End-to-end Encrypted Scalable Abstract Data Types over ICN

Christian Tschudin
University of Basel, Switzerland
christian.tschudin@unibas.ch

ABSTRACT

In Information Centric Networking (ICN), data items are made accessible through their name rather than their storage location. We extend this storage abstraction and complement it via a name lookup service for retrieving the latest version of a name binding in order to implement abstract data types (ADT), specifically "append-only logs" and "mutable key-value stores". Moreover, we make these data types scalable by choosing implementation techniques known as CRDTs (conflict-free replicated data types) and we use end-to-end encryption for protecting the content and structure against untrusted storage providers and forwarding elements. In this paper we describe our architecture, the interface to the ADTs and report on a prototype implementation for the cloud that is inspired by a real Fintech use case.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; *Cloud computing*; • **Software and its engineering** → *Abstract data types*; • **Security and privacy** → *Management and querying of encrypted data*; • **Information systems** → *Hashed file organization*;

ACM Reference Format:
Christian Tschudin. 2018. End-to-end Encrypted Scalable Abstract Data Types over ICN. In *5th ACM Conference on Information-Centric Networking (ICN '18), September 21–23, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3267955.3267962

1 INTRODUCTION

This work is inspired by a use case of Symphony Communication Services LLC[1], a high-growth technology company in Palo Alto that provides secure collaboration for financial services and other security-sensitive industries. Symphony operates a cloud-based infrastructure that enables companies to outsource their chat messaging in a secure and regulatory-compliant way: Messages are en- and decrypted on end-devices only, using a user key and company-specific escrow keys. Symphony never sees clear-text messages and, should law enforcement agency determine a need to obtain the customer's messages, must (and could only) hand over the set of encrypted messages, while the agency will need to obtain the keys from the customer or its third-party key custodian directly.

[1]https://en.wikipedia.org/wiki/Symphony_Communication

ICN '18, September 21–23, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-5959-7/18/09.
https://doi.org/10.1145/3267955.3267962

Application-level ICN. Symphony's system is currently based on standard cloud technologies like Hadoop, MongoDB, PubNub, Kafka and many more. My view[2] was that Symphony's use case would best be handled by an "application-level ICN" approach: Chat *messages* become immutable encrypted named data; chat *rooms* are ordered collections of such objects and must, for audit trail purposes, have an immutable history; finally the ICN repos in the cloud serve as an append-only store for all immutable content.

Such a re-architecting is not easy to retrofit to a running system, but Symphony has started to adopt portions of this new "Symphony 2.0" architecture when need for replacement or opportunities showed up. In this paper I report on an independent proof of concept that was written in Python and that demonstrates Symphony's use case as an application running over an extended ICN[3]. Many additional services that are part of Symphony's offerings are not covered by this PoC like e.g. information barriers, encrypted search, entitlement and access control systems, or the web browser- and smartphone-based client apps.

The Network becomes the Database. In the new architecture there is no need for stand-alone database servers: Instead, all digital assets are handled as "linked encrypted data" accessed via a flat namespace from an Information Centric Network (in this work, we focus on ICN as represented by NDN [37], CCNx [14] or CICN [16]). A chat message is referenced by its self-certifying name (the hash of its binary representation), while a collection of chat messages, i.e. a chat room, is a linked list of such hash names, and the room's state being represented by the most recently added chat message. Chat messages are immutable, hence their name never changes. But chat rooms have dynamic content, hence a name derived from a binary representation of the latest chat message not sufficient. This is solved by creating a permanent "base name" from the empty room, to use this as durable reference for a chat room, and to provide a lookup service for getting hold of the latest version's name based on the base name. In our prototype we call this lookup the HEAD service, while we call ICN's storage for immutable content the WORM service (write-once-read-many). Similar to GitHub's role as a version directory [10], or IOTA's "set of tip nodes" which is also called "edge set" [21], the HEAD service documents the dynamic frontier of the append-only WORM.

ICN and a Service Architecture for ADTs

A canonical ICN network à la NDN or CCNx provides a single service called `read()`. Our position is that the ICN layer must present itself as much richer than a mere data packet delivery service: The Information-Centric Network has to become "the" database and

[2]During an industrial leave at Symphony I was part of an architecture team that looked into re-designing Symphony's system.
[3] The mapping of Symphony's use case to ICN concepts, as well as the techniques presented in this paper, do not necessarily reflect Symphony's views but rather my own.

a platform, potentially cloud-based, for services that refine this database' content. A first step consists in complementing the current "READ" primitive (implemented with Interest/Data messages) with (a) a "WRITE" primitive on equal footing (WORM service), (b) Remote Procedure Call support (RPC) as integral part of ICN, (c) a name translation service for updating and disseminating latest version information (HEAD service), and finally (d) libraries that turn these base services into high-level abstract data types which hide the tedious details of implementing them with scaling and encryption in mind (see Figure 1).

