Global Data Plane: A Federated Vision for Secure Data in Edge Computing

Nitesh Mor
UC Berkeley
mor@eecs.berkeley.edu

Richard Pratt
UC Berkeley
rpratt@berkeley.edu

Eric Allman
UC Berkeley
eric@cs.berkeley.edu

Kenneth Lutz
UC Berkeley
lutz@berkeley.edu

John Kubiatowicz
UC Berkeley
kubitron@cs.berkeley.edu

***Abstract*—We propose a federated edge-computing architecture for management of data. Our vision is to enable a service provider model for "data-services", where a user can enter into economic agreements with an infrastructure maintainer to provide storage and communication of data, without necessarily trusting the infrastructure provider. Toward this vision, we present cryptographically hardened cohesive collections of data items called DataCapsules, and an overview of the underlying federated architecture, called Global Data Plane.**

***Index Terms*—Edge Computing, Data Security, Distributed Systems.**

I. Introduction

Edge computing, the next major paradigm of computing after cloud computing, is the use of resources at the edge of the network to augment or supplant resources in the cloud. Gartner estimates that by 2025, 75% of enterprise-generated data will be created and processed outside a traditional centralized data center or cloud [6]. The appeal is obvious. Resources at the edge provide an opportunity for low-latency and high-bandwidth communication, lower energy consumption, and (potentially) improved data security, privacy, and isolation.

Between an end user's device and the cloud are a variety of decentralized resources, managed by individuals, small corporations, municipalities and others (see Figure 1). Many are in homes, offices, public infrastructure, factory floors, enterprises, nano/micro data centers, and elsewhere; a large fraction are behind firewalls or NATs and aren't even directly reachable on the public Internet. In an ideal world, these *federated* resources could be exploited in a unified manner by users and applications designers. In fact, some have concluded that the resources in this hidden layer are crucial to enable rich applications of tomorrow [10], [37]. Unfortunately, edge environments are extremely heterogeneous, not just in terms of capabilities, but also in terms of resource ownership, security, privacy, reliability, and other such soft factors [31], [13]. Worse, it is extremely difficult to exploit them in a secure, understandable, and cost-effective way.

To start with, this extended vision of edge computing has two rather fundamental issues. First, the edge environment is often not as trustworthy as the cloud; resources at the edge of the network may be owned and maintained by novice users or malicious third parties, leading to a distinct possibility that devices may be unreliable or compromised. Further, *physical* security is less prevalent in edge environments, leading to a variety of physical attack vectors. Compromised devices could steal information in transit, covertly monitor communication, or deny service. Standard application frameworks do not provide sufficient mechanism to reason about or act on the ownership of resources at the edge. Further, even when environments are ostensibly secured through professional management, many distributed applications make use of middleware that assumes by default that it operates in a trusted, non-malicious environment (*i.e.*, relying on border security from firewalls both in the network and onboard multi-tenant hosts); once breached, these boundaries often expose unencrypted and unsecured information[1].

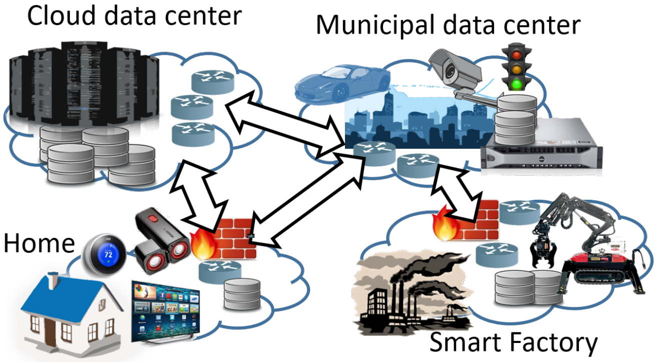

Fig. 1: A Case for Federation: Pervasive network resources partitioned by ownership *could* be exploited for secure edge computing. However, to avoid vendor lock-in and stove-pipe solutions, we must *refactor* the network around *information* and *trust*, abstracting above the level of packet communication to adopt a standardized environment for protecting, managing, and migrating information.

Second, modern connected and cyber-physical applications that *could* benefit from the edge environment (such as robots, autonomous vehicles, or smart-manufacturing environments) are extremely vulnerable to data corruption, insertion attacks, and failure-induced data loss or inconsistency; under attack, these applications could cause property damage, injury, or even loss of life. Such vulnerabilities already occur in cloud-only deployments—without introducing the complexities of edge environments. Since the need for universal attribution (provenance) and protection of information has become increasingly acute in today's world, edge environments seem less and less appropriate for the very applications that could benefit most

This work is supported by NSF/VMware Partnership on Edge Computing Data Infrastructure (ECDI), NSF award 1838833. This work is also sponsored by the The Ubiquitous Swarm Lab at UC Berkeley.

[1] Note that the notion of "securing communication" through cryptography (*e.g. TLS*) has at its root the misconception that end hosts (including those in large datacenters) are somehow secure against breach.

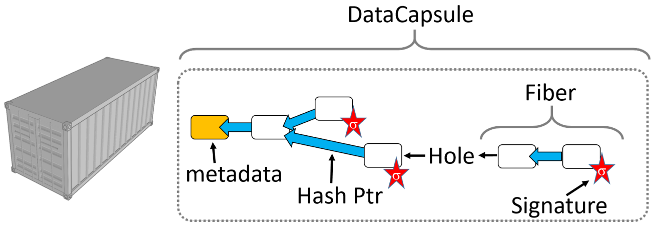

Fig. 2: DataCapsule: a *cohesive* representation of information analogous to a shipping container. Internally, it is an ordered collection of immutable *records* that are linked to each other and signed. Their unique names and identity are derived from hashes over metadata that include ownership credentials.

from their resources.

Consequently, any approach that attempts to provide edge resources to reliable applications must: (1) provide a standardized way in which to access heterogeneous resources, thereby freeing application writers from ad-hoc interfaces and stovepipe solutions; (2) respect the natural boundaries of ownership and trust as shown in Figure 1, giving applications and users the ability to make choices based on these boundaries; and (most important) (3) address the *security* and *provenance* of information independently from physical device boundaries. While such an approach would be desirable for any system–including one owned and maintained entirely by a single entity such as a large corporation–these requirements are essential to freeing applications to exploit resources in a federated environment, one in which public or semi-public resources may be bought and sold, borrowed and donated, to supplement the private resources of users and application providers.

A. Refactoring Around Information and Trust

To solve these problems, we advocate a fundamental refactoring of the information and communication infrastructure into a data-centric "narrow waist" focused on *information* rather than *physical, destination-based packet communication*. Unlike other data-centric proposals (*e.g.,* NDN [38]), we provide data update and attribution semantics by introducing a new entity, called a DataCapsule, that operates on top of a data-centric infrastructure called the Global Data Plane. As shown in Figure 2, DataCapsules are the information equivalent of the ubiquitous shipping containers seen at every port and on ships, trains, and trucks. They are standardized metadata containers, with unique names derived as hashes over their metadata, that hold cryptographically linked information histories. These histories are directed graphs of transactions with provenance (*e.g.,* signatures).

