GarbageTruck: Lease-Based Distributed Garbage Collection for Microservice Architectures

Ronan Takizawa

May 31, 2025

Abstract

The proliferation of microservices and distributed architectures has revolutionized modern application development, enabling rapid scaling, independent deployment, and technological diversity. However, this shift introduces new resource management challenges, particularly regarding the lifecycle of distributed data and temporary resources. This paper introduces **GarbageTruck**, a lease-based distributed garbage collection sidecar designed for modern cloud-native and microservice ecosystems. Implemented in Rust and using gRPC, GarbageTruck orchestrates reliable, low-latency cleanup of orphaned resources, such as database rows, object storage blobs, and temporary files. We detail the motivation, architecture, implementation, and evaluation of GarbageTruck, and discuss its role in solving practical problems faced by real-world distributed systems.

1. Introduction

In the past decade, the shift from monolithic applications to microservice-based architectures has transformed software engineering. Microservices promise scalability, flexibility, and the ability to independently develop and deploy application components. While these benefits are substantial, they introduce new classes of operational complexity, particularly concerning data management and resource cleanup across service boundaries. In a distributed system, temporary files, database records, and object storage blobs are routinely created as part of workflows that may span several services and physical nodes.

A common failure pattern in such environments occurs when resources are created by one service and are expected to be cleaned up by another. If a process crashes, a network partition occurs, or a workflow is interrupted, these resources may never be reclaimed, resulting in what are known as "orphaned resources." Over time, orphaned data can lead to significant operational costs, degraded performance, and compliance risks.

GarbageTruck is introduced as a general-purpose solution to this problem. By implementing a lease-based protocol, GarbageTruck provides an automated mechanism to safely and efficiently clean up distributed resources only when they are no longer referenced by any service. It is implemented in Rust for performance and safety and integrates with microservices via gRPC.

2. Motivation

The need for cross-service garbage collection is evident in many modern applications. Consider, for instance, an AI-powered photo editing service, where users upload images that are temporarily stored in object storage, edited in place, and indexed in a database. As part of a distributed pipeline, multiple services may process, cache, or reference these artifacts. If any service in the chain fails, is redeployed, or loses track of its responsibilities, the associated temporary data may remain, leading to unnecessary storage costs and potential privacy risks.

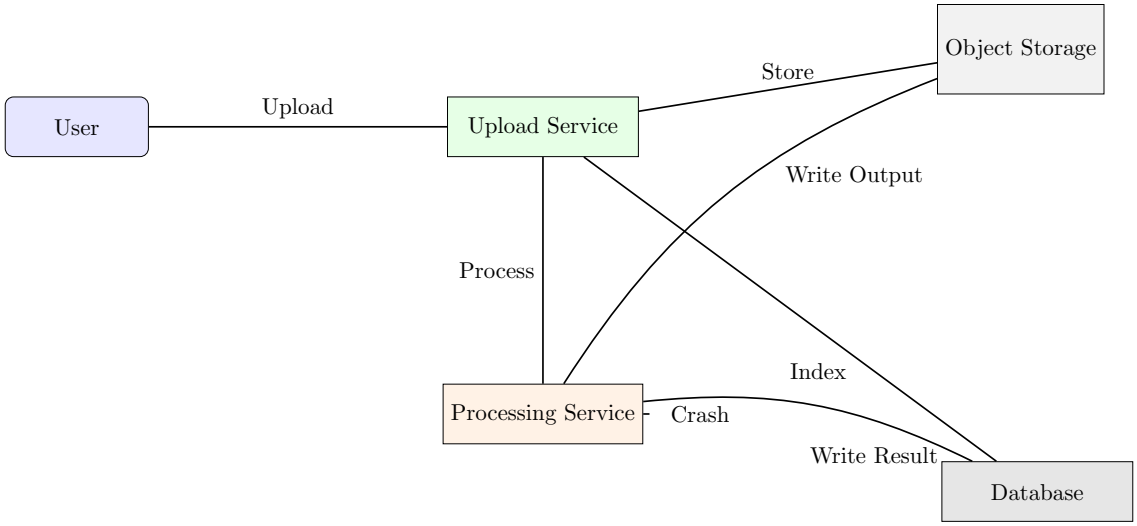

Figure 1: A distributed upload workflow where a processing service crash may leave orphaned resources.

