GitLab

Pull Mirroring knowledge sharing session

Create Deep Dive

Tiago Botelho - Backend Engineer

18th of December 2018

Purpose

- Share my knowledge of the pull mirroring feature with the entire GitLab team
- Make this “deep dive session” a reference for everyone that might need to work with pull mirroring in the future

Table of Contents

- Demo
- Workflow
  - High level lifecycle
  - Architecture
- Code dive
- Troubleshooting
  - Debugging Redis
  - Debugging Sidekiq workers
  - Useful methods
- Useful Links
- Questions

We will not talk about

- Mirroring through SSH
- Push mirroring
- Bi-directional mirroring

What is pull mirroring?

- Feature available in GitLab Starter/Bronze tier
- Automatically pulls changes from an external repository into a project in GitLab
- Makes an effort to keep healthy mirrors synchronized with the external repository every 30 minutes
- A user is also able to update more often by using the “Update now” functionality
  - Also handles common failure scenarios gracefully
- Very useful for teams that have a canonical version of their code in an external repository and want to have a secondary version hosted on either GitLab.com or their own GitLab instance
  - E.g: Users have their code hosted on an external code hosting service
  - They want to leverage our CI service
  - They set up a pull mirror that is kept in sync and runs all the pipelines that were configured for that project

Key factors

- There are a few core concepts that we need to explain in order for pull mirroring to make sense
    - Capacity
    - State transitions
    - State Management
    - Determining when a mirror update should be attempted again
- Key Metrics
    - Over 50k mirrors on GitLab.com
    - All of which were updated within the last 30 minutes

Capacity

- Redis Set
- Contains the IDs of projects that are about to or currently are being updated by a Sidekiq worker
- The total capacity is a fixed number that can be configured by the GitLab instance admin
- It is used as a way of limiting the amount of mirrors that get added in the Sidekiq queue
- The objective is to always fill that capacity with as many mirrors as we can
  - This way the workers will always have work to perform
  - Translates into more frequent updates
- It should be a number higher than the configured Sidekiq concurrency
  - Making the capacity a lot higher than the concurrency that Sidekiq enables won’t make a difference and will just translate into a bigger Sidekiq queue
- Making the value lower than the Sidekiq’s concurrency will just translate into less frequent updates

State transitions

- A mirror can be in one of the following five states:
    - None
    - Scheduled
        - Will be responsible for scheduling a worker to update the mirror
    - Started
        - Flags the time that the mirror started the update
    - Finished
        - Marks the time the mirror successfully finished
        - Will set the time when the mirror will get updated again
    - Failed
        - Marks the time the mirror finished unsuccessfully
        - Will increase the retry counter and set the time to update again
- The state machine is also useful to look for mirrors that are in inconsistent states
    - E.g: Mirrors in started state that don’t have a running Sidekiq job

State management

- There are three focal points that track the progress of the mirroring for each project
  - The Database
    - Holds information such as current status, job id, etc
  - Sidekiq
    - Provides the information about the mirroring queue
    - We are also able to know the status of each job in specific
  - Redis capacity set
    - Has the project IDs that are either in Sidekiq's queue or already being performed by one of the workers
- Spreading the information about which projects are currently getting updated helps the service become self-healing in some scenarios
  - Example: If a project in the Database says it has started
    - We can check if that job ID (stored in the Database) is still being performed or if it has finished already
    - This will tell us if Sidekiq was able to gracefully communicate with the DB in order to transition the project onto it's next stage

When should a mirror get scheduled?

- In order to determine when a mirror should be updated a formula was developed:

```
base_delay = (BACKOFF_PERIOD + rand(JITTER)) * (now() - last_update_started_at)
```