The Global Data Plane enables location-independent conversations between clients (or application components) and DataCapsules based only on DataCapsules' unique names. The Global Data Plane is organized as a graph of Trust Domains (TDs), enabling communication to be restricted to portions of the network trusted by DataCapsule owners. Since conversations with DataCapsules do not involve physical identifiers, such as IP addresses, they may be placed, moved, or replicated to satisfy requirements for availability, durability, performance, or privacy without compromising the integrity or global shareability of the enclosed information.

B. Agile and Secure Edge Computing.

An important aspect of edge computing (to complement DataCapsule mobility) is *agile secure computing*, namely the ability to instantiate a secure computation on demand in an edge environment–complete with code, cryptographic keys, and data inputs and outputs. To protect information while enabling on-demand edge computing, the Global Data Plane infrastructure is ideally coupled with secure, lightweight, virtual machine containers that can be entrusted with secret keys and attested code. Secure computational enclaves (such as constructed via Docker [28] and SGX [12], [9] or its follow-ons) are a vital component of our edge-computing vision, but beyond the scope of this current paper. DataCapsules provide an ideal mechanism with which to distribute secure, multiversioned binaries to secure enclaves as well as repositories for data used by the resulting computation.

We envision that secured computational providers at the edge could interact with applications (through a broker) to offer secure enclaves in which to execute sensitive components of applications in edge environments. Applications could judge the reliability of such resources via secure attestation. For this paper, however, we contend that refactoring applications around DataCapsules provides a significant first step toward our federated, edge-computing vision.

C. Our Contributions

In what follows, we refine our federated, edge-computing vision and discuss how an infrastructure standardized around DataCapsule metadata enables this vision. In particular:

- We identify a 'platform vision' to address the extreme heterogeneity of resources at the edge of the network.
- We explore how DataCapsules secure information within the network while simultaneously enabling a wide variety of familiar data access patterns such as databases, filesystems, and multimedia streaming.
- We discuss the design of the Global Data Plane platform that serves as a substrate for DataCapsules, and investigate how the Global Data Plane operates in a federated environment by embracing and exploiting multiple domains of trust.

II. A Platform Vision for the Edge

In this section, we explore whether a *platform* abstraction can be applied to ease the burden of application writers confronted with an edge-computing environment. As opposed to the *infrastructure* approach of forcing application developers to configure and manage systems directly, the *platform* provides developers with a way to convey properties such as performance and security to the underlying infrastructure.[2]

As shown in Figure 3, we seek a storage and communication platform that is widely distributed over a heterogeneous infrastructure managed by many administrative entities. We envision that such a system could provide a seamless integration

[2] Here, our use of the word "platform" is similar to the way it is used in "Platform as a Service" (PaaS) in cloud computing, and as opposed to the alternative "Infrastructure as a Service" (IaaS) [29].

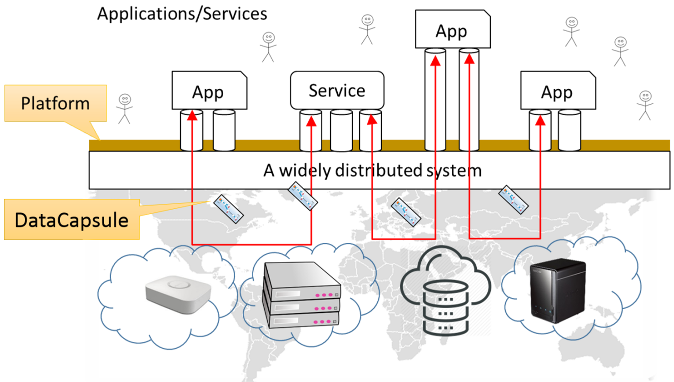

Fig. 3: A widely distributed system spread over heterogeneous infrastructure. DataCapsules are self-sufficient containers that enable our data-centric vision (see section IV).

of resources at the edge with those in the cloud if done correctly. Such a system differs from the current cloud model which is (by and large) oriented around a single owner and administrator of resources. As argued previously, we believe that refactoring applications around a *secure-storage* API is an essential step to enabling *edge computing.*

Many big-data systems, backed by cloud infrastructures, have evolved with the assumption that the infrastructure is trusted—an assumption that might be valid in the closed, single-administrator model (although a host of recent data breaches might argue to the contrary). Such assumptions must be revisited in a federated, multi-administrator environment. The DataCapsule vision outlined in the last section is an essential first step toward our platform vision, since it separates concerns of the security, authenticity, and integrity of information from the physical properties of the platform.

Other than the typical properties of a distributed storage system (scalability, fault-tolerance, durability, etc.), let us take a look at the additional requirements needed to support the edge-computing paradigm:

- **Homogeneous interface:** A homogeneous interface allows application developers to create portable applications and avoid stove-piped solutions. This property is important, given the extreme heterogeneity of resources at the edge. It is also useful if the interface can support a wide-variety of applications natively, or allows for creation of higher level interfaces.
- **Federated architecture:** The platform 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]. The edge-computing ecosystem has the potential for a vast and varied set of administrative entities. 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.
- **Locality:** Locality is extremely important for at least two reasons: First, local resources enable low-latency and real-time interactions unavailable from the cloud [10]. Second, information privacy is enhanced by limiting access exclusively to local clients or using local private resources that may be more trusted to not engage in sophisticated side-channel attacks. Finding such local resources relative to a client usually requires knowledge of the routing topology, a process best 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.
- **Secure storage on untrusted infrastructure:** The infrastructure should provide a baseline verifiable data security (confidentiality and integrity) even in the face of potentially untrusted elements. In the cloud ecosystem, there are few if any commercial storage offerings that provide 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. Enabling secure interfaces allows for a utility model of storage where smaller but local service providers can compete with larger service providers.
- **Administrative boundaries:** The system should provide visibility of administrative boundaries to an application developer. Ideally, a developer should be able to dictate what parts of the infrastructure to be used for specific data for two reasons. First, as hinted above, concerns over data privacy and security—especially for highly sensitive data—may require that data be restricted to a specific organization. Second, an application developer should be able to form economic relations with a service provider and hold them accountable if the desired Quality of Service (QoS) is not provided by the service provider.
- **Secure routing:** Data security in transit is equally important as data security at rest and simply encrypting the data is not sufficient in many cases. Encryption does provide a baseline of data confidentiality; however, any adversarial entity monitoring communication in real time can learn much information from the side-channels [8]. In a federated infrastructure where the system routes a reader/writer to a close-by resource, it becomes easy for third-party adversaries to pretend to be 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 [35]. These two requirements—that anyone can be a part of the network but that the network must be secure against timing channels or denial of service—would appear to be at odds, and a federated system must provide a solution to such conflicting goals.
- **Publish-Subscribe paradigm:** The usefulness of a storage system increases exponentially when communication is an integral part of the ecosystem; such ideas have been well studied in the economics of communication (*network effect*) [16]. Toward this goal, we believe that a *secure* publish-subscribe is crucial to enabling composability of services and applications.

[3] An *administrative entity* 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, or a large scale cloud service providers with massive data-centers.

- **Incremental Deployment:** Any system must be incrementally deployable, suggesting a hybrid combination of overlay networking for initial network penetration and native implementation (such as using an SDN infrastructure) for later performance enhancement.

