Toil Documentation

Release 3.8.0

UCSC Computational Genomics Lab

Jun 14, 2017

Contents

Toil is an open-source pure-Python workflow engine that lets people write better pipelines. You can:

- Write your workflows in Common Workflow Language (CWL)
- Run workflows on your laptop or on huge commercial clouds such as Amazon Web Services (including the spot market), Microsoft Azure, OpenStack, and Google Compute Engine
- Take advantage of high-performance computing environments with batch systems like GridEngine, Apache Mesos, and Parasol
- Run workflows concurrently at scale using thousands of nodes, managed by Toil's *Autoscaling* capabilities
- Execute workflows efficiently with caching and resource requirement specifications
- Easily link databases and services

Toil is, admittedly, not quite as good as sliced bread, but it's about as close to it as you're gonna get.

Check out our website for a more comprehensive list of Toil's features, read our paper to learn more about what Toil can do in the real world, or jump in and start with the *Installation* section. (Feel free to also join us on GitHub and Gitter.)

CHAPTER 1

Getting Started

Installation

Basic installation

At this time, Toil supports only Python 2.7.x. If that requirement is satisfied then Toil can be easily installed using pip:

```
$ pip install toil
```

Extras

Some optional features, called *extras*, are not included in the basic installation of Toil. To install Toil with all its bells and whistles, run

```
$ pip install toil[aws,mesos,azure,google,encryption,cwl]
```

Here's what each extra provides:

Extra	Description
`aws`	Provides support for storing workflow state in Amazon AWS. This extra has no native dependencies.
`google`	Experimental. Stores workflow state in Google Cloud Storage. This extra has no native dependencies.
`azure`	Stores workflow state in Microsoft Azure Storage. This extra has no native dependencies.

`mesos`	Provides support for running Toil on an Apache Mesos cluster. Note that running Toil on SGE (GridEngine), Parasol, or a single machine does not require an extra. The `mesos` extra requires the following native dependencies:
	• Apache Mesos (Tested with Mesos v1.0.0)
	• *Python headers and static libraries*
	--- 	
	Important: If you want to install Toil with the `mesos` extra in a virtualenv, be sure to create that virtualenv with the `--system-site-packages` flag:
	`$ virtualenv --system-site-packages`
	Otherwise, you'll see something like this:
	`ImportError: No module named mesos.native`

`encryption`	Provides client-side encryption for files stored in the Azure and AWS job stores. This extra requires the following native dependencies: • *Python headers and static libraries* • *libffi headers and library*
`cwl`	Provides support for running workflows written using the Common Workflow Language.

Python headers and static libraries

Only needed for the `mesos` and `encryption` extras. On Ubuntu:

```
$ sudo apt-get install build-essential python-dev
```

On macOS:

```
$ xcode-select --install
```

libffi headers and library

Only needed for the `encryption` extra. On Ubuntu:

```
$ sudo apt-get install libffi-dev
```

On macOS:

```
$ brew install libffi
```

Running Toil workflows

Quickstart: A simple workflow

Starting with Python, a Toil workflow can be run with just three steps.

1. Install Toil (see *Installation*):

```
$ pip install toil
```

2. Copy and paste the following code block into `HelloWorld.py`:

```
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

j = Job.wrapFn(helloWorld, "You did it!")

if __name__=="__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    print Job.Runner.startToil(j, options) #Prints Hello, world!, ...
```

3. Specify a job store and run the workflow like so:

```
$ python HelloWorld.py file:my-job-store
```

Now you have run Toil on the `singleMachine` batch system (the default) using the `file` job store. The job store is a place where intermediate files are written to during the workflow's execution. The `file` job store is a job store that uses the files and directories on a locally-attached filesystem - in this case, a directory called `my-job-store` in the directory that `HelloWorld.py` is run from. (Read more about *The job store interface*.)

Run `python HelloWorld.py --help` to see a complete list of available options.

For something beyond a "Hello, world!" example, refer to *A real-world example*.

Running CWL workflows

The Common Workflow Language (CWL) is an emerging standard for writing workflows that are portable across multiple workflow engines and platforms. To run workflows written using CWL, first ensure that Toil is installed with the `cwl` extra (see *Extras*). This will install the `cwl-runner` and `cwltoil` executables (these are identical - `cwl-runner` is the portable name for the default system CWL runner).

To learn more about CWL, see the CWL User Guide. Toil has nearly full support for the stable v1.0 specification, only lacking the following features:

- Directory inputs and outputs in pipelines. Currently, directory inputs must be enumerated as Files.
- File literals that specify only `contents` to a File without an explicit file name.
- Writable *InitialWorkDirRequirement <http://www.commonwl.org/v1.0/CommandLineTool.html#InitialWorkDirRequirement>* objects. Standard readable inputs do work.
- Complex file inputs – from ExpressionTool or a default value, both of which do not yet get cleanly staged into Toil file management.

To run in local batch mode, provide the CWL file and the input object file:

```
$ cwltoil example.cwl example-job.yml
```

To run in cloud and HPC configurations, you may need to provide additional command line parameters to select and configure the batch system to use.

A real-world example

For a more detailed example and explanation, we've developed a sample pipeline that merge-sorts a temporary file.

1. Download `the example code`.
2. Run it with the default settings:

   ```
   $ python toil-sort-example.py file:jobStore
   ```

3. Run with custom options:

   ```
   $ python toil-sort-example.py file:jobStore --num-lines=5000 --line-length=10 --
   ↪workDir=/tmp/
   ```

The `if __name__ == '__main__'` boilerplate is required to enable Toil to import the job functions defined in the script into the context of a Toil *worker* process. By invoking the script you created the *leader process*. A worker process is a separate process whose sole purpose is to host the execution of one or more jobs defined in that script. When using the single-machine batch system (the default), the worker processes will be running on the same machine as the leader process. With full-fledged batch systems like Mesos the worker processes will typically be started on separate machines. The boilerplate ensures that the pipeline is only started once–on the leader–but not when its job functions are imported and executed on the individual workers.

Typing `python toil-sort-example.py --help` will show the complete list of arguments for the workflow which includes both Toil's and ones defined inside `toil-sort-example.py`. A complete explanation of Toil's arguments can be found in *Command Line Interface*.

Environment variable options

There are several environment variables that affect the way Toil runs.

TOIL_WORKDIR	An absolute path to a directory where Toil will write its temporary files. This directory must exist on each worker node and may be set to a different value on each worker. The `--workDir` command line option overrides this. On Mesos nodes, `TOIL_WORKDIR` generally defaults to the Mesos sandbox, except on CGCloud-provisioned nodes where it defaults to `/var/lib/mesos`. In all other cases, the system's standard temporary directory is used.
TOIL_TEST_TEMP	An absolute path to a directory where Toil tests will write their temporary files. Defaults to the system's standard temporary directory.
TOIL_TEST_INTEGRATIVE	If True, this allows the integration tests to run. Only valid when running the tests from the source directory via `make test` or `make test_parallel`.
TOIL_TEST_EXPERIMENTAL	If True, this allows tests on experimental features to run (such as the Google and Azure) job stores. Only valid when running tests from the source directory via `make test` or `make test_parallel`.
TOIL_APPLIANCE_SELF	The fully qualified reference for the Toil Appliance you wish to use, in the form `REPO/IMAGE:TAG`. `quay.io/ucsc_cgl/toil:3.6.0` and `cket/toil:3.5.0` are both examples of valid options. Note that since Docker defaults to Dockerhub repos, only quay.io repos need to specify their registry.
TOIL_DOCKER_REGISTRY	The URL of the registry of the Toil Appliance image you wish to use. Docker will use Dockerhub by default, but the quay.io registry is also very popular and easily specifiable by settting this option to `quay.io`.
TOIL_DOCKER_NAME	The name of the Toil Appliance image you wish to use. Generally this is simply `toil` but this option is provided to override this, since the image can be built with arbitrary names.
TOIL_AWS_ZONE	The EC2 zone to provision nodes in if using Toil's provisioner.
TOIL_AWS_AMI	ID of the AMI to use in node provisioning. If in doubt, don't set this variable.
TOIL_AWS_NODE_DEBUG	Determines whether to preserve nodes that have failed health checks. If set to `True`, nodes that fail EC2 health checks won't immediately be terminated so they can be examined and the cause of failure determined. If any EC2 nodes are left behind in this manner, the security group will also be left behind by necessity as it cannot be deleted until all associated nodes have been terminated.
TOIL_SLURM_ARGS	Arguments for sbatch for the slurm batch system. Do not pass CPU or memory specifications here. Instead, define resource requirements for the job. There is no default value for this variable.
TOIL_GRIDENGINE_ARGS	Arguments for qsub for the gridengine batch system. Do not pass CPU or memory specifications here. Instead, define resource requirements for the job. There is no default value for this variable.
TOIL_GRIDENGINE_PE	Parallel environment arguments for qsub and for the gridengine batch system. There is no default value for this variable.

Logging

By default, Toil logs a lot of information related to the current environment in addition to messages from the batch system and jobs. This can be configured with the `--logLevel` flag. For example, to only log `CRITICAL` level messages to the screen:

```
$ python toil-sort-examply.py file:jobStore --logLevel=critical
```

This hides most of the information we get from the Toil run. For more detail, we can run the pipeline with `--logLevel=debug` to see a comprehensive output. For more information, see *Logging*.

Error handling and resuming pipelines

With Toil, you can recover gracefully from a bug in your pipeline without losing any progress from successfully-completed jobs. To demonstrate this, let's add a bug to our example code to see how Toil handles a failure and how

we can resume a pipeline after that happens. Add a bad assertion to line 30 of the example (the first line of `down()`):

```
def down(job, input_file_store_id, n, down_checkpoints):
    ...
    assert 1 == 2, "Test error!"
```

When we run the pipeline, Toil will show a detailed failure log with a traceback:

```
$ python toil-sort-example.py file:jobStore
...
---TOIL WORKER OUTPUT LOG---
...
m/j/jobonrSMP    Traceback (most recent call last):
m/j/jobonrSMP      File "toil/src/toil/worker.py", line 340, in main
m/j/jobonrSMP        job._runner(jobGraph=jobGraph, jobStore=jobStore,␣
↪fileStore=fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1270, in _runner
m/j/jobonrSMP        returnValues = self._run(jobGraph, fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1217, in _run
m/j/jobonrSMP        return self.run(fileStore)
m/j/jobonrSMP      File "toil/src/toil/job.py", line 1383, in run
m/j/jobonrSMP        rValue = userFunction(*((self,) + tuple(self._args)), **self._
↪kwargs)
m/j/jobonrSMP      File "toil/example.py", line 30, in down
m/j/jobonrSMP        assert 1 == 2, "Test error!"
m/j/jobonrSMP    AssertionError: Test error!
```

If we try and run the pipeline again, Toil will give us an error message saying that a job store of the same name already exists. By default, in the event of a failure, the job store is preserved so that it can be restarted from its last successful job. We can restart the pipeline by running:

```
$ python toil-sort-example.py file:jobStore --restart
```

We can also change the number of times Toil will attempt to retry a failed job:

```
$ python toil-sort-example.py --retryCount 2 --restart
```

You'll now see Toil attempt to rerun the failed job until it runs out of tries. `--retryCount` is useful for non-systemic errors, like downloading a file that may experience a sporadic interruption, or some other non-deterministic failure.

To successfully restart our pipeline, we can edit our script to comment out line 30, or remove it, and then run

```
$ python toil-sort-example.py --restart
```

The pipeline will run successfully, and the job store will be removed on the pipeline's completion.

Collecting statistics

A Toil pipeline can be run with the `--stats` flag to allows collection of statistics:

```
$ python toil-sort-example.py --stats
```

Once the pipeline finishes, the job store will be left behind, allowing us to get information on the total runtime and stats pertaining to each job function:

```
$ toil stats file:jobStore
...
```

```
Batch System: singleMachine
Default Cores: 1  Default Memory: 2097152K
...
```

Once we're done, we can clean up the job store by running

```
$ toil clean file:jobStore
```

Cloud installation

This section details how to properly set up Toil and its dependencies in various cloud environments.

Amazon Web Services

Toil includes a native AWS provisioner that can be used to start *Autoscaling* clusters. To provision static, non-autoscaling clusters we recommend using CGCloud.

Toil Provisioner

The native Toil provisioner is included in Toil alongside the `[aws]` extra and allows us to spin up a cluster without any external dependencies. It is built around the Toil Appliance, a Docker image that bundles Toil and all its requirements, e.g. Mesos. This makes deployment simple across platforms, and you can even simulate a cluster locally (see *Developing with the Toil Appliance* for details).

When using the Toil provisioner, the appliance image will be automatically chosen based on the pip installed version of Toil on your system. That choice can be overriden by setting the environment variables `TOIL_DOCKER_REGISTRY` and `TOIL_DOCKER_NAME` or `TOIL_APPLIANCE_SELF`. See *Environment variable options* for more information on these variables. If you are developing autoscaling and want to test and build your own appliance have a look at *Developing with the Toil Appliance*.

Using the provisioner to launch a Toil leader instance is simple:

```
$ toil launch-cluster CLUSTER-NAME-HERE --nodeType=t2.micro \
   -z us-west-2a --keyPairName=your-AWS-key-pair-name
```

The cluster name is used to uniquely identify your cluster and will be used to populate the instance's `Name` tag. In addition, the Toil provisioner will automatically tag your cluster with an `Owner` tag that corresponds to your keypair name to facilitate cost tracking.

The `-z` parameter is important since it specifies which EC2 availability zone to launch the cluster in. Alternatively, you can specify this option via the `TOIL_AWS_ZONE` environment variable. This is generally preferable since it lets us avoid repeating the `-z` option for every subsequent cluster command. We will assume this environment variable is set for the rest of the tutorial. Note: the zone is different from an EC2 region. A region corresponds to a geographical area like `us-west-2 (Oregon)`, and availability zones are partitions of this area like `us-west-2a`.

An important caveat to note here is that there is no currently parameter to specify the size of the instance's root volume, which is currently set to 50 Gb. This support will be added soon, but in the mean time instances with ephemeral SSD volumes should be used if > 50 Gb of disk will be needed by any job in the pipeline. See here for a full selection of EC2 instance types.

Once the leader is running, the `ssh-cluster` and `rsync-cluster` utilities can be used to interact with the instance:

```
$ toil rsync-cluster CLUSTER-NAME-HERE \
   ~/localFile :/remoteDestination
```

The most frequent use case for the `rsync-cluster` utility is deploying your Toil script to the Toil leader. Note that the syntax is the same as traditional rsync with the exception of the hostname before the colon. This is not needed in `toil rsync-cluster` since the hostname is automatically determined by Toil.

The last utility provided by the Toil Provisioner is `ssh-cluster` and it can be used as follows:

```
$ toil ssh-cluster CLUSTER-NAME-HERE
```

This will give you a shell on the Toil leader, where you proceed to start off your :ref:Autoscaling run. This shell actually originates from within the Toil leader container, and as such has a couple restrictions involving the use of the `screen` and `tmux` commands. The shell doesn't know that it is a TTY, which prevents it from properly allocating a new screen session. This can be worked around via:

```
$ script
$ screen
```

Simply running `screen` within `script` will get things working properly again.

Finally, you can execute remote commands with the following syntax:

```
$ toil ssh-cluster CLUSTER-NAME-HERE remoteCommand
```

It is not advised that you run your Toil workflow using remote execution like this unless a tool like nohup is used to insure the process does not die if the SSH connection is interrupted.

