repository of routing information. Scalability of the GDP network, thus, directly depends on how scalable a global GLookupService can be.

[40] For this estimation, we measured round trip latency from a residential network to `1.1.1.1`—a free public DNS server. We observe the average round-trip latency to be 14.896 ms over 100 measurements. We then measure the round-trip latency for the DNS name `com`. Over 100 measurements, we find this latency to be 17.070 ms.

[41] Statistics (all time in ms): Min: 27.693, Max: 44.356, Avg: 29.610, Std: 4.682.

Just as a quick exercise, let's see roughly how much computing resources are needed for maintaining a global GLookupService that maintains the state of the entire Internet's worth of delegations. While ideally the information should be geographically replicated in a highly distributed GLookupService managed by multiple entities, let's keep it simple and treat the GLookupService as a single centralized component in a data center.

As a reasonable estimate, let's say there are 1 Trillion *public* named objects that the GLookupService needs to handle, and these names are hosted on 1 Billion log servers [89]. Let's assume that each of these 1 Trillion names requires 1 KB worth of space in the global GLookupService. The total storage cost for such names is 10^{15} bytes, or 1 PetaByte. The current operational cost for an Amazon EC2 `i3.16xlarge` instance with 15.2 TB NVMe SSDs with a 1-year service contract is $3.18/hour ($27,856/year) [90], [91]. This leads to the cost of operating 1 PB worth of storage to be $209/hour ($1.8 million/year), which seems very much in-reach for large ISPs.

The cost for keeping this information up-to-date depends on the average lifespan of the delegations. A reasonable assumption is that $AdCerts$ are valid for few days (10^5 seconds), whereas $RtCerts$ are valid only for a few minutes (10^2 seconds). Given our earlier assumption on number of objects (10^{12}) and servers (10^9), this leads to 10^7 updates/second for both $AdCerts$ and $RtCerts$. Each delegation takes about 10^3 bytes of network traffic and 10^{-3} seconds for verification on a single CPU core, which leads to approximately 10GB/s incoming network traffic handled collectively by roughly 10^4 CPU cores. Once again, given the going rate of $0.101/hour ($882/year) for a 4 core `c5.xlarge` instance from Amazon EC2 for a 1-year contract term, the computational cost of verification is $252/hour ($2.2 million/year) on commodity hardware [90], [91].

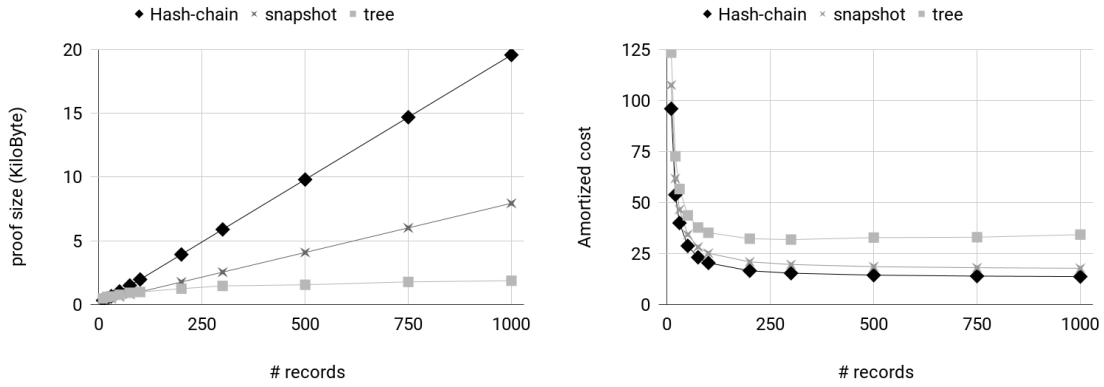
While these are extremely rough estimates for the resources required, they do demonstrate that even when running on off-the-shelf servers, the costs of operating a global deployment of the GDP network are practical. With customized hardware, these costs can be further reduced.

