Zanzibar: Google's Consistent, Global Authorization System

Ruoming Pang,[1] Ramón Cáceres,[1] Mike Burrows,[1] Zhifeng Chen,[1] Pratik Dave,[1]
Nathan Germer,[1] Alexander Golynski,[1] Kevin Graney,[1] Nina Kang,[1] Lea Kissner,[2*]
Jeffrey L. Korn,[1] Abhishek Parmar,[3*] Christina D. Richards,[1] Mengzhi Wang[1]
Google, LLC;[1] Humu, Inc.;[2] Carbon, Inc.[3]
{rpang,caceres}@google.com

Abstract

Determining whether online users are authorized to access digital objects is central to preserving privacy. This paper presents the design, implementation, and deployment of Zanzibar, a global system for storing and evaluating access control lists. Zanzibar provides a uniform data model and configuration language for expressing a wide range of access control policies from hundreds of client services at Google, including Calendar, Cloud, Drive, Maps, Photos, and YouTube. Its authorization decisions respect causal ordering of user actions and thus provide external consistency amid changes to access control lists and object contents. Zanzibar scales to trillions of access control lists and millions of authorization requests per second to support services used by billions of people. It has maintained 95th-percentile latency of less than 10 milliseconds and availability of greater than 99.999% over 3 years of production use.

1 Introduction

Many online interactions require authorization checks to confirm that a user has permission to carry out an operation on a digital object. For example, web-based photo storage services typically allow photo owners to share some photos with friends while keeping other photos private. Such a service must check whether a photo has been shared with a user before allowing that user to view the photo. Robust authorization checks are central to preserving online privacy.

This paper presents Zanzibar, a system for storing permissions and performing authorization checks based on the stored permissions. It is used by a wide array of services offered by Google, including Calendar, Cloud, Drive, Maps, Photos, and YouTube. Several of these services manage billions of objects on behalf of more than a billion users.

A unified authorization system offers important advantages over maintaining separate access control mechanisms for individual applications. First, it helps establish consistent semantics and user experience across applications. Second, it makes it easier for applications to interoperate, for example, to coordinate access control when an object from one application embeds an object from another application. Third, useful common infrastructure can be built on top of a unified access control system, in particular, a search index that respects access control and works across applications. Finally, as we show below, authorization poses unique challenges involving data consistency and scalability. It saves engineering resources to tackle them once across applications.

We have the following goals for the Zanzibar system:

- *Correctness*: It must ensure consistency of access control decisions to respect user intentions.
- *Flexibility*: It must support a rich set of access control policies as required by both consumer and enterprise applications.
- *Low latency*: It must respond quickly because authorization checks are often in the critical path of user interactions. Low latency at the tail is particularly important for serving search results, which often require tens to hundreds of checks.
- *High availability*: It must reliably respond to requests because, in the absence of explicit authorizations, client services would be forced to deny their users access.
- *Large scale*: It needs to protect billions of objects shared by billions of users. It must be deployed around the globe to be near its clients and their end users.

Zanzibar achieves these goals through a combination of notable features. To provide flexibility, Zanzibar pairs a simple data model with a powerful configuration language. The language allows clients to define arbitrary relations between users and objects, such as `owner`, `editor`, `commenter`, and `viewer`. It includes set-algebraic operators such as intersection and union for specifying potentially complex access control policies in terms of those user-object relations. For example, an application can specify that users granted editing rights on a document are also allowed to comment on the

*Work done while at Google.

document, but not all commenters are given editing rights.

At runtime, Zanzibar allows clients to create, modify, and evaluate access control lists (ACLs) through a remote procedure call (RPC) interface. A simple ACL takes the form of "user U has relation R to object O". More complex ACLs take the form of "set of users S has relation R to object O", where S is itself specified in terms of another object-relation pair. ACLs can thus refer to other ACLs, for example to specify that the set of users who can comment on a video consists of the users who have been granted viewing rights on that specific video along with those with viewing permissions on the video channel.

Group memberships are an important class of ACL where the object is a group and the relation is semantically equivalent to `member`. Groups can contain other groups, which illustrates one of the challenges facing Zanzibar, namely that evaluating whether a user belongs to a group can entail following a long chain of nested group memberships.

Authorization checks take the form of "does user U have relation R to object O?" and are evaluated by a collection of distributed servers. When a check request arrives to Zanzibar, the work to evaluate the check may fan out to multiple servers, for example when a group contains both individual members and other groups. Each of those servers may in turn contact other servers, for example to recursively traverse a hierarchy of group memberships.

Zanzibar operates at a global scale along multiple dimensions. It stores more than two trillion ACLs and performs millions of authorization checks per second. The ACL data does not lend itself to geographic partitioning because authorization checks for any object can come from anywhere in the world. Therefore, Zanzibar replicates all ACL data in tens of geographically distributed data centers and distributes load across thousands of servers around the world.

Zanzibar supports global consistency of access control decisions through two interrelated features. One, it respects the order in which ACL changes are committed to the underlying data store. Two, it can ensure that authorization checks are based on ACL data no older than a client-specified change. Thus, for example, a client can remove a user from a group and be assured that subsequent membership checks reflect that removal. Zanzibar provides these ordering properties by storing ACLs in a globally distributed database system with external consistency guarantees [15, 18].

Zanzibar employs an array of techniques to achieve low latency and high availability in this globally distributed environment. Its consistency protocol allows the vast majority of requests to be served with locally replicated data, without requiring cross-region round trips. Zanzibar stores its data in normalized forms for consistency. It handles hot spots on normalized data by caching final and intermediate results, and by deduplicating simultaneous requests. It also applies techniques such as hedging requests and optimizing computations on deeply nested sets with limited denormalization. Zanzibar responds to more than 95% of authorization checks within 10 milliseconds and has maintained more than 99.999% availability for the last 3 years.

The main contributions of this paper lie in conveying the engineering challenges in building and deploying a consistent, world-scale authorization system. While most elements of Zanzibar's design have their roots in previous research, this paper provides a record of the features and techniques Zanzibar brings together to satisfy its stringent requirements for correctness, flexibility, latency, availability, and scalability. The paper also highlights lessons learned from operating Zanzibar in service of a diverse set of demanding clients.

2 Model, Language, and API

This section describes Zanzibar's data model, configuration language, and application programming interface (API).

2.1 Relation Tuples

In Zanzibar, ACLs are collections of object-user or object-object relations represented as *relation tuples*. Groups are simply ACLs with membership semantics. Relation tuples have efficient binary encodings, but in this paper we represent them using a convenient text notation:

$\langle tuple \rangle$ ::= $\langle object \rangle$ '`#`' $\langle relation \rangle$ '`@`' $\langle user \rangle$

$\langle object \rangle$ ::= $\langle namespace \rangle$ '`:`' $\langle object\_id \rangle$

$\langle user \rangle$ ::= $\langle user\_id \rangle$ | $\langle userset \rangle$

$\langle userset \rangle$ ::= $\langle object \rangle$ '`#`' $\langle relation \rangle$

where $\langle namespace \rangle$ and $\langle relation \rangle$ are predefined in client configurations (§2.3), $\langle object\_id \rangle$ is a string, and $\langle user\_id \rangle$ is an integer. The primary keys required to identify a relation tuple are $\langle namespace \rangle$, $\langle object\_id \rangle$, $\langle relation \rangle$, and $\langle user \rangle$. One feature worth noting is that a $\langle userset \rangle$ allows ACLs to refer to groups and thus supports representing nested group membership.