CGCloud Quickstart

Setting up clusters with CGCloud has the benefit of coming pre-packaged with Toil and Mesos, our preferred batch system for running on AWS.

CGCloud documentation

Users of CGCloud may want to refer to the documentation for CGCloud-core and CGCloud-toil.

---

1. Create and activate a virtualenv:

   ```
   $ virtualenv ~/cgcloud
   $ source ~/cgcloud/bin/activate
   ```

2. Install CGCloud and the CGCloud Toil plugin:

   ```
   $ pip install cgcloud-toil
   ```

3. Add the following to your `~/.profile`, using the appropriate region for your account:

   ```
   export CGCLOUD_ZONE=us-west-2a
   export CGCLOUD_PLUGINS="cgcloud.toil:$CGCLOUD_PLUGINS"
   ```

4. Setup credentials for your AWS account in `~/.aws/credentials`:

   ```
   [default]
   aws_access_key_id=PASTE_YOUR_FOO_ACCESS_KEY_ID_HERE
   ```

```
aws_secret_access_key=PASTE_YOUR_FOO_SECRET_KEY_ID_HERE
region=us-west-2
```

5. Register your SSH key. If you don't have one, create it with `ssh-keygen`:

```
$ cgcloud register-key ~/.ssh/id_rsa.pub
```

6. Create a template *toil-box* which will contain necessary prerequisites:

```
$ cgcloud create -IT toil-box
```

7. Create a small leader/worker cluster:

```
$ cgcloud create-cluster toil -s 2 -t m3.large
```

8. SSH into the leader:

```
$ cgcloud ssh toil-leader
```

At this point, any Toil script can be run on the distributed AWS cluster by following instructions in *Running on AWS*.

Finally, if you wish to tear down the cluster and remove all its data permanently, CGCloud allows you to do so without logging into the AWS web interface:

```
$ cgcloud terminate-cluster toil
```

Azure

Toil comes with a cluster template to facilitate easy deployment of clusters running Toil on Microsoft Azure. The template allows these clusters to be created and managed through the Azure portal. To use the template to set up a Toil Mesos cluster on Azure, use the deploy button above, or open the deploy link in your browser.

For more information, see the cluster template's documentation, or read our walkthrough on azure-walkthrough.

OpenStack

Our group is working to expand distributed cluster support to OpenStack by providing convenient Docker containers to launch Mesos from. Currently, OpenStack nodes can be set up to run Toil in single machine mode by following the *Installation*.

Google Compute Engine

Support for running on Google Cloud is currently experimental. Our group is working to expand distributed cluster support to Google Compute with a cluster provisioning tool based around a Dockerized Mesos setup. Currently, Google Compute Engine nodes can be configured to run Toil in single machine mode by following the *Installation*.

Running in the cloud

Toil jobs can be run on a variety of cloud platforms. Of these, Amazon Web Services is currently the best-supported solution.

On all cloud providers, it is recommended that you run long-running jobs on remote systems using a terminal multiplexer such as screen or tmux.

Screen

Simply type `screen` to open a new `screen` session. Later, type `ctrl-a` and then `d` to disconnect from it, and run `screen -r` to reconnect to it. Commands running under `screen` will continue running even when you are disconnected, allowing you to unplug your laptop and take it home without ending your Toil jobs. See *Toil Provisioner* for complications that can occur when using screen within the Toil Appliance.

Autoscaling

The fastest way to get started with Toil in a cloud environment is by using Toil's autoscaling capabilities to handle node provisioning. You can do this by using the *Toil Provisioner* or CGCloud.

To begin, launch a Toil leader instance using your choice of provisioners.

Once we have our leader instance launched, the only remaining step is to kick off our Toil run with special autoscaling options. Now might be an opportune time to read up on Toil's extensive configuration options by passing `--help` to your toil script invocation.

There are a number of autoscaling specific options, but only 2 options are strictly necessary to enable autoscaling: `--provisioner=aws` and `--nodeType=<>`. These options, respectively, tell Toil that we are running on AWS (currently the only supported autoscaling environment) and which instance type to use for the Toil worker instances.

Preemptability

Toil can run on a heterogenous cluster of both preemptable and non-preemptable nodes. Our preemptable node type can be set by using the `--preemptableNodeType=<>` flag. While individual jobs can each explicitly specify whether or not they should be run on preemptable nodes via the boolean `preemptable` resource requirement, the `--defaultPreemptable` flag will allow jobs without a `preemptable` requirement to run on preemptable machines.

We can set the maximum number of preemptable and non-preemptable nodes via the flags `--maxNodes=<>` and `--maxPreemptableNodes=<>`.

Specify Preemptability Carefully

Ensure that your choices for `--maxNodes=<>` and `--maxPreemptableNodes=<>` make sense for your workflow and won't cause it to hang - if the workflow requires preemptable nodes set `--maxPreemptableNodes` to some non-zero value and if any job requires non-preemptable nodes set `--maxNodes` to some non-zero value.

Finally, the `--preemptableCompensation` flag can be used to handle cases where preemptable nodes may not be available but are required for your workflow.

Using Mesos with Toil on AWS

The mesos master and agent processes bind to the private IP addresses of their EC2 instance, so be sure to use the master's private IP when specifying `--mesosMaster`. Using the public IP will prevent the nodes from properly discovering each other.

Running on AWS

See *Amazon Web Services* to get setup for running on AWS.

Having followed the *Quickstart: A simple workflow* guide, the user can run their `HelloWorld.py` script on a distributed cluster just by modifying the run command. Since our cluster is distributed, we'll use the `aws` job store which uses a combination of one S3 bucket and a couple of SimpleDB domains. This allows all nodes in the cluster access to the job store which would not be possible if we were to use the `file` job store with a locally mounted file system on the leader.

Copy `HelloWorld.py` to the leader node, and run:

```
$ python HelloWorld.py \
       --batchSystem=mesos \
       --mesosMaster=master-private-ip:5050 \
       aws:us-west-2:my-aws-jobstore
```

Alternatively, to run a CWL workflow:

```
$ cwltoil --batchSystem=mesos  \
        --mesosMaster=master-private-ip:5050 \
        --jobStore=aws:us-west-2:my-aws-jobstore \
        example.cwl \
        example-job.yml
```

When running a CWL workflow on AWS, input files can be provided either on the local file system or in S3 buckets using `s3://` URL references. Final output files will be copied to the local file system of the leader node.

Running on Azure

See *Azure* to get setup for running on Azure. This section assumes that you are SSHed into your cluster's leader node.

The Azure templates do not create a shared filesystem; you need to use the `azure` job store for which you need to create an *Azure storage account*. You can store multiple job stores in a single storage account.

To create a new storage account, if you do not already have one:

1. Click here, or navigate to `https://portal.azure.com/#create/Microsoft.StorageAccount` in your browser.
2. If necessary, log into the Microsoft Account that you use for Azure.
3. Fill out the presented form. The *Name* for the account, notably, must be a 3-to-24-character string of letters and lowercase numbers that is globally unique. For *Deployment model*, choose *Resource manager*. For *Resource group*, choose or create a resource group **different than** the one in which you created your cluster. For *Location*, choose the **same** region that you used for your cluster.
4. Press the *Create* button. Wait for your storage account to be created; you should get a notification in the notifications area at the upper right when that is done.

Once you have a storage account, you need to authorize the cluster to access the storage account, by giving it the access key. To do find your storage account's access key:

1. When your storage account has been created, open it up and click the "Settings" icon.
2. In the *Settings* panel, select *Access keys*.
3. Select the text in the *Key1* box and copy it to the clipboard, or use the copy-to-clipboard icon.

You then need to share the key with the cluster. To do this temporarily, for the duration of an SSH or screen session:

1. On the leader node, run `export AZURE_ACCOUNT_KEY="<KEY>"`, replacing `<KEY>` with the access key you copied from the Azure portal.

To do this permanently:

1. On the leader node, run `nano ~/.toilAzureCredentials`.
2. In the editor that opens, navigate with the arrow keys, and give the file the following contents

```
[AzureStorageCredentials]
<accountname>=<accountkey>
```

   Be sure to replace `<accountname>` with the name that you used for your Azure storage account, and `<accountkey>` with the key you obtained above. (If you want, you can have multiple accounts with different keys in this file, by adding multipe lines. If you do this, be sure to leave the `AZURE_ACCOUNT_KEY` environment variable unset.)

3. Press `ctrl-o` to save the file, and `ctrl-x` to exit the editor.

Once that's done, you are now ready to actually execute a job, storing your job store in that Azure storage account. Assuming you followed the *Quickstart: A simple workflow* guide above, you have an Azure storage account created, and you have placed the storage account's access key on the cluster, you can run the `HelloWorld.py` script by doing the following:

1. Place your script on the leader node, either by downloading it from the command line or typing or copying it into a command-line editor.
2. Run the command:

```
$ python HelloWorld.py \
       --batchSystem=mesos \
       --mesosMaster=10.0.0.5:5050 \
       azure:<accountname>:hello-world-001
```

   To run a CWL workflow:

```
$ cwltoil --batchSystem=mesos \
        --mesosMaster=10.0.0.5:5050 \
        --jobStore=azure:<accountname>:hello-world-001 \
        example.cwl \
        example-job.yml
```

   Be sure to replace `<accountname>` with the name of your Azure storage account.

Note that once you run a job with a particular job store name (the part after the account name) in a particular storage account, you cannot re-use that name in that account unless one of the following happens:

1. You are restarting the same job with the `--restart` option.
2. You clean the job store with `toil clean azure:<accountname>:<jobstore>`.
3. You delete all the items created by that job, and the main job store table used by Toil, from the account (destroying all other job stores using the account).
4. The job finishes successfully and cleans itself up.

Running on Open Stack

After setting up Toil on *OpenStack*, Toil scripts can be run by designating a job store location as shown in *Quickstart: A simple workflow*. Be sure to specify a temporary directory that Toil can use to run jobs in with the `--workDir` argument:

```
$ python HelloWorld.py --workDir=/tmp file:jobStore
```

Running on Google Compute Engine

After setting up Toil on *Google Compute Engine*, Toil scripts can be run just by designating a job store location as shown in *Quickstart: A simple workflow*.

If you wish to use the Google Storage job store, install Toil with the `google` extra (*Extras*). Then, create a file named `.boto` with your credentials and some configuration:

```
[Credentials]
gs_access_key_id = KEY_ID
gs_secret_access_key = SECRET_KEY

[Boto]
https_validate_certificates = True

[GSUtil]
content_language = en
default_api_version = 2
```

`gs_access_key_id` and `gs_secret_access_key` can be generated by navigating to your Google Cloud Storage console and clicking on *Settings*. On the *Settings* page, navigate to the *Interoperability* tab and click *Enable interoperability access*. On this page you can now click *Create a new key* to generate an access key and a matching secret. Insert these into their respective places in the `.boto` file and you will be able to use a Google job store when invoking a Toil script, as in the following example:

```
$ python HelloWorld.py google:projectID:jobStore
```

The `projectID` component of the job store argument above refers your Google Cloud Project ID in the Google Cloud Console, and will be visible in the console's banner at the top of the screen. The `jobStore` component is a name of your choosing that you will use to refer to this job store.

CHAPTER 2

User Guide

Command Line Interface

Toil provides many command line options when running a toil script (see *Running Toil workflows*), or using Toil to run a CWL script. Many of these are described below. For most Toil scripts, executing `--help` will show this list of options.

It is also possible to set and manipulate the options described when invoking a Toil workflow from within Python using *toil.job.Job.Runner.getDefaultOptions()*, e.g.:

```
options = Job.Runner.getDefaultOptions("./toilWorkflow") # Get the options object
options.logLevel = "INFO" # Set the log level to the info level.

Job.Runner.startToil(Job(), options) # Run the script
```

Logging

Toil hides stdout and stderr by default except in case of job failure. For more robust logging options (default is INFO), use `--logDebug` or more generally, use `--logLevel=`, which may be set to either `OFF` (or `CRITICAL`), `ERROR`, `WARN` (or `WARNING`), `INFO` or `DEBUG`. Logs can be directed to a file with `--logFile=`.

If large logfiles are a problem, `--maxLogFileSize` (in bytes) can be set as well as `--rotatingLogging`, which prevents logfiles from getting too large.

Stats

The `--stats` argument records statistics about the Toil workflow in the job store. After a Toil run has finished, the entrypoint `toil stats <jobStore>` can be used to return statistics about cpu, memory, job duration, and more. The job store will never be deleted with `--stats`, as it overrides `--clean`.

Cluster Utilities

There are several utilites used for starting and managing a Toil cluster using the AWS provisioner. They make up the *Toil Provisioner*, and they use the `toil launch-cluster`, `toil rsync-cluster`, `toil ssh-cluster`, and `toil destroy-cluster` entry points. For more information, see *Toil Provisioner*

Note: Boto must be configured with AWS credentials before using cluster utilities.

Restart

In the event of failure, Toil can resume the pipeline by adding the argument `--restart` and rerunning the python script. Toil pipelines can even be edited and resumed which is useful for development or troubleshooting.

Clean

If a Toil pipeline didn't finish successfully, or is using a variation of `--clean`, the job store will exist until it is deleted. `toil clean <jobStore>` ensures that all artifacts associated with a job store are removed. This is particularly useful for deleting AWS job stores, which reserves an SDB domain as well as an S3 bucket.

The deletion of the job store can be modified by the `--clean` argument, and may be set to `always`, `onError`, `never`, or `onSuccess` (default).

Temporary directories where jobs are running can also be saved from deletion using the `--cleanWorkDir`, which has the same options as `--clean`. This option should only be run when debugging, as intermediate jobs will fill up disk space.

Batch system

Toil supports several different batch systems using the `--batchSystem` argument. More information in the *The batch system interface*.

Default cores, disk, and memory

Toil uses resource requirements to intelligently schedule jobs. The defaults for cores (1), disk (2G), and memory (2G), can all be changed using `--defaultCores`, `--defaultDisk`, and `--defaultMemory`. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

Job store

Running toil scripts has one required positional argument: the job store. The default job store is just a path to where the user would like the job store to be created. To use the *quick start* example, if you're on a node that has a large **/scratch** volume, you can specify the jobstore be created there by executing: `python HelloWorld.py /scratch/my-job-store`, or more explicitly, `python HelloWorld.py file:/scratch/my-job-store`. Toil uses the colon as way to explicitly name what type of job store the user would like. The other job store types are AWS (`aws:region-here:job-store-name`), Azure (`azure:account-name-here:job-store-name`), and the experimental Google job store (`google:projectID-here:job-store-name`). More information on these job store can be found at *Running in the cloud*. Different types of job store options can be looked up in *The job store interface*.

Miscellaneous

Here are some additional useful arguments that don't fit into another category.