C DataCapsule performance: A deeper look at cost of hash-pointers

As we discussed in previous chapters, compared to existing ADS proposals, a DataCapsule interface provides flexibility to the application to include additional hash-pointers. While the state-of-the-art already provides logarithmic sized cryptographic proofs for random access, providing flexibility to applications enables such applications to tune the cost of proofs even further depending on the use case. We discuss the costs incurred by readers in terms of proof sizes, and the overhead incurred by the writer in terms of the additional time needed for bookkeeping.

For measuring the cost on readers, we measure proof sizes for the three simple hash-pointer strategies that we discussed in section 4.1: (1) a simple 'hash chain', (2) a 'snapshot' strategy where in addition to a hash chain, an extra hash-pointer points to snapshots done at fixed intervals, and (3) a 'tree' strategy, where hash-pointers effectively look like a tree. For each of the strategies, we used a number of DataCapsules with varying number of records.

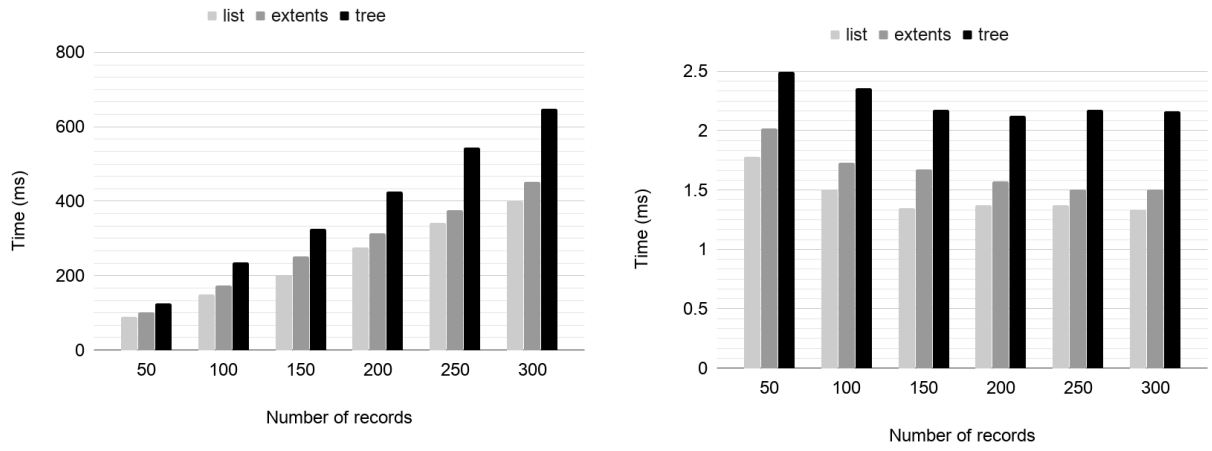
To emulate an application with random read access, we first measure average proof size (in bytes) when querying for a random record. In Figure 7.5a, we can see how the proof size reaches *O(log(n))* for a 'tree' but grows linearly when the 'hash chain' grows too long. For a slightly



(a) Average proof size for a random read as the number of records grows in a DataCapsule.

(b) Amortized proof size per record for a bulk read of all the records in a DataCapsule.

Figure 7.5: Proof sizes for various hash-pointer linking strategies.



(a) Total time taken for appending a given number of records.

(b) Amortized time taken per record.

Figure 7.6: Time taken by the writer for a given number of records, and how this time varies depending on the hash-pointer linking strategy. In (a), we measure the total time a writer takes to append the given number of records. Using the total time taken and the number of records created, we calculate the amortized time per record, which we report in (b).

different scenario where an application might want to read a large amount of data in bulk, the cost of proofs can be amortized over a number of records. For the same DataCapsules, Figure 7.5b shows the amortized cost of proof (in bytes) per record when an application reads the entire DataCapsule sequentially. As we can see, the simple 'hash chain' works much better in terms of amortized proof cost.

To see the cost of writes on the writer, we measure the total time taken for a given number of records for the same three hash-pointer strategies. Each record is 100 bytes in size, and we use only a single record per append request. We report the total time taken for a given number of records, and the amortized time taken for per append in Figure 7.6.