A system that meets these high-level requirements—security and locality being the two most important—provides application developers a *secure ubiquitous storage* platform that supports a wide variety of deployment scenarios: a service provider model with a high density of resources; private corporations with restricted data flows; strict control loops in industrial environments with explicitly provisioned resources; and a strict data-center model similar to the cloud.

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.

III. Existing Work: Where Does It Fall Short?

To realize such a vision, can we modify existing systems and applications and make them work in a federated environment and untrusted infrastructure? Often, when faced with such a question, application developers present a simplistic viewpoint: 'let's add encryption to an existing system'. Such an approach, while perhaps appealing, is insufficient, since encryption provides confidentiality, but not data integrity and provenance. Further, information objects that update over time require additional considerations to ensure protection against replays, reorders, message drops, etc. An approach like this leads to insecure applications in the worst case and point solutions in the best case.

An edge computing storage platform has to address a number of challenges all at the same time, whereas the alternatives can get away with addressing only a subset of such challenges. e.g. cloud-based data management systems can work based on 'reputation', whereas such a platform for edge computing absolutely must provide verifiable security. Similarly, on-premise system management by application developers typically do not even assume untrusted infrastructure, since the infrastructure is under their direct control.

Quite a few tools and systems have been proposed in the past to address the challenge of secure data-storage or secure communication in the face of untrusted infrastructure. However, when applied to the edge-computing platform vision, often there is a mismatch between the performance requirements and the security requirements; we present some such examples here. Note that we do not necessarily claim novelty in identifying or addressing individual challenges; instead, our novelty claim is the high-level vision of the federated platform that requires these challenges to be solved in a specific context.

A. Existing IoT Platforms

A number of ecosystems and application frameworks have emerged for the Internet of Things (IoT) to alleviate challenges very similar to those present in edge-computing [1], [4], [5], [3]. While these ecosystems are helpful to some extent, they only address the short-term—and rather shallow—challenge of providing API-uniformity and do not necessarily address the underlying issues of data security and trust, especially when the resources at the edge are owned by many different entities. Rather than providing the flexibility to work with multiple trust domains and enabling users to maintain the ownership/control of their own data, these frameworks create walled gardens trapping the data that belongs to users, leading to stove-pipes and system lock-in. In our opinion, they are solving a very different problem: how to convince as many users as possible to hand-over their data that can be stored in cloud data centers.

B. Information Centric Networks

Information Centric Networks (ICNs) are hailed as an architecture to enable locality, mobility, and various other useful properties in a data-centric architecture [38]. ICNs raise the abstractions to the information itself rather than the host-centric messaging networks such as IP. The defining property of an ICN is information objects can be addressed as a first class citizen. Various ICN architectures propose pervasive caching where anyone can respond to a client seeking a specific object, and provide a way for the client to verify the correctness of the object [14]. There are two challenges with applying such ICN architectures to our federated edge-computing platform:

- Pervasive caching, or uncontrolled caching in general, is bad for privacy, especially when dealing with sensitive data. It is important to control where data flows. Routing to the closest copy of an object, while allowing anyone to host a copy, gives rise to sybil attacks where malicious entities can insert themselves at strategic points in the network and effectively perform man in the middle attacks.
- Existing ICNs work well with static objects, but not with dynamic objects (such as streams). For objects that change over time (and thus have multiple versions with the same name), it is difficult for a client to know whether or not it is getting the most recent version of the object. No entity in the infrastructure is explicitly responsible for providing the most up-to-date copy or providing semantics for the update of information.

We believe that a strict delegation of caching, storage and routing of objects to specific service providers can alleviate both of these problems. As we will discuss later, we use explicit cryptographic delegations in a service provider context to obviate both of these limitations.

C. Distributed Storage and Untrusted Servers

Storage on untrusted infrastructure is a well-studied problem [18], [26], [23]. Distributed storage brings many challenges related to failures, network partitions, and data consistency. The presence of potentially untrusted entities makes the distributed storage problem even more challenging. Various existing systems have addressed distributed storage on untrusted infrastructure [21], [11], [36], [15].

A common theme among these solutions is to assume Byzantine failures and create a quorum that decides which

TABLE I: A summary of how Global Data Plane meets the platform requirements (see section II).

Goal	**Enabling feature**
Homogeneous interface	DataCapsule interface that supports diverse applications by providing limited yet sufficient flexibility to enable applications achieve their desired performance.
Federated architecture	Using the flat name for a DataCapsule as the trust anchor and does not rely on traditional PKI infrastructure.
Locality	Hierarchical structure for routing domains that mimics physical network topology.
Secure storage	DataCapsule as an authenticated data structure that enables clients to verify the confidentiality and integrity of information.
Administrative boundaries	Explicit cryptographic delegations to organizations at a DataCapsule-level.
Secure routing	Secure advertisements and explicit cryptographic delegations.
Publish-subscribe paradigm	Publish-subscribe as a native mode of access for a DataCapsule.
Incremental Deployment	Routing over existing IP networks as an overlay.

new writes are to be accepted and in what order[4]. While this is an excellent solution for certain use cases, it is somewhat contradictory to the edge-computing vision: on the one hand, we'd like to use local resources as much as possible, and on the other we'd like to consult multiple network resources (owned by different providers and geographically separated).

In comparison to existing systems, we explore a relatively unexplored design point: a cryptographically hardened, single-writer, append-only data structure where the ordering of writes is decided by the single writer.

D. Secure Transport

While we focus on object security, it is important to assert that the responses to various requests come from the correct parts of the infrastructure, and not an adversary that has somehow managed to insert itself in-between the path, or just happens to be in the path. In a service-provider model, it is important to ensure that an honest infrastructure provider can't be framed by an adversary.

A seemingly obvious solution is to create a secure channel to the infrastructure using TLS. However, this approach is contradictory to the notion of an ICN, since it is going back to securing the host-to-host communication. Even if one were to use TLS-like mechanisms, a solution for mapping an object name to a host name is needed. Another solution is for an infrastructure provider to simply sign the responses [23]. However, such an approach leads to practical challenges of obtaining the public key of the server, especially when dealing with a large number of information objects.

A final engineering challenge is to be able to work with multiple replicas of the same object (potentially hosted by multiple service-providers simultaneously). As with IP-networks, 'anycast' works well with connectionless protocols (such as DNS); connection-oriented protocols run into the issue of routes flipping in the middle of the connection. As a result, a protocol that requires explicit key-exchange before any data transmission may not work well.

Toward this goal, we propose a connectionless mechanism for securing responses from the infrastructure; our protocol starts the chain of trust from the name of the object itself and quickly translates to efficient HMAC based secure acknowledgments.

IV. Global Data Plane and DataCapsules

A. DataCapsules

A DataCapsule is a globally addressable *cohesive* encapsulation of information that lives in a widely distributed system and provides a unified storage and communication primitive. DataCapsules are cryptographically hardened so that they can be migrated across domains of trust while maintaining the integrity and confidentiality of information. Internally, a DataCapsule is an authenticated data structure (ADS) [32] that works well with a distributed infrastructure underneath[5].