- `--workDir` sets the location where temporary directories are created for running jobs.
- `--retryCount` sets the number of times to retry a job in case of failure. Useful for non-systemic failures like HTTP requests.
- `--sseKey` accepts a path to a 32-byte key that is used for server-side encryption when using the AWS job store.
- `--cseKey` accepts a path to a 256-bit key to be used for client-side encryption on Azure job store.
- `--setEnv <NAME=VALUE>` sets an environment variable early on in the worker

For implementation-specific flags for schedulers like timelimits, queues, accounts, etc.. An environment variable can be defined before launching the Job, i.e:

```
export TOIL_SLURM_ARGS="-t 1:00:00 -q fatq"
```

Running Workflows with Services

Toil supports jobs, or clusters of jobs, that run as *services* (see *Services*) to other *accessor* jobs. Example services include server databases or Apache Spark Clusters. As service jobs exist to provide services to accessor jobs their runtime is dependent on the concurrent running of their accessor jobs. The dependencies between services and their accessor jobs can create potential deadlock scenarios, where the running of the workflow hangs because only service jobs are being run and their accessor jobs can not be scheduled because of too limited resources to run both simultaneously. To cope with this situation Toil attempts to schedule services and accessors intelligently, however to avoid a deadlock with workflows running service jobs it is advisable to use the following parameters:

- `--maxServiceJobs` The maximum number of service jobs that can be run concurrently, excluding service jobs running on preemptable nodes.
- `--maxPreemptableServiceJobs` The maximum number of service jobs that can run concurrently on preemptable nodes.

Specifying these parameters so that at a maximum cluster size there will be sufficient resources to run accessors in addition to services will ensure that such a deadlock can not occur.

If too low a limit is specified then a deadlock can occur in which toil can not schedule sufficient service jobs concurrently to complete the workflow. Toil will detect this situation if it occurs and throw a `toil.leader.DeadlockException` exception. Increasing the cluster size and these limits will resolve the issue.

Developing a workflow

This tutorial walks through the features of Toil necessary for developing a workflow using the Toil Python API.

Scripting quick start

To begin, consider this short toil script which illustrates defining a workflow:

```
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message
```

```
j = Job.wrapFn(helloWorld, "woot")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflow")
    print Job.Runner.startToil(j, options) #Prints Hello, world!, ...
```

The workflow consists of a single job. The resource requirements for that job are (optionally) specified by keyword arguments (memory, cores, disk). The script is run using *toil.job.Job.Runner.getDefaultOptions()*. Below we explain the components of this code in detail.

Job basics

The atomic unit of work in a Toil workflow is a *Job*. User scripts inherit from this base class to define units of work. For example, here is a more long-winded class-based version of the job in the quick start example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message
```

In the example a class, HelloWorld, is defined. The constructor requests 2 gigabytes of memory, 2 cores and 3 gigabytes of local disk to complete the work.

The *toil.job.Job.run()* method is the function the user overrides to get work done. Here it just logs a message using *toil.fileStore.FileStore.logToMaster()*, which will be registered in the log output of the leader process of the workflow.

Invoking a workflow

We can add to the previous example to turn it into a complete workflow by adding the necessary function calls to create an instance of HelloWorld and to run this as a workflow containing a single job. This uses the *toil.job.Job.Runner* class, which is used to start and resume Toil workflows. For example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    print Job.Runner.startToil(HelloWorld("woot"), options)
```

Alternatively, the more powerful *toil.common.Toil* class can be used to run and resume workflows. It is used as a context manager and allows for preliminary setup, such as staging of files into the job store on the leader node. An

instance of the class is initialized by specifying an options object. The actual workflow is then invoked by calling the *toil.common.Toil.start()* method, passing the root job of the workflow, or, if a workflow is being restarted, *toil.common.Toil.restart()* should be used. Note that the context manager should have explicit if else branches addressing restart and non restart cases. The boolean value for these if else blocks is toil.options.restart.

For example:

```
from toil.job import Job
from toil.common import Toil

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        fileStore.logToMaster("Hello, world!, I have a message: %s"
                              % self.message)
if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"

    with Toil(options) as toil:
        if not toil.options.restart:
            job = HelloWorld("Smitty Werbenmanjensen, he was #1")
            toil.start(job)
        else:
            toil.restart()
```

The call to *toil.job.Job.Runner.getDefaultOptions()* creates a set of default options for the workflow. The only argument is a description of how to store the workflow's state in what we call a *job-store*. Here the job-store is contained in a directory within the current working directory called "toilWorkflowRun". Alternatively this string can encode other ways to store the necessary state, e.g. an S3 bucket or Azure object store location. By default the job-store is deleted if the workflow completes successfully.

The workflow is executed in the final line, which creates an instance of HelloWorld and runs it as a workflow. Note all Toil workflows start from a single starting job, referred to as the *root* job. The return value of the root job is returned as the result of the completed workflow (see promises below to see how this is a useful feature!).

Specifying arguments via the command line

To allow command line control of the options we can use the *toil.job.Job.Runner.getDefaultArgumentParser()* method to create a *argparse.ArgumentParser* object which can be used to parse command line options for a Toil script. For example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message

if __name__=="__main__":
    parser = Job.Runner.getDefaultArgumentParser()
```

```
    options = parser.parse_args()
    print Job.Runner.startToil(HelloWorld("woot"), options)
```

Creates a fully fledged script with all the options Toil exposed as command line arguments. Running this script with "–help" will print the full list of options.

Alternatively an existing `argparse.ArgumentParser` or `optparse.OptionParser` object can have Toil script command line options added to it with the *`toil.job.Job.Runner.addToilOptions()`* method.

Resuming a workflow

In the event that a workflow fails, either because of programmatic error within the jobs being run, or because of node failure, the workflow can be resumed. Workflows can only not be reliably resumed if the job-store itself becomes corrupt.

Critical to resumption is that jobs can be rerun, even if they have apparently completed successfully. Put succinctly, a user defined job should not corrupt its input arguments. That way, regardless of node, network or leader failure the job can be restarted and the workflow resumed.

To resume a workflow specify the "restart" option in the options object passed to *`toil.job.Job.Runner.startToil()`*. If node failures are expected it can also be useful to use the integer "retryCount" option, which will attempt to rerun a job retryCount number of times before marking it fully failed.

In the common scenario that a small subset of jobs fail (including retry attempts) within a workflow Toil will continue to run other jobs until it can do no more, at which point *`toil.job.Job.Runner.startToil()`* will raise a `toil.leader.FailedJobsException` exception. Typically at this point the user can decide to fix the script and resume the workflow or delete the job-store manually and rerun the complete workflow.

Functions and job functions

Defining jobs by creating class definitions generally involves the boilerplate of creating a constructor. To avoid this the classes *`toil.job.FunctionWrappingJob`* and `toil.job.JobFunctionWrappingTarget` allow functions to be directly converted to jobs. For example, the quick start example (repeated here):

```
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

j = Job.wrapFn(helloWorld, "woot")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    print Job.Runner.startToil(j, options)
```

Is equivalent to the previous example, but using a function to define the job.

The function call:

```
Job.wrapFn(helloWorld, "woot")
```

Creates the instance of the `toil.job.FunctionWrappingTarget` that wraps the function.

The keyword arguments *memory*, *cores* and *disk* allow resource requirements to be specified as before. Even if they are not included as keyword arguments within a function header they can be passed as arguments when wrapping a function as a job and will be used to specify resource requirements.

We can also use the function wrapping syntax to a *job function*, a function whose first argument is a reference to the wrapping job. Just like a *self* argument in a class, this allows access to the methods of the wrapping job, see `toil.job.JobFunctionWrappingTarget`. For example:

```
from toil.job import Job

def helloWorld(job, message):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the toil.fileStore.FileStore class

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    print Job.Runner.startToil(Job.wrapJobFn(helloWorld, "woot"), options)
```

Here `helloWorld()` is a job function. It accesses the *toil.fileStore.FileStore* attribute of the job to log a message that will be printed to the output console. Here the only subtle difference to note is the line:

```
Job.Runner.startToil(Job.wrapJobFn(helloWorld, "woot"), options)
```

Which uses the function *toil.job.Job.wrapJobFn()* to wrap the job function instead of *toil.job.Job.wrapFn()* which wraps a vanilla function.

Workflows with multiple jobs

A *parent* job can have *child* jobs and *follow-on* jobs. These relationships are specified by methods of the job class, e.g. *toil.job.Job.addChild()* and *toil.job.Job.addFollowOn()*.

Considering a set of jobs the nodes in a job graph and the child and follow-on relationships the directed edges of the graph, we say that a job B that is on a directed path of child/follow-on edges from a job `A` in the job graph is a *successor* of `A`, similarly `A` is a *predecessor* of `B`.

A parent job's child jobs are run directly after the parent job has completed, and in parallel. The follow-on jobs of a job are run after its child jobs and their successors have completed. They are also run in parallel. Follow-ons allow the easy specification of cleanup tasks that happen after a set of parallel child tasks. The following shows a simple example that uses the earlier `helloWorld()` job function:

```
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the toil.fileStore.FileStore class

j1 = Job.wrapJobFn(helloWorld, "first")
j2 = Job.wrapJobFn(helloWorld, "second or third")
j3 = Job.wrapJobFn(helloWorld, "second or third")
j4 = Job.wrapJobFn(helloWorld, "last")
j1.addChild(j2)
j1.addChild(j3)
j1.addFollowOn(j4)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)
```

In the example four jobs are created, first `j1` is run, then `j2` and `j3` are run in parallel as children of `j1`, finally `j4` is run as a follow-on of `j1`.

There are multiple short hand functions to achieve the same workflow, for example:

```
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the toil.fileStore.FileStore class

j1 = Job.wrapJobFn(helloWorld, "first")
j2 = j1.addChildJobFn(helloWorld, "second or third")
j3 = j1.addChildJobFn(helloWorld, "second or third")
j4 = j1.addFollowOnJobFn(helloWorld, "last")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)
```

Equivalently defines the workflow, where the functions *toil.job.Job.addChildJobFn()* and *toil.job.Job.addFollowOnJobFn()* are used to create job functions as children or follow-ons of an earlier job.

Jobs graphs are not limited to trees, and can express arbitrary directed acylic graphs. For a precise definition of legal graphs see *toil.job.Job.checkJobGraphForDeadlocks()*. The previous example could be specified as a DAG as follows:

```
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the toil.fileStore.FileStore class

j1 = Job.wrapJobFn(helloWorld, "first")
j2 = j1.addChildJobFn(helloWorld, "second or third")
j3 = j1.addChildJobFn(helloWorld, "second or third")
j4 = j2.addChildJobFn(helloWorld, "last")
j3.addChild(j4)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)
```

Note the use of an extra child edge to make `j4` a child of both `j2` and `j3`.

Dynamic job creation

The previous examples show a workflow being defined outside of a job. However, Toil also allows jobs to be created dynamically within jobs. For example:

```
from toil.job import Job

def binaryStringFn(job, depth, message=""):
    if depth > 0:
```

```
        job.addChildJobFn(binaryStringFn, depth-1, message + "0")
        job.addChildJobFn(binaryStringFn, depth-1, message + "1")
    else:
        job.fileStore.logToMaster("Binary string: %s" % message)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(Job.wrapJobFn(binaryStringFn, depth=5), options)
```

The job function `binaryStringFn` logs all possible binary strings of length `n` (here `n=5`), creating a total of `2^(n+2) - 1` jobs dynamically and recursively. Static and dynamic creation of jobs can be mixed in a Toil workflow, with jobs defined within a job or job function being created at run time.

Promises

The previous example of dynamic job creation shows variables from a parent job being passed to a child job. Such forward variable passing is naturally specified by recursive invocation of successor jobs within parent jobs. This can also be achieved statically by passing around references to the return variables of jobs. In Toil this is achieved with promises, as illustrated in the following example:

```
from toil.job import Job

def fn(job, i):
    job.fileStore.logToMaster("i is: %s" % i, level=100)
    return i+1

j1 = Job.wrapJobFn(fn, 1)
j2 = j1.addChildJobFn(fn, j1.rv())
j3 = j1.addFollowOnJobFn(fn, j2.rv())

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)
```

Running this workflow results in three log messages from the jobs: `i is 1` from `j1`, `i is 2` from `j2` and `i is 3` from `j3`.

The return value from the first job is *promised* to the second job by the call to *`toil.job.Job.rv()`* in the line:

```
j2 = j1.addChildFn(fn, j1.rv())
```

The value of `j1.rv()` is a *promise*, rather than the actual return value of the function, because `j1` for the given input has at that point not been evaluated. A promise (*`toil.job.Promise`*) is essentially a pointer to for the return value that is replaced by the actual return value once it has been evaluated. Therefore, when `j2` is run the promise becomes 2.

Promises also support indexing of return values:

```
def parent(job):
    indexable = Job.wrapJobFn(fn)
    job.addChild(indexable)
    job.addFollowOnFn(raiseWrap, indexable.rv(2))

def raiseWrap(arg):
```

```
    raise RuntimeError(arg) # raises "2"

def fn(job):
    return (0, 1, 2, 3)
```

Promises can be quite useful. For example, we can combine dynamic job creation with promises to achieve a job creation process that mimics the functional patterns possible in many programming languages:

```
from toil.job import Job

def binaryStrings(job, message="", depth):
    if depth > 0:
        s = [ job.addChildJobFn(binaryStrings, message + "0",
                                depth-1).rv(),
              job.addChildJobFn(binaryStrings, message + "1",
                                depth-1).rv() ]
        return job.addFollowOnFn(merge, s).rv()
    return [message]

def merge(strings):
    return strings[0] + strings[1]

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    l = Job.Runner.startToil(Job.wrapJobFn(binaryStrings, depth=5), options)
    print l #Prints a list of all binary strings of length 5
```

The return value `l` of the workflow is a list of all binary strings of length 10, computed recursively. Although a toy example, it demonstrates how closely Toil workflows can mimic typical programming patterns.

Promised Requirements

Promised requirements are a special case of *Promises* that allow a job's return value to be used as another job's resource requirements.

This is useful when, for example, a job's storage requirement is determined by a file staged to the job store by an earlier job:

```
from toil.job import Job, PromisedRequirement
from toil.common import Toil
import os

def parentJob(job):
    downloadJob = Job.wrapJobFn(stageFn, "File://"+os.path.realpath(__file__),␣
↪cores=0.1, memory='32M', disk='1M')
    job.addChild(downloadJob)

    analysis = Job.wrapJobFn(analysisJob, fileStoreID=downloadJob.rv(0),
                             disk=PromisedRequirement(downloadJob.rv(1)))
    job.addFollowOn(analysis)

def stageFn(job, url, cores=1):
    importedFile = job.fileStore.importFile(url)
    return importedFile, importedFile.size

def analysisJob(job, fileStoreID, cores=2):
    # now do some analysis on the file
```

```
    pass

if __name__ == "__main__":
    with Toil(Job.Runner.getDefaultOptions("./toilWorkflowRun")) as toil:
        toil.start(Job.wrapJobFn(parentJob))
```

Note that this also makes use of the `size` attribute of the *FileID* object. This promised requirements mechanism can also be used in combination with an aggregator for multiple jobs' output values:

```
def parentJob(job):
    aggregator = []
    for fileNum in range(0,10):
        downloadJob = Job.wrapJobFn(stageFn, "File://"+os.path.realpath(__file__),␣
↪cores=0.1, memory='32M', disk='1M')
        job.addChild(downloadJob)
        aggregator.append(downloadJob)

    analysis = Job.wrapJobFn(analysisJob, fileStoreID=downloadJob.rv(0),
                             disk=PromisedRequirement(lambda xs: sum(xs), [j.rv(1)␣
↪for j in aggregator]))
    job.addFollowOn(analysis)
```

Limitations

Just like regular promises, the return value must be determined prior to scheduling any job that depends on the return value. In our example above, notice how the dependant jobs were follow ons to the parent while promising jobs are children of the parent. This ordering ensures that all promises are properly fulfilled.

FileID

This object is a small wrapper around Python's builtin string class. It is used to represent a file's ID in the file store, and has a `size` attribute that is the file's size in bytes. This object is returned by `importFile` and `writeGlobalFile`.

Managing files within a workflow

It is frequently the case that a workflow will want to create files, both persistent and temporary, during its run. The *toil.fileStore.FileStore* class is used by jobs to manage these files in a manner that guarantees cleanup and resumption on failure.

The *toil.job.Job.run()* method has a file store instance as an argument. The following example shows how this can be used to create temporary files that persist for the length of the job, be placed in a specified local disk of the node and that will be cleaned up, regardless of failure, when the job finishes:

```
from toil.job import Job

class LocalFileStoreJob(Job):
    def run(self, fileStore):
        scratchDir = fileStore.getLocalTempDir() #Create a temporary
        # directory safely within the allocated disk space
        # reserved for the job.

        scratchFile = fileStore.getLocalTempFile() #Similarly
        # create a temporary file.
```

```
if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    #Create an instance of FooJob which will
    # have at least 10 gigabytes of storage space.
    j = LocalFileStoreJob(disk="10G")
    #Run the workflow
    Job.Runner.startToil(j, options)
```

Job functions can also access the file store for the job. The equivalent of the `LocalFileStoreJob` class is:

```
def localFileStoreJobFn(job):
    scratchDir = job.fileStore.getLocalTempDir()
    scratchFile = job.fileStore.getLocalTempFile()
```

Note that the `fileStore` attribute is accessed as an attribute of the `job` argument.

In addition to temporary files that exist for the duration of a job, the file store allows the creation of files in a *global* store, which persists during the workflow and are globally accessible (hence the name) between jobs. For example:

```
from toil.job import Job
import os

def globalFileStoreJobFn(job):
    job.fileStore.logToMaster("The following example exercises all the"
                              " methods provided by the"
                              " toil.fileStore.FileStore class")

    scratchFile = job.fileStore.getLocalTempFile() # Create a local
    # temporary file.

    with open(scratchFile, 'w') as fH: # Write something in the
        # scratch file.
        fH.write("What a tangled web we weave")

    # Write a copy of the file into the file-store;
    # fileID is the key that can be used to retrieve the file.
    fileID = job.fileStore.writeGlobalFile(scratchFile) #This write
    # is asynchronous by default

    # Write another file using a stream; fileID2 is the
    # key for this second file.
    with job.fileStore.writeGlobalFileStream(cleanup=True) as (fH, fileID2):
        fH.write("Out brief candle")

    # Now read the first file; scratchFile2 is a local copy of the file
    # that is read only by default.
    scratchFile2 = job.fileStore.readGlobalFile(fileID)

    # Read the second file to a desired location: scratchFile3.
    scratchFile3 = os.path.join(job.fileStore.getLocalTempDir(), "foo.txt")
    job.fileStore.readGlobalFile(fileID, userPath=scratchFile3)

    # Read the second file again using a stream.
    with job.fileStore.readGlobalFileStream(fileID2) as fH:
        print fH.read() #This prints "Out brief candle"

    # Delete the first file from the global file-store.
```

```
    job.fileStore.deleteGlobalFile(fileID)

    # It is unnecessary to delete the file keyed by fileID2
    # because we used the cleanup flag, which removes the file after this
    # job and all its successors have run (if the file still exists)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(Job.wrapJobFn(globalFileStoreJobFn), options)
```

The example demonstrates the global read, write and delete functionality of the file-store, using both local copies of the files and streams to read and write the files. It covers all the methods provided by the file store interface.

What is obvious is that the file-store provides no functionality to update an existing "global" file, meaning that files are, barring deletion, immutable. Also worth noting is that there is no file system hierarchy for files in the global file store. These limitations allow us to fairly easily support different object stores and to use caching to limit the amount of network file transfer between jobs.

Staging of files into the job store

External files can be imported into or exported out of the job store prior to running a workflow when the *toil.common.Toil* context manager is used on the leader. The context manager provides methods `toil.common.Toil.importFile()`, and `toil.common.Toil.exportFile()` for this purpose. The destination and source locations of such files are described with URLs passed to the two methods. A list of the currently supported URLs can be found at *toil.jobStores.abstractJobStore.AbstractJobStore.importFile()*. To import an external file into the job store as a shared file, pass the optional `sharedFileName` parameter to that method.

If a workflow fails for any reason an imported file acts as any other file in the job store. If the workflow was configured such that it not be cleaned up on a failed run, the file will persist in the job store and needs not be staged again when the workflow is resumed.

Example:

```
from toil.common import Toil
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, inputFileID):
        Job.__init__(self,  memory="2G", cores=2, disk="3G")
        self.inputFileID = inputFileID

    with fileStore.readGlobalFileStream(self.inputFileID) as fi:
        with fileStore.writeGlobalFileStream() as (fo, outputFileID):
            fo.write(fi.read() + 'World!')
        return outputFileID


if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"


    with Toil(options) as toil:
        if not toil.options.restart:
            inputFileID = toil.importFile('file:///some/local/path')
            outputFileID = toil.start(HelloWorld(inputFileID))
```

```
        else:
            outputFileID = toil.restart()

        toil.exportFile(outputFileID, 'file:///some/other/local/path')
```

Using Docker containers in Toil

Docker containers are commonly used with Toil. The combination of Toil and Docker allows for pipelines to be fully portable between any platform that has both Toil and Docker installed. Docker eliminates the need for the user to do any other tool installation or environment setup.

In order to use Docker containers with Toil, Docker must be installed on all workers of the cluster. Instructions for installing Docker can be found on the Docker website.

When using CGCloud or Toil-based autoscaling, Docker will be automatically set up on the cluster's worker nodes, so no additional installation steps are necessary. Further information on using Toil-based autoscaling can be found in the *Autoscaling* documentation.

In order to use docker containers in a Toil workflow, the container can be built locally or downloaded in real time from an online docker repository like Quay. If the container is not in a repository, the container's layers must be accessible on each node of the cluster.

When invoking docker containers from within a Toil workflow, it is strongly recommended that you use `dockerCall()`, a toil job function provided in `toil.lib.docker`. `dockerCall` provides a layer of abstraction over using the `subprocess` module to call Docker directly, and provides container cleanup on job failure. When docker containers are run without this feature, failed jobs can result in resource leaks.

In order to use `dockerCall`, your installation of Docker must be set up to run without `sudo`. Instructions for setting this up can be found here.

An example of a basic `dockerCall` is below:

> **dockerCall(job=job,** tool='quay.io/ucsc_cgl/bwa', work_dir=job.fileStore.getLocalTempDir(), parameters=['index', '/data/reference.fa'])

`dockerCall` can also be added to workflows like any other job function:

> from toil.job import Job
>
> **align = Job.wrapJobFn(dockerCall,** tool='quay.io/ucsc_cgl/bwa', work_dir=job.fileStore.getLocalTempDir(), parameters=['index', '/data/reference.fa']))
>
> **if __name__=="__main__":** options = Job.Runner.getDefaultOptions("./toilWorkflowRun") options.logLevel = "INFO" Job.Runner.startToil(align, options)

cgl-docker-lib contains `dockerCall`-compatible Dockerized tools that are commonly used in bioinformatics analysis.

The documentation provides guidelines for developing your own Docker containers that can be used with Toil and `dockerCall`. In order for a container to be compatible with `dockerCall`, it must have an `ENTRYPOINT` set to a wrapper script, as described in cgl-docker-lib containerization standards. Alternately, the entrypoint to the container can be set using the docker option `--entrypoint`. The container should be runnable directly with Docker as:

```
$ docker run <docker parameters> <tool name> <tool parameters>
```

For example:

```
$ docker run -d quay.io/ucsc-cgl/bwa -s -o /data/aligned /data/ref.fa
```

Services

It is sometimes desirable to run *services*, such as a database or server, concurrently with a workflow. The *toil.job.Job.Service* class provides a simple mechanism for spawning such a service within a Toil workflow, allowing precise specification of the start and end time of the service, and providing start and end methods to use for initialization and cleanup. The following simple, conceptual example illustrates how services work:

```
from toil.job import Job

class DemoService(Job.Service):

    def start(self, fileStore):
        # Start up a database/service here
        return "loginCredentials" # Return a value that enables another
        # process to connect to the database

    def check(self):
        # A function that if it returns False causes the service to quit
        # If it raises an exception the service is killed and an error is reported
        return True

    def stop(self, fileStore):
        # Cleanup the database here
        pass

j = Job()
s = DemoService()
loginCredentialsPromise = j.addService(s)

def dbFn(loginCredentials):
    # Use the login credentials returned from the service's start method
    # to connect to the service
    pass

j.addChildFn(dbFn, loginCredentialsPromise)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j, options)
```

In this example the DemoService starts a database in the start method, returning an object from the start method indicating how a client job would access the database. The service's stop method cleans up the database, while the service's check method is polled periodically to check the service is alive.

A DemoService instance is added as a service of the root job `j`, with resource requirements specified. The return value from *toil.job.Job.addService()* is a promise to the return value of the service's start method. When the promised is fulfilled it will represent how to connect to the database. The promise is passed to a child job of `j`, which uses it to make a database connection. The services of a job are started before any of its successors have been run and stopped after all the successors of the job have completed successfully.

Multiple services can be created per job, all run in parallel. Additionally, services can define sub-services using `toil.job.Job.Service.addChild()`. This allows complex networks of services to be created, e.g. Apache Spark clusters, within a workflow.

Checkpoints

Services complicate resuming a workflow after failure, because they can create complex dependencies between jobs. For example, consider a service that provides a database that multiple jobs update. If the database service fails and loses state, it is not clear that just restarting the service will allow the workflow to be resumed, because jobs that created that state may have already finished. To get around this problem Toil supports *checkpoint* jobs, specified as the boolean keyword argument `checkpoint` to a job or wrapped function, e.g.:

```
j = Job(checkpoint=True)
```

A checkpoint job is rerun if one or more of its successors fails its retry attempts, until it itself has exhausted its retry attempts. Upon restarting a checkpoint job all its existing successors are first deleted, and then the job is rerun to define new successors. By checkpointing a job that defines a service, upon failure of the service the database and the jobs that access the service can be redefined and rerun.

To make the implementation of checkpoint jobs simple, a job can only be a checkpoint if when first defined it has no successors, i.e. it can only define successors within its run method.

Encapsulation

Let `A` be a root job potentially with children and follow-ons. Without an encapsulated job the simplest way to specify a job `B` which runs after `A` and all its successors is to create a parent of `A`, call it `Ap`, and then make `B` a follow-on of `Ap`. e.g.:

```
from toil.job import Job

# A is a job with children and follow-ons, for example:
A = Job()
A.addChild(Job())
A.addFollowOn(Job())

# B is a job which needs to run after A and its successors
B = Job()

# The way to do this without encapsulation is to make a
# parent of A, Ap, and make B a follow-on of Ap.
Ap = Job()
Ap.addChild(A)
Ap.addFollowOn(B)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(Ap, options)
```

An *encapsulated job* `E(A)` of `A` saves making `Ap`, instead we can write:

```
from toil.job import Job

# A
A = Job()
A.addChild(Job())
A.addFollowOn(Job())

#Encapsulate A
A = A.encapsulate()
```

```
# B is a job which needs to run after A and its successors
B = Job()

# With encapsulation A and its successor subgraph appear
# to be a single job, hence:
A.addChild(B)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(A, options)
```

Note the call to *toil.job.Job.encapsulate()* creates the `toil.job.Job.EncapsulatedJob`.

Depending on Toil

If you are packing your workflow(s) as a pip-installable distribution on PyPI, you might be tempted to declare Toil as a dependency in your `setup.py`, via the `install_requires` keyword argument to `setup()`. Unfortunately, this does not work, for two reasons: For one, Toil uses Setuptools' *extra* mechanism to manage its own optional dependencies. If you explicitly declared a dependency on Toil, you would have to hard-code a particular combination of extras (or no extras at all), robbing the user of the choice what Toil extras to install. Secondly, and more importantly, declaring a dependency on Toil would only lead to Toil being installed on the leader node of a cluster, but not the worker nodes. Hot-deployment does not work here because Toil cannot hot-deploy itself, the classic "Which came first, chicken or egg?" problem.

In other words, you shouldn't explicitly depend on Toil. Document the dependency instead (as in "This workflow needs Toil version X.Y.Z to be installed") and optionally add a version check to your `setup.py`. Refer to the `check_version()` function in the `toil-lib` project's setup.py for an example. Alternatively, you can also just depend on `toil-lib` and you'll get that check for free.

If your workflow depends on a dependency of Toil, e.g. `bd2k-python-lib`, consider not making that dependency explicit either. If you do, you risk a version conflict between your project and Toil. The `pip` utility may silently ignore that conflict, breaking either Toil or your workflow. It is safest to simply assume that Toil installs that dependency for you. The only downside is that you are locked into the exact version of that dependency that Toil declares. But such is life with Python, which, unlike Java, has no means of dependencies belonging to different software components within the same process, and whose favored software distribution utility is incapable of properly resolving overlapping dependencies and detecting conflicts.

Best practices for Dockerizing Toil workflows

Computational Genomics Lab's Dockstore based production system provides workflow authors a way to run Dockerized versions of their pipeline in an automated, scalable fashion. To be compatible with this system of a workflow should meet the following requirements. In addition to the Docker container, a common workflow language descriptor file is needed. For inputs:

- Only command line arguments should be used for configuring the workflow. If the workflow relies on a configuration file, like Toil-RNAseq or ProTECT, a wrapper script inside the Docker container can be used to parse the CLI and generate the necessary configuration file.
- All inputs to the pipeline should be explicitly enumerated rather than implicit. For example, don't rely on one FASTQ read's path to discover the location of its pair. This is necessary since all inputs are mapped to their own isolated directories when the Docker is called via Dockstore.
- All inputs must be documented in the CWL descriptor file. Examples of this file can be seen in both Toil-RNAseq and ProTECT.

For outputs:

- All outputs should be written to a local path rather than S3.
- Take care to package outputs in a local and user-friendly way. For example, don't tar up all output if there are specific files that will care to see individually.
- All output file names should be deterministic and predictable. For example, don't prepend the name of an output file with PASS/FAIL depending on the outcome of the pipeline.
- All outputs must be documented in the CWL descriptor file. Examples of this file can be seen in both Toil-RNAseq and ProTECT.

Deploying a workflow

If a Toil workflow is run on a single machine (that is, single machine mode), there is nothing special you need to do. You change into the directory containing your user script and invoke it like any Python script:

```
$ cd my_project
$ ls
userScript.py ...
$ ./userScript.py ...
```

This assumes that your script has the executable permission bit set and contains a *shebang*, i.e. a line of the form

```
#!/usr/bin/env python
```

Alternatively, the shebang can be omitted and the script invoked as a module via

