Global Real-Time Resource Pool Allocation

Abstract

System1 utilizes multiple subsystems providing functionality such as display content, advertisement, user tracking, site tracking, click behavior, bot detection, and experimentation. Each of these subsystems use limited and scarce resources from a shared pool. Historically, the subsystems would have these resources statically assigned for long-term use. This led to resource underutilization and loss of efficiency. The Resource Allocation System (RAS) dynamically allocates these resources in a reusable manner, globally, in real-time. In doing so, we have demonstrated an increase in resource usage by a 200x factor, greatly increasing the efficiency of each subsystem. While the system we built is managing a specific type of resource, the design of the Resource Allocation System can be generally applied to manage many types of resources by simply considering the resource as a generic container.

Introduction

At *System1*, we host multiple websites for ourselves and our many partners. These websites utilize multiple subsystems and each of these various subsystems have some methodology to identify and report their usage. The methodologies can usually be generalized to just a set of parameters. The parameters may be a single set of values, e.g. ints, longs, guids, or they may be a set of tuples including other dimensions, such as website id, partner id, affiliation id, or traffic segment id. Fundamentally, the parameters control how data is aggregated and reported on by subsequent downstream ETL processes.

For simple use cases, this set of parameters was adequate. The parameter usages could be predefined. However as our testing needs grew, the set of parameters and predefining how they would be used became a limiting factor. For various reasons, the parameter sets are of fixed size and unlimited growth was not an option. Because of this, a new test could require updating code and/or databases in possibly many combinations of various subsystems, the serving code, or the the ETL code all just to plumb the parameters.

Additionally, through advanced testing techniques, our tests start and stop randomly based on many factors, e.g. traffic patterns, test performance, test prioritization. Due to this, we were under utilizing the limited parameter sets because they would be used for a test that wasn't running or for which there was no traffic.

Problem Definition

To address these issues we needed to improve the duty cycle of the resources to support the ability to reallocate existing tracking parameters in real time based on live website traffic. As assignment is in real time, it needs to be highly responsive, sub 10 mSec. Additionally, as our sites are global and are hosted in multiple data centers across US and Europe, the allocation must be eventually-consistent globally.

Solution

Approach

To improve the duty cycle of the resources the following approach is used:

- Clients “lease” exclusive rights to resource.
- Clients specify the duration they require the resource for.
- When the lease expires, the same resource is re-allocated to another client.
- Trans-continental latency is too high, so data must be replicated.
- Effects of eventual-consistency must be minimized

Architecture

The RAS architecture is shown in Figure 1. It consists of three AWS regions, with DynamoDB in each region. It uses DynamoDB replication between regions. There are multiple clients/agents running within regions. Finally there is an admin CLI (command line interface) and a single jenkins server running offline jobs.