A DataCapsule is extremely flexible the information that it can contain; it can be used for a static information dump or an evolving stream of information. Similarly, there is no inherent restriction on the frequency or volume of information updates; the only constraints are the amount of available resources. Thus, a DataCapsule could be used to store a short text file, time-series data representing ambient temperature, streaming video, or anything else. Depending on the information that a DataCapsule contains, the level of durability and consistency requirements can be varied to match well with the infrastructure underneath[6].

The inspiration for DataCapsules is a shipping-containers equivalence for data, an idea elegantly introduced by Lucas, et al [24]. The introduction of a standardized shipping container produces a strong network effect [16]: the underlying infrastructure of ships, trains, trucks, cranes and storage locations, can all be developed in support of this form factor for all containers regardless of source or destination. Each new port benefits all the preexisting ports. Thus, the standardized interface benefits both users *and* shippers. Further, during the shipping process, the goods are handled by many parties. Not only is it challenging to keep track of loose goods, but they are prone to theft as well. A shipping container makes it much easier to keep track of goods; a locked container not only protects against accidental loss but also prevents opportunistic thefts. Of course this analogy is may only be stretch so far.

In a federated and widely distributed computation infrastructure where data is almost invariably handled by multiple parties at various stages of its lifetime, we envision a DataCapsule to be the data container that enables one to *manage* the

[4] An exception is blockchain based solutions that rely on proof-of-work. While interesting from a theoretical standpoint, they have limited practicality to serve as a data-storage infrastructure because of scalability challenges.

[5] An ADS allows a client to get a proof of operations carried out on a remote data structure stored on an untrusted server. A number of ADSs have been proposed in the past, e.g. Merkle trees, authenticated dictionaries, etc.

[6] For example, a DataCapsule representing a streaming video can tolerate a few missing frames, but another DataCapsule that supports a file-system can not tolerate any missing data.

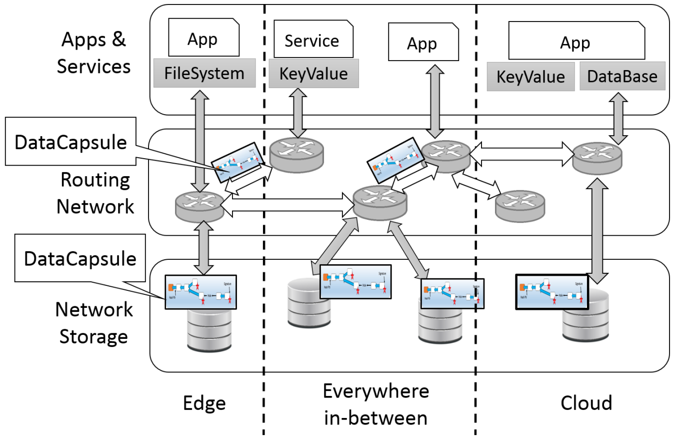

Fig. 4: DataCapsules rest on storage available in the network and become durable through replication. Storage provided by DataCapsule-servers and routing provided by GDP-routers collectively form the infrastructure: Global Data Plane.

information through various domains of trust while enabling interoperability across rich and diverse applications.

B. Global Data Plane: An Ecosystem for DataCapsules

While a DataCapsule is an abstract bundle of information, the physical backing for the DataCapsule is provided by the Global Data Plane (GDP). At a high level, the GDP is to a DataCapsule is what a distributed file-system is to a file. The GDP is a widely distributed system of storage and routing nodes that enables an ecosystem for such DataCapsules (see Figure 4). While an application decides what information goes in a DataCapsule and in what order, the infrastructure is tasked with making the information durable and available to appropriate readers.

The infrastructure is owned and operated by *organizations*. An organization can operate DataCapsule-servers and act as a storage organization; or it can operate GDP-routers along with the associated routing infrastructure and act as a routing domain; or both. Anyone can create their own organization; all they need is to put up appropriate infrastructure. Similar to a DataCapsule, an organization is identified by a unique identifier. Not only organizations, even individual DataCapsule-servers and GDP-routers also have their own unique identity. These organizations are what define the Trust Domains that we mentioned earlier.

Note that these names/identities for various addressable entities (organizations, DataCapsules, DataCapsule-servers, GDP-routers) are all part of the same flat name-space, which is also their address in the underlying GDP network: a routing fabric consisting of GDP-routers that collectively enable communication between these flat addresses. Instead of being exclusively a host-to-host network, this is a more abstract network where one can communicate directly with services, data, or in the general case—principals. Since DataCapsules representing information are a first class citizen of this network, this network fits the definition of an Information-Centric Network (ICN).

DataCapsules reside on DataCapsule-servers in Global Data Plane, as assigned by the DataCapsule-owner. The task of DataCapsule-servers is to make information durable and available to the appropriate readers while maintaining the integrity of data. The durability is ensured by creating replicas; a single DataCapsule may have a number of replicas throughout the GDP network across domains.

A DataCapsule-owner assigns the responsibility for storing a DataCapsule and providing a route to the name of DataCapsule by using cryptographic delegations. Thus, such organizations define the administrative boundaries for where a DataCapsule can reside or be routed through. Organizations can have hierarchies (sub-organizations, and so on) to enable fine-grained administrative controls. Such hierarchies allow flexibility on the granularity of delegation; a DataCapsule-owner can selectively delegate DataCapsule related responsibilities to specific sub-organizations. Just like redundancy for a utility, the architecture allows a single DataCapsule to be delegated to multiple service providers at the same time. The GDP network natively supports locality and anycast to the closest replica and enables clients to satisfy their performance requirements.

C. Threat model

A DataCapsule targets a separation of integrity and confidentiality from service availability. Appropriate entities in the infrastructure act on service requests from a client (e.g. sending a message to a destination, reading data from a DataCapsule-server), and honest infrastructure operators fulfill their contractual obligation to provide service to honest clients. In the most general case, a client does not trust the routing/storage infrastructure for data-integrity or confidentiality. The infrastructure can behave incorrectly either accidentally or deliberately, but a client can detect such deviations. As examples of what is permissible under such threat model, any messages can be arbitrarily delayed, replayed at a later time, tampered with during transit, or sent to the wrong destination. Similarly, a DataCapsule-server can attempt to tamper with individual records or the order of records when stored on disk.

We assume service availability from the infrastructure, which is achieved with a utility provider model. If a client does not receive the expected level of service (e.g. excessive message loss, low resiliency to component failures, etc.), it can find a different service provider without compromising the security of data. For mission-critical data, the DataCapsule-owner preemptively delegate multiple service-providers to ensure no accidental or malicious disruption in service. The GDP natively supports such redundant delegation.

The infrastructure may receive requests for DataCapsule operations from not just appropriately authorized clients, but also from third parties pretending to be legitimate clients. DataCapsule is designed to allow honest service providers implement appropriate access control, and for honest clients to detect when such access control has been violated.

In summary, the GDP ensures that data security (i.e. integrity, confidentiality, and provenance) can be managed by the users, whereas durability and availability are the responsibility of an underlying infrastructure enabled by a utility provider model.

V. Internals and Mechanisms

In this section, we describe the internal details and mechanisms of the architecture. Note that unless otherwise specified, 'hash' refers to a SHA256 hash function. Also, 'signatures' refer to ECDSA instead of a non-EC algorithm (RSA, DSA) because of smaller key sizes.