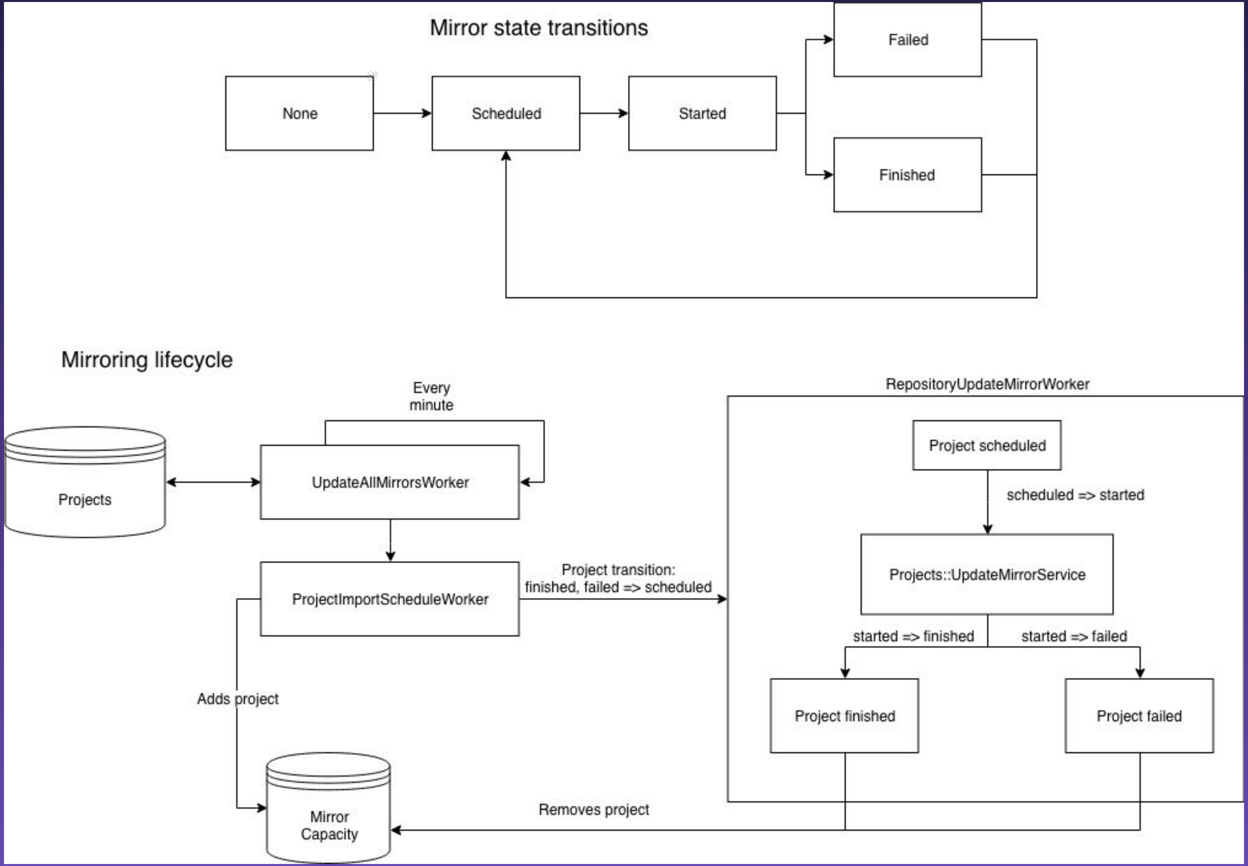
```
def set_next_execution_timestamp
  timestamp = Time.now
  retry_factor = [1, self.retry_count].max
  delay = [base_delay(timestamp), ::Gitlab::Mirror.min_delay].max
  delay = [delay * retry_factor, ::Gitlab::Mirror.max_delay].min

  self.next_execution_timestamp = timestamp + delay
end
```

- We want to penalise mirrors that fail often from running as frequently as healthy mirrors
- If a mirror reaches the maximum amount of retries, it will transition into a hard failed state where it won’t get scheduled until a user takes action and solves the issue

Workflow

1. The scheduler worker will pick all the mirrors that have the next execution time < now()
2. It will schedule mirrors until there is no more capacity available or when there are no more mirrors ready to be updated at the moment
3. After mirroring starts:
   a. Fetch the changes from the provided remote URL
   b. Update the respective branches with the new information
4. After updating the mirror:
   a. Remove the project from the capacity list
   b. Set the next execution time
   c. Mirror finishes
      i. Clear retry counter
   d. Mirror fails
      i. Retry counter gets incremented

Architecture



Mirror state transitions

Mirroring lifecycle

Questions?