In Section 2, we first point to the many domains (in distributed systems) that are relevant to this paper and single out CRDTs (Conflict-Free Replicated Data Types) in Section 3. Section 4 describes our service architecture while Section 5 shows code excerpts of two demo applications (chat and DropBox replacement).

2 RELATED WORK

In this Section we review technologies that relate to this work and which mostly stem from the domain of distributed systems (rather than networking) For each domain we highlight the problems (with regard to the use of an ICN network) that the corresponding concepts address. The CRDT topic is handled in an independent Section directly afterwards.

2.1 ICN APIs beyond interest/data messaging

Several attempts have been made for wrapping the Interest/Data messaging into more conveniently usable primitives, the motivation being that for example "reliable and secure application-level message exchange" or "file replication" are the relevant services in which application developers are interested in. Support for Producer/Consumer communication has been the focus of several authors (Moiseenko [13], PARC's CCNx 1.0 lib and now CICN sockets [16]). ChronoSync [39] can be seen as way to automatically replicate hierarchically organized data over ICN. The focus of these efforts is on the data itself and but not on a well-defined set of manipulation procedures. For example, ChronoSync implements "set reconciliation" but does not offer a `set` data type where one can easily add elements, test for presence, or compute the difference between two sets. Similarly, manifests (e.g. FLIC [36]) only capture the linked nature of a data collection but do not implement an abstract data type proper e.g., how to `append()`, `seek()` or `write()` into such a collection.

2.2 Coupling Distributed Apps with Mailboxes and Queues

From a networking perspective, "shipping the bits" is the core challenge when coupling processes. However, more support is needed for running a distributed application. In this context, the use of mailboxes (Agha's actors concept [2]) or a message bus is a well established concept that goes beyond mere communication as it includes reliable *storage* of transient messages. In an evolved version, the mailbox concept is known in the ICN context in form of PubSub [5, 32]. Examples of available offerings today are Google's PubSub service [9] or Amazon's SNS+SQS [3]. Again, the focus is on disseminating content but this time it permits to trigger and chain RPCs together and to implement workflows. For building

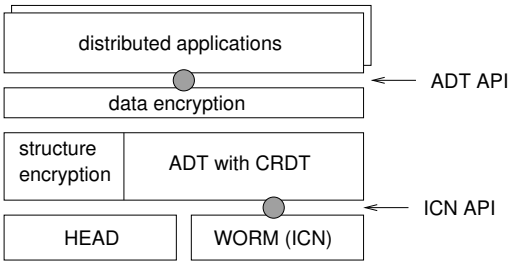

Figure 1: A software stack for implementing scalable ADTs over HEAD and WORM services.

distributed applications these are extremely valuable high-level services, but (unlike the ICN flavor of PubSub) they *need an active element in the cloud* which can persist items with strong guarantees like "exactly-once" semantics (while the ICN pubsub typically is best effort): Having "published" an items means that responsibility of delivery has fully passed to the service provider, although there might be limits for how long an item is stored (e.g., up to four days). In this paper we study another coupling of the distributed elements, specifically a HEAD name service similar to what GitHub provides.

2.3 Append-only (and Log-Structured) Storage

ICN so far is very much focused on single name-data chunks rather than how *multiple* chunks are actually persisted, both at the retrieving end (in a repo) or on the writing side – which is relevant for performance and for data consistency reasons. In operating systems, append-only data structures are a well known concept and have been implemented multiple times, for example in the `venti` file system [24] of Plan9. Log-structured file systems [26] also operate along this line and have become an important part in highly reliable systems and consensus services like e.g. RAFT [20]. Blockchains [17] (and distributed ledgers in general) also are append-only data structures, but come with extreme assumptions regarding the open set of "contributors" to these data structures, involving proof-of-work or similar to prevent malicious actors to rewrite the stored content. In our work we assume cooperative software e.g., controlled by a trusted entity or open source as well as trustworthy service provisioning (which is independent of provenance guarantees e.g., in form of messages being signed by the end device). Again, ChronoSync should be mentioned here, too, because it operates on synchronizing ever-growing Merkle trees.

It should be noted here that major ICN projects often present themselves to cover immutable data by pointing to cryptographic binding between the data and its name. However, only `READ(name)` is implemented inside information-centric networks like NDN or CICN: The "publishing", as it is unfortunately called, happens out-of-band or on top of the read() service. The NDN Common Client Library and its examples [18], for example, lack a `WRITE(data)` method: Only via ChronoSync is it possible to add content to a name space, although repos can register their prefix with the routing substrate, which leaves it to some mechanism *outside* the basic NDN protocol how data is written (over the network) to a repo. Recent work [22] seem to indicate a shift. In the present paper we make the point that in an ICN there must be "persistent write" operation

at the same level as the `read()`, in order to implement collaborative shared data structures.

