CodaLab Worker System

Konstantin Lopyrev
klopyrev@gmail.com

1 Introduction

Research in Computer Science is often hindered by a lack of executable code and data that are published along with papers. Even if code and data are provided, it is often unclear how to get the same results as in the papers. Anyone wanting to continue the work needs to spend significant amounts of time reproducing the results.

CodaLab is a system for researchers to upload their code and data, and to run that code in a sandboxed environment, demonstrating to others how to reproduce the results they publish in their papers. CodaLab makes it easy for peers to re-run the same code on the same data, to run the code on new data, or to even run the code with some modifications.

At the core of CodaLab is a job execution system, called the worker system, which takes code and all dependent data, runs it in a sandboxed environment such that two deterministic runs yield the same outputs, and saves the outputs in long-term storage. This paper describes a re-implementation of the CodaLab worker system, the system used to run code.

2 Goals

At the beginning of this project, CodaLab used a worker system that had a number of problems, including:

- **Confusing design**: The system was built on top of an existing job management program designed for a purpose other than CodaLab. That made the code difficult to understand.
- **Lack of flexibility**: Any significant new features would require making significant changes to the existing system and would be hacks on top of an already confusing design.
- **Bugs and stability issues**: Due to the loose coupling between CodaLab and the job management program it was difficult to ensure robustness. Additionally, the complexity resulted in various bugs.

The primary goal of this project was to change the worker system to allow users to run their own workers using their own compute resources, something that was not possible to do in a clean way with the old implementation. The secondary goal of the project was to simplify the design, make it more flexible and more stable. This report describes what was an almost complete rewrite of the worker system.

3 Design Overview

The worker system consists of 3 components:

- **REST server**: This highly concurrent REST RPC server is responsible for answering client requests, including those from workers, for storing any metadata in a database and for storing any data on disk.
- **Workers**: Workers run on machines with available compute resources. User commands are executed using Docker, a platform for running programs inside isolated containers. User commands, known as bundles, include information about which environment (i.e. Docker image) to run the commands in, any data dependencies and the command line string.
- **Bundle manager**: This process continuously checks the database for bundles to run and schedules them to run on workers.

4 Communication Model

At the heart of the worker system is a communication model that allows for messages to be sent to and from the workers. Sending messages from the workers to the REST server is necessary to support:

- Reporting information about the workers for scheduling bundles
- Reporting the status of bundle execution
- Downloading any run dependencies
- Uploading the outputs of finished run bundles

Sending messages to the workers is necessary to support:

- Scheduling bundles to run on workers
- Reading data from running bundles
- Writing data to running bundles
- Killing running bundles
- Upgrading workers

4.1 Sending messages from workers

The worker sends messages to the REST server via a REST API, supporting the following methods:

1. **Check in**: Updates scheduling information about the worker and receives messages sent to the worker.
2. **Check out**: Indicates that a worker is shutting down.
3. **Reply**: Replies to a read request with a JSON message.
4. **Reply data**: Replies to a read request with binary data.
5. **Start bundle**: Indicates that a bundle is about to start running.
6. **Get bundle contents**: Downloads the contents of a dependency.
7. **Update bundle metadata**: Updates information such as run status and resource consumption of a bundle.
8. **Update bundle contents**: Uploads the contents of a bundle upon completion of a run.
9. **Finalize bundle**: Indicates that a bundle has finished running.
10. **Get code**: Downloads the new worker code in order to upgrade a running worker.

Where these are used are described in more detail in the following sections.

All requests are encrypted using SSL to ensure that the data is protected. To call all of these the worker first authorizes using the OAuth2 password grant flow, receiving an access token that can be used when making requests until it expires. Downloading and uploading of directories is done by archiving the directories using the tar tool. Any binary data sent during downloading and uploading is GZipped to minimize bandwidth use and to speed up transfer. Finally, uploads of an archived directory or file are done using chunked encoding since archiving is done on the fly and the size of the archive is not available at the beginning of the upload.

4.2 Sending messages to workers

Some of the more complex code is used for sending messages to workers. One of the requirements of the worker system is to support running workers on diverse computers. Such computers could be behind firewalls and NATs, making connecting to a worker difficult. To support sending requests to workers I've implemented a generic messaging system that allows for messages to be sent to a worker without creating a connection to the worker.

