How we designed Dropbox ATF: an async task framework

I joined Dropbox not long after graduating with a Master's degree in computer science. Aside from an internship, this was my first big-league engineering job. My team had already begun designing a critical internal service that most of our software would use: It would handle asynchronous computing requests behind the scenes, powering everything from dragging a file into a Dropbox folder to scheduling a marketing campaign.

This Asynchronous Task Framework (ATF) would replace multiple bespoke async systems used by different engineering teams. It would reduce redundant development, incompatibilities, and reliance on legacy software. There were no open-source projects or buy-not-build solutions that worked well for our use case and scale, so we had to create our own. ATF is both an important and interesting challenge, though, so we were happy to design, build and deploy our own in-house service.

ATF not only had to work well, it had to work well at scale: It would be a foundational building block of Dropbox infrastructure. It would need to handle 10,000 async tasks per second from the start, and be architected for future growth. It would need to support nearly 100 unique async task types from the start, again with room to grow. There were at least two dozen engineering teams that would want to use it for entirely different parts of our codebase, for many products and services.

As any engineer would, we Googled to see what other companies with mega-scale services had done to handle async tasks. We were disappointed to find little material published by engineers who built supersized async services.

Now that ATF is deployed and currently serving 9,000 async tasks scheduled per second and in use by 28 engineering teams internally, we're glad to fill that information gap. We've documented Dropbox ATF thoroughly, as a reference and guide for the engineering community seeking their own async solutions.

Introduction

Scheduling asynchronous tasks on-demand is a critical capability that powers many features and internal platforms at Dropbox. Async Task Framework (ATF) is the infrastructural system that supports this capability at Dropbox through a callback-based architecture. ATF enables developers to define callbacks, and schedule tasks that execute against these pre-defined callbacks.

Since its introduction over a year ago, ATF has gone on to become an important building block in the Dropbox infrastructure, used by nearly 30 internal teams across our codebase. It currently supports 100+ use cases which require either immediate or delayed task scheduling.

Glossary

Some basic terms repeatedly used in this post, defined as used in the context of this discussion.

Lambda: A callback implementing business logic.

Task: Unit of execution of a lambda. Each asynchronous job scheduled with ATF is a task.

Collection: A labeled subset of tasks belonging to a lambda. If send email is implemented as a lambda, then password reset email and marketing email would be collections.

Priority: Labels defining priority of execution of tasks within a lambda.

Features

Task scheduling
Clients can schedule tasks to execute at a specified time. Tasks can be scheduled for immediate execution, or delayed to fit the use case.

Priority based execution
Tasks should be associated with a priority. Tasks with higher priority should get executed before tasks with a lower priority once they are ready for execution.

Task gating
ATF enables the the gating of tasks based on lambda, or a subset of tasks on a lambda based on collection. Tasks can be gated to be completely dropped or paused until a suitable time for execution.

Track task status
Clients can query the status of a scheduled task.

System guarantees

At-least once task execution
The ATF system guarantees that a task is executed at least once after being scheduled. Execution is said to be complete once the user-defined callback signals task completion to the ATF system.

No concurrent task execution
The ATF system guarantees that at most one instance of a task will be actively executing at any given in point. This helps users write their callbacks without designing for concurrent execution of the same task from different locations.

Isolation
Tasks in a given lambda are isolated from the tasks in other lambdas. This isolation spans across several dimensions, including worker capacity for task execution and resource use for task scheduling. Tasks on the same lambda but different priority levels are also isolated in their resource use for task scheduling.

Delivery latency
95% of tasks begin execution within five seconds from their scheduled execution time.

High availability for task scheduling
The ATF service is 99.9% available to accept task scheduling requests from any client.