The recent push towards a *decentralized Web* includes named-data access at application-level and goes hand-in-hand with append-only data repositories. Using the content's hash as the data name is common practice and is used for the versioning large data sets, see e.g. `dat` [19] or Quilt [23] which offers "Git for data". The latter operates a logically centralized registry (like GitHub is for code) where clients can upload or learn about the latest changes. Secure Scuttlebutt (SSB) [33] follows a radically distributed approach: Peers only publish to their *own* append-only signed hash chain (log) instead of some central repository. Nevertheless can this system be used to implement e.g. a shared chat application. A core element of SSB is the so called "subjective reader" where each client reconstructs the chat dialogue by extracting the relevant cross-linked posts (which are referenced by the hash of the corresponding log entry) from the logs of the involved peers. As an application-level ICN, SSB occupies an interesting position in the caching space: The append-only logs are eagerly replicated in totality based on a binary decision whether another peer's log is (subjectively) relevant or not: A set difference protocol is used in SSB to check for and retrieve changes from a peer, in a fully receiver-driven fashion. Due to the eager replication of the relevant logs, a chat dialogue is reconstructed locally, without requiring network access.

2.4 End-to-end Encryption

End-to-end encryption is considered the silver standard for distributed applications (for gold standard, privacy of meta-data would be required, too) and even has gained mass popularity once applications like WhatsApp moved to the corresponding technology (see the SIGNAL protocol [6]). Symphony's solution predates these efforts: From the beginning Symphony offered end-to-end encryption and meta-data protection but added escrow keys for regulatory reasons. In our work we have also used this concept, but do not explore the implications of re-keying and simply assume that data is encrypted once, with keys only known to the entitled parties.

The classic way of implementing end-to-end encryption for "secure broadcast" is to use a random symmetric data-encryption-key (DEK) which is encrypted for each recipient using their public key, these keys being called key-encryption-keys (KEK). Escrowing is implemented by adding an escrow-specific KEK: In case of a chat room, the group of members is expanded to also include the escrow; a one-to-one communication is turned into a group of three members that includes the original correspondants plus the escrow. In ICN, the use of DEK and KEK is a well established technique e.g., in name-based access control [38].

3 CONFLICT-FREE REPLICATED DATA TYPES (CRDT)

With ICN's focus on *retrieving* bits, less attention has been paid so far on the multi-*writer* problem where conflicts can occur and show up at the naming level: How are version numbers allocated e.g., when redundant sensors can concurrently update the temperature readings of a room? How can one solve the problem that two independently posted chat messages receive the same sequence number? An important discovery was that in many cases it is not necessary to resolve such an ordering conflict as they can be avoided with appropriate techniques, an approach that has received the label of CRDT. Results in Conflict-free Replicated Data Types appeared in 2009 by work done at INRIA [11, 31]: The main statement is that certain data structures can be modified concurrently without coordinator or global consensus yet enjoy the property of being eventually consistent. As a consequence, distributed applications relying on such data structures will be *fully* scalable (active conflict resolution inevitably introduces a bottleneck element). The original work demonstrated this concept with shared editors, counters and key-value maps, among others. Databases like Riak [25] have included these concepts which continue to undergo refinement.

The core of CRDT are commutative update operations (which can be done and propagated anytime), specially crafted merge operations and read operations that "decode" from the collected data trails the current values.

For example, **a scalable counter** can be implemented by a vector where each component stands for the current value of a party in the system. The read operation consists in summing up all components, updating is achieved by letting each party increment its component and periodically exchanging its vector to neighbors e.g. using a gossip protocol, merging is done by copying the maximum of conflicting component values. Eventually, an increment will have propagated to all parties. Unlike other eventually consistent systems (in databases), CRDTs enjoy deterministic eventual consistency: that is, the outcome does not depend on the order in which the parties perform an update or merge operation (which is easy to see in the above example).

3.1 Auto-Merge Log (AMlog)

We have implemented an append-only log file abstraction (or "list" abstract data type) as a CRDT in form of a directed acyclic graph (DAG). Starting from the empty list node, each update appends a "log node" that refers to the predecessor node via its hash name. If two updaters append at the same time, two such nodes exist in the graph, representing two different "branches". Later, one of the updaters, or a third party, can add a "merge node" that reduces the two or more nodes to one again. These merge operations can also happen in parallel, wherefore several rounds of merging may be necessary if many log or merge nodes are appended in parallel. The concept is very similar to IOTA's tangle [21], a DAG where each transaction node always merges two predecessor nodes. Ultimately, when no updates are being made and all merging is done, the DAG has a single frontier node (called "tip" in IOTA). If the AMlog-reading clients need a serialized view of the DAG's entry, it suffices to traverse the DAG in the same agreed-on order e.g. after running a topological sort and breaking ties based on some node property (e.g. hash value of its binary representation).