Table 1 shows some example tuples and corresponding semantics. While some relations (e.g. `viewer`) define access control directly, others (e.g. `parent`, pointing to a folder) only define abstract relations between objects. These abstract relations may indirectly affect access control given userset rewrite rules specified in namespace configs (§2.3.1).

Defining our data model around tuples, instead of per-object ACLs, allows us to unify the concepts of ACLs and groups and to support efficient reads and incremental updates, as we will see in §2.4.

2.2 Consistency Model

ACL checks must respect the order in which users modify ACLs and object contents to avoid unexpected sharing behaviors. Specifically, our clients care about preventing the

Example Tuple in Text Notation	Semantics
`doc:readme#owner@10`	User 10 is an owner of `doc:readme`
`group:eng#member@11`	User 11 is a member of `group:eng`
`doc:readme#viewer@group:eng#member`	Members of `group:eng` are viewers of `doc:readme`
`doc:readme#parent@folder:A#...`	`doc:readme` is in `folder:A`

Table 1: Example relation tuples. "#..." represents a relation that does not affect the semantics of the tuple.

"new enemy" problem, which can arise when we fail to respect the ordering between ACL updates or when we apply old ACLs to new content. Consider these two examples:

Example A: Neglecting ACL update order

1. Alice removes Bob from the ACL of a folder;
2. Alice then asks Charlie to move new documents to the folder, where document ACLs inherit from folder ACLs;
3. Bob should not be able to see the new documents, but may do so if the ACL check neglects the ordering between the two ACL changes.

Example B: Misapplying old ACL to new content

1. Alice removes Bob from the ACL of a document;
2. Alice then asks Charlie to add new contents to the document;
3. Bob should not be able to see the new contents, but may do so if the ACL check is evaluated with a stale ACL from before Bob's removal.

Preventing the "new enemy" problem requires Zanzibar to understand and respect the causal ordering between ACL or content updates, including updates on different ACLs or objects and those coordinated via channels invisible to Zanzibar. Hence Zanzibar must provide two key consistency properties: *external consistency* [18] and *snapshot reads with bounded staleness*.

External consistency allows Zanzibar to assign a timestamp to each ACL or content update, such that two causally related updates $x \prec y$ will be assigned timestamps that reflect the causal order: $T_x < T_y$. With causally meaningful timestamps, a snapshot read of the ACL database at timestamp T, which observes all updates with timestamps $\leq T$, will respect ordering between ACL updates. That is, if the read observes an update x, it will observe all updates that happen causally before x.

Furthermore, to avoid applying old ACLs to new contents, the ACL check evaluation snapshot must not be staler than the causal timestamp assigned to the content update. Given a content update at timestamp T_c, a snapshot read at timestamp $\geq T_c$ ensures that all ACL updates that happen causally before the content update will be observed by the ACL check.

To provide external consistency and snapshot reads with bounded staleness, we store ACLs in the Spanner global database system [15]. Spanner's TrueTime mechanism assigns each ACL write a microsecond-resolution timestamp, such that the timestamps of writes reflect the causal ordering between writes, and thereby provide external consistency. We evaluate each ACL check at a single snapshot timestamp across multiple database reads, so that all writes with timestamps up to the check snapshot, and only those writes, are visible to the ACL check.

To avoid evaluating checks for new contents using stale ACLs, one could try to always evaluate at the latest snapshot such that the check result reflects all ACL writes up to the check call. However, such evaluation would require global data synchronization with high-latency round trips and limited availability. Instead, we design the following protocol to allow most checks to be evaluated on already replicated data with cooperation from Zanzibar clients:

1. A Zanzibar client requests an opaque consistency token called a *zookie* for each content version, via a *content-change* ACL check (§2.4.4) when the content modification is about to be saved. Zanzibar encodes a current global timestamp in the zookie and ensures that all prior ACL writes have lower timestamps. The client stores the zookie with the content change in an atomic write to the client storage. Note that the content-change check does *not* need to be evaluated in the same transaction as the application content modification, but only has to be triggered when the user modifies the contents.
2. The client sends this zookie in subsequent ACL check requests to ensure that the check snapshot is at least as fresh as the timestamp for the content version.

External consistency and snapshot reads with staleness bounded by zookie prevent the "new enemy" problem. In Example A, ACL updates $A1$ and $A2$ will be assigned timestamps $T_{A1} < T_{A2}$, respectively. Bob will not be able to see the new documents added by Charlie: if a check is evaluated at $T < T_{A2}$, the document ACLs will not include the folder ACL; if a check is evaluated at $T \geq T_{A2} > T_{A1}$, the check will observe update $A1$, which removed Bob from the

folder ACL. In Example B, Bob will not see the new contents added to the document. For Bob to see the new contents, the check must be evaluated with a zookie $\geq T_{B2}$, the timestamp assigned to the content update. Because $T_{B2} > T_{B1}$, such a check will also observe the ACL update $B1$, which removed Bob from the ACL.

The zookie protocol is a key feature of Zanzibar's consistency model. It ensures that Zanzibar respects causal ordering between ACL and content updates, but otherwise grants Zanzibar freedom to choose evaluation timestamps so as to meet its latency and availability goals. The freedom arises from the protocol's at-least-as-fresh semantics, which allow Zanzibar to choose any timestamp fresher than the one encoded in a zookie. Such freedom in turn allows Zanzibar to serve most checks at a default staleness with already replicated data (§3.2.1) and to quantize evaluation timestamps to avoid hot spots (§3.2.5).

2.3 Namespace Configuration

Before clients can store relation tuples in Zanzibar, they must configure their namespaces. A namespace configuration specifies its relations as well as its storage parameters. Each relation has a name, which is a client-defined string such as `viewer` or `editor`, and a relation config. Storage parameters include sharding settings and an encoding for object IDs that helps Zanzibar optimize storage of integer, string, and other object ID formats.

2.3.1 Relation Configs and Userset Rewrites

While relation tuples reflect relationships between objects and users, they do not completely define the effective ACLs. For example, some clients specify that users with `editor` permissions on each object should have `viewer` permission on the same object. While such relationships between relations can be represented by a relation tuple per object, storing a tuple for each object in a namespace would be wasteful and make it hard to make modifications across all objects. Instead, we let clients define object-agnostic relationships via *userset rewrite rules* in relation configs. Figure 1 demonstrates a simple namespace configuration with concentric relations, where `viewer` contains `editor`, and `editor` contains `owner`.