As we can see from the results, the total cost grows somewhat linearly as the number of records increases. The pattern for amortized cost per record, however, is a lot more revealing. First, we can see that for all three strategies, the amortized cost per record is larger when writing a smaller number of records. We attribute this pattern to the fixed costs incurred at startup time where a writer must load state from its non-volatile storage. Second, the 'tree' pattern performs poorly as compared to the simpler strategies. In fact, the per record time for 300 records is almost 50% greater for the 'tree' strategy as compared to the simple hash-chain.

From these results, we conclude that while it is not always possible to predict the access pattern, allowing a writer to make the best guess is a good idea. If such access patterns are not known in advance, a fall back to a generic 'good enough' strategy can be adopted. If a reader cannot get the desired performance, it can always create a derivative DataCapsule with hash-pointers more appropriate to the specific use-case. We consider more optimizations in this direction as future work.

Chapter 8

Conclusions

Motivated by the need for a secure ubiquitous storage infrastructure, we presented 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 dissertation 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 data structure, called DataCapsule, the can provide secure storage in the presence of potentially untrusted infrastructure and enable a leaderless replication strategy.
- The design of a scalable and secure routing network, called the GDP network, for inter- and intra-domain routing in the presence of mutually distrustful routing domains with a flat address space.

This dissertation is only a beginning for the GDP vision; it merely scratches the surface of a number of broader issues related to data security in a service-provider ecosystem. We hope that the future iterations of the GDP and DataCapsules can address some of the shortcomings of the design presented in this dissertation. At the very least, we hope that the production system provides a solution to the software engineering issues identified in the research prototype.

Bibliography

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

[2] Ben Zhang, Nitesh Mor, John Kolb, et al. "The cloud is not enough: saving iot from the cloud". In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. 2015.

[3] *What Edge Computing Means for Infrastructure and Operations Leaders*. https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders/. 2018.

[4] Weisong Shi, Jie Cao, Quan Zhang, et al. "Edge computing: Vision and challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

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

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

[7] Armando Fox, Rean Griffith, Anthony Joseph, et al. "Above the clouds: A berkeley view of cloud computing". In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), p. 2009.

[8] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. "A Taxonomy and Survey of Cloud Computing Systems." In: *NCM* 9 (2009), pp. 44–51.

[9] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, et al. "Cloud programming simplified: a berkeley view on serverless computing". In: *arXiv preprint arXiv:1902.03383* (2019).

[10] Brandon Butler. "PaaS Primer: What is platform as a service and why does it matter". In: *Network World, February* 11 (2013), p. 2013.

[11] Oliver Gass, Hendrik Meth, and Alexander Maedche. "PaaS characteristics for productive software development: an evaluation framework". In: *IEEE Internet Computing* 18.1 (2014), pp. 56–64.

[12] Michael Boniface, Bassem Nasser, Juri Papay, 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*. IEEE. 2010, pp. 155–160.

[13] Noah Apthorpe, Dillon Reisman, and Nick Feamster. "A Smart Home is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic". In: *arXiv preprint arXiv:1705.06805* (2017).

[14] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. "A survey of DHT security techniques". In: *ACM Computing Surveys (CSUR)* 43.2 (2011), p. 8.

[15] James Hendler and Jennifer Golbeck. "Metcalfe's law, Web 2.0, and the Semantic Web". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 6.1 (2008), pp. 14–20.

[16] Jinyuan Li, Maxwell N Krohn, David Mazières, et al. "Secure Untrusted Data Repository (SUNDR)." In: *OSDI*. Vol. 4. 2004, pp. 9–9.

[17] John Kubiatowicz, David Bindel, Yan Chen, et al. "Oceanstore: An architecture for global-scale persistent storage". In: *ACM Sigplan Notices* 35.11 (2000), pp. 190–201.

[18] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, et al. "Plutus: Scalable Secure File Sharing on Untrusted Storage." In: *Fast*. Vol. 3. 2003, pp. 29–42.