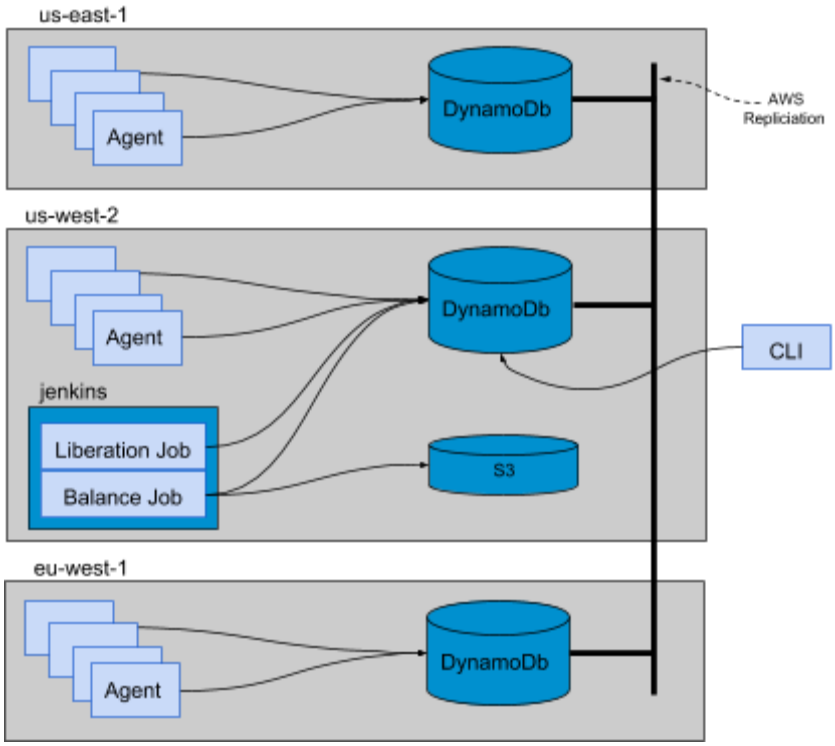

Figure 1: RAS Architecture

DynamoDB

DynamoDB was chosen as the primary data store for its simplicity as a managed service, low latency, high throughput, and global replication. RAS uses auto scaling for both RCU and WCU. The access pattern is Read dominated. RAS uses a single table.

Agent

The agent is a python module that clients import. It is the only means in which clients interface with the system. It provides a single method, `get_lease`, which returns an existing leased resource or a newly acquired leased resource, where “resource” is a string that contains the tracking parameters but is opaque to RAS, e.g. serialized json.

The client supplies four parameters, `client_id`, `pool_id`, `key`, and `lease_expires`. The `client_id` and `pool_id` is a fixed set of predetermined values. The `client_id` may represent the client itself or one of our syndication partners. The `pool_id` sub-divides the client’s pool space into arbitrary sub-pools. Sub-pools provides a dimension of flexibility to the clients and only has limited significance to RAS. The `key` is the dynamic parameter that is known only at run time. To the client, it may represent a test id or a particular traffic segment. To RAS it is just string used to map to a leased resource. In the ETL reporting infrastructure (not

discussed in this paper) the `key` is the mechanism to join and aggregate data across the various subsystems. `Lease_expires` is the GMT time the client is requesting the lease until.

The basic DynamoDB document is:

```
{
        "client_id": "{client id}",
        "pool_id": "{pool id}",
        "resource": "{opaque data string}",
        "key": "{client dynamic key}",
        "lease_expires": {datetime},
}
```

The `client_id` is the primary key and the `resource` is the sort key. When a document is first inserted into the table `client_id`, `pool_id`, and `resource` are all defined, `key` and `lease_expires` are left null. The table has a "Leased Resources" global secondary index with `key` as the primary key and `pool_id` as the sort key. This index allows the agent to call DynamoDB's `GetItem(key)`, which takes approximately 7 mSecs to return an existing lease.

To acquire a new lease, the agent must first call `GetItem(key)` and when that fails it must perform a DynamoDB `scan` to find an available resource. To make the scan efficient, we add the fields `region`, `partition`, and `lease_available` along with an additional "Available Resources" global secondary index (GSI). This index has `partition` as the primary key and `lease_available` as the sort key.

```
{
        "client_id": "{client id}",
        "pool_id": {pool id}",
        "resource": "{opaque data string}",
        "key": "{client dynamic key}",
        "lease_expires": {datetime},
        "region": "{aws region}",
        "partition": "{client id}.{pool id}.{aws region}.[0..9],
        "lease_available": {datetime}
}
```

The `region` field is used to prevent race conditions where multiple regions could update the same document. The agent is aware of its region and only scans for records within the region. The `partition` field value are divided into 10 sub-partitions, [0..9]. The resources are evenly divided across these sub-partitions. When scanning, each agent randomly chooses a sub partition. This keeps the scan fairly consistent in speed regardless of whether the overall number of resources increases or the real-time load increases. The document also has a `version` field. This field is an integer incremented on every write. When an agent wants to acquire the lease, it uses a conditional write incrementing the `version` asserting no other agent has claimed it. It also sets the `key` and removes the `lease_available` datetime, thereby removing it from the Available Resources GSI.

With this partitioning strategy, even with the `GetItem` call, subsequent scan and write of the retrieved document, the acquisition of a new lease is consistently 21 mSec tested up to 10K resources per sub-partition. While this is slower than the desired 10 mSec, the cost is only on the initial acquisition. All subsequent calls for the key will be 7 mSec.

