Data Management for Internet-Scale Single-Sign-On

Sharon E. Perl
Google Inc.

Margo Seltzer
Harvard University & Oracle Corporation

Abstract

Google offers a variety of Internet services that require user authentication. These services rely on a single-sign-on service, called Google Accounts, that has been in active deployment since 2002. As of 2006, Google has tens of applications with millions of user accounts worldwide. We describe the data management requirements and architecture for this service, the problems we encountered, and the experience we've had running it. In doing so we provide perspective on "where theory meets practice." The success of the system comes from combining good algorithms with practical engineering tradeoffs.

1 Introduction

All of Google's applications that require sign-on or the delivery of enhanced functionality to identified users depend upon a common single sign-on system (SSO). The SSO's availability sets an upper bound on the availability of these applications. Google has extremely high availability goals for all but their most experimental services (e.g., Google Labs), so the availability goals for SSO are aggressive. One way to achieve high availability in a distributed system is to sacrifice data consistency. Exposing an end-user to inconsistent views of a single logical data item (e.g., a username/password combination) can make the system frustratingly unpredictable. Because Google strives to provide the best user experience possible, trading off usability for availability is a decision that is not taken lightly. Early in the design of the SSO architecture we decided that single-copy consistency was a usability requirement.

Achieving single-copy consistency in a highly-available, fault-tolerant distributed computing system is a well-studied problem. We were attracted to the replicated state machine approach embodied in Lamport's Paxos algorithm [3] and Oki's and Liskov's Viewstamped Replication work [5] because of its generality and practicality. Lampson describes how to use the approach along with other mechanisms to build high-performance distributed systems [4].

As we considered various implementation alternatives for the Google SSO system, we learned of the new replication-based high-availability functionality of Berkeley DB (BDB-HA) from Sleepycat Software (now Oracle). BDB-HA has a replication model that is compatible with Paxos and Viewstamped Replication. It is an embedded database that leaves configuration and policy decisions to the application, which makes it easy to adapt to Google's highly distributed and somewhat unconventional production environment. The underlying BDB database is well-suited to the task of efficiently storing and retrieving small, random key-value pairs, which is a good match for the requirements of the SSO system.

The rest of this paper is organized as follows. Section 2 gives a high-level overview of Berkeley DB High Availability for background. Section 3 discusses the storage aspects of the SSO architecture and how it incorporates BDB-HA. In Section 4, we discuss abstractions at the boundary of the SSO service and BDB-HA. In Section 5, we discuss our experience building and running the system, and in Section 6, we conclude.

2 Berkeley DB Overview

Berkeley DB is an embedded, high-performance, scalable, transactional storage system for key/data pairs. "Embedded" indicates that Berkeley DB is a library linking directly into an application's address space, avoiding the costly IPC that reduces performance for client/server systems. On a commodity x86 platform, Berkeley DB returns millions of key/data pairs per second. Berkeley DB is scalable in a number of dimensions: it is used to store bytes to terabytes; its replication is used in systems

ranging from two to many tens of sites; it can be used as a simple data repository or as a highly concurrent, transactional engine.

Berkeley DB provides both keyed and sequential lookup. It does not support any data model (e.g., relational or object-oriented), but different data models can be implemented on top of it. Its simple storage model provides applications with the flexibility to store data in whatever format is most convenient.

Berkeley DB supports a primary copy replication model, with a single writer (called the *master*) and multiple read-only replicas. Conceptually, Berkeley DB's replication is simple: all writes are directed to the master. The master applies the update(s) and then propagates the corresponding database log records to the replicas, each of which applies the changes, as if it were running "redo" recovery [2]. The replicas apply operations on a transactional basis, so transactions that abort on the master require no work on the replicas other than writing log records. The replicas and the master maintain identical logs. When the master fails, the replicas hold an election, and the winning replica becomes the new master. All the other replicas synchronize with this new master, rolling their logs backwards and forwards as necessary to ensure that the replicas have identical logs.

