Release Consistent Locking for Parallel Programs on Distributed Stateful Lambdas

William Mullen
wmullen@berkeley.edu

Willis Wang
williswang@berkeley.edu

1 Introduction

The Global Data Plane (GDP) [5] is a data-centric system that enables secure management (transport, storage) of data among clients on both the edge and in the cloud. This federated infrastructure enables the seamless access and management of computational resources through a structure called a DataCapsule. DataCapsules are a blockchain like structure that provide integrity, confidentially, and provenance on data. As a result, the GDP can provide security guarantees to services that run on top of it.

However, while the GDP introduces a comprehensive communications layer, it does not natively contain any frameworks for building applications on top of it. The Paranoid Stateful Lambda (PSL) [2] project is a function-as-a-service (FaaS) framework that leverages the GDP's architecture to enable large-scale parallel, secure, and stateful computation among edge and cloud environments. PSLs use DataCapsules and Trusted Execution Environments (TEEs) [4] to share data between nodes securely. Using a in-memory key-value store (KVS) called a memtable, lambdas can multicast values to ensure the entire system has access to any changes. These lambdas are also lightweight enough to run on edge devices in a secure environment while the GDP enables access to additional storage and computation resources. Using PSLs, functions can be run in a cloud environment without needing to trust the underlying hardware provider.

While the PSL project achieves our goal of secure computation on the edge, it does have a major limitation for parallel computing applications. Due to its distributed nature, PSLs only implement a loose consistency guarantee. As a result, nodes may not receive updates immediately. For certain classes of applications, this eventual consistency is more than enough for shared values. However, as we show in section 5, some parallel applications require much stronger guarantees.

1.1 Problem Statement

In this project, we present a release-consistent locking scheme for PSLs. Our algorithm enforces stronger consistency guarantees on the memtable for parallel applications. We demonstrate the correctness of the algorithm as well as evaluate its impact on performance in comparison to the loose consistency model.

2 Background

To provide context for our work, we begin by discussing the GDP, DataCapsules, the PSL project, and two relevant consistency models. Both the GDP and DataCapsules are critical components from a structural standpoint, and function as described in the paper *Global Data Plane: A Federated Vision for Secure Data in Edge Computing* by Mor et al [5].

2.1 Global Data Plane

The GDP [5] is a federated data-centric infrastructure network. There is no central authority. Instead, GDP servers route requests to specific services and clients using GDP names as identifiers. The basic unit of data in the GDP is a structure called a DataCapsule, a data-agnostic and cryptographically secure container for data. See the next section for a more detailed description of DataCapsules. These capsules are replicated and distributed throughout the GDP, acting as storage, communication channels, transaction records, and entity identifiers. Together with the GDP's infrastructure, they provide a robust distributed computing architecture, able to draw on computing resources regardless of physical location. Since each capsule follows a standardized format, they are universally routable and manageable, regardless of the actual data contained within them. Through additional locality optimizations, clients are also able to satisfy additional performance requirements by pulling DataCapsule replicas close to them. This paper will not discuss details of or improvements to the GDP's implementation.

2.2 DataCapsules

A DataCapsule is a cryptographically hardened, append-only data structure that is globally addressable in the GDP and acts as the atomic unit of data transfer. Each capsule is made up of a metadata wrapper and a series of records. In their most basic form, these records contain arbitrary data, a hash over the data, and a series of hash pointers to older records. Each record must also specifically contain a signed hash pointer of the previous record, thus creating a well-ordered chain of hashes as shown in figure 1. The first record in the capsule is a special metadata record that uniquely defines the DataCapsule itself. It contains ownership information, a public key for signing records, and other application specific information. The GDP name of this capsule is defined as the hash of this metadata record. In turn, each record in the capsule can also be uniquely identified by its hash. DataCapsules are also extremely flexible in the type and amount of data each can carry, constrained only by the available resources.

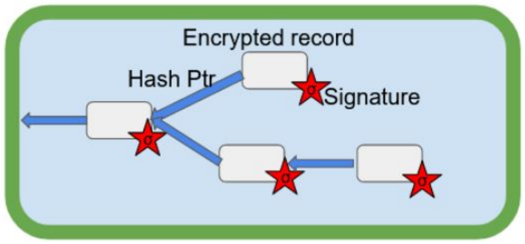

Figure 1: A basic DataCapsule with hash pointers and signatures.