Lambda requirements

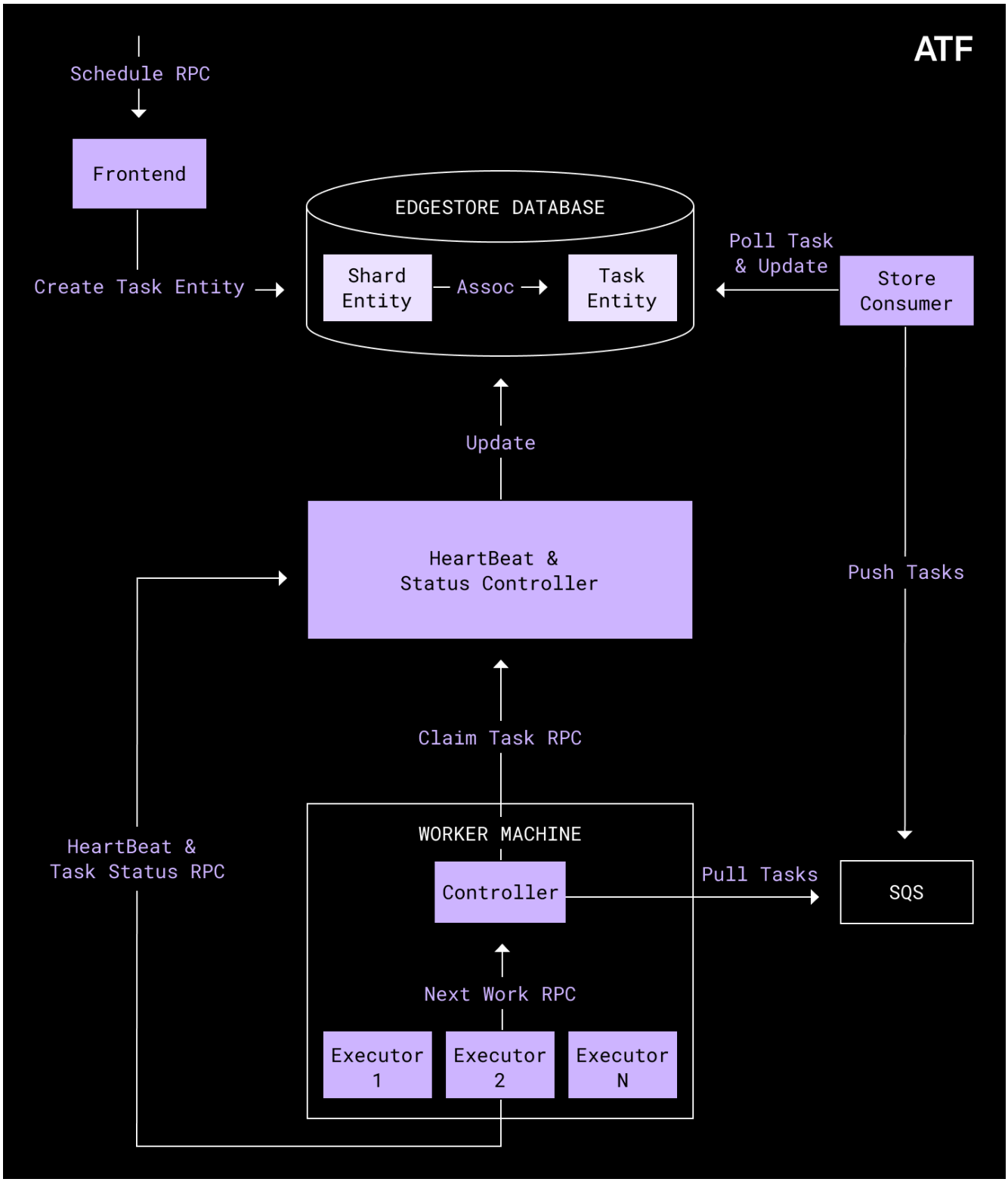
Following are some restrictions we place on the callback logic (lambda):

Idempotence
A single task on a lambda can be executed multiple times within the ATF system. Developers should ensure that their lambda logic and correctness of task execution in clients are not affected by this.

Resiliency
Worker processes which execute tasks might die at any point during task execution. ATF retries abruptly interrupted tasks, which could also be retried on different hosts. Lambda owners must design their lambdas such that retries on different hosts do not affect lambda correctness.

Terminal state handling
ATF retries tasks until they are signaled to be complete from the lambda logic. Client code can mark a task as successfully completed, fatally terminated, or retriable. It is critical that lambda owners design clients to signal task completion appropriately to avoid misbehavior such as infinite retries.