3 SSO Architecture

Figure 1 illustrates the SSO data architecture. The SSO service maps usernames to user account data and services to service-specific data.[1] These mappings are stored in the SSO database, which is partitioned into hundreds of pieces (called *shards*) for load balancing and data localization. Each shard is a replicated Berkeley DB database composed of between 5 and 15 replicas, depending on the shard's purpose. The SSO data in each replica is stored in a single Berkeley DB Btree database [2].

Smaller shards have five full replicas, any of which is capable of becoming a master. All updates must go to the master. Consistent reads must also go to the master. We sometimes allow "stale reads", which may be slightly out-of-date by an amount of time that we control, and which can be performed at non-master replicas. The larger replication groups typically have five replicas capable of becoming masters ("electable replicas") plus additional read-only replicas. Read-only replicas receive updates from the master, but do not participate in elections or contribute to commit quorums for updates, so the number of read-only replicas and their distance from other replicas does not affect the latency or availability of operations. When the system is running well (the normal case) the state of read-only replicas will be fairly closely synchronized with the master. A shard can have a master as long as more than half its electable replicas are up and communicating.

We spread replicas across multiple, geographically distributed data centers for availability in the face of failures of machines, networks, or data centers. At the same time, we try to keep replicas within a shard fairly close to one another because the communication latency between replicas affects how long it takes to commit a write operation to a shard or to elect a new master. The set of shards is geographically dispersed for data locality. We try to assign new users to shards based on where their data is likely to be accessed. This becomes tricky when the user data is shared by a variety of services that also may be spread over geographically dispersed data centers. We could do more optimization of data placement than we currently do, however it has not turned out to be a high priority for system performance.

As illustrated in Figure 1, there are logically two different kinds of shards. The vast majority of shards are independent databases that map a set of userids to account data and service ids to user-independent service data. The remaining shards implement the *ID-map*, which maps usernames to userids and userids to shards.

The ID-map is used for login, e-mail delivery, and at other times when we need to find a user's account data given a username. The ID-map shards are chained together in a doubly-linked list to store an extensible map, for scalability. Each shard in the chain handles a sub-range of the key space. Adjacent shards store adjacent ranges. Client library code keeps hints for the names of the component shards of the ID-map and their corresponding key ranges, so that we do not have to traverse the list for each key access. If the keys get rebal-

[1] To give a sense of scale, the distributed database currently contains over a billion keys and averages about one kilobyte of data per user account. While the system does not contain a huge volume of data, the need for scalability comes from supporting real-time updates, a sustained, high request rate, and high availability for a growing user base.