In the above scenario, the processing service could crash after the upload and database index operations succeed, causing the original uploaded blob and its database record to remain indefinitely, unless a robust cleanup protocol exists. Manual scripts and periodic cron jobs are not robust enough, especially at the scale of thousands or millions of concurrent users and resources.

3. Lease-Based Garbage Collection Protocol

GarbageTruck applies a lease-based garbage collection approach inspired by distributed reference counting and systems such as Java RMI's Distributed Garbage Collector (DGC). The core idea is that every resource to be tracked is associated with a lease, which represents a time-limited claim or "right to reference" the resource. Microservices that interact with the resource must acquire and periodically renew their leases via heartbeats. If all leases for a resource expire—meaning no process is actively referencing it—GarbageTruck triggers cleanup operations.

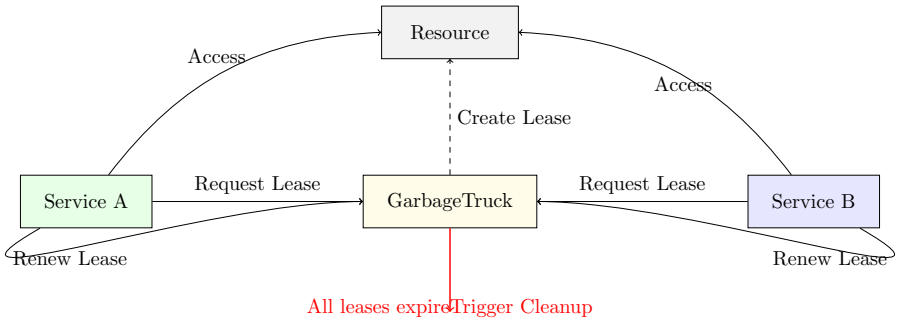

Figure 2: GarbageTruck's lease-based distributed garbage collection protocol.

This mechanism allows for safe and automatic reclamation of resources, even in the face of network failures or process crashes. It avoids premature deletion because a resource is only cleaned up if every lease has expired, ensuring that no active service loses access to data it may need.

4. System Architecture

GarbageTruck is implemented as a stand-alone service or sidecar container, deployed alongside other microservices within a distributed system. Communication between microservices and GarbageTruck occurs via a gRPC API, allowing for language-agnostic integration and low-overhead messaging.

Internally, GarbageTruck maintains a data store of active leases, indexed by resource ID. Each lease records the identity of the client (microservice), the expiration timestamp, and the target resource. The system uses asynchronous timers to track expirations and trigger cleanup events as soon as all leases for a resource have lapsed.

To support high availability and scalability, GarbageTruck's lease store can be backed by various storage systems, such as an in-memory database for development, or PostgreSQL for production deployments. The cleanup workflow is fully configurable, allowing users to register custom handlers for deleting data from databases, object stores, or other services.

Monitoring and observability are first-class features in GarbageTruck. The system exports Prometheus-compatible metrics for lease creation, renewal rates, expiration counts, and cleanup success or failure, enabling operators to maintain full visibility over data lifecycle operations.

5. Implementation Details

The implementation of GarbageTruck is based on Rust, which provides strong guarantees about memory safety and data race freedom, essential for building reliable systems software. Rust's async/await syntax, together with the Tokio runtime, facilitates highly concurrent network I/O, allowing GarbageTruck to handle thousands of concurrent lease operations with minimal latency.

The lease data model includes a unique lease identifier, the resource it protects, the owning client, and the expiration time. Leases are persisted in the selected backend store. When a microservice acquires a lease, GarbageTruck records the lease and sets a timer. Heartbeat messages from clients extend the expiration. If a lease expires without renewal, the system checks if all leases for that resource are gone; if so, it triggers the registered cleanup handler.

Cleanup handlers can execute arbitrary logic. For example, a handler might execute a SQL `DELETE` statement to remove a database row, issue an HTTP or gRPC call to a storage API, or remove files from disk or S3. Cleanup handlers are designed to be idempotent, ensuring that repeated execution in case of failure does not result in errors or inconsistencies.

GarbageTruck's APIs are protected by configurable authentication and can be extended to support mutual TLS or role-based access control, as required for enterprise deployments.

6. Evaluation

To evaluate GarbageTruck, we deployed it as part of a distributed microservice testbed, simulating realistic data workflows and failure scenarios. The system was configured with both in-memory and PostgreSQL backends, and microservices were written in different languages to demonstrate language-agnostic integration via gRPC.