Architecture



Async Task Framework (ATF) [Fig 1]

In this section, we describe the high-level architecture of ATF and give brief description of its different components. (See Fig. 1 above.) In this section, we describe the high-level architecture of ATF and give brief description of its different components. (See Fig. 1 above.) Dropbox uses gRPC for remote calls and our in-house Edgestore to store tasks.

ATF consists of the following components:

- Frontend
- Task Store

- Store Consumer
- Queue
- Controller
- Executor
- Heartbeat and Status Controller (HSC)

Frontend
This is the service that schedules requests via an RPC interface. The frontend accepts RPC requests from clients and schedules tasks by interacting with ATF's task store described below.

Task Store
ATF tasks are stored in and triggered from the task store. The task store could be any generic data store with indexed querying capability. In ATF's case, We use our in-house metadata store Edgestore to power the task store. More details can be found in the Data Model section below.

Store Consumer
The Store Consumer is a service that periodically polls the task store to find tasks that are ready for execution and pushes them onto the right queues, as described in the queue section below. These could be tasks that are newly ready for execution, or older tasks that are ready for execution again because they either failed in a retriable way on execution, or were dropped elsewhere within the ATF system.

Below is a simple walkthrough of the Store Consumer's function:

```
repeat every second:
  1. poll tasks ready for execution from task store
  2. push tasks onto the right queues
  3. update task statuses
```

The Store Consumer polls tasks that failed in earlier execution attempts. This helps with the at-least-once guarantee that the ATF system provides. More details on how the Store Consumer polls new and previously failed tasks is presented in the Lifecycle of a task section below.

Queue
ATF uses AWS Simple Queue Service (SQS) to queue tasks internally. These queues act as a buffer between the Store Consumer and Controllers (described below). Each <lambda, priority> pair gets a dedicated SQS queue. The total number of SQS queues used by ATF is #lambdas x #priorities.

Controller
Worker hosts are physical hosts dedicated for task execution. Each worker host has one controller process responsible for polling tasks from SQS queues in a background thread, and then pushing them onto process local buffered queues. The Controller is only aware of the lambdas it is serving and thus polls only the limited set of necessary queues.

The Controller serves tasks from its process local queue as a response to NextWork RPCs. This is the layer where execution level task prioritization occurs. The Controller has different process level queues for tasks of different priorities and can thus prioritize tasks in response to NextWork RPCs.

Executor
The Executor is a process with multiple threads, responsible for the actual task execution. Each thread within an Executor process follows this simple loop:

```
while True:
  w = get_next_work()
  do_work(w)
```

Each worker host has a single Controller process and multiple executor processes. Both the Controller and Executors work in a "pull" model, in which active loops continuously long-poll for new work to be

done.

Heartbeat and Status Controller (HSC)
The HSC serves RPCs for claiming a task for execution (ClaimTask), setting task status after execution (SetResults) and heartbeats during task execution (Heartbeat). ClaimTask requests originate from the Controllers in response to NextWork requests. Heartbeat and SetResults requests originate from executor processes during and after task execution. The HSC interacts with the task store to update the task status on the kind of request it receives.

Data model

ATF uses our in-house metadata store, Edgestore, as a task store. Edgestore objects can be Entities or Associations (assoc), each of which can have user-defined attributes. Associations are used to represent relationships between entities. Edgestore supports indexing only on attributes of associations.

Based on this design, we have two kinds of ATF-related objects in Edgestore. The ATF association stores scheduling information, such as the next scheduled timestamp at which the Store Consumer should poll a given task (either for the first time or for a retry). The ATF entity stores all task related information that is used to track the task state and payload for task execution. We query on associations from the Store Consumer in a pull model to pick up tasks ready for execution.

Lifecycle of a task

1. Client performs a Schedule RPC call to **Frontend** with task information, including execution time.
2. Frontend creates Edgestore entity and assoc for the task.
3. When it is time to process the task, **Store Consumer** pulls the task from **Edgestore** and pushes it to a related **SQS** queue.
4. **Executor** makes NextWork RPC call to **Controller**, which pulls tasks from the **SQS** queue, makes a ClaimTask RPC to the HSC and then returns the task to the **Executor**.
5. **Executor** invokes the callback for the task. While processing, **Executor** performs Heartbeat RPC calls to **Heartbeat and Status Controller (HSC)**. Once processing is done, **Executor** performs TaskStatus RPC call to **HSC**.
6. Upon getting Heartbeat and TaskStatus RPC calls, **HSC** updates the **Edgestore** entity and assoc.