This system consists of a long polling loop. The worker has a messaging threading which continuously calls a "check in" REST method on the server. After updating scheduling metadata about the worker, the REST server blocks the request until either a message sent to the worker is available or the call times out. Upon return from request, the worker handles the message, if any, and then calls the "check in" method again to check for any more messages. The "check in" call has a timeout period of 2 seconds to ensure that any scheduling information is fresh and that if the user shuts down an idle worker at most 2 seconds pass before the worker actually shuts down. Note that there is also a "check out" call which is used to tell the server that a worker is shutting down and prevent it from scheduling bundles to it.

On the server side, the sending of messages is done synchronously using Unix domain sockets. A database table and a folder on the file system are used for allocating and creating Unix domain sockets. Each Unix domain socket has a unique ID which is used in the path to the socket file to ensure that the path is unique. These unique IDs are assigned using the socket database table that also keeps track of all sockets allocated to a worker so that they can be cleaned up when a worker is shut down.

The process for receiving messages is to start listening on a socket, accept a single connection and then to receive one of more messages. The process for sending messages is to connect to a socket and then to send one or more messages. We support two types of messages: JSON messages and binary streams of data. Note that all uses of this mechanism either send just a single JSON message, or send a header JSON message followed by a stream of data unless the header message indicates that an error has occurred.

When the worker "checks in" for the first time, it allocates a single socket ID. Then, every time it "checks in" it receives a message by listening on the corresponding file. Note that however Unix domain sockets are not well designed to support the use case where a server continuously starts and stops listening on a single socket. I use a retry mechanism to ensure that messages sent to a worker are received reliably.

First, note that all messages sent to a worker are JSON messages and sending JSON messages can be retried, unlike sending binary stream which can only be read from once on the sending side. Before a worker starts listening it deletes any existing socket file, causing any connection attempts blocked on that file to fail. Then, it creates a new socket file and starts listening on that file. The code sending messages, if it finds that the socket file got deleted, will retry sending the message. Note that due to what appears to be a bug in the socket code a connection is sometimes accepted spuriously after the worker has already received a message and is in the process of handling that message. To account for this issue I have the listening code acknowledge that it is ready to receive the message by sending a single byte on that connection. The code sending the message then checks for the presence of that byte, retrying sending the message if it doesn't arrive.

These sockets are also used when a worker needs to reply to a message. Specifically, replying is needed to read the contents of a running bundle from the worker. There are several types of read that the worker needs to support:

- Reading the directory structure of the bundle.
- Reading the truncated version of the stardard output and standard error.
- Reading an entire directory or a file.
- Tailing a file.

All of these reads are supported by a design where to make the read the request code first allocates a new socket for the worker, sends a request message to the worker containing the ID of this socket, and then waits for the worker to reply. Replying to a message is implemented using two REST methods: one that replies with just a JSON message transmitted in the body of the HTTP request and a second that takes the contents of a header JSON message from the query parameters and sends the body as a stream.

Reading a directory structure uses the first reply type where the structure is sent in the JSON message. Before reading the contents of files and directories, all requests first read the directory structure to figure out whether the specified path is a file, a directory or a symbolic link. Dereferrencing symbolic links is not allowed for security reasons. The directory structure response does contain all the information to display the target of the link, though.

The rest of the read types send errors as just a JSON message, but otherwise send the response data as a stream. Tailing files is implemented using an API to read a section of a file. The code then uses this API to repeatedly read the end of the file until the bundle finishes running, getting any new contents that appear as they are generated.

4.3 Alternative Designs

Note that this worker communication model is fairly complex. One alternative that I considered is to make the messaging system asynchronous by using a database table to store messages. The main problems with this design are:

- In order to make the workers reasonably responsive to messages they would need to poll the database for messages at a high rate. With many workers this is expensive.
- It is not clear how to support streaming large amounts of data from the worker to the user through the REST server without an extra layer of buffering.

5 Threading Model

The worker is multi-threaded to ensure that it is generally responsive. The different threads in use are described below.