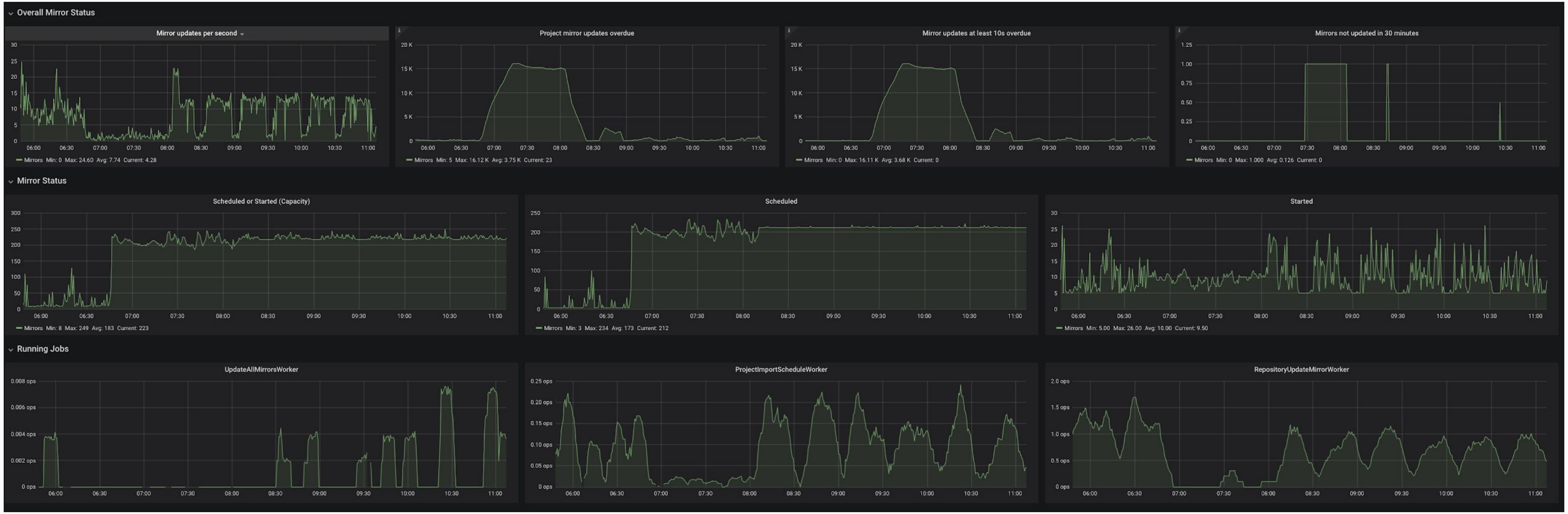
Code Dive

Seeing what is behind the curtains

Questions?

Grafana

- Example of an unhealthy mirroring system (link)



Overall Mirror Status

Mirror updates per second

Mirrors Min: 0 Max: 24.60 Avg: 7.74 Current: 4.28

Project mirror updates overdue

Mirrors Min: 5 Max: 16.12 K Avg: 3.75 K Current: 23

Mirror updates at least 10s overdue

Mirrors Min: 0 Max: 16.11 K Avg: 3.68 K Current: 0

Mirrors not updated in 30 minutes

Mirrors Min: 0 Max: 1.000 Avg: 0.126 Current: 0

Mirror Status

Scheduled or Started (Capacity)

Mirrors Min: 8 Max: 249 Avg: 183 Current: 223

Scheduled

Mirrors Min: 3 Max: 234 Avg: 173 Current: 212

Started

Mirrors Min: 5.00 Max: 26.00 Avg: 10.00 Current: 9.50

Running Jobs

UpdateAllMirrorsWorker

ProjectImportScheduleWorker

RepositoryUpdateMirrorWorker

Troubleshooting

- Always refer to the “project_mirror_data” table or the “ProjectImportState” model to check the state of your mirrors such as:
  - last_error
  - retry_count
  - jid
  - last_updated_at
  - last_successful_update_at
  - next_execution_timestamp
- The DB table is called “project_mirror_data” for legacy reasons even though ProjectImportState is used jointly by imports and forks as well

Troubleshooting

- Checking the available capacity
  - Gitlab::Mirror.available_capacity
  - Helps us debug situations where we might not be removing projects from the capacity
- Project.mirrors_to_sync(Time.now) will return all the mirrors ready to be picked for an update
  - Along with Gitlab::Mirror.available_capacity we are able to see if we have enough mirrors to completely fill the capacity up