The AMlog is our central data structure that allows multiple writers to append to a shared log without need to coordinate. If two or more writers append concurrently, these additions are recorded in the DAG and the existence of such branches will be made visible through the HEAD service. How these branches are merged is explained in the "Merging Name-Sets" paragraph of Section 4.4.

3.2 Observed-Remove Maps (ORmap)

In order to have a dictionary-like object, we implemented the "observed-remove maps" where removal is recorded in dependency of the latest state that an updater sees. This permits to capture cases where an element is removed and added "concurrently": The party having deleted an element, and discovering that the same entry shows up again, would have to delete that new entry. Note that the operating system plan9 had similar semantics on its implementation of directories where it was possible to have multiple entries with the same name, as another consequence. We refer the reader to the original work in [4] where the "observed-remove set" protocol is formally described.

4 A SERVICE ARCHITECTURE FOR SCALABLE ADTS OVER ICN

In this paper we abstract from the classic *Interest/Content* protocol that is characteristic for many papers on ICN. Instead, we define basic APIs for the WORM service as well as the new HEAD service. In order to support Symphony's use case of end-to-end encrypted chat we then layer abstract data types (ADT) on top of these services, namely a append-only list and a key-value-store abstraction, as is shown in Figure 1.

In this section we step through and motivate the design decisions of our service architecture, shown in Figure 2. The second subfigure shows the preferred configuration which consists of cloud/edge-based services that mediate between end devices and the persistence and name lookup services (WORM and HEAD) which, together, form our "enhanced ICN".

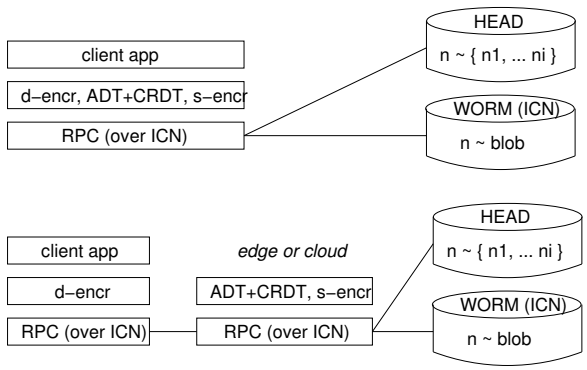

Figure 2: Service Architecture: Running ADT logic on end devices only vs using cloud- or edge-based assists.

4.1 Linked Encrypted Data, Data Structure Encryption and Storage-level Encryption

Similar to Name-based Access Control [38] we use content encryption to prevent unauthorized access to content, but distinguish between encrypting *data* content (e.g. fields of a chat message) vs encrypting *structure* information (fields required to manage linked list or index tables). This distinction has to do with Symphony's service of relieving end-devices from the sometimes heavy-lifting job of providing reliability and high availability despite all possible cloud failure scenarios: Semi-trusted edge nodes (i.e. run by Symphony and trusted by the customers) can then operate on linked lists of encrypted content and other aggregating data types without having access to the chat messages' content. The operator's separate and orthogonal "data structure encryption" protects the logical relationship between single blobs and permits the operator to offer computationally intensive services like fast retrieval via index tables e.g. "fetch the 20 most recent items".

Such semi-trusted nodes are functionally not mandatory, though: Indexing tasks, for example, can run on each end device, too. This results in a fully serverless mode[4] as shown in the upper subfigure of Fig. 2. However, in light of the high chat message volume and churn (and considering that some end devices consist of JavaScript running in a web browser), this is not practical. In our prototype we made the placement of ADT support configurable in order to test both configurations as shown in Figure 2.

Potentially, a third level of encryption takes place at the level of HEAD and WORM. While the current configuration assumes that Symphony operates the edge/cloud *and* the WORM and HEAD services, the cloud providers also encrypt user data independently in order to protect against theft of the physical disks. In fact, storage for WORM and HEAD could be provisioned by a third party that offers an additional level of "encryption-at-rest" guarantee[5]. This configuration is transparent to the end devices, which continue to use their end-to-end encryption and self-certified names.

4.2 Remote Procedure Call, Streaming Mode

As can be seen in Figure 2, RPC is used for the communication between end devices and edge/cloud services, as well as between edge/cloud services and HEAD+WORM service.

The prominent use of RPC has several reasons. First, it enables to group one or more service requests in a session where the service provider can authenticate an end device (similar: edge/cloud services authenticate themselves against the HEAD+WORM providers). The underlying motivation is contractual as each request consumes resources and must be backed by monetary compensation.