The structure of DataCapsules enforce several powerful security guarantees. Records are immutable, well-ordered, and cryptographically signed. In addition, they contain hashes over their data as well as hash pointers to previous records. Thus, it is straightforward to quickly verify the integrity of records and the overall capsule. This structure also establishes a chain of provenance, storing the data's history for the life of the capsule. For use cases that require even stricter security, proofs can be returned that verify a DataCapsule's validity. Since each record is data agnostic, it is also straightforward to encrypt the stored data, providing confidentiality without disrupting the capsule's structure. Finally, because DataCapsules have the owner's public key embedded into the metadata, only authorized users are able to write to it. Thus, the structure of DataCapsules provides strong integrity, confidentiality, authenticity, and provenance guarantees without needing to rely on third-party infrastructure.

2.3 PSL

The Paranoid Stateful Lambda project (PSL) is an implementation of a function-as-a-service (FaaS) framework similar to AWS Lambda [1] and Google Cloud Functions [3], but built on the GDP. However, in contrast to the aforementioned services, PSLs can maintain state between remote nodes. Each worker also maintains an internal eventually consistent and durable KVS called a memtable. Workers interact with their local memtable using the standard KVS interface: `put(key, value)` to store a key-value pair and `get(key)` to retrieve the stored value for a given key. These memtables enable the statefulness of PSLs. Through a system called the Secure Concurrency Layer (SCL), workers can multicast new key-value (KV) pairs to other workers.

However, PSLs may be operating on sensitive data. For example, in an AI use case, a user may not want their data uploaded to an third-party's server, but the service provider may not want to put their proprietary model directly onto a user device for fear of intellectual property loss. To overcome this problem, PSL executes almost entirely in TEEs, specifically Intel SGX [4] enclaves. These TEEs allow for service providers and users to attest the enclave as well as that the code inside is being executed as expected. Furthermore, enclaves protect the data they contain from the host OS and applications. As a result, PSLs can safely run on the edge or in the cloud, regardless of who owns the underlying hardware, without leaking any information. The only portion of the system that is not inside the enclave is the DataCapsule itself, as its structure natively protects its data as discussed above.

Finally, the SCL also is responsible for connecting the various components of the PSL system together. In addition to the worker enclaves, there is also a coordinating enclave that takes requests from clients and manages the overall execution. Paired with it is a key-management enclave that distributes identity information to remote enclaves and manages the attestation process. With all these components, PSLs provides a robust and secure FaaS framework that has a wide range of applications.

2.4 Consistency Models

For our project, there are two relevant consistency models. The first is the eventual consistency model natively implemented in the PSL project. Under this model, PUT requests to the memtable are immediately available on the node which executed the request. However, the newly added value must be multicast out to the other lambda nodes. Given this networking time, under this model PSL only guarantees that the value will eventually be available to all nodes. Using special SYNC messages, we can upper bound the time until a KV pair reaches all nodes to be the SYNC time: the time between SYNC messages. When a SYNC occurs, the nodes pause computation and communicate the hash of the last DataCapsule record each node received. Since DataCapsules are well-ordered, nodes can detect if they are missing KV pairs and communicate to bring their internal memtables up to date. During operation, timestamps are used to compare whether incoming values are newer and, if so, replace any values with the same key currently in a node's memtable.

The second model is a release consistent locking model which we define to be the following. When a lambda wants to update or insert a KV pair, it first acquires a lock. When that lock is released, all nodes in the system will have the correct KV pair and nodes can use the value immediately. Before the lock is released, it is not possible for any node to retrieve an out of date value. This is a much stronger guarantee than the eventual consistency model as it completely removes the possibility of nodes using old values for computation.

3 Design

In the following sections, we describe the general PSL system structure and distributed locking protocol followed by all worker nodes. Furthermore, we discuss potential cases of race conditions for our locking protocol, and the approaches necessary to solve the given issues.

Our goal was to ensure compatibility with the core PSL system. As part of our changes, we integrated the locking protocol into a blocking API used by the worker application to interact with its local memtable. Specifically, when a worker performs a put or get to an entry in the memtable, the worker will sleep until the put or get operation succeeds. By our release consistent model defined above, the operation only succeeds when the most up-to-date value between all workers is returned by the operation. In order to do so, an individual lock state is maintained for every entry in the memtable.