Every state update in the lifecycle of a task is accompanied by an update to the next trigger timestamp in the assoc. This ensures that the Store Consumer pulls the task again if there is no change in state of the task within the next trigger timestamp. This helps ATF achieve its at-least-once delivery guarantee by ensuring that no task is dropped.

Following are the task entity and association states in ATF and their corresponding timestamp updates:

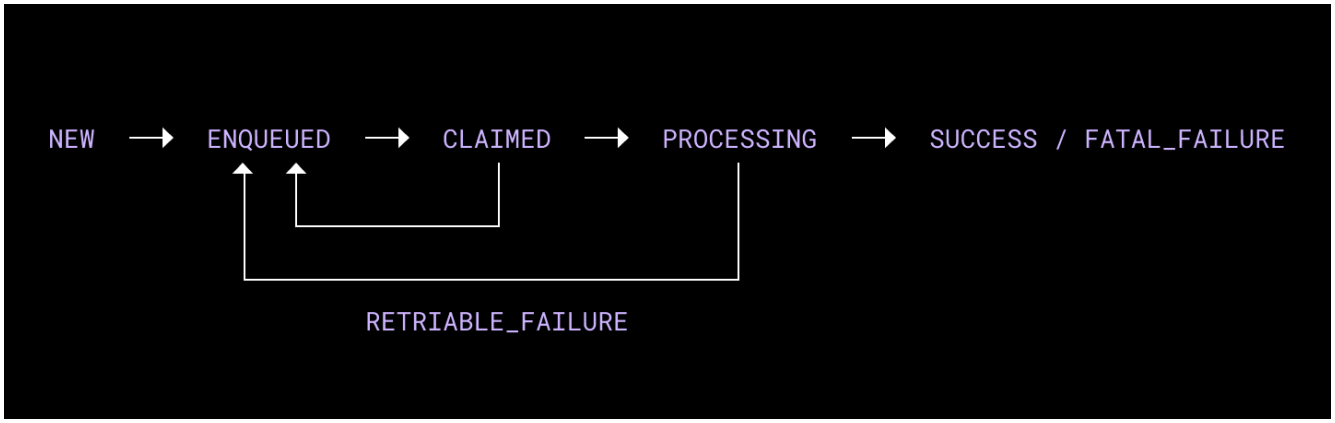
Entity status	**Assoc status**	**next trigger timestamp in Assoc**	**Comment**
new	new	scheduled_timestamp of	Pick up new tasks that are ready.

		the task	
enqueued	started	enqueued_timestamp + enqueue_timeout	Re-enqueue task if it has been in enqueued state for too long. This can happen if the queue loses data or the controller goes down after polling the queue and before the task is claimed.
claimed	started	claimed_timestamp + claim_timeout	Re-enqueue if task is claimed but never transfered to processing. This can happen if Controller is down after claiming a task. Task status is changed to enqueued after re-enqueue.
processing	started	heartbeat_timestamp + heartbeat_timeout`	Re-enqueue if task hasn't sent heartbeat for too long. This can happen if Executor is down. Task status is changed to enqueued after re-enqueue.
retriable failure	started	compute next_timestamp according to backoff logic	Exponential backoff for tasks with retriable failure.
success	completed	N/A	
fatal_failure	completed	N/A	

The store consumer polls for tasks based on the following query:

assoc_status= && next_timestamp<=time.now()

Below is the state machine that defines task state transitions:



Achieving guarantees

At-least-once task execution
At-least-once execution is guaranteed in ATF by retrying a task until it completes execution (which is signaled by a Success or a FatalFailure state). All ATF system errors are implicitly considered retriable failures, and lambda owners have an option of marking tasks with a RetriableFailure state. Tasks might be dropped from the ATF execution pipeline in different parts of the system through transient RPC failures and failures on dependencies like Edgestore or SQS. These transient failures at different parts of the system do not affect the at-least-once guarantee, though, because of the system of timeouts and re-polling from Store Consumer.