A. DataCapsule: A configurable ADS

In an abstract sense, a DataCapsule is a single-writer, append-only data structure stored on a distributed infrastructure and identified by a unique flat name. This flat name serves as a cryptographic trust anchor for verifying everything related to the DataCapsule. The unit of read or write to a DataCapsule is called a *record*. Internally, a DataCapsule is an ordered collection of variable sized immutable *records* linked together with hash-pointers.

The key operations available to a client are: 'append', 'read' and 'subscribe'. An 'append' (or publish) is adding new records to the DataCapsule. 'Read' is for fetching records by addressing them individually or by a range, while 'subscribe' allows a client to request future records as they arrive, enabling an event-driven programming model. Thus, a DataCapsule not only enables a persistent information repository by providing efficient random reads for old data, it also supports real-time communication with a pub-sub paradigm and secure replays at a later time (a time-shift property).

The absolute simplest DataCapsule is essentially a hash-list of records; each new record has a hash-pointer to the previous record. An append operation is adding new records at the end of such a list. Each update is associated with a signed 'heartbeat' generated by the writer; such signature is over the most-recent record using a key known only to the writer. Read queries can be verified against a particular state of the data-structure, identified by the 'heartbeat'. As we will describe next, each read comes with a cryptographic proof of correctness created using signatures and hash-pointers.

While there is only a single writer for a DataCapsule, there can be multiple readers. The readers and the single writer of a DataCapsule are collectively referred to as *clients*. At a cryptographic level, the write access control is maintained by the writer's signature key, and read access control is maintained by selective sharing of decryption keys.[7] Clients use digital signatures and encryption as the fundamental tools to enable trust in data than in infrastructure.

DataCapsule Name as A Trust Anchor: The globally unique name of the DataCapsule is derived by computing a hash of the 'metadata'; metadata is a special record at the beginning of a DataCapsule. Metadata is essentially a list of key-value pairs signed by the DataCapsule-owner, that describe immutable properties about a DataCapsule. One such property is a public signature key belonging to the designated single writer; another property is the owner's signature key.

The name of a DataCapsule serves as a cryptographic trust anchor for everything related to the DataCapsule. e.g. given the metadata of DataCapsule, a reader can get the correct signature key and verify the entire history of DataCapsule up to a specific point in time against a specific heartbeat. In addition to verifying entire history, a reader can also get cryptographic proofs for specific records from a DataCapsule in a similar way as the well-known Merkle hash trees [32].

Why use signatures? Signatures are computationally expensive. However, they have two benefits: (a) they enable write access control, which can be verified by DataCapsule-servers or anyone else. (b) Because of the hash-pointers, each signed heartbeat effectively attests the entire history of updates (both the content and the ordering), thus providing a data-provenance.

Secure Responses: Not only does a DataCapsule name enable verification of DataCapsule contents, it also enables a client to verify that only a designated DataCapsule-server is responding to the requests. As we discussed in subsection III-D, this requires careful thought. We address it as follows.

The creation of a DataCapsule involves two operations by the DataCapsule-owner: (a) placing the signed metadata on appropriate DataCapsule-servers, and (b) creating a cryptographic delegation to specific servers, allowing them to respond to queries for the specific DataCapsule. Such delegations are called AdCerts and are essentially a signed statement by the DataCapsule-owner that a certain DataCapsule-server is allowed to respond for the DataCapsule in question.[8] DataCapsule-servers are identified by their names, which is derived in a similar way as the DataCapsule, i.e. by a computing a cryptographic hash over a list of key-value pairs that includes a public key of the DataCapsule-server.

With such delegations in place, a DataCapsule-server can respond to clients on behalf of a DataCapsule and include signatures with its own signature key. In order to facilitate verification by the client, the DataCapsule-server provides its own metadata and the corresponding AdCert along with the signed response. As an optimization, a client and a DataCapsule-server dynamically establish a shared secret in parallel with actual request/response, which they can use to create HMAC instead of signatures and achieve a steady state byte overhead roughly similar to TLS.

Configuration Flexibility: A DataCapsule goes beyond just a simple hash-list and allows for a variable number of additional hash-pointers to past records for efficiency and fault-tolerance. With such additional hash-pointers, a DataCapsule looks more like an authenticated skip-list that allows skipping over records for creating more efficient proofs [25]. Our ingenuity is in exposing the flexibility of which hash-pointers to include to the application. Regardless of the hash-pointers chosen by the writer, all invariants and proofs work with a generalized validation scheme.

[7] Note that infrastructure ensures that the data does not leave specified routing domains as controlled by policies. Encryption provides the final level of defense in the case when the entire infrastructure is compromised.

[8] Of course, a DataCapsule-owner can issue such delegations to a number of DataCapsule-servers. While we do not discuss the mechanisms in detail, in practice, a DataCapsule-owner issues such delegations to storage organizations instead of individual DataCapsule-servers.

These additional hash-pointers provide a very configurable mechanism to support desired performance for high-level interfaces; a file-system interface on a DataCapsule may make all records to include a hash-pointer to a checkpoint record; a video stream in a DataCapsule may use such hash-pointers to allow for records missing in transmission while maintaining integrity properties; and so on.

How to choose the hash-pointers? Depending on the usage scenario, the performance of a DataCapsule can be tuned by choosing an appropriate strategy for hash-pointers. Typically, it's a trade-off between the cost of 'append' and integrity proofs for 'read'. The simplest strategy without any additional hash-pointers results in hash-chain, where 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 in a linked-list design is self-verifying with respect to the newest record in the range).

B. Toward Richer Interfaces: CAAPIs

Many applications are likely to need more common interfaces than an append-only single-writer interface. It should come as no surprise that DataCapsules are sufficient to implement any convenient, mutable data storage repository. The DataCapsule-interface is rather open to system integrators and they can put together an interface of their choice that uses these DataCapsules underneath to meet application specific requirements such as the desired semantics for reads/writes, consistency issues, access control, update ordering, etc. We call such interfaces as Common Access APIs (CAAPIs), see Figure 4.

Recall that in our vision of the world, the applications make the decision of *what* information should be added to DataCapsules and *how to order* such information; the infrastructure merely makes the information durable and available wherever needed. Such CAAPIs are consistent with this vision. As we discussed earlier, DataCapsules provide additional flexibility to ensure efficient performance to match the needs of an application. While we expect that a number of applications will rely on CAAPIs, the simplest of the applications *can* use the DataCapsules directly. Regardless, because DataCapsule serves as the ground truth, the benefit of integrity, confidentiality, and access control are easily carried over to such interfaces.

VI. Replication: Durability and Consistency

As we discussed earlier, a DataCapsule has multiple copies in the infrastructure for durability scalability, fault-tolerance, etc.. The DataCapsule-name represents the DataCapsule and all its replicas. This means that a DataCapsule replica can be reached by using this name as an address; one does not need to know a priori what physical machine(s) does this DataCapsule reside on. Also, recall that a DataCapsule-owner explicitly delegates responsibility for a DataCapsule to specific DataCapsule-server(s)/organizations using AdCerts. For highly replicated DataCapsules, the underlying routing network ensures that the requests are automatically directed to the closest replica. Replicas can be migrated and new replicas can be created based on usage patterns; such placement decisions are made by the owner of a DataCapsule.