[19] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, et al. "CryptDB: protecting confidentiality with encrypted query processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 85–100.

[20] Prince Mahajan, Srinath Setty, Sangmin Lee, et al. "Depot: Cloud storage with minimal trust". In: *ACM Transactions on Computer Systems (TOCS)* 29.4 (2011), p. 12.

[21] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, et al. "Antiquity: exploiting a secure log for wide-area distributed storage". In: *ACM SIGOPS Operating Systems Review* 41.3 (2007), pp. 371–384.

[22] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, et al. "SiRiUS: Securing Remote Untrusted Storage." In: *NDSS*. Vol. 3. 2003, pp. 131–145.

[23] Raluca Ada Popa, Jacob R Lorch, David Molnar, et al. "Enabling Security in Cloud Storage SLAs with CloudProof." In: *USENIX Annual Technical Conference*. Vol. 242. 2011, pp. 355–368.

[24] Roberto Tamassia. "Authenticated data structures". In: *Algorithms-ESA 2003*. Springer, 2003, pp. 2–5.

[25] Miguel Castro, Barbara Liskov, et al. "Practical Byzantine fault tolerance". In: *OSDI*. Vol. 99. 1999, pp. 173–186.

[26] Byung-Gon Chun, Petros Maniatis, Scott Shenker, et al. "Attested append-only memory: Making adversaries stick to their word". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 189–204.

[27] *AWS IoT*. `https://aws.amazon.com/iot/`.

[28] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: a Distributed Messaging System for Log Processing". In: (2011).

[29] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, et al. "The many faces of publish/subscribe". In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131.

[30] Chenxi Wang, Antonio Carzaniga, David Evans, et al. “Security issues and requirements for internet-scale publish-subscribe systems”. In: *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*. IEEE. 2002, pp. 3940–3947.

[31] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[32] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, et al. “Unikernels: Library operating systems for the cloud”. In: *Acm Sigplan Notices* 48.4 (2013), pp. 461–472.

[33] Victor Costan and Srinivas Devadas. “Intel SGX Explained.” In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[34] Ali Ghodsi, Scott Shenker, Teemu Koponen, et al. “Information-centric networking: seeing the forest for the trees”. In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM. 2011, p. 1.

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

[36] Martín Abadi, Paul Barham, Jianmin Chen, et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[37] Guoqiang Hu, Wee Peng Tay, and Yonggang Wen. “Cloud robotics: architecture, challenges and applications”. In: *IEEE network* 26.3 (2012), pp. 21–28.

[38] Ajay Kumar Tanwani, Nitesh Mor, John Kubiatowicz, et al. “A Fog Robotics Approach to Deep Robot Learning: Application to Object Recognition and Grasp Planning in Surface Decluttering”. In: *arXiv preprint arXiv:1903.09589* (2019).

[39] Ralph C Merkle. *Method of providing digital signatures*. US Patent 4,309,569. 1982.

[40] Ralph C Merkle. “Protocols for public key cryptosystems”. In: *Security and Privacy, 1980 IEEE Symposium on*. IEEE. 1980, pp. 122–122.

[41] Ralph C Merkle. “A certified digital signature”. In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 218–238.

[42] Marten Van Dijk, Luis FG Sarmenta, Charles W OâĂŹdonnell, et al. *Proof of freshness: How to efficiently use an online single secure clock to secure shared untrusted memory*. Tech. rep. Citeseer, 2006.

[43] Leslie Lamport et al. “Paxos made simple”. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[44] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[45] Marc Shapiro, Nuno Preguiça, Carlos Baquero, et al. “Conflict-free replicated data types”. In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[46] Mike Burrows. “The Chubby lock service for loosely-coupled distributed systems”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 335–350.

[47] James C Corbett, Jeffrey Dean, Michael Epstein, et al. “Spanner: GoogleâĂŹs globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[48] *LogCabin*. `https://github.com/logcabin/logcabin`. 2015.

[49] Natacha Crooks, Youer Pu, Nancy Estrada, et al. “Tardis: A branch-and-merge approach to weak consistency”. In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 1615–1628.