5.1 Main Thread

The main thread is responsible for receiving messages sent to the worker. This thread runs a loop that calls the REST server, updating scheduling information and fetching new messages for the worker.

When the worker receives a message telling it to start running a bundle it creates a new thread to start and manage this run. Having a separate thread for starting the run ensures that the worker is responsive to other messages.

Messages to read the contents of a running bundle are also handled in a separate thread started for each read since such reads can take a non-negligible amount of time. Messages to write a short string to a file in the running bundle are handled by the main thread since writes are quick. Finally, messages to kill a running bundle are passed off to the thread managing the running bundle as described below.

When the worker needs to shut down, whether to upgrade as described in the section 10 or because it is told to shut down via one of several Unix signals, it first waits for any runs to finish. During this time it reports to the REST server that it does not have any slots for starting new runs, preventing the scheduling of new runs to the worker.

Note that the main work of this thread is wrapped in a try / except block to ensure that the worker continues running even when, for example, the REST server is down temporarily.

5.2 Dependency Management Thread

This thread manages the dependencies cached on the worker. It is described in detail in section 8.

5.3 Run Threads

2 threads are started for each running bundle. The first main run thread starts, monitors and finishes the run. The second disk monitoring thread performs the disk use computation since that computation could potentially be much slower than the other monitoring steps.

After starting the run the run thread wakes up every few seconds, reporting resource utilization, checking and handling any kill requests, and checking and reporting if the run finished.

There are several situations where the run can be killed. If the user requests for the run to be killed when it is just starting, the request will interrupt any ongoing download of the Docker image or dependency. If the user requests for the run to be killed when it has already started, the worker code simply sends a kill request to Docker and then handles the termination of the Docker container as normal. A container is also killed via a kill request to Docker if at any point in time it has used more time, memory or disk than allowed.

Finally, note that the run thread is robust to exceptions. Any exceptions when the run is starting are reported as failures to the REST server. When it runs, the code ignores exceptions since such exceptions are expected to be temporary (e.g. the REST server is down). In case the REST server is down when the run is finishing, the code retries until the REST server comes back up.

5.4 Alternative Designs

One alternative design is to have a single thread for monitoring all of the runs. However, with this alternative extra logic is needed for keeping track of which runs need to be monitored as well as for finishing a run and uploading the results without blocking the monitoring of other runs.

6 Running Bundles

In order to sandbox the execution of bundles the worker system uses Docker, a platform for running code inside containers that provides resource isolation. A bundle runs inside a container built from a user-specified image. The process to start running a bundle proceeds as follows:

1. All dependencies of the bundle are downloaded if necessary. The management of these dependencies is described further in 8.
2. The Docker image is downloaded if necessary.
3. The output directory is created and mounted as read-write inside the container.
4. Dependencies are mounted as read-only volumes inside the container.
5. Some GPU set up is done, described in section 7.
6. The container is started.

Note that Docker containers run as the root user. However, the output directory may be owned by a different user. To ensure that any new files created inside the output directory belong to the correct user the user-specified command is executed inside the Docker container as part of a ”sudo” call which runs that command as the user that owns the output directory.

Once the container is running, memory and CPU utilization is monitored by reading this information from a set of files created by Docker called cgroup files. Total amount of disk used is computed by a separate thread by reading the sizes of all the files, with some throttling to ensure that for runs with a lot of files this computation doesn't use significant disk I/O. Note that I decided not to use the container statistics Docker API since it seems to be very slow to read and hard to parse.

The monitoring thread also checks periodically whether the Docker container corresponding to the run has finished executing. Once it has, the process to finish running the bundle is as follows:

1. The Docker container is deleted.
2. The contents of the bundle are uploaded, retrying on server errors. This step updates the total amount of bundle storage space used by the user.
3. The bundle metadata is updated to report that the run has finished, also retrying on server errors. This step updates the total amount of time used by the user.
4. The new bundle is added as an available dependency on the worker.

7 GPU Set Up

The new worker system supports using the GPUs available on a worker machines along with CUDA. The setup is the same as suggested in the Tensorflow Docker documentation. Generally, CUDA consists of several components:

1. The NVIDIA driver, a kernel module, working with the /dev/nvidia device files.
2. CUDA driver, a shared library, working with the NVIDIA driver. This driver is different for different versions of the NVIDIA driver and thus the two need to agree.
3. CUDA Toolkit containing code for using CUDA.