[2] More precisely, Berkeley DB provides B+-link-trees

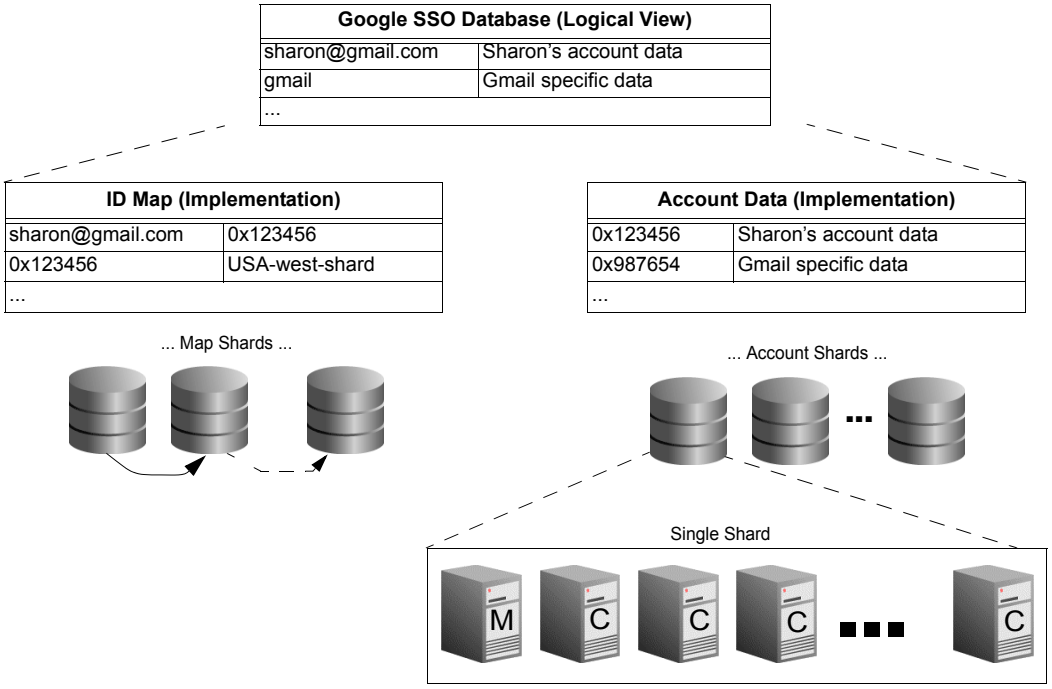

Figure 1: Single-Sign-On Database Architecture. Logically, the database consists of a mapping between user or service names and detailed user or service data. The logical database is realized by two physical databases: the ID-Map and the Account-Data. The ID-Map contains mappings from user names to userids and from userids to shards. The Account-Data contains mappings from userids to user data and service-ids to service data. The ID-Map comprises a linked list of ID shards. The Account-Data comprises the vast majority of independent account shards. For example, an account shard might contain account details for customers in the western United States. Each shard is implemented as a Berkeley DB replication group.

anced among the shards (which they can using offline tools), clients of the storage system will notice the changes and adjust their cached locations.

4 Database Integration

Berkeley DB leaves many policy decisions up to the application. For example, the application is responsible for providing the communication infrastructure. The application registers a callback function that Berkeley DB uses when it wants to send messages; the application receives messages and calls a Berkeley DB function to process messages. Because the application owns the communication infrastructure, it is responsible for deciding how synchronously the replicas run. At one extreme, the master can dispatch messages to the replicas and continue immediately, assuming at least one of the replicas receives the messages. At the other extreme, the master can wait until all the replicas acknowledge that they have applied the transmitted records. The choice affects the performance and semantics of the system, of course.

4.1 Quorums

Following the Paxos and Viewstamped Replication algorithms, Google implements a quorum protocol to guarantee that updates are never lost, even if a site—including the master—suffers a catastrophic data loss. Whenever Berkeley DB indicates to the SSO application that the message it is sending is essential to transactional semantics, the master waits until it has received a positive acknowledgement from a majority (over half) of the replicas, including itself. Only after it has received those acknowledgements does the SSO application consider a trans-

action completed. Therefore, even if the master crashes and loses all its data, the data is guaranteed to reside on the other sites that acknowledged the transaction. Similarly, SSO requires a majority of replicas to agree when electing a new master. Because any two majorities overlap, we are guaranteed that at least one replica participating in the election will have seen the most recent update from the last master. Berkeley DB elections always select a replica with the latest log entry during an election, so we are guaranteed that a new master's log will include all updates committed by the previous master.

4.2 Leases

A pure Paxos-based system requires the involvement of a majority of replicas for reads as well as writes, making reads prohibitively expensive. Practical systems typically use a mechanism called *leases* to allow a master to perform reads locally without the danger of returning stale data if a partition or other failure causes a master to lose its mastership without noticing for some period of time [1, 4].

Google implemented a lease mechanism on top of Berkeley DB replication. The master must hold a *master lease* whenever responding to a read request. The master refreshes its lease every *lease timeout interval* (currently a small number of seconds) by successfully communicating with a majority of replicas. The application must also prohibit elections from completing within the lease timeout interval. Leases are not required to ensure proper write behavior, because the application must obtain successful replies from a majority of the replicas before considering a write complete. Leases renew automatically as long as a master is in communication with a majority of replicas.