Second, RPC (in a suitable implementation) establishes a security context where all parameters and results are encrypted. Recall that end devices are responsible for handling data encryption but structure information must be communicated forth and back between the end device and some edge/cloud service instance, hence must be in clear for the edge/cloud service but secured against eavesdroppers.

Third, RPC establishes a context for transient state that can be amortized: Authentication and the establishment of per-RPC session keys will apply to several method invocations. Following Google ProtoBuf framework [7], we designed the RPC interface such that parameters as well as results are streamed. The following ProtoBuf specification shows the WORM methods of our "Symphony 2.0" proof of concept:

```
service S2worm {
  rpc write(stream s2wire.Opaque)
      returns (stream s2wire.S2pointer);
```

[4] This refers to the *serverless web*, not the recent term of *serverless computing*.
[5] AWS's Simple Storage Service, which can be seen as a prototypical WORM service, persists data in encrypted form.

```
  rpc read(stream s2wire.S2pointer)
      returns (stream s2wire.Opaque);
}
```

Opting for the RPC streaming mode has also to do with the desired service quality: Writing to the WORM must be reliably persistent, which requires this service to use a replicated storage system in order to survive the crash of a whole data center. This mandates some form of consensus in a High Availability zone, e.g. using RAFT, and drives service latency to the range of 100 to 200 msec per write operation (e.g. these are numbers from Google's PubSub which has the same reliability requirement). However, streaming permits to pipeline these writes such that the per-write latency in a batch of writes is much lower than a RTT.

4.3 WORM

The WORM API, as shown in the previous paragraph, may surprise at first view: The `write()` operation receives an opaque byte array and returns the object's name, which is a self-certifying name. Why not having the end device directly derive the self-certifying name, without paying the price of a full round-trip? One motivation is that, for regulatory reasons, Symphony's customers require that Symphony have an audit trail of each data piece stored in Symphony's systems. Second, Symphony can add meta-data and repo-specific (locator) info to the name, which the end-device is not aware of. Third, only by forcing the round-trip to the WORM service with its reliability guarantee, can the end device be sure that the item has been persisted and will survive even catastrophic system failures. Finally, the service providers can keep track of `write()` requests in order to handle "storage leak" which can happen if an end device creates a data chunk but crashes before being able to write a reference to it into a higher-level data structure.

4.4 HEAD

Two prominent ICN systems, NDN and CCNx 0.7, championed the use of version numbers as part of a data item's name. Using content discovery features, applications could then start with a stem name and find out the latest version of the corresponding data item. As we use self-certifying names, this is not an option: Instead, we need an explicit naming service that returns the latest version.

A straight-forward approach would be to implement a DNS-like distributed translation service from one name to another, NDNS comes to mind [1]. However, inspired by Git, we designed a "HEAD" service that maps a name to a *set of names*. This requires some explanation and careful distinctions:

In our "Symphony 2.0" context, names are hash pointers. This is fine for immutable object like a chat message (and commits, in Git). For objects that can be superseded by a more recent version, we introduce the notion of a base name. For a list, an empty object is created (including some nonce to make it unique) and the hash of this object becomes the (self-certifying) base name. In Git, our base name corresponds to a repo name. As end devices create new versions and want to commit their changes to the system, they do so by referring to the base name for which they register a commit pointing to the precedent version.

We use the HEAD service to collect these commits but the service will do nothing to merge or even resolve conflicts. The clients can call *append()* for new commits, which in Git corresponds to creating new branches. The second, and preferred method of a client to interact with the HEAD service, is to call *replace()* which lets the HEAD service remove a given name set and replace it by a single new commit. The HEAD API, using the ProtoBuf language, looks like this:

```
service S2head {
  rpc getHead(s2wire.S2pointer) // base name
      returns (HeadSet);
  rpc append(s2wire.S2pointer,  // base name
             s2wire.S2pointer)  // name of commit
      returns (google.protobuf.Empty);
  rpc replace(s2wire.S2pointer, // base name
           HeadSet,        // set of names to be removed
          s2wire.S2pointer) // name of replacing commit
      returns (google.protobuf.Empty);
}
```

Merging Name-Sets: The reason to have a "name-to-set-of-names" map (instead of a "name-to-name" map) is concurrency: Unlike GitHub, we avoid making HEAD a serializing service (which would have to decide which branch becomes the latest master branch). For example, `append()`ing the same branch name to an AMlog is idempotent due to the use of a name set. Concurrent replacement requests with different removal sets can be done gracefully (i.e., execute a `replace()` even if a name in the removal set already has been removed). The standard configuration will be that a AMlog-reading client will see multiple branch names via the `getHead()` method and will have to merge them at application level, effectively replacing a set of names with a single name. If two merges are done in parallel, two branch names will emerge which the next client will have to merge, etc. Note that there is no `remove()` method because a base name always has to map to a non-empty set of names (initially the base name itself).