Userset rewrite rules are defined per relation in a namespace. Each rule specifies a function that takes an object ID as input and outputs a userset expression tree. Each leaf node of the tree can be any of the following:

- `_this`: Returns all users from stored relation tuples for the ⟨*object#relation*⟩ pair, including indirect ACLs referenced by usersets from the tuples. This is the default behavior when no rewrite rule is specified.
- `computed_userset`: Computes, for the input object, a new userset. For example, this allows the userset expression for a `viewer` relation to refer to the `editor` userset on the same object, thus offering an ACL inheritance capability between relations.
- `tuple_to_userset`: Computes a tupleset (§2.4.1) from the input object, fetches relation tuples matching the tupleset, and computes a userset from every fetched relation tuple. This flexible primitive allows our clients to express complex policies such as "look up the parent folder of the document and inherit its viewers".

```
name: "doc"

relation { name: "owner" }

relation {
 name: "editor"
 userset_rewrite {
  union {
   child { _this {} }
   child { computed_userset { relation: "owner" } }
} } }

relation {
 name: "viewer"
 userset_rewrite {
  union {
   child { _this {} }
   child { computed_userset { relation: "editor" } }
   child { tuple_to_userset {
     tupleset { relation: "parent" }
     computed_userset {
       object: $TUPLE_USERSET_OBJECT  # parent folder
       relation: "viewer"
   } } }
} } }
```

Figure 1: Simple namespace configuration with concentric relations on documents. All owners are editors, and all editors are viewers. Further, viewers of the parent folder are also viewers of the document.

A userset expression can also be composed of multiple sub-expressions, combined by operations such as union, intersection, and exclusion.

2.4 API

In addition to supporting ACL checks, Zanzibar also provides APIs for clients to read and write relation tuples, watch tuple updates, and inspect the effective ACLs.

A concept used throughout these API methods is that of a *zookie*. A zookie is an opaque byte sequence encoding a globally meaningful timestamp that reflects an ACL write, a client content version, or a read snapshot. Zookies in ACL read and check requests specify staleness bounds for snapshot reads, thus providing one of Zanzibar's core consistency properties. We choose to use an opaque cookie instead of the actual timestamp to discourage our clients from choosing arbitrary timestamps and to allow future extensions.

2.4.1 Read

Our clients read relation tuples to display ACLs or group membership to users, or to prepare for a subsequent write. A read request specifies one or multiple *tuplesets* and an optional zookie.

Each *tupleset* specifies keys of a set of relation tuples. The set can include a single tuple key, or all tuples with a given object ID or userset in a namespace, optionally constrained by a relation name. With the tuplesets, clients can look up a specific membership entry, read all entries in an ACL or group, or look up all groups with a given user as a direct member. All tuplesets in a read request are processed at a single snapshot.

With the zookie, clients can request a read snapshot no earlier than a previous write if the zookie from the write response is given in the read request, or at the same snapshot as a previous read if the zookie from the earlier read response is given in the subsequent request. If the request doesn't contain a zookie, Zanzibar will choose a reasonably recent snapshot, possibly offering a lower-latency response than if a zookie were provided.

Read results only depend on contents of relation tuples and do not reflect userset rewrite rules. For example, even if the `viewer` userset always includes the `owner` userset, reading tuples with the `viewer` relation will not return tuples with the `owner` relation. Clients that need to understand the effective userset can use the Expand API (§2.4.5).

2.4.2 Write

Clients may modify a single relation tuple to add or remove an ACL. They may also modify all tuples related to an object via a read-modify-write process with optimistic concurrency control [21] that uses a read RPC followed by a write RPC:

1. Read all relation tuples of an object, including a per-object "lock" tuple.
2. Generate the tuples to write or delete. Send the writes, along with a touch on the lock tuple, to Zanzibar, with the condition that the writes will be committed only if the lock tuple has not been modified since the read.
3. If the write condition is not met, go back to step 1.

The lock tuple is just a regular relation tuple used by clients to detect write races.

2.4.3 Watch

Some clients maintain secondary indices of relation tuples in Zanzibar. They can do so with our Watch API. A watch request specifies one or more namespaces and a zookie representing the time to start watching. A watch response contains all tuple modification events in ascending timestamp order, from the requested start timestamp to a timestamp encoded in a *heartbeat zookie* included in the watch response. The client can use the heartbeat zookie to resume watching where the previous watch response left off.

2.4.4 Check

A check request specifies a userset, represented by $\langle$*object#relation*$\rangle$, a putative user, often represented by an authentication token, and a zookie corresponding to the desired object version. Like reads, a check is always evaluated at a consistent snapshot no earlier than the given zookie.

To authorize application content modifications, our clients send a special type of check request, a *content-change* check. A content-change check request does not carry a zookie and is evaluated at the latest snapshot. If a content change is authorized, the check response includes a zookie for clients to store along with object contents and use for subsequent checks of the content version. The zookie encodes the evaluation snapshot and captures any possible causality from ACL changes to content changes, because the zookie's timestamp will be greater than that of the ACL updates that protect the new content (§2.2).

2.4.5 Expand

The Expand API returns the effective userset given an $\langle$*object#relation*$\rangle$ pair and an optional zookie. Unlike the Read API, Expand follows indirect references expressed through userset rewrite rules. The result is represented by a *userset tree* whose leaf nodes are user IDs or usersets pointing to other $\langle$*object#relation*$\rangle$ pairs, and intermediate nodes represent union, intersection, or exclusion operators. Expand is crucial for our clients to reason about the complete set of users and groups that have access to their objects, which allows them to build efficient search indices for access-controlled content.

3 Architecture and Implementation

Figure 2 shows the architecture of the Zanzibar system. `aclservers` are the main server type. They are organized in clusters and respond to Check, Read, Expand, and Write requests. Requests arrive at any server in a cluster and that server fans out the work to other servers in the cluster as necessary. Those servers may in turn contact other servers to compute intermediate results. The initial server gathers the final result and returns it to the client.

Zanzibar stores ACLs and their metadata in Spanner databases. There is one database to store relation tuples for each client namespace, one database to hold all namespace configurations, and one changelog database shared across all namespaces. `aclservers` read and write those databases in the course of responding to client requests.

`watchservers` are a specialized server type that respond to Watch requests. They tail the changelog and serve a stream of namespace changes to clients in near real time.

Zanzibar periodically runs a data processing pipeline to perform a variety of offline functions across all Zanzibar data in Spanner. One such function is to produce dumps of the relation tuples in each namespace at a known snapshot times-