[50] Douglas B Terry, Marvin M Theimer, Karin Petersen, et al. “Managing update conflicts in Bayou, a weakly connected replicated storage system”. In: *ACM SIGOPS Operating Systems Review*. Vol. 29. 5. ACM. 1995, pp. 172–182.

[51] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. “Dynamo: amazon’s highly available key-value store”. In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[52] Eric Brewer. “A certain freedom: thoughts on the CAP theorem”. In: *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM. 2010, pp. 335–335.

[53] Daniel Abadi. “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story”. In: *Computer* 45.2 (2012), pp. 37–42.

[54] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, et al. “Order preserving encryption for numeric data”. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. 2004, pp. 563–574.

[55] Philip Hunter. “Pakistan YouTube block exposes fundamental internet security weakness: Concern that pakistani action affected youtube access elsewhere in world”. In: *Computer Fraud & Security* 2008.4 (2008), pp. 10–11.

[56] *14,000 Incidents: A 2017 Routing Security Year in Review*. `https://www.internetsociety.org/blog/2018/01/14000-incidents-2017-routing-security-year-review/`. 2018.

[57] *Internet Assigned Numbers Authority*.

[58] Ola Nordström and Constantinos Dovrolis. “Beware of BGP attacks”. In: *ACM SIGCOMM Computer Communication Review* 34.2 (2004), pp. 1–8.

[59] Emil Sit and Robert Morris. “Security considerations for peer-to-peer distributed hash tables”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 261–269.

[60] Stephen Kent, Charles Lynn, and Karen Seo. “Secure border gateway protocol (S-BGP)”. In: *IEEE Journal on Selected areas in Communications* 18.4 (2000), pp. 582–592.

[61] Carlo Contavalli, Wilmer van der Gaast, David C Lawrence, et al. "RFC 7871-Client Subnet in DNS Queries". In: ().

[62] James D Solomon. *Mobile IP: the Internet unplugged*. PTR Prentice Hall Upper Saddle River, New Jersey, 1998.

[63] Kishore Ramachandran. "Mobile IP-deployment after a decade". In: *White Paper* (2006).

[64] Danilo Cicalese, Jordan Augé, Diana Joumblatt, et al. "Characterizing IPv4 anycast adoption and deployment". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM. 2015, p. 16.

[65] David David Folkman Mazières. "Self-certifying file system". PhD thesis. Massachusetts Institute of Technology, 2000.

[66] Van Jacobson, Marc Mosko, D Smetters, et al. "Content-centric networking". In: *Whitepaper, Palo Alto Research Center* (2007), pp. 2–4.

[67] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, et al. "Named Data Networking". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 66–73.

[68] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, et al. "A data-oriented (and beyond) network architecture". In: *ACM SIGCOMM Computer Communication Review*. Vol. 37. 4. ACM. 2007, pp. 181–192.

[69] Nikos Fotiou, Pekka Nikander, Dirk Trossen, et al. "Developing information networking further: From PSIRP to PURSUIT". In: *International Conference on Broadband Communications, Networks and Systems*. Springer. 2010, pp. 1–13.

[70] Christian Dannewitz, Dirk Kutscher, BöRje Ohlman, et al. "Network of information (netinf)–an information-centric networking architecture". In: *Computer Communications* 36.7 (2013), pp. 721–735.

[71] David R Cheriton and Mark Gritter. "TRIAD: A new next-generation Internet architecture". In: ().

[72] Ivan Seskar, Kiran Nagaraja, Sam Nelson, et al. "Mobilityfirst future internet architecture project". In: *Proceedings of the 7th Asian Internet Engineering Conference*. ACM. 2011, pp. 1–3.

[73] Ashok Anand, Fahad Dogar, Dongsu Han, et al. "XIA: An architecture for an evolvable and trustworthy Internet". In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM. 2011, p. 2.

[74] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *Middleware 2001*. Springer. 2001, pp. 329–350.

[75] Ben Yanbin Zhao, John Kubiatowicz, Anthony D Joseph, et al. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing". In: (2001).