A. Why a single-writer append-only mode?

An append-only design is an elegant choice for distributed storage systems. With an append-only interface, the data can not be changed once it is written, which greatly simplifies replication across as a distributed infrastructure; the replication problem translates to keeping the replicas up-to-date. As such, an append-only design is used by many existing systems [20]. The choice for only allowing a single writer, however, enables us to move the serialization responsibilities to the writer/application, and keep the infrastructure design simple while requiring minimum coordination between various servers. In addition, a single-writer signing the records and their ordering allows reasoning about data integrity and provenance without trusting the infrastructure.

Not only does such a design avoid the problem of active coordination between DataCapsule-servers, any append operations from the single writer can be easily forwarded as is to all the DataCapsule-servers in arbitrary order; since the writer is the only point of serialization, and each record has hash-pointers to older records, every signature by the writer is attesting an order of all the records so far, which can be verified by any DataCapsule-server and reader. For any missing records, DataCapsule-servers can synchronize their state in the background. This effectively leads us to a leaderless replication design, which is much more efficient in presence of failures. Note that a DataCapsule meets the definition of a Conflict-Free Replicated Data Type [30].

The single-writer design comes with slightly increased responsibilities for the writer; this design translates to the writer performing two additional tasks: (a) keep some local state, which at the very least includes the hash of the most recent record (potentially in non-volatile memory to recover after writer failures), and any additional hashes the writer might need in near future; and (b) ensure that the durability requirements for the DataCapsule are met. We call this the Strict Single-Writer (SSW) mode.

Multiple writers can be accommodated in two ways: (a) by using a distributed commit service [22], [11] that accepts updates from multiple writers, serializes them, and appends them to a DataCapsule, or (b) by creating an aggregation service that subscribes to multiple single-writer DataCapsules and combines them based on some application-level logic. In the first case, such a distributed commit service *is* the single writer, and represents a separation of write decisions from durability responsibilities.

Note that durability should not be confused with consistency; while they are intertwined, it is possible to achieve one separately from the other. e.g. a number of readers can all agree that a certain record is permanently lost (i.e. consistency without durability), or as another example, a number of readers may not agree on the most recent state of a DataCapsule because of stale replicas (i.e. durability without consistency).

B. Durability

In the simplest case, the writer receives a single acknowledgment from the closest DataCapsule-server; the writer can make progress while the DataCapsule-server propagates the new updates to other DataCapsule-servers hosting a replica in the background. However, during a small window of time, some part of the DataCapsule is stored on only one single DataCapsule-server. In case this DataCapsule-server crashes, the hash-pointer chain can result in a 'hole' (see Figure 2). Such holes can be transient or permanent, depending on failure mode of the DataCapsule-server.

For certain applications such as video streams, minor transient (or even permanent) loss of data is acceptable; the additional hash-pointers in the DataCapsule enable resilience against such failures. On the other hand, applications that can not tolerate such loss, the writer can indicate that the DataCapsule-server must collect additional acknowledgments from other replicas and return it to the writer. The writer must block and retry unless it is satisfied with the durability of a record; such a mode results in a reduced performance at the cost of greater durability.

C. Consistency

In a strict single-writer (SSW) mode (such as an IoT sensor recording data in a DataCapsule), records are *strictly ordered* as defined by the hash-chain ordering. Thus, readers that see different states from replicas can simply discard stale information. The resulting consistency semantics observed by readers are similar to that of sequential consistency. On the other hand, a reader interested in the most up-to-date state of a DataCapsule can query all replicas of a DataCapsule and achieve read semantics similar to that of strict consistency at the risk of losing fault tolerance; such a reader must block if any single replica is unavailable.

In cases where SSW mode isn't available (e.g. because of writer failures in the absence of non-volatile storage to store most-recent hash), a Quasi-Single Writer (QSW) mode can be used. The assumption in QSW mode is that there *can* be more than one concurrent writers from time to time, but such situations are rare, and that each such writer attempts to synchronize its state out-of-band (e.g. a personal file-system mounted on multiple devices, where it is rare, but possible, that there are multiple concurrent writers). In QSW mode, there is a chance of *branches* in the DataCapsule. Ignoring the additional hash pointers, a branch is a condition when two or more records have hash pointers that point to the same record. Such branches result in a *partial order* of records. In such a case, a reader can only expect strong eventual consistency.

Note that updates *across* DataCapsules can be ordered using entanglement schemes described by [25].

VII. Secure Routing in GDP Wilderness

So far, we have assumed an underlying routing network that enables clients to address DataCapsules directly. Such routing network is composed of GDP-routers that enable efficient routing in a flat address space (see Figure 4). Like DataCapsule-servers, GDP-routers are owned and operated by various organizations called 'routing domains' that collectively form a federated network. Recall the challenges of ICNs from subsection III-B: traditional ICNs suffer from privacy issues and do not support freshness properties. Such challenges are even more pronounced in a federated network with no central authority for assigning names.

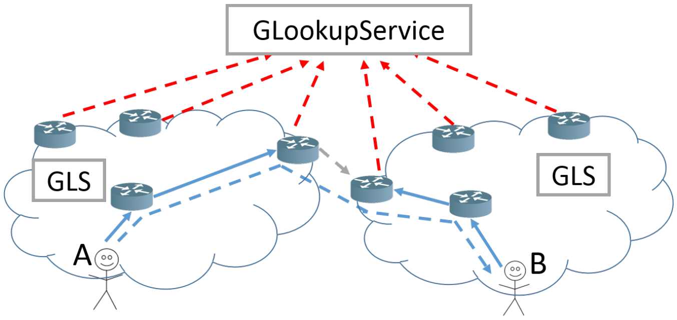

Fig. 5: Routing architecture for the GDP network. Two routing domains, their corresponding GLookupService, the global GLookupService are shown. Communication between two entities with names 'A' and 'B' is enabled by a series of GDP-routers.

The routing related goals for this network are twofold: (a) provide locality of access and enable 'anycast' for the layer above, and (b) ensure routing security to prevent trivial man-in-the-middle attacks, i.e. ensure that people can not simply claim any name they desire. Of course, this network needs to be scalable as well. At a system level, we also desire to maintain control over data, i.e. (a) ensure that data does not leave specific boundaries, and (b) control on who can serve (or advertise for) a specific object (freshness issue). The GDP network enables such control as well.

While it is a work in progress, we briefly describe the architecture of the GDP network. To ensure scalability, locality of access, and security of routing, we use two principles: (a) a hierarchical structure for routing enabled by routing-domains, and (b) independently verifiable routing state maintained separately from GDP-routers.

Routing delegations: In order to achieve the routing security goal and maintain control over data, we use strict cryptographic delegation for being able to route to a name; we call such a routing delegations RtCert. A RtCert is a signed statement issued by a physical machine (e.g. a DataCapsule-server) to a GDP-router authorizing the GDP-router to send/receive messages on behalf of DataCapsule-server. RtCerts are issued during 'secure advertisements'.