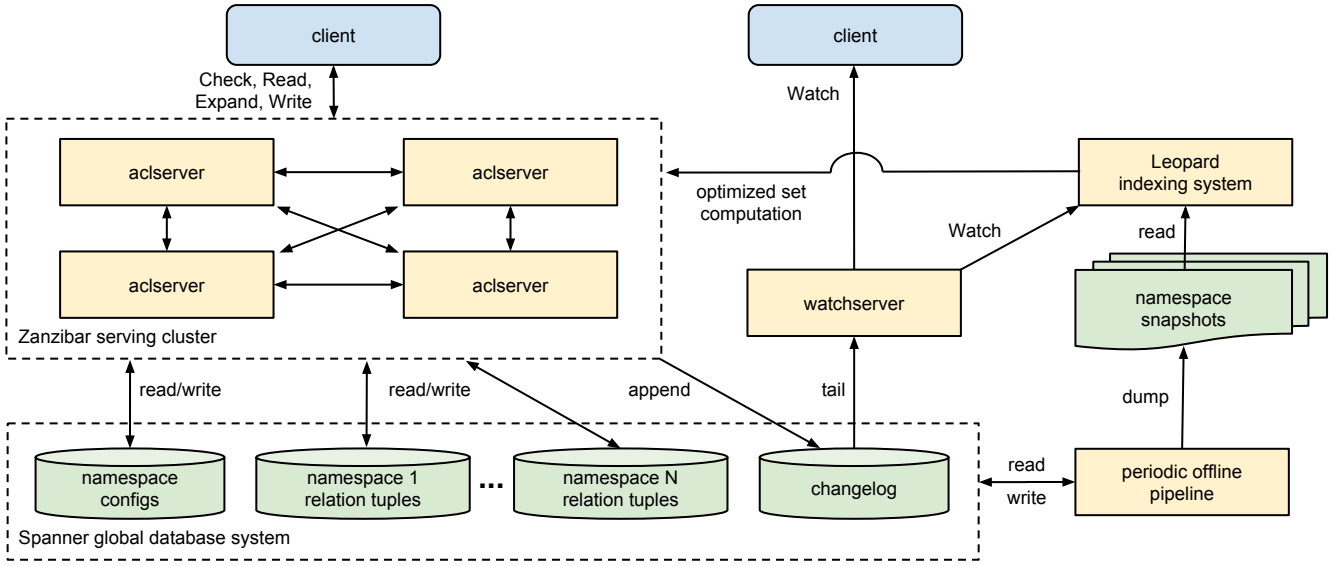

Figure 2: Zanzibar architecture. Arrows indicate the direction of data flow.

tamp. Another is to garbage-collect tuple versions older than a threshold configured per namespace.

Leopard is an indexing system used to optimize operations on large and deeply nested sets. It reads periodic snapshots of ACL data and watches for changes between snapshots. It performs transformations on that data, such as denormalization, and responds to requests from `aclservers`.

The rest of this section presents the implementation of these architectural elements in more detail.

3.1 Storage

3.1.1 Relation Tuple Storage

We store relation tuples of each namespace in a separate database, where each row is identified by primary key *(shard ID, object ID, relation, user, commit timestamp)*. Multiple tuple versions are stored on different rows, so that we can evaluate checks and reads at any timestamp within the garbage collection window. The ordering of primary keys allows us to look up all relation tuples for a given object ID or *(object ID, relation)* pair.

Our clients configure sharding of a namespace according to its data pattern. Usually the shard ID is determined solely by the object ID. In some cases, for example, when a namespace stores groups with very large numbers of members, the shard ID is computed from both object ID and user.

3.1.2 Changelog

Zanzibar also maintains a changelog database that stores a history of tuple updates for the Watch API. The primary keys are *(changelog shard ID, timestamp, unique update ID)*, where a changelog shard is randomly selected for each write.

Every Zanzibar write is committed to both the tuple storage and the changelog shard in a single transaction. We designate the Spanner server hosting the changelog shard as the transaction coordinator to minimize blocking of changelog reads on pending transactions.

3.1.3 Namespace Config Storage

Namespace configs are stored in a database with two tables. One table contains the configs and is keyed by namespace IDs. The other is a changelog of config updates and is keyed by commit timestamps. This structure allows a Zanzibar server to load all configs upon startup and monitor the changelog to refresh configs continuously.

3.1.4 Replication

To reduce latency, Zanzibar data is replicated to be close to our clients. Replicas exist in dozens of locations around the world, with multiple replicas per region. The 5 voting replicas are in eastern and central United States, in 3 different metropolitan areas to isolate failures but within 25 milliseconds of each other so that Paxos transactions commit quickly.

3.2 Serving

3.2.1 Evaluation Timestamp

As noted in §2.4, clients can provide zookies to ensure a minimum snapshot timestamp for request evaluation. When a zookie is not provided, the server uses a default staleness chosen to ensure that all transactions are evaluated at a timestamp that is as recent as possible without impacting latency.

On each read request it makes to Spanner, Zanzibar receives a hint about whether or not the data at that timestamp required an out-of-zone read and thus incurred additional latency. Each server tracks the frequency of such out-of-zone reads for data at a default staleness as well as for fresher

and staler data, and uses these frequencies to compute a binomial proportion confidence interval of the probability that any given piece of data is available locally at each staleness.

Upon collecting enough data, the server checks to see if each staleness value has a sufficiently low probability of incurring an out-of-zone read, and thus will be low-latency. If so, it updates the default staleness bound to the lowest "safe" value. If no known staleness values are safe, we use a two-proportion z-test to see if increasing the default will be a statistically significant amount safer. In that case, we increase the default value in the hopes of improving latency. This default staleness mechanism is purely a performance optimization. It does not violate consistency semantics because Zanzibar always respects zookies when provided.

3.2.2 Config Consistency

Because changes to namespace configs can change the results of ACL evaluations, and therefore their correctness, Zanzibar chooses a single snapshot timestamp for config metadata when evaluating each client request. All `aclservers` in a cluster use that same timestamp for the same request, including for any subrequests that fan out from the original client request.

Each server independently loads namespace configs from storage continuously as they change (§3.1.3). Therefore, each server in a cluster may have access to a different range of config timestamps due to restarts or network latency. Zanzibar must pick a timestamp that is available across all of them. To facilitate this, a monitoring job tracks the timestamp range available to every server and aggregates them, reporting a globally available range to every other server. On each incoming request the server picks a time from this range, ensuring that all servers can continue serving even if they are no longer able to read from the config storage.

3.2.3 Check Evaluation

Zanzibar evaluates ACL checks by converting check requests to boolean expressions. In a simple case, when there are no userset rewrite rules, checking a user U against a userset $\langle object\#relation \rangle$ can be expressed as

$$\begin{aligned} &\text{CHECK}(U, \langle object\#relation \rangle) = \\ &\quad \exists \text{ tuple } \langle object\#relation@U \rangle \\ &\quad \vee \exists \text{ tuple } \langle object\#relation@U' \rangle, \text{ where} \\ &\qquad U' = \langle object'\#relation' \rangle \text{ s.t. } \text{CHECK}(U, U'). \end{aligned}$$

Finding a valid $U' = \langle object'\#relation' \rangle$ involves evaluating membership on all indirect ACLs or groups, recursively. This kind of "pointer chasing" works well for most types of ACLs and groups, but can be expensive when indirect ACLs or groups are deep or wide. §3.2.4 explains how we handle this problem. Userset rewrite rules are also translated to boolean expressions as part of check evaluation.

To minimize check latency, we evaluate all leaf nodes of the boolean expression tree concurrently. When the outcome of one node determines the result of a subtree, evaluation of other nodes in the subtree is cancelled.

Evaluation of leaf nodes usually involves reading relation tuples from databases. We apply a pooling mechanism to group reads for the same ACL check to minimize the number of read RPCs to Spanner.

3.2.4 Leopard Indexing System

Recursive pointer chasing during check evaluation has difficulty maintaining low latency with groups that are deeply nested or have a large number of child groups. For selected namespaces that exhibit such structure, Zanzibar handles checks using Leopard, a specialized index that supports efficient set computation.

A Leopard index represents a collection of named sets using (T, s, e) tuples, where T is an enum representing the set type and s and e are 64-bit integers representing the set ID and the element ID, respectively. A query evaluates an expression of union, intersection, or exclusion of named sets and returns the result set ordered by the element ID up to a specified number of results.