Talk about general setup: worker nodes, coordinator node, root router node. All worker node's memtables are synchronized - thus we implement a locking layer interacting with local memtable operations.

3.1 Algorithm

Our locking scheme, at a high level, implicitly maintains two different lock types held by worker nodes for different memtable operations:

- **Exclusive Lock:** Only one worker may hold this lock at any point in time. Guarantees full ownership for the memtable entry for the duration the lock is held, and therefore held when performing a put operation on a given memtable entry.
- **Shared Lock:** Multiple worker nodes may simultaneously hold the same shared lock, but may not be acquired when an exclusive lock for the entry is held. Held when performing a get operation on a given memtable entry, thereby avoiding read-write race conditions.

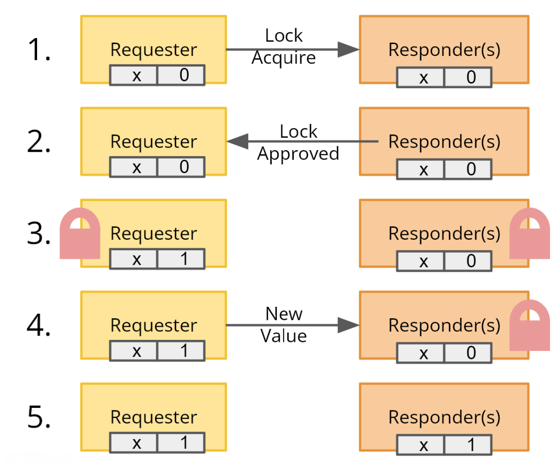

Figure 2: The steps of the locking algorithm.

To implement shared locks, knowledge is needed on whether an exclusive lock of the same entry is acquired by any other worker node at the given point in time. This can be done by tracking exclusive lock requests and releases from other workers, which is feasible given that all communications occur through multicast. Additionally, other worker nodes do not need to know that a shared lock being held by other workers; therefore, a query to other workers is not necessary and the process for acquiring a shared lock is a local process to the worker itself. For the remainder of this section, we focus on the implementation of exclusive locks for the remainder of the paper, which require coordination of all worker nodes. The implementation of shared locks is described more fully in the next section.

Our algorithm consists of five core steps outlined in figure 2. In the first step, a node requests a lock on a key by multicasting a `LOCK_ACQUIRE` message to all other workers and locks its own local copy. In step two, all other nodes check their local memtable, and respond with either a `LOCK_DENY` or `LOCK_ACCEPT`message, depending on local state. In step three, the requesting node checks the messages it received. If there is a single `LOCK_DENY`, it releases its local lock, multicasts a notice to release the locks held by other nodes, and retries a `LOCK_ACQUIRE` after a delay. If all other nodes respond with a `LOCK_ACCEPT`, the requesting node successfully acquires the lock, and makes the update on its local memtable. In step four, the requesting node releases the lock and multicasts the new value to other nodes. Finally, in step five, the other lambdas receive the new value, update their local tables, and unlock the values. Note that all values are guaranteed to unlock at the next SYNC point, given that all outstanding messages are resolved during that process.

3.2 State Machine Implementation

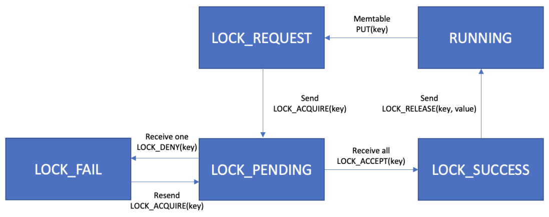

Figure 3: State machine for exclusive locking on PUT operations.

From the protocol described above, each worker node must participate in requesting for locks, as well as accepting or denying these requests. To simplify the protocol as well as to better account for various race conditions, we adopt a state machine abstraction for worker nodes. As PUT operations are a blocking procedure, each worker node will be in one of the four states as described below. Each state dictates the possible operations and responses the worker node may perform. Doing so allows us to to adhere to the properties of shared and exclusive locks. Figure 3 describes the possible states assumed by any worker node. Below, we discuss the meaning of each state, as well as possible message responses specific to each state.