[76] Ion Stoica, Robert Morris, David Karger, et al. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[77] Sylvia Ratnasamy, Paul Francis, Mark Handley, et al. *A scalable content-addressable network*. Vol. 31. 4. ACM, 2001.

[78] Dirk Kutscher, Suyong Eum, Kostas Pentikousis, et al. *Information-centric networking (ICN) research challenges*. Tech. rep. 2016.

[79] Kostas Pentikousis, Borje Ohlman, E Davies, et al. *Information-Centric Networking: Evaluation and Security Considerations*. Tech. rep. 2016.

[80] David Barrera, Laurent Chuat, Adrian Perrig, et al. "The SCION internet architecture". In: *Communications of the ACM* 60.6 (2017), pp. 56–65.

[81] Nick McKeown. "Software-defined networking". In: *INFOCOM keynote talk* 17.2 (2009), pp. 30–32.

[82] *Introducing Open/R: a new modular routing platform*. https://code.fb.com/connectivity/introducing-open-r-a-new-modular-routing-platform/. 2016.

[83] *Redis*. https://redis.io/.

[84] Robert Morris, Eddie Kohler, John Jannotti, et al. "The Click modular router". In: *ACM SIGOPS Operating Systems Review*. Vol. 33. 5. ACM. 1999, pp. 217–231.

[85] *Measuring Fixed Broadband Report - 2016*. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-report-2016.

[86] Matthew E Hoskins. "Sshfs: super easy file access over ssh". In: *Linux Journal* 2006.146 (2006), p. 4.

[87] Ajay Tirumala, Feng Qin, Jon Dugan, et al. "Iperf: the TCP/UDP bandwidth measurement tool". In: (2005).

[88] David Naylor, Alessandro Finamore, Ilias Leontiadis, et al. "The cost of the S in HTTPS". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, pp. 133–140.

[89] Cisco Visual Networking Index. "Cisco visual networking index: Forecast and methodology 2017-2022". In: *White paper, CISCO* (2017).

[90] *EC2 Instance Pricing*. [Online; accessed 29-Jan-2019].

[91] *Amazon EC2 Instance Types*. [Online; accessed 29-Jan-2019].

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

[93] Edward A Lee, Jan Rabaey, D Blaauw, et al. "The Swarm at the Edge of the Cloud". In: ().

[94] Michael Armbrust, Armando Fox, Rean Griffith, et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[95] *Carriots*. https://www.carriots.com/.

[96] *GroveStreams*. https://www.grovestreams.com/.

[97] *Samsung SAMI*. https://developer.samsungsami.io/.

[98] *Xively*. https://xively.com/.

[99] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, et al. "sMAP: a simple measurement and actuation profile for physical information". In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2010, pp. 197–210.

[100] Andrea Zanella, Nicola Bui, Angelo Castellani, et al. "Internet of things for smart cities". In: *IEEE Internet of Things journal* 1.1 (2014), pp. 22–32.

[101] *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/. [Online; accessed 27-March-2016]. 2016.

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

[103] Branden Ghena, William Beyer, Allen Hillaker, et al. "Green lights forever: analyzing the security of traffic infrastructure". In: *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. 2014.

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

[105] *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. [Online; accessed 27-March-2016]. 2016.

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

[107] Alma Whitten and J Doug Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0." In: *Usenix Security*. Vol. 1999. 1999.

Appendix 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 [15]. 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 section 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 section 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 [93]. 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 [94]. 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 [27], [95]–[98]. 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 [99], air quality monitoring [100], 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 [1]. 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 cannot 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.[1]

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.

A.2 Security of IoT devices: A heterogeneity challenge

The general state of security of IoT devices isn't very good [101]–[104]. 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 [105]. 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[2] of the top 10 IoT vulnerabilities [106]. None of these

[1] 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.

[2] 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.

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 [107].

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 DataCapsules provide a standardized narrow waist with reduced attack surface for communicating with physical devices (see section 2.1). DataCapsules 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 2.1). Using DataCapsules, 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.