Secure advertisements: When clients and DataCapsule-servers connect to GDP-routers, they advertise the names that they can service to the routing infrastructure. The set of available names is advertised via one or more a *naming catalogs* in the form of DataCapsules containing individual advertisements and access-control credentials[9]. The advertiser

[9] Such credentials enable network-level routing restrictions, such as restricting subscription to DataCapsule updates (i.e. who can join a secure multicast tree associated with a given name) or to stop denial of service attacks at the border of a trust domain.

must prove to the routing infrastructure that it possesses authorized delegations for each of its advertised names; we call this mechanism 'secure advertisement'. All such proof is included in a catalog, signed by the advertiser.

Advertisements have corresponding expiration times, which can be deferred as a group by appending extension records to the catalog. This architecture is particularly optimized for transient failure and re-establishment of DataCapsule-service (e.g. when a DataCapsule-server fails and restarts, or when a trust domain becomes unavailable due to transient network failure). It also allows names and access control certificates to be easily synchronized with routing elements within the network (such as the GLookupService mentioned below).

As an example, when a DataCapsule-server connects to a GDP-router, the DataCapsule-server engages in a challenge-response process with the GDP-router to prove that it possesses the private key corresponding to the name of the DataCapsule-server and that it possesses various properties referenced in delegations (e.g. membership in a given organization). Once this process succeeds, the DataCapsule-server issues a RtCert to the GDP-router (or to the routing domain, depending on the granularity policies). Once a DataCapsule-server has proven to the GDP-router that it is who it claims to be, it can then export the AdCerts for various DataCapsules by making naming catalogs available. The routing infrastructure can thus verify the chain of trust created by AdCerts and RtCerts to ensure secure routing to such names.

Within a routing domain, all routing information is kept in a shared database that we call a GLookupService. Such information includes any RtCert issued by the DataCapsule-server, AdCerts presented for each of the DataCapsules hosted by the DataCapsule-server, etc. The GLookupService is populated during the 'secure advertisement' process. When a GDP-router receives a request for a destination it does not know the route to, it can query this GLookupService, verify the routes by following the chain of trust, and update its local state. Such a model is similar to those of SDNs [27], where an SDN-controller plays a similar role to the GLookupService.

Recall that routing domains are hierarchical in nature. When a specific name cannot be found in the local GLookupService, such a name is queried in the GLookupService of the parent routing domain, and so on. Finally, there is a global GLookupService that contains verifiable routing information for all publicly available DataCapsules; this top-level GLookupService corresponds roughly to a tier-1 service provider in the general Internet. It requires that any information acquired during the advertisement process also be propagated to the parent GLookupService. This is where any policies for the scope of a DataCapsule are adhered to; any restriction on where can a DataCapsule be routed through are specified by the DataCapsule-owner at the time of issuance of AdCert to the DataCapsule-servers; the routing infrastructure adheres to such restrictions when propagating such routing information.

Note that the GLookupService is essentially a key-value store and is not required to be trusted; existing technologies such as distributed hash tables (DHTs) can be used to implement a highly distributed and scalable GLookupService. Also note that the GLookupService architecture is similar to the hierarchical nature of DNS, with the exceptions that (i) we query flat names and (ii) the returned information is independently verifiable.

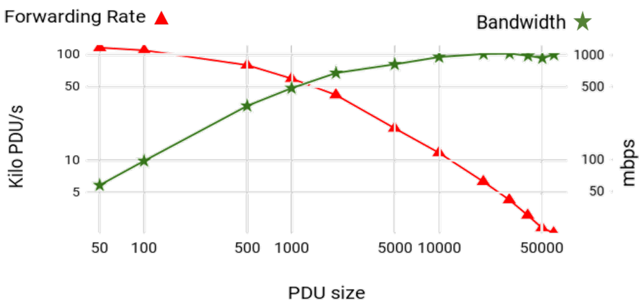

Fig. 6: Forwarding rate in PDUs/sec and throughput as a function of changing PDU size. At steady state, a preliminary version of our GDP-router is able to achieve close to 1 Gbps throughput as PDU size reaches close to 10k bytes. The PDU processing rate is 120k PDU/s even for very small sized PDUs.

VIII. System Implementation

We have implemented a prototype to validate the feasibility of the Global Data Plane and associated concepts. We also maintain a small infrastructure at our university that has been running since late 2014, and has gone through a number of major upgrades. Currently, it implements a subset of features but is usable and has enabled us to understand the application developers' needs. While our implementation is still a work in progress, it is very much a real system with real users and supports a number of applications ranging from time-series environmental sensors, visualization of time-series data via a web-browser, audio/video support via GStreamer [34], file-system support for TensorFlow [7], web gateways using REST and websockets, and many more.

The core component of our routing implementation is a Click [19] based router that supports flat namespace routing. GDP-routers route PDUs in the flat namespace network. GDP-routers expose a TCP-based interface to clients and DataCapsule-servers, but use a UDP tunneling protocol for inter-router communication. Each GDP-router maintains a local forwarding table (FIB), and also publishes the information to a GLookupService. Our GLookupService is a limited application that responds to UDP queries and uses a database server for back-end storage. When a destination can not be found in the local FIB, a GDP-router queries the GLookupService and looks up the destination on-demand.

The DataCapsule-server is a daemon written in C which uses SQLite for the back-end storage; each DataCapsule is stored in its own separate SQLite database. SQLite enables a DataCapsule-server to respond to random reads efficiently.

Client applications primarily link against an event-driven C-based GDP library. The GDP library takes care of connecting to a GDP-router using TCP, advertise the desired names, and

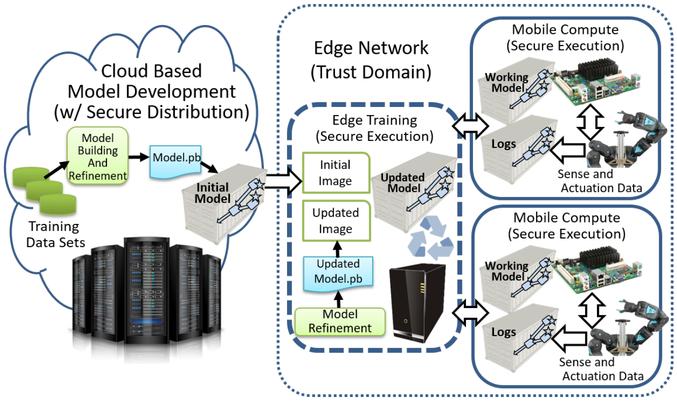

Fig. 7: Robotics and machine learning at the edge. General purpose robots are trained in the cloud and refined at the edge. DataCapsules serve as the information containers for both models and episode history of actual events for future training/debugging. The GDP enables partitioning resources based on ownership, and allows reasoning about flow of information by a clear separation of policy from mechanisms.

provide the desired interface of a DataCapsule as an object that can be appended to, read from, or subscribed to. In addition, we have developed Python and Java language bindings for client applications, which are simply wrappers around the C-library. These higher-level wrappers provide an object-oriented interface and enable quick prototyping.

In order to achieve good performance from the overall system, it is of utmost importance to have a high-performance network underneath. We claim that it is possible to achieve high performance even with the security mechanisms, because the additional cost of cryptographic validation is incurred only once per flow per router at the beginning of flow establishment. To validate this assumption, we ran a simple experiment with an unoptimized preliminary version of our GDP-router running on a 4-core EC2 `c5.xlarge` instance. We measured the PDU forwarding rate and sustained bandwidth that a single GDP-router can support in a steady state. To test this, we a total of 32 client processes and 32 server process spread uniformly over 4 16-core `c5.4xlarge` instances in the same data center and all connected to the GDP-router. After advertising, each client simply sends messages of a given size destined to a specific server as fast as it can. As seen in Figure 6, the throughput increases as the PDU size increases and easily reaches 1 Gbps.