To index and evaluate group membership, Zanzibar represents group membership with two set types, GROUP2GROUP and MEMBER2GROUP, which we show here as functions mapping from a set ID to element IDs:

- $\text{GROUP2GROUP}(s) \rightarrow \{e\}$, where s represents an ancestor group and e represents a descendent group that is *directly or indirectly* a sub-group of the ancestor group.
- $\text{MEMBER2GROUP}(s) \rightarrow \{e\}$, where s represents an individual user and e represents a parent group in which the user is a *direct* member.

To evaluate whether user U is a member of group G, we check whether

$$(\text{MEMBER2GROUP}(U) \cap \text{GROUP2GROUP}(G)) \neq \emptyset$$

Group membership can be considered as a reachability problem in a graph, where nodes represent groups and users and edges represent direct membership. Flattening group-to-group paths allows reachability to be efficently evaluated by Leopard, though other types of denormalization can also be applied as data patterns demand.

The Leopard system consists of three discrete parts: a serving system capable of consistent and low-latency operations across sets; an offline, periodic index building system; and an online real-time layer capable of continuously updating the serving system as tuple changes occur.

Index tuples are stored as ordered lists of integers in a structure such as a skip list, thus allowing for efficient union and intersections among sets. For example, evaluating the intersection between two sets, A and B, requires only $O(\min(|A|, |B|))$ skip-list seeks. The index is sharded by element IDs and can be distributed across multiple servers. Shards are usually served entirely from memory, but they

can also be served from a mix of hot and cold data spread between memory and remote solid-state devices.

The offline index builder generates index shards from a snapshot of Zanzibar relation tuples and configs, and replicates the shards globally. It respects userset rewrite rules and recursively expands edges in an ACL graph to form Leopard index tuples. The Leopard servers continously watch for new shards and swap old shards with new ones when they become available.

The Leopard system described thus far is able to efficiently evaluate deeply and widely nested group membership, but cannot do so at a fresh and consistent snapshot due to offline index generation and shard swapping. To support consistent ACL evaluation, Leopard servers maintain an incremental layer that indexes all updates since the offline snapshot, where each update is represented by a (T, s, e, t, d) tuple, where t is the timestamp of the update and d is a deletion marker. Updates with timestamps less than or equal to the query timestamp are merged on top of the offline index during query processing.

To maintain the incremental layer, the Leopard incremental indexer calls Zanzibar's Watch API to receive a temporally ordered stream of Zanzibar tuple modifications and transforms the updates into a temporally ordered stream of Leopard tuple additions, updates, and deletions. Generating updates for the GROUP2GROUP tuples requires the incremental indexer to maintain group-to-group membership for denormalizing the effects of a relation tuple update to potentially multiple index updates.

In practice, a single Zanzibar tuple addition or deletion may yield potentially tens of thousands of discrete Leopard tuple events. Each Leopard serving instance receives the complete stream of these Zanzibar tuple changes through the Watch API. The Leopard serving system is designed to continuously ingest this stream and update its various posting lists with minimal impact to query serving.

3.2.5 Handling Hot Spots

The workload of ACL reads and checks is often bursty and subject to hot spots. For example, answering a search query requires conducting ACL checks for all candidate results, whose ACLs often share common groups or indirect ACLs. To facilitate consistency, Zanzibar avoids storage denormalization and relies only on normalized data (except for the cases described in §3.2.4). With normalized data, hot spots on common ACLs (e.g., popular groups) may overload the underlying database servers. We found the handling of hot spots to be the most critical frontier in our pursuit of low latency and high availability.

Zanzibar servers in each cluster form a distributed cache for both reads and check evaluations, including intermediate check results evaluated during pointer chasing. Cache entries are distributed across Zanzibar servers with consistent hashing [20]. To process checks or reads, we fan out requests to the corresponding Zanzibar servers via an internal RPC interface. To minimize the number of internal RPCs, for most namespaces we compute the forwarding key from the object ID, since processing a check on $\langle object\#relation \rangle$ often involves indirect ACL checks on other relations of the same object and reading relation tuples of the object. These checks and reads can be processed by the same server since they share the same forwarding key with the parent check request. To handle hot forwarding keys, we cache results at both the caller and the callee of internal RPCs, effectively forming cache trees. We also use Slicer [12] to help distribute hot keys to multiple servers.

We avoid reusing results evaluated from a different snapshot by encoding snapshot timestamps in cache keys. We choose evaluation timestamps rounded up to a coarse granularity, such as one or ten seconds, while respecting staleness constraints from request zookies. This timestamp quantization allows the vast majority of recent checks and reads to be evaluated at the same timestamps and to share cache results, despite having microsecond-resolution timestamps in cache keys. It is worth noting that rounding up timestamps does not affect Zanzibar's consistency properties, since Spanner ensures that a snapshot read at timestamp T will observe all writes up to T—this holds even if T is in the future, in which case the read will wait until TrueTime has moved past T.

To handle the "cache stampede" problem [3], where concurrent requests create flash hot spots before the cache is populated with results, we maintain a *lock table* on each server to track outstanding reads and checks. Among requests sharing the same cache key only one request will begin processing; the rest block until the cache is populated.

We can effectively handle the vast majority of hot spots with distributed caches and lock tables. Over time we made the following two improvements.

First, direct membership checks of a user for an object and relation (i.e. $\langle object\#relation@user \rangle$) are usually handled by a single relation tuple lookup. However, occasionally a very popular object invites many concurrent checks for different users, causing a hot spot on the storage server hosting relation tuples for the object. To avoid these hot spots, we read and cache *all* relation tuples of $\langle object\#relation \rangle$ for the hot object, trading read bandwidth for cacheability. We dynamically detect hot objects to apply this method to by tracking the number of outstanding reads on each object.

Second, indirect ACL checks are frequently cancelled when the result of the parent ACL check is already determined. This leaves the cache key unpopulated. While eager cancellation reduces resource usage significantly, it negatively affects latency of concurrent requests that are blocked by the lock table entry. To prevent this latency impact, we delay eager cancellation when there are waiters on the corresponding lock table entry.

3.2.6 Performance Isolation

Performance isolation is indispensable for shared services targeting low latency and high availability. If Zanzibar or one of its clients occasionally fails to provision enough resources to handle an unexpected usage pattern, the following isolation mechanisms ensure that performance problems are isolated to the problematic use case and do not adversely affect other clients.

First, to ensure proper allocation of CPU capacity, Zanzibar measures the cost of each RPC in terms of generic *cpu-seconds*, a hardware-agnostic metric. Each client has a global limit on maximum CPU usage per second; its RPCs will be throttled if it exceeds the limit *and* there is no spare capacity in the overall system.

Each Zanzibar server also limits the total number of outstanding RPCs to control its memory usage. Likewise it limits the number of oustanding RPCs per client.

Zanzibar further limits the maximum number of concurrent reads per *(object, client)* and per client on each Spanner server. This ensures that no single object or client can monopolize a Spanner server.

Finally, we use different lock table keys for requests from different clients to prevent any throttling that Spanner applies to one client from affecting other clients.