The NVIDIA driver works at the kernel level and needs to be loaded as a kernel module with only one such module loaded at the same time per device. Thus, that needs to be done outside of Docker. Since the CUDA driver and the NVIDIA driver need to agree, the CUDA driver also needs to be available from outside Docker to prevent version mismatch. Thus, when setting up the Docker container the /dev/nvidia device files are added as devices for the container, and the CUDA driver shared library is mounted as a read-only volume. To use CUDA on CodaLab one simply needs to use a Docker image that has the CUDA Toolkit installed. One example is the Tensorflow Docker image.

8 Managing Dependencies

Workers that do not share a file system with the main server (i.e. user-owned workers or workers running for the worksheets.codalab.org deployment, more below) need to download all necessary dependencies to run a bundle. Dependencies can be specified at a granularity of a specific path within a bundle. In such a case, only that path is downloaded and saved. It is possible that two dependencies need to be downloaded that overlap, that is one specifies a path that contains the other. The current design does not optimize this case by ensuring that the same data is downloaded only once due to the complexity involved in this optimization.

The logic to download dependencies is complicated by the fact that multiple bundles starting to run at the same time could need the same dependency. This case is handled by having one of the threads starting the bundle actually download the dependency and the other thread block on a condition variable until the download finishes.

To ensure quick re-execution of a bundle with large dependencies the workers keep a cache of downloaded dependencies, deleting old dependencies once the size of the cache reaches a maximum size. The management of this cache is done by a separate thread that wakes up every 10 seconds and checks the total size of all dependencies. If the total size exceeds the maximum it starts evicting the least recently used dependencies until either the size falls below the limit or there are no unused dependencies. Note that this clean-up thread figures out and caches the sizes of any newly added dependencies, a design that is slightly simpler than having the code adding dependencies specify the size.

Finally, it is important to ensure that if the worker is restarted (e.g. when upgrading or when a user shuts it down temporarily) any downloaded dependencies are preserved. This ensures that a user working on a large dataset doesn't have to download the whole dataset every time the worker is

restarted. To handle this case, the dependency management code saves its in-memory state into a file that is updated every time something changes, including when dependencies are added and when the size of a dependency is computed.

9 Scheduling

All scheduling decisions are made by the bundle manager process which runs on the main server. This process wakes up every 0.5 seconds and checks to see if any bundles are in a state where an action needs to be taken. The bundle manager takes bundles through a state machine which differs slightly depending on the CodaLab deployment.

9.1 worksheets.codalab.org

On the public worksheets.codalab.org deployment there are a few dedicated worker machines which regularly check in, asking for bundles to run. The state machine for this deployment is shown in figure 1, with the state transitions executed by the bundle manager shown in black and those executed by request from a worker shown in red.