We have built our HEAD service with the semantics as explained above. Our hypothesis is that manipulating a set of names can be made in a distributed fashion without (global) serialization. Given this is true, the HEAD service can be implemented in a peer-to-peer fashion among brokers where the peers' current name sets are only exchanged periodically and added (via union) to the local name set e.g., using a gossip style protocol. Name replacement operations would *not* be exchanged i.e., applications at other peers will have to re-merge commits or remove superfluous names, before the next gossip round. Note that such a HEAD-maintained map (that temporarily associates a name with a set of names) is not a CRDT as it is not able nor supposed to merge the accumulated state by itself. An interesting reference point is Secure Scuttlebutt [33] where the eager and gossip-based replication automatically leads to the propagating of all log changes, hence provides an implicit implementation of the HEAD service *and* notification.

5 RESULTS (PROTOTYPE IMPLEMENTATION)

We have implemented WORM, HEAD, AMlog and ORmap in Python, all communication between parties (end device, edge/cloud server and HEAD+WORM service) is carried over Google RPC [8]. Two demo applications are Chat and a DropBox replacement (see [34] for the GitHub repo) which we describe in this section.

5.1 End device "boot record"

In both demo applications, end devices need a set of information in order to engage in collaborative work. We call this the "boot record" and detail here its content.

Beside connectivity details (because of Google RPC), the boot record contains the base name of a personal object called "root map": This map collects pointers into the various collaboration spaces a user is member of. In Symphony this corresponds to a tenant's "database" but in our database-less system, this is replaced by yet another map object used as top directory (for this employer) from which an end device can find the links to all applications that the tenant supports.

Instead of using plain self-certifying names as resource pointers we use "descriptors" that contain additional information like e.g. the credentials necessary to access to referenced objects.

Because objects are end-to-end encrypted, end-devices belonging to the same tenant (or, in a non-Symphony context, simply wanting to collaborate) must possess the shared secret in order to read the root map. This is another (per shared collaboration space) element of the boot record.

Finally, an end device must have access to its private and public key in order to identify itself and sign commits. Typically, the elements of the boot record are stored in a safe fashion on e.g. a smartphone. If end users trust a third party (e.g. employees their employer), one can also use "key manager servers" operated by the third party from which end devices can fetch the bits to boot into the distributed app.

5.2 A Chat Example

The purpose of the following code excerpts is to show the linked-data character of the abstract data types where several lookups in key-value store ADTs must be made, all mediated and potentially cached by an ICN. Note that unlike NDN there is no need for a namespace design as these object references (hash values) are internal to the ADT while the field names of application-level data structures are stored in ORmaps.

The first example shows the simplified code of a chat client for booting and listing all messages in the chat room "Forum" of the "icn2018" collaboration space and appending a note(we omit the use of keys via "hierarchical security contexts", a concept originally described in [27] and more recently in [12]):

```
bootRec = boot.getByUserName(myName)
rootMap = net.loadORmap(bootRec['collabSpaces'])
icn2018 = net.loadORmap(rootMap['icn2018'])
roomDir = net.loadORmap(icn2018['chat'])
forum   = net.loadAMlog(roomDir['Forum'])
for msg in forum.traverse():
    print(msg)
forum.append("hello ICN")
forum.refresh()
```

The `loadORmap()` and `loadAMlog()` methods create a local proxy object in Python and potentially fetch, via ICN, one or two items of the log or map DAG from the WORM service.

Updates to the local proxy object are not flushed immediately. Instead, an explicit `refresh()` is needed to trigger a writeback, possibly involving a merge operation of the AMlog and loading of the newest chat room state. Note that the `refresh()` silently returns in case of lost connectivity because CRDT updates can be played back at any (delayed) time. The consequence of a delayed writeback is that once it has happened, traversals of the DAG will have more leeway about where a chat message has to be inserted into the linearized log.

5.3 A DropBox Example

The boot phase for the DropBox example is identical as above, except that we fetch the "DropBox Object" which is a ORmap representing the root folder visible to all collaborators. We show here the simplified implementation of the "`ls`" command which prints the name and length of the referenced file:

```
bootRec = boot.getByUserName(myName)
rootMap = net.loadORmap(bootRec['collabSpaces'])
icn2018 = net.loadORmap(rootMap['icn2018'])
dropbox = net.loadORmap(icn2018['dropbox'])
for nm,direntry in dropbox.items():
    print("%-20s %d" % (nm, direntry['size']))
```

The example shows access to the size attribute of a directory entry. One other attribute is a pointer to a FLIC manifest in case of a file, or a pointer to another ORmap in case of a subdirectory.