3.2.7 Tail Latency Mitigation

Zanzibar's distributed processing requires measures to accommodate slow tasks. For calls to Spanner and to the Leopard index we rely on request hedging [16] (i.e. we send the same request to multiple servers, use whichever response comes back first, and cancel the other requests). To reduce round-trip times, we try to place at least two replicas of these backend services in every geographical region where we have Zanzibar servers. To avoid unnecessarily multiplying load, we first send one request and defer sending hedged requests until the initial request is known to be slow.

To determine the appropriate hedging delay threshold, each server maintains a delay estimator that dynamically computes an Nth percentile latency based on recent measurements. This mechanism allows us to limit the additional traffic incurred by hedging to a small fraction of total traffic.

Effective hedging requires the requests to have similar costs. In the case of Zanzibar's authorization checks, some checks are inherently more time-consuming than others because they require more work. Hedging check requests would result in duplicating the most expensive workloads and, ironically, worsening latency. Therefore we do not hedge requests between Zanzibar servers, but rely on the previously discussed sharding among multiple replicas and on monitoring mechanisms to detect and avoid slow servers.

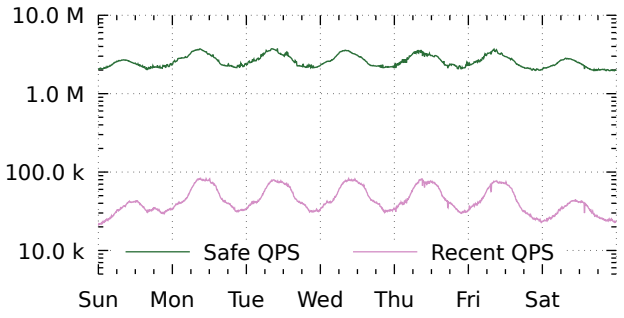

Figure 3: Rate of Check Safe and Check Recent requests over a 7-day period in December 2018.

4 Experience

Zanzibar has been in production use for more than 5 years. Throughout that time, the number of clients using Zanzibar and the load they place on Zanzibar have grown steadily. This section discusses our experience operating Zanzibar as a globally distributed authorization system.

Zanzibar manages more than 1,500 namespaces defined by hundreds of client applications. The size of a namespace configuration file serves as a rough measure of the complexity of the access control policy implemented by that namespace. These configuration files range from tens of lines to thousands of lines, with the median near 500 lines.

These namespaces contain more than 2 trillion relation tuples that occupy close to 100 terabytes. The number of tuples per namespace ranges over many orders of magnitude, from tens to a trillion, with the median near 15,000. This data is fully replicated in more than 30 locations around the world to maintain both proximity to users and high availability.

Zanzibar serves more than 10 million client queries per second (QPS). Over a sample 7-day period in December 2018, Check requests peak at roughly 4.2M QPS, Read at 8.2M, Expand at 760K, and Write at 25K. Queries that read data are thus two orders of magnitude more frequent than those that write data.

Zanzibar distributes this load across more than 10,000 servers organized in several dozen clusters around the world. The number of servers per cluster ranges from fewer than 100 to more than 1,000, with the median near 500. Clusters are sized in proportion to load in their geographic regions.

4.1 Requests

We divide requests into two categories according to the required data freshness, which can have a large impact on latency and availability of the requests. Specifically, Check, Read, and Expand requests carry zookies to specify lower bounds on evaluation timestamps. When a zookie timestamp is higher than that of the most recent data replicated to the region, the storage reads require cross-region round trips to the leader replica to retrieve fresher data. As our storage

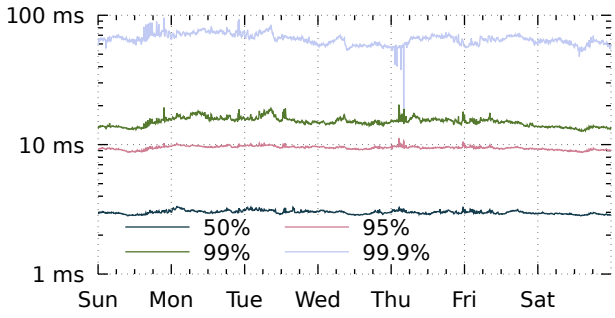

Figure 4: Latency of Check Safe responses at different percentiles over a 7-day period in December 2018.

is configured with replication heartbeats with 8-second intervals, we divide our requests into two categories: *Safe* requests have zookies more than 10 seconds old and can be served within the region most of time, while *Recent* requests have zookies less than 10 seconds old and often require inter-region round trips. We report separate statistics for each.

Figure 3 shows the rate of Check Safe and Check Recent requests over 7 days. Both exhibit a diurnal cycle. The rate of Safe requests is about two orders of magnitude larger than that of Recent requests, which allows Zanzibar to serve the vast majority of ACL checks locally.

4.2 Latency

Zanzibar's latency budget is generally a small fraction of the few hundreds of milliseconds of total response time that its clients must provide to be viable interactive services. Consider for example a client that performs authorization checks on multiple documents before it can show the results of a search on those documents.

We measure latency on the server side using live traffic because (1) latency is heavily influenced by our caching and de-duplication mechanisms so that it is only realistically reflected by live traffic, and (2) accurately measuring latency from clients requires well-behaving clients. Provisioning of client jobs is outside of Zanzibar's control and sometimes client jobs are overloaded.

Figure 4 shows the latency of Check Safe responses over 7 days. At the 50th, 95th, 99th, and 99.9th percentiles it peaks at roughly 3, 11, 20, and 93 msec, respectively. This performance meets our latency goals for an operation that is frequently in the critical path of user interactions.

Table 2 summarizes the latency distributions of Check, Read, Expand, and Write responses over the same 7 days. As intended, the more frequently used Safe versions of Check, Read, and Expand are significantly faster than the less frequently used Recent versions. Writes are the least frequently used of all the APIs, and the slowest because they always require distributed coordination among Spanner servers.

		Latency in milliseconds, μ (σ)		
	API	**50%ile**	**95%ile**	**99%ile**
Safe	Check	3.0 (0.091)	9.46 (0.3)	15.0 (1.19)
	Read	2.18 (0.031)	3.71 (0.094)	8.03 (3.28)
	Expand	4.27 (0.313)	8.84 (0.586)	34.1 (4.35)
Recent	Check	2.86 (0.087)	60.0 (2.1)	76.3 (2.59)
	Read	2.21 (0.054)	40.1 (2.03)	86.2 (3.84)
	Expand	5.79 (0.224)	45.6 (3.44)	121.0 (2.38)
	Write	127.0 (3.65)	233.0 (23.0)	401.0 (133.0)

Table 2: Mean and standard deviation of RPC response latency over a 7-day period in December 2018.

4.3 Availability

We define availability as the fraction of "qualified" RPCs the service answers successfully within latency thresholds: 5 seconds for a Safe request, and 15 seconds for a Recent request as leader re-election in Spanner may take up to 10 seconds. For an RPC to be qualified, the request must be well-formed and have a deadline longer than the latency threshold. In addition, the client must stay within its resource quota.