```
$ python -m userScript
```

in which case the executable permission is not required either. Both are common methods for invoking Python scripts.

The script can have dependencies, as long as those are installed on the machine, either globally, in a user-specific location or in a virtualenv. In the latter case, the virtualenv must of course be active when you run the user script.

If, however, you want to run your workflow in a distributed environment, on multiple worker machines, either in the cloud or on a bare-metal cluster, your script needs to be made available to those other machines. If your script imports other modules, those modules also need to be made available on the workers. Toil can automatically do that for you, with a little help on your part. We call this feature *hot-deployment* of a workflow.

Let's first examine various scenarios of hot-deploying a workflow and then take a look at *deploying Toil*, which, as we'll see shortly cannot be hot-deployed. Lastly we'll deal with the issue of declaring *Toil as a dependency* of a workflow that is packaged as a setuptools distribution.

Hot-deploying Toil

Toil can be easily deployed to a remote host, given that both Python and Toil are present. The first order of business after copying your workflow to each host is to create and activate a virtualenv:

```
$ virtualenv --system-site-packages venv
$ . venv/bin/activate
```

Note that the virtualenv was created with the `--system-site-packages` option, which ensures that globally-installed packages are accessible inside the virtualenv. This is necessary as Toil and its dependencies must be installed globally.

From here, you can install your project and its dependencies:

```
$ tree
.
- util
|    - __init__.py
|    - sort
|        - __init__.py
|        - quick.py
- workflow
     - __init__.py
     - main.py

3 directories, 5 files
$ pip install fairydust
$ cp -R workflow util venv/lib/python2.7/site-packages
```

Ideally, your project would have a `setup.py` file (see setuptools) which streamlines the installation process:

```
$ tree
.
- util
|    - __init__.py
|    - sort
|        - __init__.py
|        - quick.py
- workflow
|    - __init__.py
|    - main.py
- setup.py

3 directories, 6 files
$ pip install .
```

Or, if your project has been published to PyPI:

```
$ pip install my-project
```

In each case, we have created a virtualenv with the `--system-site-packages` flag in the `venv` subdirectory then installed the `fairydust` distribution from PyPI along with the two packages that our project consists of. (Again, both Python and Toil are assumed to be present on the leader and all worker nodes.) We can now run our workflow:

```
$ python -m workflow.main --batchSystem=mesos ...
```

Important: If workflow's external dependencies contain native code (i.e. are not pure Python) then they must be manually installed on each worker.

Note: Neither `python setup.py develop` nor `pip install -e .` can be used in this process as, instead of copying the source files, they create `.egg-link` files that Toil can't hot-deploy. Similarly, `python setup.py install` doesn't work either as it installs the project as a Python `.egg` which is also not currently supported by Toil (though it could be in the future).

It should also be noted that while using the `--single-version-externally-managed` flag with `setup.py` will prevent the installation of your package as an `.egg`, it will also disable the automatic installation of your project's

dependencies.

Hot-deployment with sibling modules

This scenario applies if the user script imports modules that are its siblings:

```
$ cd my_project
$ ls
userScript.py utilities.py
$ ./userScript.py --batchSystem=mesos ...
```

Here `userScript.py` imports additional functionality from `utilities.py`. Toil detects that `userScript.py` has sibling modules and copies them to the workers, alongside the user script. Note that sibling modules will be hot-deployed regardless of whether they are actually imported by the user script–all .py files residing in the same directory as the user script will automatically be hot-deployed.

Sibling modules are a suitable method of organizing the source code of reasonably complicated workflows.

Hot-deploying a package hierarchy

Recall that in Python, a package is a directory containing one or more `.py` files—one of which must be called `__init__.py`—and optionally other packages. For more involved workflows that contain a significant amount of code, this is the recommended way of organizing the source code. Because we use a package hierarchy, we can't really refer to the user script as such, we call it the user *module* instead. It is merely one of the modules in the package hierarchy. We need to inform Toil that we want to use a package hierarchy by invoking Python's `-m` option. That enables Toil to identify the entire set of modules belonging to the workflow and copy all of them to each worker. Note that while using the `-m` option is optional in the scenarios above, it is mandatory in this one.

The following shell session illustrates this:

```
$ cd my_project
$ tree
.
- utils
|    - __init__.py
|    - sort
|        - __init__.py
|        - quick.py
- workflow
     - __init__.py
     - main.py

3 directories, 5 files
$ python -m workflow.main --batchSystem=mesos ...
```

Here the user module `main.py` does not reside in the current directory, but is part of a package called `util`, in a subdirectory of the current directory. Additional functionality is in a separate module called `util.sort.quick` which corresponds to `util/sort/quick.py`. Because we invoke the user module via `python -m workflow.main`, Toil can determine the root directory of the hierarchy–`my_project` in this case–and copy all Python modules underneath it to each worker. The `-m` option is documented here

When `-m` is passed, Python adds the current working directory to `sys.path`, the list of root directories to be considered when resolving a module name like `workflow.main`. Without that added convenience we'd have to run the workflow as `PYTHONPATH="$PWD" python -m workflow.main`. This also means that Toil can detect the

root directory of the user module's package hierarchy even if it isn't the current working directory. In other words we could do this:

```
$ cd my_project
$ export PYTHONPATH="$PWD"
$ cd /some/other/dir
$ python -m workflow.main --batchSystem=mesos ...
```

Also note that the root directory itself must not be package, i.e. must not contain an `__init__.py`.

Relying on shared filesystems

Bare-metal clusters typically mount a shared file system like NFS on each node. If every node has that file system mounted at the same path, you can place your project on that shared filesystem and run your user script from there. Additionally, you can clone the Toil source tree into a directory on that shared file system and you won't even need to install Toil on every worker. Be sure to add both your project directory and the Toil clone to `PYTHONPATH`. Toil replicates `PYTHONPATH` from the leader to every worker.

Using a shared filesystem

Toil currently only supports a `tempdir` set to a local, non-shared directory.

Deploying Toil

Toil comes with the Toil Appliance, a Docker image with Mesos and Toil baked in. It's easily deployed, only needs Docker, and allows for workflows to be run in single-machine mode and for clusters of VMs to be provisioned. For more information, see the *Cloud installation* section.

CHAPTER 3

API and Architecture

Toil API

Job methods

Jobs are the units of work in Toil which are composed into workflows.

class toil.job.**Job**(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*, *checkpoint=False*)

Class represents a unit of work in toil.

__init__(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*, *checkpoint=False*)

This method must be called by any overriding constructor.

Parameters

- **memory** (*int or string convertable by bd2k.util.humanize.human2bytes to an int*) – the maximum number of bytes of memory the job will require to run.
- **cores** (*int or string convertable by bd2k.util.humanize.human2bytes to an int*) – the number of CPU cores required.
- **disk** (*int or string convertable by bd2k.util.humanize.human2bytes to an int*) – the amount of local disk space required by the job, expressed in bytes.
- **preemptable** (*bool*) – if the job can be run on a preemptable node.
- **checkpoint** – if any of this job's successor jobs completely fails, exhausting all their retries, remove any successor jobs and rerun this job to restart the subtree. Job must be a leaf vertex in the job graph when initially defined, see toil.job.Job.checkNewCheckpointsAreCutVertices().

run(*fileStore*)

Override this function to perform work and dynamically create successor jobs.

Parameters **fileStore** (toil.fileStore.FileStore) – Used to create local and globally sharable temporary files and to send log messages to the leader process.

Returns The return value of the function can be passed to other jobs by means of *toil.job.Job.rv()*.

addChild(*childJob*)

Adds childJob to be run as child of this job. Child jobs will be run directly after this job's *toil.job.Job.run()* method has completed.

Parameters **childJob** (toil.job.Job) –

Returns childJob

Return type *toil.job.Job*

hasChild(*childJob*)

Check if childJob is already a child of this job.

Parameters **childJob** (toil.job.Job) –

Returns True if childJob is a child of the job, else False.

Return type bool

addFollowOn(*followOnJob*)

Adds a follow-on job, follow-on jobs will be run after the child jobs and their successors have been run.

Parameters **followOnJob** (toil.job.Job) –

Returns followOnJob

Return type *toil.job.Job*

addService(*service*, *parentService=None*)

Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

Raises ***toil.job.JobException*** – If service has already been made the child of a job or another service.

Parameters

- **service** (toil.job.Job.Service) – Service to add.
- **parentService** (toil.job.Job.Service) – Service that will be started before 'service' is started. Allows trees of services to be established. parentService must be a service of this job.

Returns a promise that will be replaced with the return value from *toil.job.Job.Service.start()* of service in any successor of the job.

Return type *toil.job.Promise*

addChildFn(*fn*, **args*, ***kwargs*)

Adds a function as a child job.

Parameters **fn** – Function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new child job that wraps fn.

Return type *toil.job.FunctionWrappingJob*

addFollowOnFn (*fn*, **args*, ***kwargs*)
Adds a function as a follow-on job.

Parameters **fn** – Function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.FunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new follow-on job that wraps fn.

Return type *toil.job.FunctionWrappingJob*

addChildJobFn (*fn*, **args*, ***kwargs*)
Adds a job function as a child job. See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters **fn** – Job function to be run as a child job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new child job that wraps fn.

Return type *toil.job.JobFunctionWrappingJob*

addFollowOnJobFn (*fn*, **args*, ***kwargs*)
Add a follow-on job function. See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

Parameters **fn** – Job function to be run as a follow-on job with `*args` and `**kwargs` as arguments to this function. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new follow-on job that wraps fn.

Return type *toil.job.JobFunctionWrappingJob*

static **wrapFn** (*fn*, **args*, ***kwargs*)
Makes a Job out of a function. Convenience function for constructor of *toil.job.FunctionWrappingJob*.

Parameters **fn** – Function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new function that wraps fn.

Return type *toil.job.FunctionWrappingJob*

static **wrapJobFn** (*fn*, **args*, ***kwargs*)
Makes a Job out of a job function. Convenience function for constructor of *toil.job.JobFunctionWrappingJob*.

Parameters **fn** – Job function to be run with `*args` and `**kwargs` as arguments. See toil.job.JobFunctionWrappingJob for reserved keyword arguments used to specify resource requirements.

Returns The new job function that wraps fn.

Return type *toil.job.JobFunctionWrappingJob*

encapsulate()
Encapsulates the job, see *toil.job.EncapsulatedJob*. Convenience function for constructor of *toil.job.EncapsulatedJob*.

Returns an encapsulated version of this job.

Return type *toil.job.EncapsulatedJob*

rv(**path*)
Creates a *promise* (*toil.job.Promise*) representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

Parameters **path** (*(Any)*) – Optional path for selecting a component of the promised return value. If absent or empty, the entire return value will be used. Otherwise, the first element of the path is used to select an individual item of the return value. For that to work, the return value must be a list, dictionary or of any other type implementing the *__getitem__()* magic method. If the selected item is yet another composite value, the second element of the path can be used to select an item from it, and so on. For example, if the return value is *[6,{'a':42}]*, *.rv(0)* would select *6* , *rv(1)* would select *{'a':3}* while *rv(1,'a')* would select *3*. To select a slice from a return value that is slicable, e.g. tuple or list, the path element should be a *slice* object. For example, assuming that the return value is *[6, 7, 8, 9]* then *.rv(slice(1, 3))* would select *[7, 8]*. Note that slicing really only makes sense at the end of path.

Returns A promise representing the return value of this jobs *toil.job.Job.run()* method.

Return type *toil.job.Promise*

prepareForPromiseRegistration(*jobStore*)
Ensure that a promise by this job (the promissor) can register with the promissor when another job referring to the promise (the promissee) is being serialized. The promissee holds the reference to the promise (usually as part of the the job arguments) and when it is being pickled, so will the promises it refers to. Pickling a promise triggers it to be registered with the promissor.

Returns

checkJobGraphForDeadlocks()
See *toil.job.Job.checkJobGraphConnected()*, toil.job.Job.checkJobGraphAcyclic() and *toil.job.Job.checkNewCheckpointsAreLeafVertices()* for more info.

Raises ***toil.job.JobGraphDeadlockException*** – if the job graph is cyclic, contains multiple roots or contains checkpoint jobs that are not leaf vertices when defined (see toil.job.Job.checkNewCheckpointsAreLeaves()).

getRootJobs()

Returns The roots of the connected component of jobs that contains this job. A root is a job with no predecessors.

:rtype : set of toil.job.Job instances

checkJobGraphConnected()

Raises ***toil.job.JobGraphDeadlockException*** – if *toil.job.Job.getRootJobs()* does not contain exactly one root job.

As execution always starts from one root job, having multiple root jobs will cause a deadlock to occur.

checkJobGraphAcylic()

Raises ***toil.job.JobGraphDeadlockException*** – if the connected component of jobs containing this job contains any cycles of child/followOn dependencies in the *augmented job graph* (see below). Such cycles are not allowed in valid job graphs.

A follow-on edge (A, B) between two jobs A and B is equivalent to adding a child edge to B from (1) A, (2) from each child of A, and (3) from the successors of each child of A. We call each such edge an edge an "implied" edge. The augmented job graph is a job graph including all the implied edges.

For a job graph G = (V, E) the algorithm is `O(|V|^2)`. It is `O(|V| + |E|)` for a graph with no follow-ons. The former follow-on case could be improved!

checkNewCheckpointsAreLeafVertices()

A checkpoint job is a job that is restarted if either it fails, or if any of its successors completely fails, exhausting their retries.

A job is a leaf it is has no successors.

A checkpoint job must be a leaf when initially added to the job graph. When its run method is invoked it can then create direct successors. This restriction is made to simplify implementation.

Raises *toil.job.JobGraphDeadlockException* – if there exists a job being added to the graph for which checkpoint=True and which is not a leaf.

defer(*function*, **args*, ***kwargs*)

Register a deferred function, i.e. a callable that will be invoked after the current attempt at running this job concludes. A job attempt is said to conclude when the job function (or the *toil.job.Job.run()* method for class-based jobs) returns, raises an exception or after the process running it terminates abnormally. A deferred function will be called on the node that attempted to run the job, even if a subsequent attempt is made on another node. A deferred function should be idempotent because it may be called multiple times on the same node or even in the same process. More than one deferred function may be registered per job attempt by calling this method repeatedly with different arguments. If the same function is registered twice with the same or different arguments, it will be called twice per job attempt.

Examples for deferred functions are ones that handle cleanup of resources external to Toil, like Docker containers, files outside the work directory, etc.

Parameters

- **function** (*callable*) – The function to be called after this job concludes.
- **args** (*list*) – The arguments to the function
- **kwargs** (*dict*) – The keyword arguments to the function

getTopologicalOrderingOfJobs()

Returns a list of jobs such that for all pairs of indices i, j for which i < j, the job at index i can be run before the job at index j.

Return type list

Job.FileStore

The FileStore is an abstraction of a Toil run's shared storage.

class toil.fileStore.**FileStore**(*jobStore*, *jobGraph*, *localTempDir*, *inputBlockFn*)

An abstract base class to represent the interface between a worker and the job store. Concrete subclasses will be used to manage temporary files, read and write files from the job store and log messages, passed as argument to the *toil.job.Job.run()* method.

__init__(*jobStore*, *jobGraph*, *localTempDir*, *inputBlockFn*)

open(**args*, ***kwds*)

The context manager used to conduct tasks prior-to, and after a job has been run.

Parameters **job** (toil.job.Job) – The job instance of the toil job to run.

getLocalTempDir()
Get a new local temporary directory in which to write files that persist for the duration of the job.

Returns The absolute path to a new local temporary directory. This directory will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates, removing all files it contains recursively.

Return type str

getLocalTempFile()
Get a new local temporary file that will persist for the duration of the job.

Returns The absolute path to a local temporary file. This file will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates.

Return type str

getLocalTempFileName()
Get a valid name for a new local file. Don't actually create a file at the path.

Returns Path to valid file

Return type str

writeGlobalFile(*localFileName*, *cleanup=False*)
Takes a file (as a path) and uploads it to the job store.

Parameters

- **localFileName** (*string*) – The path to the local file to upload.
- **cleanup** (*bool*) – if True then the copy of the global file will be deleted once the job and all its successors have completed running. If not the global file must be deleted manually.

Returns an ID that can be used to retrieve the file.

Return type toil.fileStore.FileID

writeGlobalFileStream(*cleanup=False*)
Similar to writeGlobalFile, but allows the writing of a stream to the job store. The yielded file handle does not need to and should not be closed explicitly.

Parameters **cleanup** (*bool*) – is as in *toil.fileStore.FileStore.writeGlobalFile()*.

Returns A context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store.

readGlobalFile(*fileStoreID*, *userPath=None*, *cache=True*, *mutable=None*)
Downloads a file described by fileStoreID from the file store to the local directory.

If a user path is specified, it is used as the destination. If a user path isn't specified, the file is stored in the local temp directory with an encoded name.

Parameters

- **fileStoreID** (*toil.fileStore.FileID*) – job store id for the file
- **userPath** (*string*) – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **cache** (*bool*) – Described in *readGlobalFile()*
- **mutable** (*bool*) – Described in *readGlobalFile()*

Returns An absolute path to a local, temporary copy of the file keyed by fileStoreID.

Return type str

readGlobalFileStream(*fileStoreID*)
Similar to readGlobalFile, but allows a stream to be read from the job store. The yielded file handle does not need to and should not be closed explicitly.

Returns a context manager yielding a file handle which can be read from.

deleteLocalFile(*fileStoreID*)
Deletes Local copies of files associated with the provided job store ID.

Parameters **fileStoreID** (*str*) – File Store ID of the file to be deleted.

deleteGlobalFile(*fileStoreID*)
Deletes local files with the provided job store ID and then permanently deletes them from the job store. To ensure that the job can be restarted if necessary, the delete will not happen until after the job's run method has completed.

Parameters **fileStoreID** – the job store ID of the file to be deleted.

classmethod findAndHandleDeadJobs(*nodeInfo*, *batchSystemShutdown=False*)
This function looks at the state of all jobs registered on the node and will handle them (clean up their presence ont he node, and run any registered defer functions)

Parameters

- **nodeInfo** – Information regarding the node required for identifying dead jobs.
- **batchSystemShutdown** (*bool*) – Is the batch system in the process of shutting down?

logToMaster(*text*, *level=20*)
Send a logging message to the leader. The message will also be logged by the worker at the same level.

Parameters

- **text** – The string to log.
- **level** (*int*) – The logging level.

classmethod shutdown(*dir_*)
Shutdown the filestore on this node.

This is intended to be called on batch system shutdown.

Parameters **dir** – The jeystone directory containing the required information for fixing the state of failed workers on the node before cleaning up.

Job.Runner

The Runner contains the methods needed to configure and start a Toil run.

class Job.**Runner**
Used to setup and run Toil workflow.

static getDefaultArgumentParser()
Get argument parser with added toil workflow options.

Returns The argument parser used by a toil workflow with added Toil options.

Return type argparse.ArgumentParser

static getDefaultOptions(*jobStore*)
Get default options for a toil workflow.

Parameters **jobStore** (*string*) – A string describing the jobStore for the workflow.

Returns The options used by a toil workflow.

Return type argparse.ArgumentParser values object

static addToilOptions(*parser*)
Adds the default toil options to an optparse or argparse parser object.

Parameters **parser** (*optparse.OptionParser or argparse.ArgumentParser*) – Options object to add toil options to.

static startToil(*job*, *options*)
Deprecated by toil.common.Toil.run. Runs the toil workflow using the given options (see Job.Runner.getDefaultOptions and Job.Runner.addToilOptions) starting with this job. :param toil.job.Job job: root job of the workflow :raises: toil.leader.FailedJobsException if at the end of function their remain failed jobs. :return: The return value of the root job's run function. :rtype: Any

Toil

The Toil class provides for a more general way to configure and start a Toil run.

class toil.common.**Toil**(*options*)
A context manager that represents a Toil workflow, specifically the batch system, job store, and its configuration.

__init__(*options*)
Initialize a Toil object from the given options. Note that this is very light-weight and that the bulk of the work is done when the context is entered.

Parameters **options** (*argparse.Namespace*) – command line options specified by the user

config = None

Type toil.common.Config

start(*rootJob*)
Invoke a Toil workflow with the given job as the root for an initial run. This method must be called in the body of a `with Toil(...) as toil:` statement. This method should not be called more than once for a workflow that has not finished.

Parameters **rootJob** (toil.job.Job) – The root job of the workflow

Returns The root job's return value

restart()
Restarts a workflow that has been interrupted. This method should be called if and only if a workflow has previously been started and has not finished.

Returns The root job's return value

classmethod getJobStore(*locator*)
Create an instance of the concrete job store implementation that matches the given locator.

Parameters **locator** (*str*) – The location of the job store to be represent by the instance

Returns an instance of a concrete subclass of AbstractJobStore

Return type *toil.jobStores.abstractJobStore.AbstractJobStore*

static createBatchSystem(*config*)
Creates an instance of the batch system specified in the given config.

Parameters **config** (*toil.common.Config*) – the current configuration

Return type *batchSystems.abstractBatchSystem.AbstractBatchSystem*

Returns an instance of a concrete subclass of AbstractBatchSystem

static getWorkflowDir(*workflowID*, *configWorkDir=None*)
Returns a path to the directory where worker directories and the cache will be located for this workflow.

Parameters

- **workflowID** (*str*) – Unique identifier for the workflow
- **configWorkDir** (*str*) – Value passed to the program using the –workDir flag

Returns Path to the workflow directory

Return type str

Job.Service

The Service class allows databases and servers to be spawned within a Toil workflow.

class Job.**Service**(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*)
Abstract class used to define the interface to a service.

__init__(*memory=None*, *cores=None*, *disk=None*, *preemptable=None*, *unitName=None*)
Memory, core and disk requirements are specified identically to as in *toil.job.Job.__init__()*.

start(*job*)
Start the service.

Parameters **job** (toil.job.Job) – The underlying job that is being run. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

Returns An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see *toil.job.Job.addService()*).

stop(*job*)
Stops the service. Function can block until complete.

Parameters **job** (toil.job.Job) – The underlying job that is being run. Can be used to register deferred functions, or to access the fileStore for creating temporary files.

check()
Checks the service is still running.

Raises **exceptions.RuntimeError** – If the service failed, this will cause the service job to be labeled failed.

Returns True if the service is still running, else False. If False then the service job will be terminated, and considered a success. Important point: if the service job exits due to a failure, it should raise a RuntimeError, not return False!

FunctionWrappingJob

The subclass of Job for wrapping user functions.

class toil.job.**FunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)
Job used to wrap a function. In its *run* method the wrapped function is called.

__init__(*userFunction*, **args*, ***kwargs*)

Parameters **userFunction** (*callable*) – The function to wrap. It will be called with `*args` and `**kwargs` as arguments.

The keywords `memory`, `cores`, `disk`, `preemptable` and `checkpoint` are reserved keyword arguments that if specified will be used to determine the resources required for the job, as *toil.job.Job.__init__()*. If they are keyword arguments to the function they will be extracted from the function definition, but may be overridden by the user (as you would expect).

JobFunctionWrappingJob

The subclass of FunctionWrappingJob for wrapping user job functions.

class toil.job.**JobFunctionWrappingJob**(*userFunction*, **args*, ***kwargs*)

A job function is a function whose first argument is a *Job* instance that is the wrapping job for the function. This can be used to add successor jobs for the function and perform all the functions the *Job* class provides.

To enable the job function to get access to the *toil.fileStore.FileStore* instance (see *toil.job.Job.run()*), it is made a variable of the wrapping job called fileStore.

EncapsulatedJob

The subclass of Job for *encapsulating* a job, allowing a subgraph of jobs to be treated as a single job.

class toil.job.**EncapsulatedJob**(*job*)

A convenience Job class used to make a job subgraph appear to be a single job.

Let A be the root job of a job subgraph and B be another job we'd like to run after A and all its successors have completed, for this use encapsulate:

```
#  Job A and subgraph, Job B
A, B = A(), B()
A' = A.encapsulate()
A'.addChild(B)
#  B will run after A and all its successors have completed, A and its subgraph of
# successors in effect appear to be just one job.
```

The return value of an encapsulatd job (as accessed by the *toil.job.Job.rv()* function) is the return value of the root job, e.g. A().encapsulate().rv() and A().rv() will resolve to the same value after A or A.encapsulate() has been run.

__init__(*job*)

Parameters **job** (toil.job.Job) – the job to encapsulate.

Promise

The class used to reference return values of jobs/services not yet run/started.

class toil.job.**Promise**(*job*, *path*)

References a return value from a *toil.job.Job.run()* or *toil.job.Job.Service.start()* method as a *promise* before the method itself is run.

Let T be a job. Instances of *Promise* (termed a *promise*) are returned by T.rv(), which is used to reference the return value of T's run function. When the promise is passed to the constructor (or as an argument to a wrapped function) of a different, successor job the promise will be replaced by the actual referenced return value. This mechanism allows a return values from one job's run method to be input argument to job before the former job's run function has been executed.

filesToDelete = set([])
A set of IDs of files containing promised values when we know we won't need them anymore

__init__(*job*, *path*)

Parameters

- **job** (Job) – the job whose return value this promise references
- **path** – see *Job.rv()*

class toil.job.**PromisedRequirement**(*valueOrCallable*, **args*)

__init__(*valueOrCallable*, **args*)
Class for dynamically allocating job function resource requirements involving *toil.job.Promise* instances.

Use when resource requirements depend on the return value of a parent function. PromisedRequirements can be modified by passing a function that takes the *Promise* as input.

For example, let f, g, and h be functions. Then a Toil workflow can be defined as follows:: A = Job.wrapFn(f) B = A.addChildFn(g, cores=PromisedRequirement(A.rv()) C = B.addChildFn(h, cores=PromisedRequirement(lambda x: 2*x, B.rv()))

Parameters

- **valueOrCallable** – A single Promise instance or a function that takes *args as input parameters.
- ***args** (*int or Promise*) – variable length argument list

getValue()
Returns PromisedRequirement value

static convertPromises(*kwargs*)
Returns True if reserved resource keyword is a Promise or PromisedRequirement instance. Converts Promise instance to PromisedRequirement.

Parameters **kwargs** – function keyword arguments

Returns bool

Exceptions

Toil specific exceptions.

exception toil.job.**JobException**(*message*)
General job exception.

__init__(*message*)

exception toil.job.**JobGraphDeadlockException**(*string*)
An exception raised in the event that a workflow contains an unresolvable dependency, such as a cycle. See *toil.job.Job.checkJobGraphForDeadlocks()*.

__init__(*string*)

exception toil.jobStores.abstractJobStore.**ConcurrentFileModificationException**(*jobStoreFileID*)
Indicates that the file was attempted to be modified by multiple processes at once.

__init__(*jobStoreFileID*)

Parameters **jobStoreFileID** (*str*) – the ID of the file that was modified by multiple workers or processes concurrently

exception toil.jobStores.abstractJobStore.**JobStoreExistsException**(*locator*)
Indicates that the specified job store already exists.

__init__(*locator*)

exception toil.jobStores.abstractJobStore.**NoSuchFileException**(*jobStoreFileID*, *customName=None*)
Indicates that the specified file does not exist.

__init__(*jobStoreFileID*, *customName=None*)

Parameters

- **jobStoreFileID** (*str*) – the ID of the file that was mistakenly assumed to exist
- **customName** (*str*) – optionally, an alternate name for the nonexistent file

exception toil.jobStores.abstractJobStore.**NoSuchJobException**(*jobStoreID*)
Indicates that the specified job does not exist.

__init__(*jobStoreID*)

Parameters **jobStoreID** (*str*) – the jobStoreID that was mistakenly assumed to exist

exception toil.jobStores.abstractJobStore.**NoSuchJobStoreException**(*locator*)
Indicates that the specified job store does not exist.

__init__(*locator*)

Toil architecture

The following diagram layouts out the software architecture of Toil.

These components are described below:

- **the leader:** The leader is responsible for deciding which jobs should be run. To do this it traverses the job graph. Currently this is a single threaded process, but we make aggressive steps to prevent it becoming a bottleneck (see *Read-only Leader* described below).
- **the job-store:** Handles all files shared between the components. Files in the job-store are the means by which the state of the workflow is maintained. Each job is backed by a file in the job store, and atomic updates to this state are used to ensure the workflow can always be resumed upon failure. The job-store can also store all user files, allowing them to be shared between jobs. The job-store is defined by the *AbstractJobStore* class. Multiple implementations of this class allow Toil to support different back-end file stores, e.g.: S3, network file systems, Azure file store, etc.
- **workers:** The workers are temporary processes responsible for running jobs, one at a time per worker. Each worker process is invoked with a job argument that it is responsible for running. The worker monitors this job and reports back success or failure to the leader by editing the job's state in the file-store. If the job defines successor jobs the worker may choose to immediately run them (see *Job Chaining* below).
- **the batch-system:** Responsible for scheduling the jobs given to it by the leader, creating a worker command for each job. The batch-system is defined by the *AbstractBatchSystem* class. Toil uses multiple existing batch systems to schedule jobs, including Apache Mesos, GridEngine and a multi-process single node implementation that allows workflows to be run without any of these frameworks. Toil can therefore fairly easily be made to run a workflow using an existing cluster.

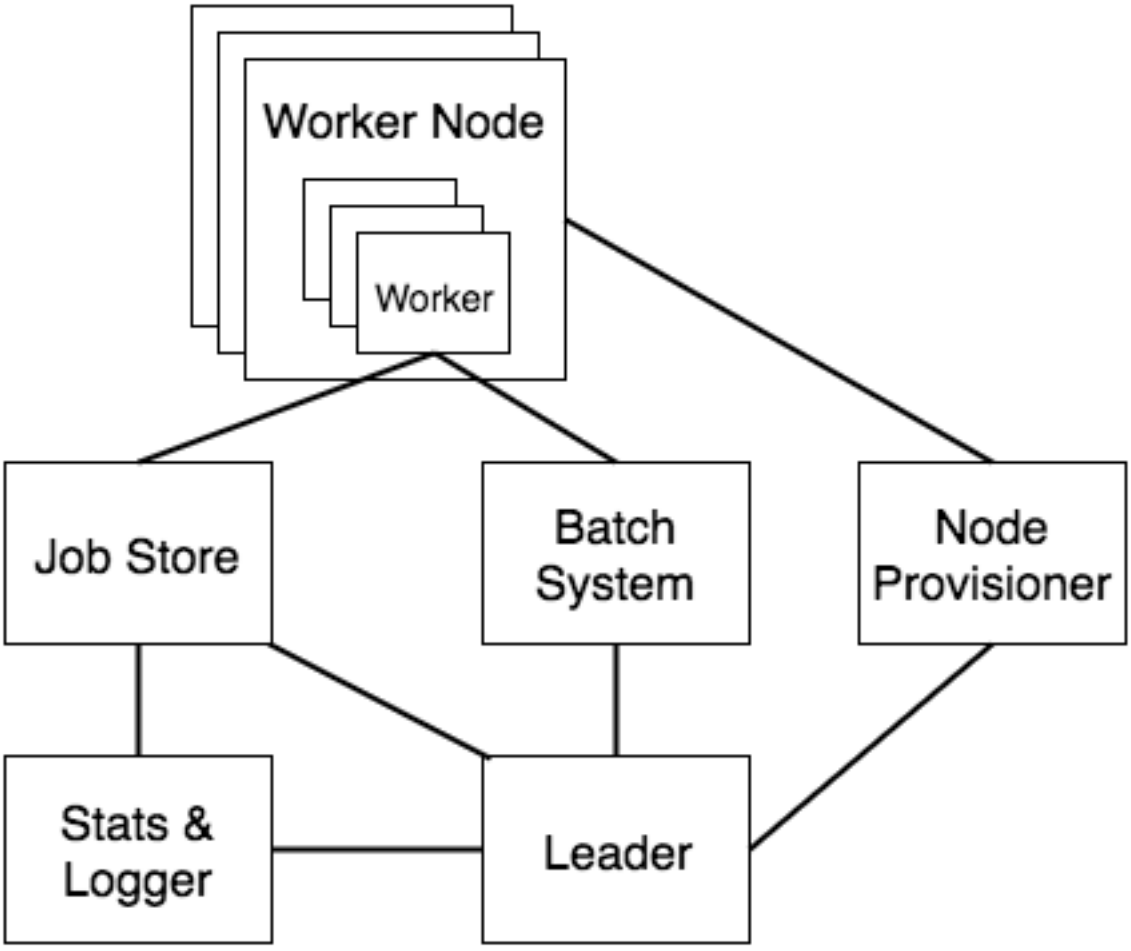


Fig. 3.1: Figure 1: The basic components of Toil's architecture.

- **the node provisioner:** Creates worker nodes in which the batch system schedules workers. It is defined by the `AbstractProvisioner` class.
- **the statistics and logging monitor:** Monitors logging and statistics produced by the workers and reports them. Uses the job-store to gather this information.

Optimizations

Toil implements lots of optimizations designed for scalability. Here we detail some of the key optimizations.

Read-only leader

The leader process is currently implemented as a single thread. Most of the leader's tasks revolve around processing the state of jobs, each stored as a file within the job-store. To minimise the load on this thread, each worker does as much work as possible to manage the state of the job it is running. As a result, with a couple of minor exceptions, the leader process never needs to write or update the state of a job within the job-store. For example, when a job is complete and has no further successors the responsible worker deletes the job from the job-store, marking it complete. The leader then only has to check for the existence of the file when it receives a signal from the batch-system to know that the job is complete. This off-loading of state management is orthogonal to future parallelization of the leader.

Job chaining

The scheduling of successor jobs is partially managed by the worker, reducing the number of individual jobs the leader needs to process. Currently this is very simple: if the there is a single next successor job to run and it's resources fit within the resources of the current job and closely match the resources of the current job then the job is run immediately on the worker without returning to the leader. Further extensions of this strategy are possible, but for many workflows which define a series of serial successors (e.g. map sequencing reads, post-process mapped reads, etc.) this pattern is very effective at reducing leader workload.

Preemptable node support

Critical to running at large-scale is dealing with intermittent node failures. Toil is therefore designed to always be resumable providing the job-store does not become corrupt. This robustness allows Toil to run on preemptible nodes, which are only available when others are not willing to pay more to use them. Designing workflows that divide into many short individual jobs that can use preemptable nodes allows for workflows to be efficiently scheduled and executed.

Caching

Running bioinformatic pipelines often require the passing of large datasets between jobs. Toil caches the results from jobs such that child jobs running on the same node can directly use the same file objects, thereby eliminating the need for an intermediary transfer to the job store. Caching also reduces the burden on the local disks, because multiple jobs can share a single file. The resulting drop in I/O allows pipelines to run faster, and, by the sharing of files, allows users to run more jobs in parallel by reducing overall disk requirements.

To demonstrate the efficiency of caching, we ran an experimental internal pipeline on 3 samples from the TCGA Lung Squamous Carcinoma (LUSC) dataset. The pipeline takes the tumor and normal exome fastqs, and the tumor rna fastq and input, and predicts MHC presented neoepitopes in the patient that are potential targets for T-cell based immunotherapies. The pipeline was run individually on the samples on c3.8xlarge machines on AWS (60GB RAM,600GB SSD storage, 32 cores). The pipeline aligns the data to hg19-based references, predicts MHC haplotypes using PHLAT, calls mutations using 2 callers (MuTect and RADIA) and annotates them using SnpEff, then predicts

MHC:peptide binding using the IEDB suite of tools before running an in-house rank boosting algorithm on the final calls.

To optimize time taken, The pipeline is written such that mutations are called on a per-chromosome basis from the whole-exome bams and are merged into a complete vcf. Running mutect in parallel on whole exome bams requires each mutect job to download the complete Tumor and Normal Bams to their working directories – An operation that quickly fills the disk and limits the parallelizability of jobs. The script was run in Toil, with and without caching, and Figure 2 shows that the workflow finishes faster in the cached case while using less disk on average than the uncached run. We believe that benefits of caching arising from file transfers will be much higher on magnetic disk-based storage systems as compared to the SSD systems we tested this on.

The batch system interface

The batch system interface is used by Toil to abstract over different ways of running batches of jobs, for example Slurm, GridEngine, Mesos, Parasol and a single node. The *toil.batchSystems.abstractBatchSystem.AbstractBatchSystem* API is implemented to run jobs using a given job management system, e.g. Mesos.

Environmental variables allow passing of scheduler specific parameters.

For SLURM:

```
export TOIL_SLURM_ARGS="-t 1:00:00 -q fatq"
```

For GridEngine (SGE, UGE), there is an additional environmental variable to define the parallel environment for running multicore jobs:

```
export TOIL_GRIDENGINE_PE='smp'
export TOIL_GRIDENGINE_ARGS='-q batch.q'
```

class `toil.batchSystems.abstractBatchSystem.`**`AbstractBatchSystem`**

An abstract (as far as Python currently allows) base class to represent the interface the batch system must provide to Toil.

classmethod `supportsHotDeployment``()`

Whether this batch system supports hot deployment of the user script itself. If it does, the *setUserScript()* can be invoked to set the resource object representing the user script.

Note to implementors: If your implementation returns True here, it should also override

Return type bool

classmethod `supportsWorkerCleanup``()`

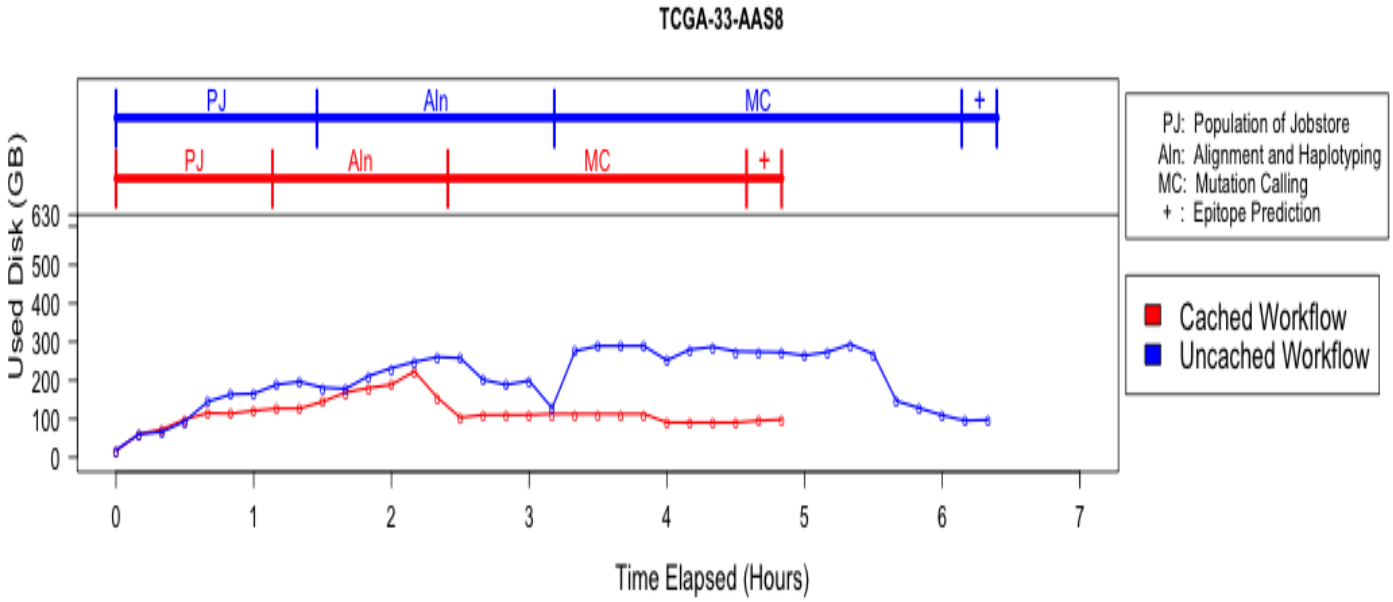
Indicates whether this batch system invokes `workerCleanup()` after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

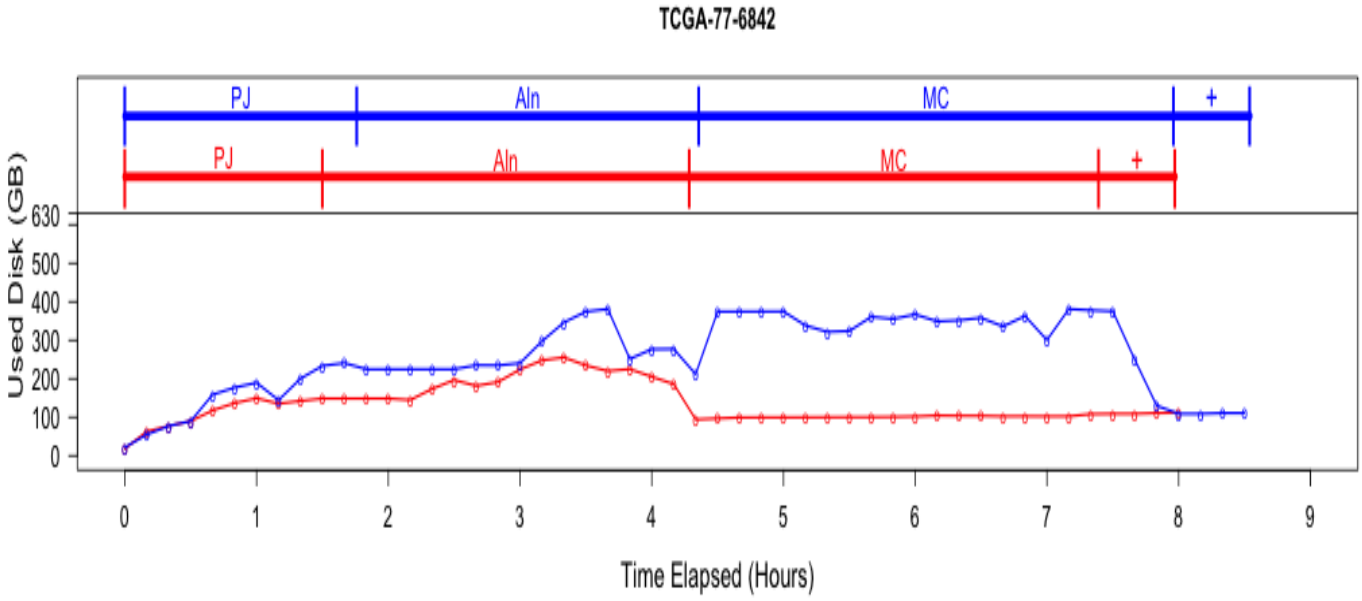
Return type bool

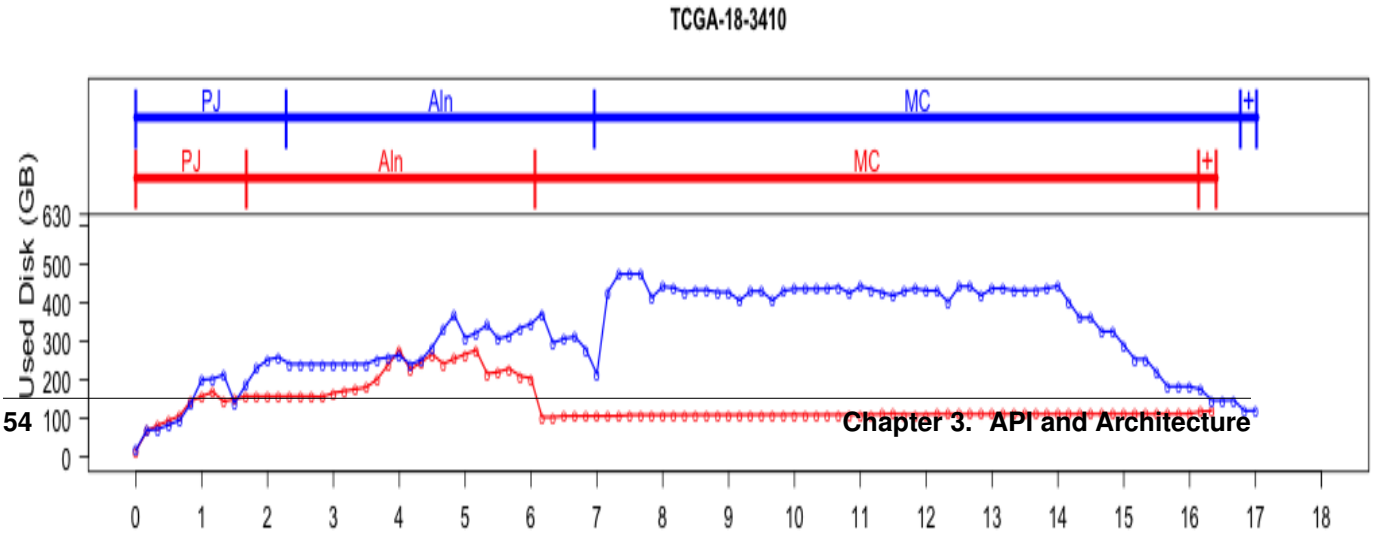
`setUserScript`(*userScript*)

Set the user script for this workflow. This method must be called before the first job is issued to this batch system, and only if *supportsHotDeployment()* returns True, otherwise it will raise an exception.

Parameters **`userScript`** (*toil.resource.Resource*) – the resource object representing the user script or module and the modules it depends on.


TCGA-33-AAS8
PJ
Aln
MC
+
PJ
Aln
MC
+
Used Disk (GB)
630
500
400
300
200
100
0
0
1
2
3
4
5
6
7
Time Elapsed (Hours)
PJ: Population of Jobstore
Aln: Alignment and Haplotyping
MC: Mutation Calling
+ : Epitope Prediction
Cached Workflow
Uncached Workflow


TCGA-77-6842
PJ
Aln
MC
+
PJ
Aln
MC
+
Used Disk (GB)
630
500
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9
Time Elapsed (Hours)


TCGA-18-3410
PJ
Aln
MC
+
PJ
Aln
MC
+
Used Disk (GB)
630
500
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

issueBatchJob(*jobNode*)
Issues a job with the specified command to the batch system and returns a unique jobID.

Parameters

- **command** (*str*) – the string to run as a command,
- **memory** (*int*) – int giving the number of bytes of memory the job needs to run
- **cores** (*float*) – the number of cores needed for the job
- **disk** (*int*) – int giving the number of bytes of disk space the job needs to run
- **preemptable** (*bool*) – True if the job can be run on a preemptable node

Returns a unique jobID that can be used to reference the newly issued job

Return type int

killBatchJobs(*jobIDs*)
Kills the given job IDs.

Parameters **jobIDs** (*list[int]*) – list of IDs of jobs to kill

getIssuedBatchJobIDs()
Gets all currently issued jobs

Returns A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

Return type list[str]

getRunningBatchJobIDs()
Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

Returns dictionary with currently running jobID keys and how many seconds they have been running as the value

Return type dict[str,float]

getUpdatedBatchJob(*maxWait*)
Returns a job that has updated its status.

Parameters **maxWait** (*float*) – the number of seconds to block, waiting for a result

Return type tuple(str, int) or None

Returns If a result is available, returns a tuple (jobID, exitValue, wallTime). Otherwise it returns None. wallTime is the number of seconds (a float) in wall-clock time the job ran for or None if this batch system does not support tracking wall time. Returns None for jobs that were killed.

shutdown()
Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

setEnv(*name*, *value=None*)
Set an environment variable for the worker process before it is launched. The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueuing a job.

If no value is provided it will be looked up from the current environment.

classmethod getRescueBatchJobFrequency()
Gets the period of time to wait (floating point, in seconds) between checking for missing/overlong jobs.

The job store interface

The job store interface is an abstraction layer that that hides the specific details of file storage, for example standard file systems, S3, etc. The *AbstractJobStore* API is implemented to support a give file store, e.g. S3. Implement this API to support a new file store.

class toil.jobStores.abstractJobStore.**AbstractJobStore**
Represents the physical storage for the jobs and files in a Toil workflow.

__init__()
Create an instance of the job store. The instance will not be fully functional until either *initialize()* or *resume()* is invoked. Note that the *destroy()* method may be invoked on the object with or without prior invocation of either of these two methods.

initialize(*config*)
Create the physical storage for this job store, allocate a workflow ID and persist the given Toil configuration to the store.

Parameters **config** (*toil.common.Config*) – the Toil configuration to initialize this job store with. The given configuration will be updated with the newly allocated workflow ID.

Raises ***JobStoreExistsException*** – if the physical storage for this job store already exists

writeConfig()
Persists the value of the AbstractJobStore.config attribute to the job store, so that it can be retrieved later by other instances of this class.

resume()
Connect this instance to the physical storage it represents and load the Toil configuration into the AbstractJobStore.config attribute.

Raises ***NoSuchJobStoreException*** – if the physical storage for this job store doesn't exist

config
The Toil configuration associated with this job store.

Return type toil.common.Config

setRootJob(*rootJobStoreID*)
Set the root job of the workflow backed by this job store

Parameters **rootJobStoreID** (*str*) – The ID of the job to set as root

loadRootJob()
Loads the root job in the current job store.

Raises ***toil.job.JobException*** – If no root job is set or if the root job doesn't exist in this job store

Returns The root job.

Return type toil.jobGraph.JobGraph

createRootJob(**args*, ***kwargs*)
Create a new job and set it as the root job in this job store

Return type toil.jobGraph.JobGraph

importFile(*srcUrl*, *sharedFileName=None*)

Imports the file at the given URL into job store. The ID of the newly imported file is returned. If the name of a shared file name is provided, the file will be imported as such and None is returned.

Currently supported schemes are:

- **'s3' for objects in Amazon S3** e.g. s3://bucket/key
- **'wasb' for blobs in Azure Blob Storage** e.g. wasb://container/blob
- **'file' for local files** e.g. file:///local/file/path
- **'http'** e.g. http://someurl.com/path

Parameters

- **srcUrl** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an Azure Blob Storage container.
- **sharedFileName** (*str*) – Optional name to assign to the imported file within the job store

Returns The jobStoreFileId of the imported file or None if sharedFileName was given

Return type FileID or None

exportFile(*jobStoreFileID*, *dstUrl*)

Exports file to destination pointed at by the destination URL.

Refer to *importFile()* documentation for currently supported URL schemes.

Note that the helper method _exportFile is used to read from the source and write to destination. To implement any optimizations that circumvent this, the _exportFile method should be overridden by subclasses of AbstractJobStore.

Parameters

- **jobStoreFileID** (*str*) – The id of the file in the job store that should be exported.
- **dstUrl** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an Azure Blob Storage container.

classmethod getSize(*url*)

returns the size of the file at the given URL

destroy()

The inverse of *initialize()*, this method deletes the physical storage represented by this instance. While not being atomic, this method *is* at least idempotent, as a means to counteract potential issues with eventual consistency exhibited by the underlying storage mechanisms. This means that if the method fails (raises an exception), it may (and should be) invoked again. If the underlying storage mechanism is eventually consistent, even a successful invocation is not an ironclad guarantee that the physical storage vanished completely and immediately. A successful invocation only guarantees that the deletion will eventually happen. It is therefore recommended to not immediately reuse the same job store location for a new Toil workflow.

getEnv()

Returns a dictionary of environment variables that this job store requires to be set in order to function properly on a worker.

Return type dict[str,str]

clean (*jobCache=None*)
Function to cleanup the state of a job store after a restart. Fixes jobs that might have been partially updated. Resets the try counts and removes jobs that are not successors of the current root job.

Parameters **jobCache** (*dict[str,toil.jobGraph.JobGraph]*) – if a value it must be a dict from job ID keys to JobGraph object values. Jobs will be loaded from the cache (which can be downloaded from the job store in a batch) instead of piecemeal when recursed into.

create (*jobNode*)
Creates a job graph from the given job node & writes it to the job store.

Return type toil.jobGraph.JobGraph

exists (*jobStoreID*)
Indicates whether the job with the specified jobStoreID exists in the job store

Return type bool

getPublicUrl (*fileName*)
Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters **fileName** (*str*) – the jobStoreFileID of the file to generate a URL for

Raises ***NoSuchFileException*** – if the specified file does not exist in this job store

Return type str

getSharedPublicUrl (*sharedFileName*)
Differs from `getPublicUrl()` in that this method is for generating URLs for shared files written by `writeSharedFileStream()`.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

Parameters **sharedFileName** (*str*) – The name of the shared file to generate a publically accessible url for.

Raises ***NoSuchFileException*** – raised if the specified file does not exist in the store

Return type str

load (*jobStoreID*)
Loads the job referenced by the given ID and returns it.

Parameters **jobStoreID** (*str*) – the ID of the job to load

Raises ***NoSuchJobException*** – if there is no job with the given ID

Return type toil.jobGraph.JobGraph

update (*job*)
Persists the job in this store atomically.

Parameters **job** (*toil.jobGraph.JobGraph*) – the job to write to this job store

delete (*jobStoreID*)
Removes from store atomically, can not then subsequently call load(), write(), update(), etc. with the job.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