1. `RUNNING`: The worker node is running the user application, and is not performing any memtable operation. In this state, the worker node will accept all incoming `LOCK_REQUEST` requests for any key, by sending back a `LOCK_ACCEPT` acknowledgement message.

2. `LOCK_REQUEST`: The user application issues a memtable PUT request for a particular key. The worker node records the timestamp of the issued PUT request for the given key. In this state, the worker node will accept `LOCK_ACQUIRE` requests for other keys, by sending back a `LOCK_ACCEPT` acknowledgement message. For handling `LOCK_ACQUIRE` requests from other workers on the same key, the given worker will `LOCK_ACCEPT` if the PUT timestamp of the requester worker node is earlier than that of the current worker node, or else will `LOCK_DENY` the request. We argue the correctness and properties of forward progress for time-based tiebreaking in the below section.

3. `LOCK_PENDING`: The worker node has multicast a `LOCK_ACQUIRE` request to all other worker nodes, and waits for a response. The worker node follows the same timestamp-breaking procedure as that of the `LOCK_REQUEST` state.

4. `LOCK_FAIL`: The given worker node has received a `LOCK_DENY` response from one of the other worker nodes. A `LOCK_ACQUIRE` request will be resent after a set retry delay. Do note that the PUT request's timestamp is not updated. The worker node follows the same timestamp-breaking procedure as that of the `LOCK_REQUEST` state.

5. `LOCK_SUCCESS`: The given worker node has received `LOCK_ACCEPT` responses from all other worker nodes, and thus successfully acquires the exclusive lock. In this state, the worker node will `LOCK_ACCEPT` all `LOCK_ACQUIRE` requests for other keys, and `LOCK_DENY` requests for the key corresponding to the held exclusive lock. When the given worker node finishes modifying the value, a `LOCK_RELEASE` message, including the updated value, is multicast to all other workers. This message signifies the release of the memtable entry's exclusive lock. Receiving

worker nodes update their local value of their memtable entry using the updated value of the message.

3.3 Multicast Tree

The worker nodes of the PSL system are not guaranteed to be on the same machine and may be scattered in different physical locations. As such, networking is necessary to facilitate communications between worker nodes. For our implementation, we use the PSL system's built-in multicast protocol to perform all communications between workers. As all messages sent by a worker node are received by all other worker nodes, a sender and recipient field is added to each message. Nodes who are not the recipient, for example from a directed `LOCK_ACCEPT` message, will simply ignore and drop the message. By using the general multicast API, we may utilize future bandwidth and latency improvements of the multicast protocol to improve overall performance of our locking protocol.

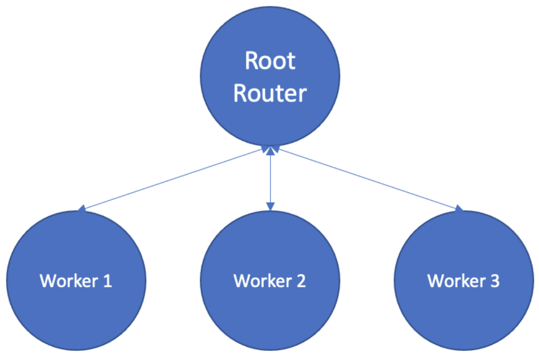

Figure 4: Test multicast setup example.

4 Synchronization

One major source of concern is the dealing of simultaneous `LOCK_ACQUIRE` requests. To informally illustrate the guarantee of forward progress for all worker nodes as well as correctness, presume that all worker nodes simultaneously request an exclusive lock on the same key. Our algorithm guarantees that one worker node will receive `LOCK_ACCEPT` acknowledgements from all other worker nodes - namely, the worker node who has first performed the PUT request, with the timestamp recorded in the `LOCK_REQUEST` stage. Furthermore, the successful worker nodes, following the similar logic, will `LOCK_DENY` the requests from all other workers. Therefore, we ensure that at least one worker node will successfully acquire the lock, while providing the guarantees of exclusive locking.

By breaking ties by initial request timestamp, our protocol guarantees that through repeated `LOCK_ACQUIRE` requests, the requester will eventually successfully acquire the lock. Do note that in the case of requests having the same timestamp, statically assigned worker IDs are used to break ties instead.