In the chat as well as in the DropBox examples it seems that a *hierarchical name* would be ideal for accessing the final AMlog or ORmap object. Note, however, that there is no static name path (e.g., `/collabSpace/icn2018/dropbox`): Instead, each resolution step involves the retrieval of crypto key material and the lookup of a map's latest binding of a string to the corresponding self-certifying hash name. Like in UNIX, path resolution is the most expensive part of (file) data access.

6 STATUS AND FUTURE WORK

The current proof of concept focuses on supporting scalable abstract data types with encryption. We can run our demo applications in different configurations (serverless apps, where the ADT libraries are run on end devices and perform all CRDT merge operations, or in edge-assisted mode where the ADT library on the end device mostly proxies requests).

A major "non-ICN" element is currently the use of **Google RPC** (and underneath TCP which is used to connect an end device with the services in our system). We have started to implement an ICN-compatible RPC replacement for which we copied the 4-way handshake of HIP [15] and mapped it to the Interest/Data messaging pattern [35].

So far we only did functional tests. A next step will be to measure **performance**. The real competitor in the room is ChronoSync and more recently VectorSync [29] where we would like to understand whether the WORM+HEAD plus ADT design can scale better than a one-size-fits-all approach with its potentially heavy set reconciliation work. Another performance test should be carried out against the cloud where it will be interesting to put some services into **AWS lambda** [28] or Google cloud functions, although we expect some limitations due to the potentially long-lived streaming RPCs and the overall runtime limit of 300 seconds, for example for AWS lambdas.

In this paper we did not discuss the need to support **retention policies** where the append-only concept is modified in the sense that old data MUST be removed for legal reasons; 7 years is a typical number in this context. While the implementation of the ADTs with linked data structures can be extended to record a "no-content-beyond-this-point" condition, which means that all subgraphs from this point on have been trimmed, two problems must be addressed which are (a) the need to do a global snapshot for assessing consensus on the trimmed data structure and (b) the danger of dangling pointers where data structures outside an AMlog of ORmap still retain references to trimmed content. CRDTs may –again– provide a solution also to this problem [30] but this remains to be explored.

It is tempting to extend the HEAD service with **notification** capabilities in order to avoid polling and have shorter reaction times. However, we are hesitant to suggest such a path as it could lead to a potentially heavy-weight service like a full-blown Google pubsub replacement. It remains to be studies whether a non-reliable (sporadic) notification system, combined with a reliable queuing service, can support the implementation of reactive workflows sufficiently well.

7 CONCLUSIONS

Symphony's core application is end-to-end encrypted chat that observes special customer and legal requirements and runs in the cloud. ICN's concept of "secure the data, not the channel" matches this use case well, but ICN cannot be used directly to support it.

In this paper we argue that collaborative applications need higher-level APIs than the bare `READ(name)` method that stands for ICN's Interest/Data messaging protocol. Starting from adding `WRITE()`, we describe in this paper a service architecture that support scalable data types (based on conflict-free replicated data types) implemented as directed acyclic graphs, using self-certifying names as pointers. One such data structure is called AMlog – its implementation resembles an (encrypted) version of Git's data structure and IOTA's tangle; the other ADT is a replicated key-value map based on the "observed-remove set" construction. We also discuss the possibility and necessity of cloud-based assists, most basically the WORM and the novel HEAD service, which can be proxied in order to lighten the computation load on end-devices (and which is legally required in Symphony's use case).

ACKNOWLEDGMENTS

We would like to thank Michał Król, Dirk Kutscher and Ioannis Psaras for helpful discussions around "queues" in the context of ICN and Lixia Zhang for her help as shepherd of this paper.