No concurrent task execution
Concurrent task execution is avoided through a combination of two methods in ATF. First, tasks are explicitly claimed through an exclusive task state (Claimed) before starting execution. Once the task execution is complete, the task status is updated to one of Success, FatalFailure or RetriableFailure. A task can be claimed only if its existing task state is Enqueued (retried tasks go to the Enqueued state as well once they are re-pushed onto SQS).

However, there might be situations where once a long running task starts execution, its heartbeats might fail repeatedly yet the task execution continues. ATF would retry this task by polling it from the store consumer because the heartbeat timeouts would've expired. This task can then be claimed by another worker and lead to concurrent execution.

To avoid this situation, there is a termination logic in the Executor processes whereby an Executor process terminates itself as soon as three consecutive heartbeat calls fail. Each heartbeat timeout is large enough to eclipse three consecutive heartbeat failures. This ensures that the Store Consumer cannot pull such tasks before the termination logic ends them—the second method that helps achieve this guarantee.

Isolation
Isolation of lambdas is achieved through dedicated worker clusters, dedicated queues, and dedicated per-lambda scheduling quotas. In addition, isolation across different priorities within the same lambda is likewise achieved through dedicated queues and scheduling bandwidth.

Delivery latency
ATF use cases do not require ultra-low task delivery latencies. Task delivery latencies on the order of a couple of seconds are acceptable. Tasks ready for execution are periodically polled by the Store Consumer and this period of polling largely controls the task delivery latency. Using this as a tuning lever, ATF can achieve different delivery latencies as required. Increasing poll frequency reduces task delivery latency and vice versa. Currently, we have calibrated ATF to poll for ready tasks once every two seconds.

Ownership model

ATF is designed to be a self-serve framework for developers at Dropbox. The design is very intentional in driving an ownership model where lambda owners own all aspects of their lambdas' operations. To promote this, all lambda worker clusters are owned by the lambda owners. They have full control over operations on these clusters, including code deployments and capacity management. Each executor process is bound to one lambda. Owners have the option of deploying multiple lambdas on their worker clusters simply by spawning new executor processes on their hosts.

Extending ATF

As described above, ATF provides an infrastructural building block for scheduling asynchronous tasks. With this foundation established, ATF can be extended to support more generic use cases and provide more features as a framework. Following are some examples of what could be built as an extension to ATF.

Periodic task execution
Currently, ATF is a system for one-time task scheduling. Building support for periodic task execution as an extension to this framework would be useful in unlocking new capabilities for our clients.

Better support for task chaining
Currently, it is possible to chain tasks on ATF by scheduling a task onto ATF that then schedules other tasks onto ATF during its execution. Although it is possible to do this in the current ATF setup, visibility and control on this chaining is absent at the framework level. Another natural extension here would be to better support task chaining through framework-level visibility and control, to make this use case a first class concept in the ATF model.

Dead letter queues for misbehaving tasks
One common source of maintenance overhead we observe on ATF is that some tasks get stuck in infinite retry loops due to occasional bugs in lambda logic. This requires manual intervention from the ATF framework owners in some cases where there are a large number of tasks stuck in such loops, occupying a lot of the scheduling bandwidth in the system. Typical manual actions in response to such a situation include pausing execution of the lambdas with misbehaving tasks, or dropping them outright.

One way to reduce this operational overhead and provide an easy interface for lambda owners to recover from such incidents would be to create dead letter queues filled with such misbehaving tasks. The ATF framework could impose a maximum number of retries before tasks are pushed onto the dead letter queue. We could create and expose tools that make it easy to reschedule tasks from the dead letter queue back into the ATF system, once the associated lambda bugs are fixed.

Conclusion

We hope this post helps engineers elsewhere to develop better async task frameworks of their own. Many thanks to everyone who worked on this project: Anirudh Jayakumar, Deepak Gupta, Dmitry Kopytkov, Koundinya Muppalla, Peng Kang, Rajiv Desai, Ryan Armstrong, Steve Rodrigues, Thomissa Comellas, Xiaonan Zhang and Yuhuan Du.