Lease Liberation

A key part of the system is the Available Resources GSI and for that to work the `lease_available` field is either null or set. This keeps the index small and highly targeted. To set this field we use the `lease_expires` field set on the original request. The majority of our leases are for an hour to an hour-and-a-half. The Leased Resources GSI, previously mentioned, only contains resources that are leased, regardless of whether that lease is expired or not. We have an offline jenkins job that runs hourly and scans the Leased Resources GSI and looks for leases that are expired. Finding an expired lease, the job conditionally increments the `version`, clears the `lease_expiration`, clears the `key`, and sets the `lease_available`. This has the ultimate effect of moving the resource from the Leased Resources GSI to the Available Resources GSI.

```
{
        “client_id”: “{client id}”,
        “pool_id”: {pool id}”,
        “resource”: “{opaque data string}”,
        “key”: “{client dynamic key}”,
        “lease_expires”: {datetime},
        “region”: “{aws region}”,
        “partition”: “{client id}.{pool id}.{aws region}.[0..9],
        “lease_available”: {datetime},
        “version”: {incremental number}
}
```

Balancing

An issue that is caused by partitioning resources by region is determining how many resources to allot to each region. Our servers see considerably more traffic in us-east-1 than the other two regions. Because of geographic locations and the natural daily cyclic traffic patterns of each region, each peaks after different GMT points. Depending on breaking news stories, one region may see a sudden spike in traffic. In addition, there is operational support where the operations team will swing traffic out of a region due to a production incident. Ideally the leasing system would be responsive to all of these.

To address this we have an offline balancing job running on jenkins every 5 minutes. It reads the DynamoDB stream from one region and maintains a metadata snapshot of traffic behavior for all clients and pools on a per region basis. Using an exponentially weighted moving average, it projects the loads out for the next ten minutes and re-allocates resources to the necessary

regions to support the projected load. In between invocations, the job persists its state and metadata snapshot on s3.

As the DynamoDB table is globally replicated, the job only needs to watch one region's table and updates that table even if it wants to influence the other regions.

Command Line Interface

We have written an admin CLI for performing CRUD tasks on clients, pools, resources. Like the balancing job, it only needs to work with one region's table.

Eventual Consistency - Caveat

Given the 21 mSec new lease acquisition time and DynamoDB fast global replication, we've found that at our max production traffic load with 99.9% certainty, the key-resource mapping made in one region will be visible in all other regions before the other regions attempt a new resource acquisition for the same key.

In other words, for our usage, approximately 0.1% of the time, multiple resources will be mapped to the same key. We chose to prioritize the accuracy of the data over the optimized resource usage. Should the readers use case differ, it is reasonable to write an offline process similar to the liberator that watches the stream from one region and looks for the same key being assigned to resources from different regions. Finding a duplicate assignment of the same key to two unique resources, it could just release one of the resources for use.

Optimization

This is one of the next items on our road map. We've found the RAS so efficient for managing resources for all of the concurrent testing needs, we've realized that the resources are underutilized the majority of the time. As the balancing job has metadata regarding client pool headroom availability, we can expose this to the client via the agent. This allows the client to adjust resource usage at runtime based on availability.

Summary

The Resource Allocation System is relatively simple and lightweight. It relies heavily on DynamoDB, exploiting its speeds, indexes, auto-scaling, and global replication. The fundamental concepts that makes Resource Allocation work are:

- Managing moving documents between "Available Resources" and "Leased Resources" indexes
- Partitioning to prevent cross-region race conditions and allow scalability

- Conditional writes to prevent inter-region race conditions

The most complex part of the system is the balancing job. Determining the current state and predicting future state is challenging given several clients and pools. It is even more challenging to determine which region to pull resources from when moving to another region when dealing with more than two regions. This is all occuring while resources are actively changing lease state.

Currently RAS is managing approximately 155K actual resources. With the 200x factors dynamic allocation provides for us, that gives us 31M pseudo resources for testing purposes. Scalability testing we've done demonstrates we can increase the actual resources being managed by two orders of magnitude with negligible impact on performance.

While the code and algorithms themselves are proprietary, hopefully we've shared enough of an architectural approach that helps the reader solve similar problems.

Authors
Curtis Beck, System1 Principal Engineer
Cameron Hawkins, System1 Software Engineer