REFERENCES

[1] A. Afanasyev, X. Jiang, Y. Yu, J. Tan, Y. Xia, A. Mankin, and L. Zhang. 2017. NDNS: A DNS-Like Name Service for NDN. In *26th International Conference on Computer Communication and Networks (ICCCN)*. 1–9.
[2] Gul Agha. 1985. A Message-Passing Paradigm for Object Management. *IEEE Database Eng. Bull.* 8, 4 (1985), 75–82.
[3] AWS. 2012. SQS Queues and SNS Notifications — Now Best Friends. https://aws.amazon.com/blogs/aws/queues-and-notifications-now-best-friends/
[4] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. *CoRR* (2012). arXiv:1210.3368
[5] Antonio Carzaniga. 1998. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* Ph.D. Dissertation. Politecnico di Milano, Italy.
[6] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 451–466.
[7] Google. 2008. Protocol Buffers. https://developers.google.com/protocol-buffers/
[8] Google. 2015-2018. GRPC – A high performance, open-source universal RPC framework. https://grpc.io/
[9] Google. 2018. Cloud Pub/Sub: A Google-Scale Messaging Service. https://cloud.google.com/pubsub/architecture
[10] Junio C. Hamano. 2006. GIT – A Stupid Content Tracker. In *2006 Linux Symposium, Volume One*. Ottawa, Canada, 385–394.
[11] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. 2009. CRDTs: Consistency without concurrency control. *CoRR* (2009). arXiv:0907.0929
[12] Claudio Marxer and Christian Tschudin. 2017. Schematized Access Control for Data Cubes and Trees. In *Proceedings of the 4th ACM Conference on Information-Centric Networking (ICN '17)*. 170–175.
[13] Ilya Moiseenko, Lijing Wang, and Lixia Zhang. 2015. Consumer/Producer Communication with Application Level Framing in Named Data Networking. In *Proc 2nd ACM Conference on Information-Centric Networking (ACM-ICN '15)*. 99–108.
[14] Marc Mosko and Ignacio Solis. 2015. *CCNx Semantics – draft-irtf-icnrg-ccnxsemantics-00.* Technical Report. Internet Research Task Force (IRTF).
[15] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson. 2015. *Host Identity Protocol Version 2 (HIPv2).* RFC 7401. Internet Engineering Task Force.
[16] L. Muscariello et al. 2017. Community ICN (CICN). https://wiki.fd.io/view/Cicn
[17] Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.
[18] NDN. 2018. Common Client Libraries (NDN-CCL) Documentation. https://named-data.net/codebase/platform/ndn-ccl/
[19] Maxwell Ogden, Karissa McKelvey, and Mathias Buus Madsen. 2017. Dat – Distributed Dataset Synchronization and Versioning. https://github.com/datproject/docs/blob/master/papers/dat-paper.pdf
[20] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 305–320.
[21] Serguei Popov. 2016. *The Tangle.* Technical Report. http://tanglereport.com/wp-content/uploads/2018/01/IOTA_Whitepaper.pdf
[22] Ioannis Psaras, Onur Ascigil, Sergi Rene, George Pavlou, Alex Afanasyev, and Lixia Zhang. 2018. Mobile Data Repositories at the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. Boston, MA.
[23] Quilt. 2016. Web site. https://quiltdata.com/
[24] Sean Quinlan and Sean Dorward. 2002. Venti: A New Approach to Archival Storage. In *Proceedings Conference on File and Storage Technologies (FAST-02)*.
[25] RIAK. 2014. Distributed Data Types — Riak 2.0. http://basho.com/posts/technical/distributed-data-types-riak-2-0/
[26] M. Rosenblum and J. K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52.
[27] R. S. Sandhu. 1987. On Some Cryptographic Solutions for Access Control in a Tree Hierarchy. In *Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow (ACM '87)*. 405–410.
[28] Amazon Web Services. 2015. AWS Serverless Multi-Tier Architectures Using Amazon API Gateway and AWS Lambda. https://d1.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf
[29] W. Shang, A. Afanasyev, and L. Zhang. 2018. *VectorSync: Distributed Dataset Synchronization over Named Data Networking.* NDN technical report. https://named-data.net/wp-content/uploads/2018/03/ndn-0056-1-vectorsync.pdf
[30] Marc Shapiro, Annette Bieniusa, Peter Zeller, and Gustavo Petri. 2018. Ensuring Referential Integrity Under Causal Consistency. In *Proceedings 5th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC'18)*. 1:1–1:5.
[31] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. LNCS 6976 (Jul 2011), 386–400.
[32] Sasu Tarkoma, Mark Ain, and Kari Visala. 2009. The Publish/Subscribe Internet Routing Paradigm (PSIRP): Designing the Future Internet Architecture. In *Towards the Future Internet - A European Research Perspective*. 102–111.
[33] Dominic Tarr et al. 2015. Scutlebutt web site. https://www.scuttlebutt.nz/
[34] Christian Tschudin. 2018. GitHub repo: Abstract Data Types over Information Centric Networking. https://github.com/cn-uofbasel/ADToverICN
[35] Christian Tschudin. March 2018. *"Re: RPC Publish method in NDN".* https://www.ietf.org/mail-archive/web/icnrg/current/msg02455.html
[36] Christian Tschudin and Christopher A Wood. 2017. *File-Like ICN Collection (FLIC) draft-icnrg-flic-00.* Technical Report. Internet Research Task Force (IRTF).
[37] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 66–73.
[38] Zhiyi Zhang, Yingdi Yu, Alexander Afanasyev, Jeff Burke, and Lixia Zhang. 2017. NAC: Name-based Access Control in Named Data Networking. In *Proceedings of the 4th ACM Conference on Information-Centric Networking (ICN '17)*. 186–187.
[39] Z. Zhu and A. Afanasyev. 2013. Let's ChronoSync: Decentralized dataset state synchronization in Named Data Networking. In *21st IEEE International Conference on Network Protocols (ICNP)*. 1–10.