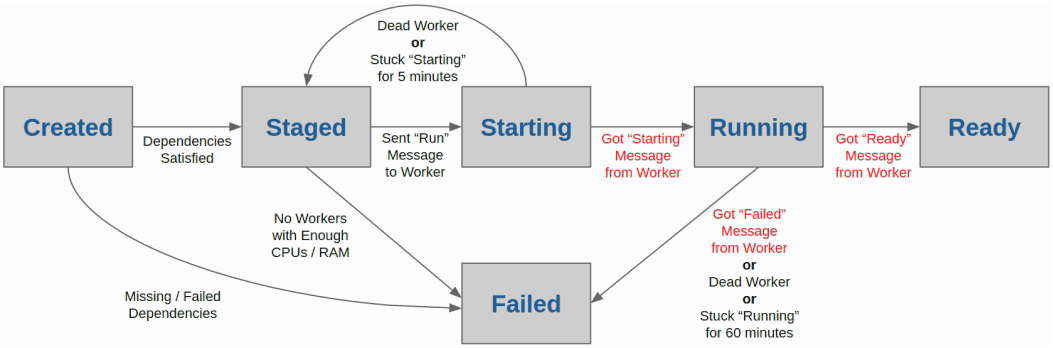

Figure 1: worksheets.codalab.org Scheduling

When "run" bundles are created, they start out in "created" state. If the bundle manager finds that all the dependencies of a bundle are satisfied, it moves it into the "staged" state, ready to be scheduled to run. When it finds an available worker to run the bundle, it sends a message to the worker, telling it to run the bundle, and then moves the bundle into the "starting" state. At this point, the worker checks in with the REST server, telling it that it is starting to run the bundle and moving the bundle into the "running" state. Once the bundle finishes running, the worker checks in with the REST server, moving the bundle into either the "ready" or into the "failed" state, depending on whether the run succeeded.

There are a few exceptional cases that need to be handled. First, if after being created a bundle has dependencies which either failed or are missing, the bundle is moved into the "failed" state, unless the bundle is run with the *–allow-failed-dependencies* flag which causes failures to be ignored. Second, if after being "staged" there are no workers with enough resources to run the bundle, that is all workers have less CPUs or RAM, the bundle is also "failed". Finally, once a bundle is "starting" or "running", if the worker that the bundle is assigned to fails to check in for 5 minutes or if the bundle has been in the given state for a long time, it is moved into the state shown in figure 1.

Note that the "starting" state serves an important purpose in the state machine. This state prevents two workers from starting to run the same bundle in some exceptional scenarios, such as network partitions or other network issues. When the worker is starting to run the bundle it checks with the REST server to see if the bundle is still in the "starting" state and if it is it moves the bundle to the "running" state in the same transaction as the check. If the bundle is not in the "starting" state, then the worker doesn't start running the bundle.

An important aspect of the implementation is which exact worker a given bundle is assigned to. The design is fairly simple. First, workers are filtered:

- **By available slots**: Each worker has a fixed number of slots for running bundles.
- **By number of CPUs**: Only workers with enough CPUs are used in case the user specifies the number of CPUs. Note that this doesn't check whether the CPUs are available, but only if the machine in total has enough CPUs.
- **By amount of RAM**: Only workers with enough RAM are used in case the user specifies the amount of RAM. Again, this is total RAM not available RAM.
- **By tag**: A user can specify a tag for the worker as well as for a bundle so that bundles are scheduled to workers with specific tags.

After the set of workers is filtered to match the above criteria, the bundle is scheduled to the worker that has the most needed dependencies available. This ensures that, for example, once a large dataset is downloaded on some machine runs that use that dataset stay on one machine as much as possible. If multiple workers have the same number of dependencies, then the worker with the most free slots is chosen.

9.2 codalab.stanford.edu

The Stanford NLP deployment uses the Torque job scheduling system which CodaLab is built on top of. The same worker code is used by both deployments. What differs is how workers are started. The worksheets.codalab.org deployment has a fixed set of worker jobs running, while on codalab.stanford.edu workers are started as required. Thus, an additional state is added, as shown in 2. After the bundle is moved into the "staged" state the bundle manager issues a command to Torque to start a new worker, with just a single worker per bundle. Once that worker starts up and checks in with the REST server, the bundle manager sends a message to it to start running the bundle. Once the worker reports that the run has finished, the bundle manager issues a command to Torque to delete the worker.