For these reasons, we cannot measure availability directly with live traffic, as our clients sometimes send RPCs with short deadlines or cancel their in-progress RPCs. Instead, we sample a small fraction of valid requests from live traffic and replay them later with our own probers. When replaying the requests, we set the timeout to be longer than the availability threshold. We also adjust the request zookie, if one is specified, so that the relative age of the zookie remains the same as when the request was received in the live traffic. Finally, we run 3 probers per cluster and exclude outliers to eliminate false alarms caused by rare prober failures.

To compute availability, we aggregate success ratios over 90-day windows averaged across clusters. Figure 5 shows Zanzibar's availability as measured by these probers. Availability has remained above 99.999% over the past 3 years of operation at Google. In other words, for every quarter, Zanzibar has less than 2 minutes of global downtime and fewer than 13 minutes when the global error ratio exceeds 10%.

4.4 Internals

Zanzibar servers delegate checks and reads to each other based on consistent hashing, and both the caller and the callee sides of the delegated operations cache the results to prevent hot spots (§3.2.5). At peak, Zanzibar handles 22 million internal "delegated" RPCs per second, split about evenly between reads and checks. In-memory caching handles approximately 200 million lookups per second at peak, 150 million from checks and 50 million from reads. Caching for

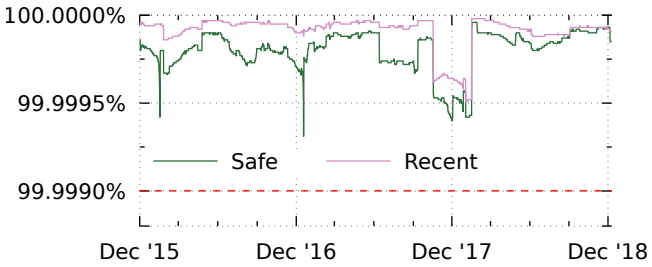

Figure 5: Zanzibar's availability over the past three years has remained above 99.999%.

checks has a 10% hit rate on the delegate's side, with an additional 12% saved by the lock table. Meanwhile, caching on the delegator's side has a 2% hit rate with an additional 3% from the lock table. While these hit rates appear low, they prevent 500K internal RPCs per second from creating hot spots.

Delegated reads see higher hit rates on the delegate's side—24% on the cache and 9% on the lock table—but the delegator's cache is hit less than 1% of the time. For super-hot groups, Zanzibar further optimizes by reading and caching the full set of members in advance—this happens for 0.1% of groups but further prevents hot spots.

This caching, along with aggressive pooling of read requests, allows Zanzibar to issue only 20 million read RPCs per second to Spanner. The median of these requests reads 1.5 rows per RPC, but at the 99th percentile they each read close to 1 thousand rows.

Zanzibar's Spanner reads take 0.5 msec at the median, and 2 msec at the 95th percentile. We find that 1% of Spanner reads, or 200K reads per second, benefit from hedging. We note that Zanzibar uses an instance of Spanner that runs internally to Google, not an instance of Cloud Spanner [6].

The Leopard index is performing 1.56M QPS at the median, or 2.22M QPS at the 99th percentile, based on data aggregated over 7 days. Over the same 7 days, Leopard servers respond in fewer than 150 µsec at the median, or under 1 msec at the 99th percentile. Leopard's incremental layer during those 7 days writes roughly 500 index updates per second at the median, and approximately 1.5K updates per second at the 99th percentile.

4.5 Lessons Learned

Zanzibar has evolved to meet the varied and heavy demands of a growing set of clients, including Google Calendar, Google Cloud, Google Drive, Google Maps, Google Photos, and YouTube. This section highlights lessons learned from this experience.

One common theme has been the importance of flexibility to accommodate differences between clients. For example:

- *Access control patterns vary widely:* Over time we have added features to support specific clients. For instance, we added `computed_userset` to allow inferring an object's owner ID from the object ID prefix, which reduces space requirements for clients such as Drive and Photos that manage many private objects. Similarly, we added `tuple_to_userset` to represent object hierarchy with only one relation tuple per hop. The benefits are both space reduction and flexibility—it allows clients such as Cloud both to express ACL inheritance compactly and to change ACL inheritance rules without having to update large numbers of tuples. See §2.3.1.
- *Freshness requirements are often but not always loose:* Clients often allow unspecified, moderate staleness during ACL evaluation, but sometimes require more precisely specified freshness. We designed our zookie protocol around this property so that we can serve most requests from a default, already replicated snapshot, while allowing clients to bound the staleness when needed. We also tuned the granularity of our snapshot timestamps to match clients' freshness requirements. The resulting coarse timestamp quanta allow us to perform the majority of authorization checks on a small number of snapshots, thus greatly reducing the frequency of database reads. See §3.2.1.

Another theme has been the need to add performance optimizations to support client behaviors observed in production. For example:

- *Request hedging is key to reducing tail latency:* Clients that offer search capabilities to their users, such as Drive, often issue tens to hundreds of authorization checks to serve a single set of search results. We introduced hedging of Spanner and Leopard requests to prevent an occasional slow operation from slowing the overall user interaction. See §3.2.7.
- *Hot-spot mitigation is critical for high availability:* Some workloads create hot spots in ACL data that can overwhelm the underlying database servers. A common pattern is a burst of ACL checks for an object that is indirectly referenced by the ACLs for many different objects. Specific instances arise from the search use case mentioned above, where the documents in the search indirectly share ACLs for a large social or work group, and Cloud use cases where many objects indirectly share ACLs for the same object high in a hierarchy. Zanzibar handles most hot spots with general mechanisms such as its distributed cache and lock table, but we have found the need to optimize specific uses cases. For example, we added cache prefetching of all relation tuples for a hot object. We also delayed cancellation of secondary ACL checks when there are concurrent requests for the same ACL data. See §3.2.5.
- *Performance isolation is indispensable to protect against misbehaving clients:* Even with hot-spot mitigation measures, unexpected and sometimes unin-

tended client behaviors could still overload our system or its underlying infrastructure. Examples include when clients launch new features that prove unexpectedly popular or exercise Zanzibar in unintended ways. Over time we have added isolation safeguards to ensure that there are no cascading failures between clients or between objects of the same client. These safeguards include fine-grained cost accounting, quotas, and throttling. See §3.2.6.

5 Related Work

Zanzibar is a planet-scale distributed ACL storage and evaluation system. Many of its authorization concepts have been explored previously within the domains of access control and social graphs, and its scaling challenges have been investigated within the field of distributed systems.

Access control is a core part of multi-user operating systems. Multics [23] supports ACLs on segments and directories. ACL entries consist of a principal identifier and a set of permissions bits. In the first edition of UNIX [9], file flags indicate whether owner and non-owner can read or write the file. By the 4th edition, the permissions bits had been expanded to read/write/execute bits for owner, group, and others. POSIX ACLs [4] add an arbitrary list of users and groups, each with up to 32 permissions bits. VMS [7, 8] supports ACL inheritance for files created within a directory tree. Zanzibar's data model supports permissions, users, groups, and inheritance as found in the above systems.

Taos [24, 10] supports compound principals that incorporate how an identity has been transformed as it passes through a distributed system. For example, if user U logged into workstation W to access file server S, S would see requests authenticated as "W for U" rather than just U. This would allow one to write an ACL on a user's e-mail that would be accessible only to the user, and only if being accessed via the mail server. Abadi et al. discuss in [11] a model of group-based ACLs with support for compound identities. Their notion of "blessings" are similar to Zanzibar tuples. However, Zanzibar adopts a unified representation for ACLs and groups using usersets, while they are separate concepts in [11].