To see the need for leases when a master performs local reads, consider a replication group with five replicas, A through E. At time 0, A is the master. At time 1, the network partitions leaving A on one side and B through E on the other. Without leases, B through E could hold an election, elect B as the new master, and process new updates. Simultaneously, A could continue to respond to read requests locally, unknowingly returning stale data. The lease mechanism prevents B through E from holding a successful election until A's master lease expires, even though A is no longer in communciation with the other replicas.

4.3 Replica Group Membership

In addition to the quorum protocols and leases, another key feature of a replicated system not handled within Berkeley DB is replication group membership. For the SSO application, we implemented our own replica group management.

A replica configuration includes the logical (DNS) name of each replica along with its current IP address and the value of the master lease timeout. To change the physical machine associated with a logical name, we change the DNS entry. To add a new logical replica, we need to add its name and IP address to the configuration.

While we don't have room to describe the complete algorithm, some of the key ideas are:

- Only the master performs DNS resolution.
- When the master sees the DNS settings change, it initiates a configuration change.
- The configuration gets stored in the database and is updated via normal database operations.
- Non-master replicas check the database to learn the configuration (certain cues tell them when to check).
- Additions or deletions to the configuration can be specified in a file that the master reads periodically.
- For critical operations (commits and elections) a replica will only process messages from other replicas that have the same configuration.

A new replica starts out as a non-voting member of its replication group. It won't participate in elections until it is deemed to be caught up to the master at least as of the time the new replica started running. So a new replica cannot cause an election outcome to be incorrect.

We require all configuration transitions to have a subset of members that is a quorum in both old and new sets. This is slightly restrictive, but makes it much easier to rule out the possibility of having more than one master at once or of being unable to elect a master.

The most common configuration change is the replacement of a single physical machine correspond-

ing to a logical replica name. This must be handled automatically for smooth system operation. Other configuration changes require some operator actions.

The correctness of Paxos and related protocols relies on replicas not losing their stable storage. If too many replicas lost state, but continued to participate in the protocols as usual, updates could potentially be lost if one of those replicas were elected master. This situation is similar to replacing one physical replica machine with another. A replica that has lost state restarts as a non-voting member of its configuration and begins participating in elections only when it has caught up to the master as of the time the replica restarted.

5 Experience

5.1 Database vs. Application

Google began development of SSO with the first Berkeley DB HA release. Sleepycat was developing a generic infrastructure component and Google was developing a specific application. Each organization needed to make different tradeoffs, but we worked closely together to determine whether various functionality should reside in the application or in the Berkeley DB library.

All of the abstractions described in the previous section—quorums, leases, and replica group management—straddle the boundary between database and application. For example, Google implemented master leases in the application, because at the time, Sleepycat did not see the demand for this feature from other customers. In retrospect, master leases should be implemented in Berkeley DB, because Berkeley DB has more complete and precise knowledge about when masters change. By adding interfaces, we were able to implement master lease support in the application, but it is an unnatural implementation.

The initial Berkeley DB election algorithm was not compatible with Paxos in its use of a concept called *election generations*. When a replica starts a new election, it chooses a generation number greater than any it has seen before. Other replicas participate in the election only if they have never participated in an election with a higher generation number. To implement Paxos-like behavior, election generations numbers must be stored stably, so that if all replicas go down (either for planned maintenance or an unplanned outage), old messages that may still be floating around the system when the replicas come back up cannot cause incorrect behavior in subsequent elections. The probability that such bad behavior occurs is low, but in an installation as large as Google's, even low probability events must be considered possible. None of Sleepycat's other customers considered such a scenario a viable threat.

Implementing stable election ids meant costly disk writes during elections and additional persistent state. Sleepycat wanted to avoid disk writes during elections, because database updates are unavailable during elections, reducing overall system availability. While a few millisecond wait for the disk write might be acceptable for Internet-based applications, it might be perceived as an intolerable delay in a backplane-based system, such as a switch managing TCP connections. The two teams discussed this feature at length to determine whether it was an essential piece of the infrastructure (in which case Sleepycat should implement it) or an application-specific behavior (in which case Google should implement it). Ultimately, Sleepycat decided to implement stable election generations because there was essentially no way to implement them correctly in the application and because Sleepycat saw value in a fully Paxos-compatible implementation.