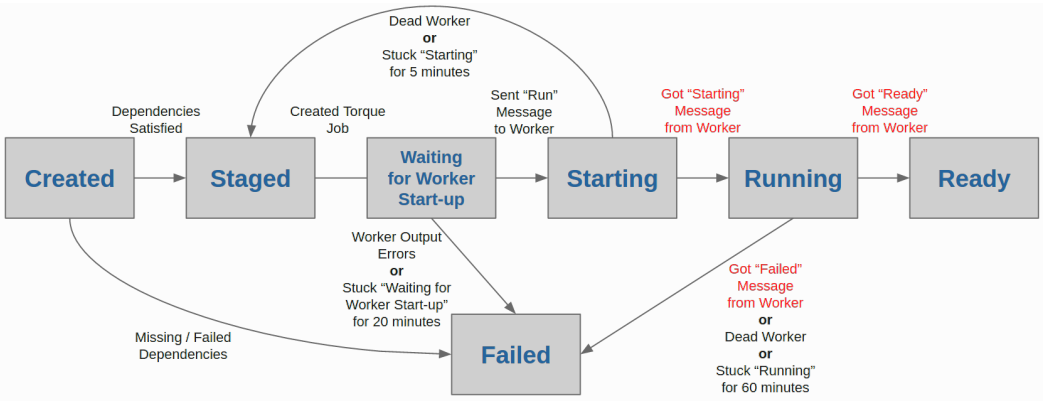

Figure 2: codalab.stanford.edu Scheduling

The codalab.stanford.edu deployment also differs in that the bundle storage file system is accessible from the workers. Thus, workers don't need to download dependencies and upload the runs after they finish.

There are also 2 additional failure scenarios: either the worker outputs some errors when it is starting or the Torque job doesn't start at all. Errors are output into a special directory on the shared file system and read by the bundle manager. The bundle manager also detects if the Torque job just doesn't start after 20 minutes. The bundle is failed in these scenarios and the Torque job deleted.

9.3 User-Owned Workers

One of the main goals of this project was to support having users run their own workers. It is now indeed possible to do so. The owner of a given worker is stored along with other information about the worker in the database. For both the deployments the bundle manager tries to schedule bundles

on user-owned workers first and falls back to CodaLab workers. The state machine and scheduling logic used for user-owned workers is exactly the same as that of the worksheets.codalab.org deployment, even on codalab.stanford.edu.

9.4 Alternative Designs

A few alternative designs were considered.

First, I considered a design that doesn't have a bundle manager and where workers make all the scheduling decisions themselves. However, with this design it is much harder to ensure robustness. Mechanisms, such as timing out bundles stuck in some state, are necessary and it is not clear how to implement these without a central process. Additionally, with this design, it is also not clear how Torque workers would be started.

Second, I considered making the scheduling more intelligent. Currently, the scheduling logic for worksheets.codalab.org doesn't look at the CPU and RAM utilization on the workers, only at the total amount of available. Writing a scheduler that takes them into account in some reasonable way is difficult and simple designs just don't work too well. For example, if the user doesn't specify the amount of RAM needed, what value is used to make scheduling decisions? In that case, the system could either pick some maximum amount and apply it to all runs, or keep track of actual usage. If a maximum is used, then workers would be kept under capacity by a large amount. If actual usage is used, it would be easy for a worker to go over capacity without special logic.

Finally, for the codalab.stanford.edu deployment I considered having a fixed set of workers running all the time, similarly to the worksheets.codalab.org deployment. However, Torque adds a lot of value, such as intelligent scheduling and managing workers, and thus there is good reason to use it. One downside is that it takes more time to start a Torque worker than to start running a bundle on an existing worker.

10 Upgrading the Worker

It is necessary to support upgrading the workers running on the worksheets.codalab.org deployment as well as those owned by a user, since the worker code may change in the future. Upgrading is done via a message that is sent to the worker when its version doesn't agree with the server's version of the worker code. When a worker receives this message, it stops accepting new bundles to run and waits for the currently running bundles to finish. Once they are finished, it downloads the new code from the REST server and restarts.

10.1 Limitations of Design

A limitation of the current design is that if all workers have bundles running when upgrading all workers will stop accepting new bundles to run and the system will not start running any new bundles. If this becomes an issue in the future the design may need to change.

11 Security

Given that we're executing arbitrary user commands, security is of utmost importance. Most of the system's security comes from using Docker. Docker runs user commands in isolated containers that don't have the same priviliges as commands outside Docker.

A few places where it is tricky to ensure security is around dependency paths and symbolic links. All dependency paths, before being mounted as read-only directories, are checked to ensure that they point inside the dependency (e.g., no ".." in the path) and that if they are symbolic links the links point inside the dependency.

Generally, communication between the worker and the REST server is authenticated using a password and all communication is encrypted since the SSL protocol is used, at least with the worksheets.codalab.org deployment. The user ID is part of the worker ID so that a given user can only affect his own workers. All socket IDs are checked to ensure that one worker can't use a socket of another worker.

12 Conclusion

The new worker design is a much more stable and robust design, offering a few new features and making it easier to incorporate new features in the future.