Furthermore, as described above, a single `LOCK_DENY` response prompts a worker node to re-request a `LOCK_ACQUIRE` for the denied lock after a delay. Due to network latency delays, it is possible for `LOCK_ACCEPT` and `LOCK_DENY` responses from the previous acquire request, to be received by the worker node after sending the new `LOCK_ACQUIRE` request. As such, a per-worker lock counter is maintained, and is incremented on each `LOCK_ACQUIRE` requests. By sending and receiving this

Number of Nodes	Release Consistency	Eventual Consistency
2	32500	5000
5	82200	14000
9	579555.6	23555.56

Table 1: Performance comparison between locking algorithms.

counter within `LOCK_ACQUIRE` and `LOCK_ACCEPT/DENY` requests respectively, worker nodes are able to filter out stale responses from other workers.

5 Evaluation

To evaluate our implementation, we created a simple marketplace. In this test, there are ten items with 200 units of each whose counts are stored in the shared KVS. As it executes, a set of lambdas attempt to purchase a random number between one and ten units of one of the items. If enough stock is available, the purchase is successful and the lambda updates the value. If not, then it tries again with another random value. The lambdas attempt 1000 purchases over the course of the test.

5.1 Experimental Setup

Our locking algorithm is evaluated on a Intel NUC 7PJYH, equipped with Intel(R) Pentium(R) Silver J5005 CPU @ 1.50GHz with 4 physical cores (4 logical threads). The processor has 96K L1 data cache, a 4MiB L2 cache, and 16GB memory. The machine uses Ubuntu 18.04.5 LTS 64bit. The machine is equipped with Intel SGX2 and runs Asylo version 0.6.2. Our test is written in JavaScript and run using the engine embedded into the PSL system. As such, all security features are enabled, including Intel SGX enclaves, key distribution, signing, encryption, and hashing. Default SYNC time is 20 seconds.

5.2 Comparison to Base Implementation

As table 1 shows, using the release consistent locking scheme imposes a major performance hit on PSL on the order of 6x worse to 24x worse in the nine node case. All times are in milliseconds. However, with the error rates as high as they are, not using our locking model guarantees incorrect behavior. Our test is very simple, and so much more complex applications with high collision rates on shared data would suffer even more erratic results. While there are likely many potential optimizations to our implementation, as well as the base PSL implementation, any application that uses our locking scheme would likely need to incur a major portion of the performance drop shown here.

5.3 Performance Scaling

We also analyzed how the number of lambda nodes impacts the overall performance of the system. As shown in figure 5, as the number of nodes increases, the runtime gets exponentially worse. This behavior is expected, as the number of collisions increases dramatically. Furthermore, requesting nodes must wait for N^2 more `LOCK_ACCEPT` messages before proceeding. Interestingly, this test forms a worst case scenario for our locking algorithm. Every value is shared, there is no additional work that can be done locally, and every node is colliding on the same keys simultaneously. In most parallel applications, we would not expect to see this worse case scenario occur.

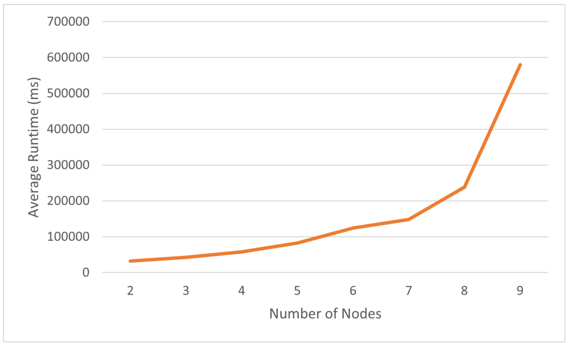

Figure 5: Performance Scaling.

References

[1] *AWS Lambda*. https://aws.amazon.com/lambda/.

[2] Kaiyuan Chen et al. "SCL: A Secure Concurrency Layer For Paranoid Stateful Lambdas". In: 2021.

[3] *Google Cloud Functions*. https://cloud.google.com/functions/.

[4] Frank McKeen et al. "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016, pp. 1–9.

[5] Nitesh Mor et al. "Global Data Plane: A Federated Vision for Secure Data in Edge Computing". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 1652–1663.