5.2 Operation

The replicated Berkeley DB-based SSO system was deployed in late 2003. Operationally, everything went extremely smoothly as we first replaced an early version of the SSO system with an unreplicated version of the system based on Berkeley DB and subsequently turned on replication. The system continued to perform well and run smoothly for months while we began to work on improvements to handle the anticipated scaling bottlenecks. For example, the initial deployment did not include the ability to manage replica group membership automatically. This was implemented about a year later, when we had enough shards that our small operations team was spending too much time manually replacing failed machines. As another example, the ID-map originally was implemented by a single shard, for simplicity and expedience. The scalable ID-map was introduced about a year after initial de-

ployment, when traffic growth predictions indicated that the existing single ID-map master would soon become overloaded.

The philosophy of delaying complexity until necessary works well for deploying reliable, scalable systems in a timely way in the setting of Internet services. Some problems need to be addressed before you can deploy anything, while others can wait. While the scaling bottlenecks in the SSO system were fairly predicatable before initial deployment, actual experience with the system helped us prioritize the various followup tasks as the system evolved. While this approach works for Internet services that exert full control over the running software, it does not necessarily work well when shipping a product that others will run and manage.

The SSO storage system is currently low maintenance from an operational standpoint, freeing the team to concentrate on supporting new applications and features. It has scaled by more than an order of magnitude since its initial deployment and still has room to grow.

6 Conclusions

Large systems require robust algorithms needing little maintenance. It is better to design for correctness in the face of as many errors as possible than to place bets on what failure scenarios will and will not occur. Highly-fault-tolerant algorithms like Paxos are valuable. It still helps to understand your operating environment, however. We decided not to use a byzantine fault-tolerant replication algorithm because we felt that the extra cost and complexity were not justified.

Trading-off availability and consistency will always require careful consideration. Experience with the running system gave us confidence that we could ease some of the consistency requirements without sacrificing the user experience. For example, the mapping from usernames to user identifiers almost never changes, and the mapping from names to shards changes only rarely. We can allow slightly stale reads for these types of data without sacrificing usability. The database design fundamentally does not support best-effort writes, however, so we have to pay latency and availability costs for all writes in this design.

An unsurprising lesson is that availability and reliability need to be considered for the system as a whole. The SSO system consists of a distributed collection of front-end servers, middle-tier servers, load balancers, etc., in addition to the many replicated database shards. It is tempting to get caught up in fascinating replication algorithms, dwelling on how available the system can be and still have single-copy consistency. While these are important issues, a highly-available, highly-reliable database is only one component; one must also consider issues like partitions outside of the database component, the location of services relative to account data, and how to set reasonable timeouts on user-level operations.

7 Acknowledgements

Many people contributed to the successful design and implementation of the Google Accounts system. We'd like to thank Mike Burrows, the Google Accounts team, the Sleepycat core team, and Susan LoVerso. Thanks also to Butler Lampson for some illuminating discussions about Paxos. Finally, thank you to the reviewers who gave us excellent feedback on an earlier draft of this paper and to our program committee shepherd, David Andersen.

References

[1] Gray, C., and Cheriton, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles* (New York, NY, USA, 1989), ACM Press, pp. 202–210.

[2] Härder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (1983), 287–317.

[3] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[4] Lampson, B. W. How to build a highly available system using consensus. In *10th International Workshop on Distributed Algorithms (WDAG 96)* (1996), Babaoglu and Marzullo, Eds., vol. 1151, Springer-Verlag, Berlin Germany, pp. 1–17.

[5] Oki, B. M., and Liskov, B. H. Viewstamped replication: a new primary copy method to support highly available distributed systems. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (New York, NY, USA, 1988), ACM Press, pp. 8–17.