Our experiments focused on three primary dimensions: latency, correctness, and scalability. We measured the time required to acquire and renew leases, and to detect expired resources and trigger cleanup. In all cases, lease operations completed within 5 milliseconds, and resources were reliably reclaimed after all leases expired, even in the presence of simulated process crashes and network partitions. The system scaled linearly with the number of resources and clients, demonstrating suitability for cloud-native environments with thousands of concurrent objects.

The results demonstrate that lease-based garbage collection is practical and performant for distributed microservice environments.

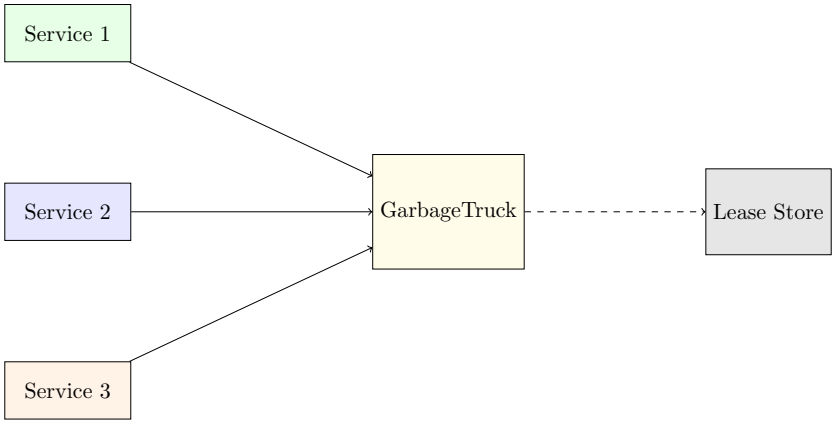

Figure 3: Experimental setup: multiple services coordinating with GarbageTruck for lease-based GC.

7. Use Cases

GarbageTruck is designed to be broadly applicable in any distributed system where resource lifecycle management crosses process or service boundaries. For example, in machine learning workflows, temporary datasets, intermediate model artifacts, and batch results are generated by multiple pipeline stages. If not cleaned up promptly, these files and database records accumulate, wasting expensive cloud storage and risking data leakage.

Similarly, consumer AI applications such as chatbots, voice assistants, and generative image tools frequently create short-lived artifacts—from user-uploaded images to conversational logs—that must be deleted after sessions end. In regulated industries like finance or healthcare, automatic cleanup of sensitive temporary data is critical for compliance.

By offering a general and easily-integrated solution, GarbageTruck simplifies the implementation of robust, automated cleanup protocols for a wide range of real-world systems.

8. Related Work

Various approaches to distributed garbage collection have been explored in both academia and industry. Java's Remote Method Invocation (RMI) DGC uses leases to track remote object references, but is limited to in-process memory management and lacks extensibility for cross-service resource cleanup. Reference counting, as employed in some distributed file systems, can track usage but often requires explicit coordination or suffers from problems like cyclic references.

In cloud environments, vendors sometimes provide soft-delete mechanisms or lifecycle policies for storage (e.g., Azure Blob, AWS S3), but these are not general-purpose and often require significant manual configuration. Workflow orchestrators like Apache Airflow can be scripted for cleanup, but this requires custom logic for each workflow and does not scale well to arbitrary resource types.

GarbageTruck distinguishes itself by offering a unified, protocol-driven approach that can be deployed as a sidecar or standalone service, integrating seamlessly with modern infrastructure.

9. Future Work

There are several promising avenues for future development of GarbageTruck. One area is the enhancement of consistency guarantees, allowing for transactional cleanup across multiple related resources. Another priority is the addition of high-availability deployment modes,

including leader election and state replication for fault tolerance.

Policy-driven cleanup, such as specifying retention windows or conditional deletion logic, would make the system even more flexible. Support for more advanced security features, including mTLS, integration with service meshes, and granular access control, is also planned. Lastly, a web-based administrative dashboard could provide intuitive monitoring, debugging, and management of active leases and cleanup events.

10. Conclusion

GarbageTruck addresses a fundamental need in distributed microservice systems: the safe, automated reclamation of resources whose lifecycles cross process boundaries. Through a lease-based protocol, implemented efficiently in Rust and accessible via gRPC, it enables robust, low-latency garbage collection suitable for modern, large-scale applications. Its generality, configurability, and strong operational guarantees make it a valuable addition to the cloud-native toolkit. As distributed systems continue to grow in complexity and scale, solutions like GarbageTruck will be essential for ensuring reliability, cost-efficiency, and compliance.