Role-based access control (RBAC), first proposed in [17], introduced the notion of *roles*, which are similar to Zanzibar relations. Roles can inherit from each other and imply permissions. A number of Zanzibar clients have implemented RBAC policies on top of Zanzibar's namespace configuration language.

A discussion of ACL stores in 2019 would be remiss without mentioning the Identity and Access Management (IAM) systems offered commercially by Amazon [1], Google [5], Microsoft [2], and others. These systems allow customers of those companies' cloud products to configure flexible access controls based on various features such as: assigning users to roles or groups; domain-specific policy languages; and APIs that allow the creation and modification of ACLs. What all of these systems have in common is unified ACL storage and an RPC-based API, a philosophy also core to Zanzibar's design. Google's Cloud IAM system [5] is built as a layer on top of Zanzibar's ACL storage and evaluation system.

TAO [13] is a distributed datastore for Facebook's social graph. Several Zanzibar clients also use Zanzibar to store their social graphs. Both Zanzibar and TAO provide authorization checks to clients. Both are deployed as single-instance services, both operate at a large scale, and both are optimized for read-only operations. TAO offers eventual global consistency with asynchronous replication and best-effort read-after-write consistency with synchronous cache updates. In contrast, Zanzibar provides external consistency and snapshot reads with bounded staleness, so that it respects causal ordering between ACL and content updates and thus protects against the "new enemy" problem.

Lamport clocks [22] provide partially ordered vector timestamps that can be used to determine the order of events. However, Lamport clocks require explicit participation of all "processes", where in Zanzibar's use cases some of the "processes" can be external clients or even human users. In contrast, Zanzibar relies on its underlying database system, Spanner [15], to offer both external consistency and snapshot reads with bounded staleness. In particular, Zanzibar builds on Spanner's TrueTime abstraction [15] to provide linearizable commit timestamps encoded as zookies.

At the same time, Zanzibar adds a number of features on top of those provided by Spanner. For one, the zookie protocol does *not* let clients read or evaluate ACLs at an arbitrary snapshot. This restriction allows Zanzibar to choose a snapshot that facilitates fast ACL evaluation. In addition, Zanzibar provides resilience to database hotspots (e.g. authorization checks on a suddenly popular video) and safe pointer chasing despite potentially deep recursion (e.g. membership checks on hierarchical groups).

The Chubby distributed lock service [14] offers reliable storage, linearizes writes, and provides access control, but it lacks features needed to support Zanzibar's use cases. In particular, it does not support high volumes of data, efficient range reads, or reads at a client-specified snapshot with bounded staleness. Its cache invalidation mechanism also limits its write throughput.

Finally, ZooKeeper offers a high-performance coordination service [19] but also lacks features required by Zanzibar. Relative to Chubby, it can handle higher read and write rates with more relaxed cache consistency. However, it does not provide external consistency for updates across different nodes since its linearizability is on a per-node basis. It also does not provide snapshot reads with bounded staleness.

6 Conclusion

The Zanzibar authorization system unifies access control data and logic for Google. Its simple yet flexible data model and configuration language support a variety of access control policies from both consumer and enterprise applications.

Zanzibar's external consistency model is one of its most salient features. It respects the ordering of user actions, yet at the same time allows authorization checks to be evaluated at distributed locations without global synchronization.

Zanzibar employs other key techniques to provide scalability, low latency, and high availability. For example, it evaluates deeply or widely nested group membership with Leopard, a specialized index for efficient computation of set operations with snapshot consistency. As another example, it combines a distributed cache with a mechanism to deduplicate in-flight requests. It thus mitigates hot spots, a critical production issue when serving data on top of normalized, consistent storage. These measures together result in a system that scales to trillions of access control rules and millions of authorization requests per second.

7 Acknowledgments

Many people have made technical contributions to Zanzibar. We thank previous and recent members of the development team, including Dan Barella, Miles Chaston, Daria Jung, Alex Mendes da Costa, Xin Pan, Scott Smith, Matthew Steffen, Riva Tropp, and Yuliya Zabiyaka. We also thank previous and current members of the Site Reliability Engineering team, including Randall Bosetti, Hannes Eder, Robert Geisberger, Tom Li, Massimo Maggi, Igor Oks, Aaron Peterson, and Andrea Yu.

In addition, a number of people have helped to improve this paper. We received insightful comments from David Bacon, Carolin Gäthke, Brad Krueger, Ari Shamash, Kai Shen, and Lawrence You. We are also grateful to Nadav Eiron and Royal Hansen for their support. Finally, we thank the anonymous reviewers and our shepherd, Eric Eide, for their constructive feedback.

References

[1] Amazon Web Services Identity and Access Management. `https://aws.amazon.com/iam/`. Accessed: 2019-04-16.

[2] Azure Identity and Access Management. `https://www.microsoft.com/en-us/cloud-platform/identity-management`. Accessed: 2019-04-16.

[3] Cache stampede. `https://en.wikipedia.org/wiki/Cache_stampede`. Accessed: 2019-04-16.

[4] DCE 1.1: Authentication and Security Services. `http://pubs.opengroup.org/onlinepubs/9668899`. Accessed: 2019-04-16.

[5] Google Cloud Identity and Access Management. `https://cloud.google.com/iam/`. Accessed: 2019-04-16.

[6] Google Cloud Spanner. `https://cloud.google.com/spanner/`. Accessed: 2019-04-16.

[7] HP OpenVMS System Management Utilities Reference Manual. `https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04622366`. Accessed: 2019-04-16.

[8] OpenVMS Guide to System Security. `http://www.itec.suny.edu/scsys/vms/ovmsdoc073/V73/6346/6346pro_006.html#acl_details`. Accessed: 2019-04-16.

[9] Unix Manual. `https://www.bell-labs.com/usr/dmr/www/pdfs/man22.pdf`. Accessed: 2019-04-16.

[10] ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst. 15*, 4 (Sept. 1993), 706–734.

[11] ABADI, M., BURROWS, M., PUCHA, H., SADOVSKY, A., SHANKAR, A., AND TALY, A. Distributed authorization with distributed grammars. In *Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security - Volume 9465* (New York, NY, USA, 2015), Springer-Verlag New York, Inc., pp. 10–26.

[12] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 739–753.

[13] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC '13, pp. 49–60.

[14] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the*

7th Symposium on Operating Systems Design and Implementation (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[15] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI '12, pp. 251–264.

[16] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74–80.

[17] FERRAIOLO, D., AND KUHN, R. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference* (1992), pp. 554–563.

[18] GIFFORD, D. K. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford, CA, USA, 1981. AAI8124072.

[19] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIX ATC '10, USENIX Association.

[20] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[21] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.

[22] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[23] SALTZER, J. H. Protection and control of information sharing in Multics. In *Proceedings of the Fourth ACM Symposium on Operating System Principles* (New York, NY, USA, 1973), SOSP '73, ACM.

[24] WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. Authentication in the Taos operating system. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 256–269.