Troubleshooting

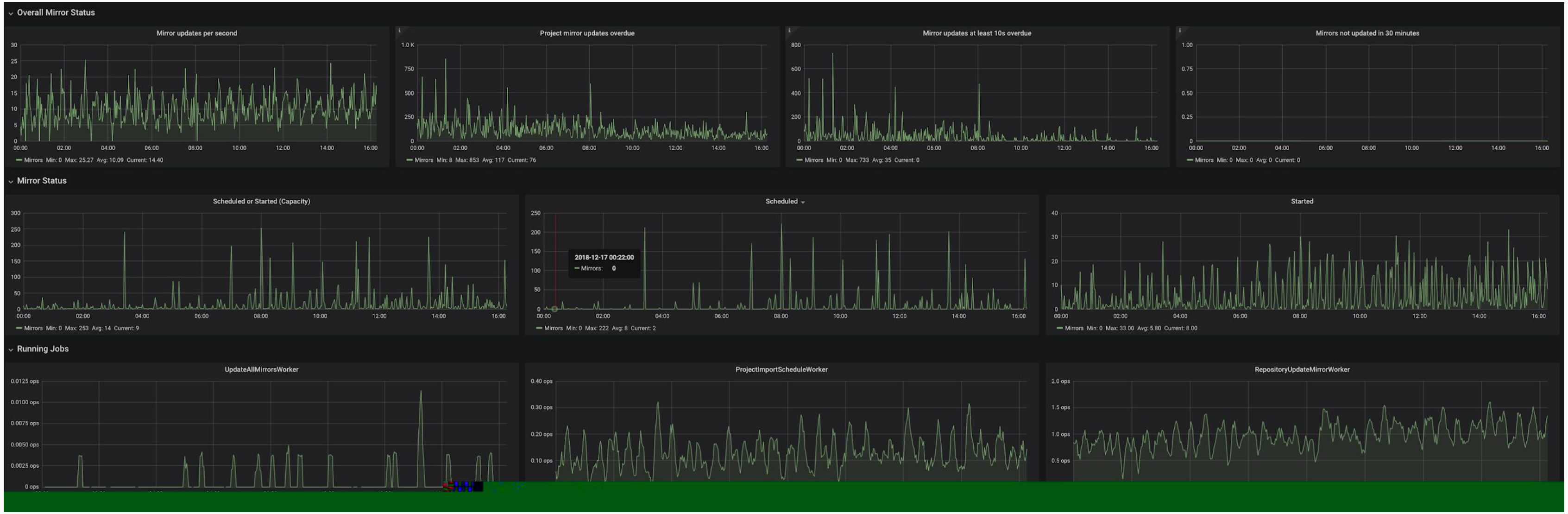
- Check the status of the workers for each mirror in the scheduled/started state
  - ProjectImportState.with_status([:scheduled, :started]).where.not(jid: nil).select(:jid)
  - Gitlab::SidekiqStatus.job_status(jids)
- Retrieve the project IDs that are currently in the Redis set
  - Gitlab::Redis::SharedState.with { |r| r.smembers(Gitlab::Mirror::PULL_CAPACITY_KEY) }
  - Useful to look for projects that are stuck or with inconsistent information
  - Example: A finished/failed project ID should never be in that list

Troubleshooting

- Clear data inconsistencies
  - When the Database is inconsistent with Sidekiq
    - A project is started in the DB, but Sidekiq already considers it finished
    - StuckImportJobsWorker will look at the job ids maintained by the Gitlab::SidekiqStatus Redis key
    - Usually a timeout is the main cause for this scenario
  - When the capacity set is inconsistent with the DB and Sidekiq
    - A project is finished but the project id is still present in the capacity set
    - The only solution might be to remove that project ID from the capacity set
    - This is currently done manually (StuckImportJobsWorker will handle this in the future)
    - `Gitlab::Redis::SharedState.with { |redis| redis.del(Gitlab::Mirror::PULL_CAPACITY_KEY) }`
      - Only use this when the capacity is completely blocked!
    - `Gitlab::Mirror.decrement_capacity(project_id)`
    - This is currently done manually (StuckImportJobsWorker will handle this in the future)

Grafana

- A healthy mirroring system


Overall Mirror Status
Mirror updates per second
30
25
20
15
10
5
0
00:00 02:00 04:00 06:00 08:00 10:00 12:00 14:00 16:00
Mirrors Min: 0 Max: 25.27 Avg: 10.09 Current: 14.40
Project mirror updates overdue
1.0 K
750
500
250
0
Mirrors Min: 8 Max: 853 Avg: 117 Current: 76
Mirror updates at least 10s overdue
800
600
400
200
0
Mirrors Min: 0 Max: 733 Avg: 35 Current: 0
Mirrors not updated in 30 minutes
1.00
0.75
0.50
0.25
0
Mirrors Min: 0 Max: 0 Avg: 0 Current: 0
Mirror Status
Scheduled or Started (Capacity)
300
250
200
150
100
50
0
Mirrors Min: 0 Max: 253 Avg: 14 Current: 9
Scheduled
250
200
150
100
50
0
2018-12-17 00:22:00
Mirrors: 0
Mirrors Min: 0 Max: 222 Avg: 8 Current: 2
Started
40
30
20
10
0
Mirrors Min: 0 Max: 33.00 Avg: 5.80 Current: 8.00
Running Jobs
UpdateAllMirrorsWorker
0.0125 ops
0.0100 ops
0.0075 ops
0.0050 ops
0.0025 ops
0 ops
ProjectImportScheduleWorker
0.40 ops
0.30 ops
0.20 ops
0.10 ops
RepositoryUpdateMirrorWorker
2.0 ops
1.5 ops
1.0 ops
0.5 ops

Useful links

- Pulling from a remote repository documentation
- StateMachine ActiveRecord module documentation
- Infrastructure Pull Mirroring Troubleshooting Guides

Questions?

Thank you

Tiago Botelho - Backend Engineer
tiago@gitlab.com