IX. A Case Study

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 [33]. Robots are increasingly embedded in the real world environments such as factory floors, households, etc. Programming these robots to work in complex environment remains a challenge, but machine learning seems a promising mechanism. However, such machine learning is driven by large volumes of potentially sensitive data, and thus requires appropriate data handling and management strategy.

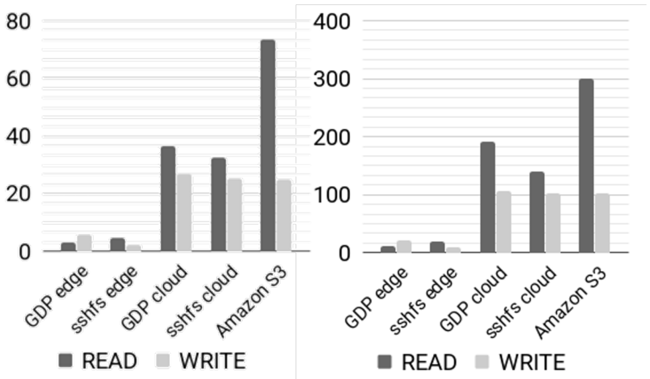

Fig. 8: Read/write times (seconds) for our case study comparing GDP to other options. We show a 28 MB (left) and a 115 MB (right) model (averaged over 5 runs). Smaller is better.

For our study, consider a factory floor with worker robots as shown in Figure 7. Although general-purpose models are trained in the cloud, they must be refined for the specific environment of the factory. It is desirable to keep the environment-specific information (e.g. refined models or episode history) restricted to the factory floor for privacy reasons. DataCapsules and the GDP provide a clean architecture where one can reason about control on both raw data and trained models.

Toward the goal of enabling secure machine learning, we developed a CAAPI for TensorFlow—a popular machine learning framework [7]. TensorFlow supports custom filesystem plugins; such plugin can be used for all file operations (loading and storing data, models, checkpoints, events, etc.) during various stages of training and deployment of an arbitrary machine learning application. Default TensorFlow distribution ships with a few such plugins including one that allows use of Amazon S3 as a storage back-end. Our CAAPI is in form of a C++ plugin (layered on top of the GDP library) that can be loaded into TensorFlow at runtime, allowing existing applications to use DataCapsules. Internally, this CAAPI maintains a top-level directory in a single DataCapsule. Each filename is represented as its own DataCapsule; the top-level directory merely maps filenames to DataCapsule-names.

We first show that given equivalent infrastructure, the GDP and DataCapsules provide comparable performance to existing cloud systems (S3). Next, we demonstrate how the GDP and DataCapsules enable the use of close-by infrastructure for better performance. For our experiments, our client is in a residential network, with the Internet bandwidth capped to 100/10 Mbps (upload/download): a good representative of an average household Internet connection in United States [2]. We compare against an Amazon S3 bucket in a specific S3 region (on the same continent). We run the GDP infrastructure in Amazon EC2 in the same region as the S3 bucket. We also compare against SSHFS [17] on the same host as our GDP infrastructure[10]. Next, we run the same experiment, but this

[10] 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.

time we use the GDP infrastructure in local environment using on-premise edge resources. Once again, we run SSHFS for comparison. We repeat storing and loading of two pre-trained and publicly available machine learning models of different sizes. As can be seen from Figure 8, the GDP provides performance somewhere between that of SSHFS and S3 when using the cloud infrastructure. As expected, the performance when using edge resources is orders of magnitude better.

In our case study, DataCapsules achieve comparable performance to existing systems while also enabling use of close resources and control over integrity and confidentiality of information. Even further, the federated nature of the Global Data Plane 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.

X. Conclusions

While the GDP infrastructure is still ongoing work, we presented a brief outline for what is needed to enable a utility model for edge computing, noting the challenges ahead and providing a high-level overview of how to address these challenges. A number of challenges remain, such as trust-based resource placement and routing; secure execution to leverage DataCapsules and achieve end-to-end security; and wide-scale deployment challenges. Regardless, we envision that a seamless combination of edge resources and data-center resources using a federated infrastructure—one that takes the domains of ownership in account—*is* the way forward. *With cloud computing, we learned how to scale things well within a single administrative domain. With edge computing, we must learn how to scale in the number of administrative domains to enable a truly federated infrastructure.*

References

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

[2] Measuring Fixed Broadband Report - 2016. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-report-2016.

[3] Nest Weave. https://nest.com/weave/.

[4] SmartThings. http://www.smartthings.com/.

[5] The Thing System. http://thethingsystem.com/.

[6] What Edge Computing Means for Infrastructure and Operations Leaders. https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders/.

[7] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

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

[9] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., et al. Scone: Secure linux containers with intel sgx. In *OSDI* (2016), vol. 16, pp. 689–703.

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

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

[12] Costan, V., and Devadas, S. Intel sgx explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[13] Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., and Riviere, E. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review 45*, 5 (2015), 37–42.

[14] Ghodsi, A., Shenker, S., Koponen, T., Singla, A., Raghavan, B., and Wilcox, J. Information-centric networking: seeing the forest for the trees. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), ACM, p. 1.

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

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

[17] Hoskins, M. E. Sshfs: super easy file access over ssh. *Linux Journal 2006*, 146 (2006), 4.

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

[19] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Transactions on Computer Systems (TOCS) 18*, 3 (2000), 263–297.

[20] Kreps, J., Narkhede, N., Rao, J., et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (2011), pp. 1–7.

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

[22] Lamport, L., et al. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

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

[24] Lucas, P., Ballay, J., and McManus, M. *Trillions: Thriving in the emerging information ecology*. John Wiley & Sons, 2012.

[25] Maniatis, P., and Baker, M. Secure history preservation through timeline entanglement. *arXiv preprint cs/0202005* (2002).

[26] Mazières, D. D. F. *Self-certifying file system*. PhD thesis, Massachusetts Institute of Technology, 2000.

[27] McKeown, N. Software-defined networking. *INFOCOM keynote talk 17*, 2 (2009), 30–32.

[28] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014), 2.

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

[30] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems* (2011), Springer, pp. 386–400.

[31] Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal 3*, 5 (2016), 637–646.

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

[33] Tanwani, A. K., Mor, N., Kubiatowicz, J., Gonzalez, J. E., and Goldberg, K. A fog robotics approach to deep robot learning: Application to object recognition and grasp planning in surface decluttering. *arXiv preprint arXiv:1903.09589* (2019).

[34] Team, G. Gstreamer: open source multimedia framework, 2016.

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

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

[37] Zhang, B., Mor, N., Kolb, J., Chan, D. S., Lutz, K., Allman, E., Wawrzynek, J., Lee, E., and Kubiatowicz, J. The cloud is not enough: saving iot from the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (2015).

[38] Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., Crowley, P., Papadopoulos, C., Wang, L., Zhang, B., et al. Named Data Networking. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 66–73.