Parameters **jobStoreID** (*str*) – the ID of the job to delete from this job store

jobs()
Best effort attempt to return iterator on all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished succesfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive ToilState object

Returns Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

Return type Iterator[toil.jobGraph.JobGraph]

writeFile(*localFilePath*, *jobStoreID=None*)
Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time.

Parameters

- **localFilePath** (*str*) – the path to the local file that will be uploaded to the job store.
- **jobStoreID** (*str or None*) – If specified the file will be associated with that job and when jobStore.delete(job) is called all files written with the given job.jobStoreID will be removed from the job store.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchJobException*** – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns an ID referencing the newly created file and can be used to read the file in the future.

Return type str

writeFileStream(**args*, ***kwds*)
Similar to writeFile, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly.

Parameters **jobStoreID** (*str*) – the id of a job, or None. If specified, the file will be associated with that job and when when jobStore.delete(job) is called all files written with the given job.jobStoreID will be removed from the job store.

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchJobException*** – if the job specified via jobStoreID does not exist

FIXME: some implementations may not raise this

Returns an ID that references the newly created file and can be used to read the file in the future.

Return type str

getEmptyFileStoreID(*jobStoreID=None*)
Creates an empty file in the job store and returns its ID. Call to fileExists(getEmptyFileStoreID(jobStoreID)) will return True.

Parameters **jobStoreID** (*str*) – the id of a job, or None. If specified, the file will be associated with that job and when jobStore.delete(job) is called a best effort attempt is made to delete all files written with the given job.jobStoreID

Returns a jobStoreFileID that references the newly created file and can be used to reference the file in the future.

Return type str

readFile(*jobStoreFileID*, *localFilePath*)
Copies the file referenced by jobStoreFileID to the given local file path. The version will be consistent with the last copy of the file written/updated.

The file at the given local path may not be modified after this method returns!

Parameters

- **jobStoreFileID** (*str*) – ID of the file to be copied
- **localFilePath** (*str*) – the local path indicating where to place the contents of the given file in the job store

readFileStream(**args*, ***kwds*)
Similar to readFile, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

Parameters **jobStoreFileID** (*str*) – ID of the file to get a readable file handle for

deleteFile(*jobStoreFileID*)
Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

Parameters **jobStoreFileID** (*str*) – ID of the file to delete

fileExists(*jobStoreFileID*)
Determine whether a file exists in this job store.

Parameters **jobStoreFileID** (*str*) – an ID referencing the file to be checked

Return type bool

updateFile(*jobStoreFileID*, *localFilePath*)
Replaces the existing version of a file in the job store. Throws an exception if the file does not exist.

Parameters

- **jobStoreFileID** (*str*) – the ID of the file in the job store to be updated
- **localFilePath** (*str*) – the local path to a file that will overwrite the current version in the job store

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method
- ***NoSuchFileException*** – if the specified file does not exist

updateFileStream(*jobStoreFileID*)
Replaces the existing version of a file in the job store. Similar to writeFile, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

Parameters **jobStoreFileID** (*str*) – the ID of the file in the job store to be updated

Raises

- ***ConcurrentFileModificationException*** – if the file was modified concurrently during an invocation of this method

- **NoSuchFileException** – if the specified file does not exist

writeSharedFileStream(**args*, ***kwds*)
Returns a context manager yielding a writable file handle to the global file referenced by the given name.

Parameters

- **sharedFileName** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store
- **isProtected** (*bool*) – True if the file must be encrypted, None if it may be encrypted or False if it must be stored in the clear.

Raises **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method

readSharedFileStream(**args*, ***kwds*)
Returns a context manager yielding a readable file handle to the global file referenced by the given name.

Parameters **sharedFileName** (*str*) – A file name matching AbstractJobStore.fileNameRegex, unique within this job store

writeStatsAndLogging(*statsAndLoggingString*)
Adds the given statistics/logging string to the store of statistics info.

Parameters **statsAndLoggingString** (*str*) – the string to be written to the stats file

Raises **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method

readStatsAndLogging(*callback*, *readAll=False*)
Reads stats/logging strings accumulated by the writeStatsAndLogging() method. For each stats/logging string this method calls the given callback function with an open, readable file handle from which the stats string can be read. Returns the number of stats/logging strings processed. Each stats/logging string is only processed once unless the readAll parameter is set, in which case the given callback will be invoked for all existing stats/logging strings, including the ones from a previous invocation of this method.

Parameters

- **callback** (*Callable*) – a function to be applied to each of the stats file handles found
- **readAll** (*bool*) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files

Raises **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method

Returns the number of stats files processed

Return type int

CHAPTER 4

Contributor's Guide

Building from Source

For developers, tinkerers, and people otherwise interested in Toil's internals, this section explains how to build Toil from source and run its test suite.

Building from master

First, clone the source:

```
$ git clone https://github.com/BD2KGenomics/toil
$ cd toil
```

Then, create and activate a virtualenv:

```
$ virtualenv venv
$ . venv/bin/activate
```

From there, you can list all available Make targets by running `make`. First and foremost, we want to install Toil's build requirements. (These are additional packages that Toil needs to be tested and built but not to be run.)

```
$ make prepare
```

Now, we can install Toil in development mode (such that changes to the source code will immediately affect the virtualenv):

```
$ make develop
```

Or, to install with support for all optional *Extras*:

```
$ make develop extras=[aws,mesos,azure,google,encryption,cwl]
```

To build the docs, run `make develop` with all extras followed by

```
$ make docs
```

Running tests

To invoke all tests (unit and integration) use

```
$ make test
```

Installing Docker with Quay

Docker is needed for some of the tests. Follow the appopriate installation instructions for your system on their website to get started.

When running `make test` you might still get the following error:

```
$ make test
Please set TOIL_DOCKER_REGISTRY, e.g. to quay.io/USER.
```

To solve, make an account with Quay and specify it like so:

```
$ TOIL_DOCKER_REGISTRY=quay.io/USER make test
```

where `USER` is your Quay username.

For convenience you may want to add this variable to your bashrc by running

```
$ echo 'export TOIL_DOCKER_REGISTRY=quay.io/USER' >> $HOME/.bashrc
```

Run an individual test with

```
$ make test tests=src/toil/test/sort/sortTest.py::SortTest::testSort
```

The default value for `tests` is `"src"` which includes all tests in the `src/` subdirectory of the project root. Tests that require a particular feature will be skipped implicitly. If you want to explicitly skip tests that depend on a currently installed *feature*, use

```
$ make test tests="-m 'not azure' src"
```

This will run only the tests that don't depend on the `azure` extra, even if that extra is currently installed. Note the distinction between the terms *feature* and *extra*. Every extra is a feature but there are features that are not extras, such as the `gridengine` and `parasol` features. To skip tests involving both the Parasol feature and the Azure extra, use the following:

```
$ make test tests="-m 'not azure and not parasol' src"
```

Running Mesos tests

If you're running Toil's Mesos tests, be sure to create the virtualenv with `--system-site-packages` to include the Mesos Python bindings. Verify this by activating the virtualenv and running `pip list | grep mesos`. On macOS, this may come up empty. To fix it, run the following:

```
for i in /usr/local/lib/python2.7/site-packages/*mesos*; do ln -snf $i venv/lib/
↪python2.7/site-packages/; done
```

Developing with the Toil Appliance

To develop on features reliant on the Toil Appliance (i.e. autoscaling), you should consider setting up a personal registry on Quay or Docker Hub. Because the Toil Appliance images are tagged with the Git commit they are based on and because only commits on our master branch trigger an appliance build on Quay, as soon as a developer makes a commit or dirties the working copy they will no longer be able to rely on Toil to automatically detect the proper Toil Appliance image. Instead, developers wishing to test any appliance changes in autoscaling should build and push their own appliance image to a personal Docker registry.

Here is a general workflow: (similar instructions apply when using Docker Hub)

1. Make some changes to the provisioner of your local version of Toil.
2. Go to the location where you installed the Toil source code and run:

   ```
   $ make docker
   ```

   to automatically build a docker image that can now be uploaded to your personal Quay account. If you have not installed Toil source code yet check out *Building from Source*.
3. If it's not already you will need Docker installed and need to log into Quay. Also you will want to make sure that your Quay account is public.
4. Set the environment variable `TOIL_DOCKER_REGISTRY` to your Quay account. If you find yourself doing this often you may want to add:

   ```
   export TOIL_DOCKER_REGISTRY=quay.io/<MY_QUAY_USERNAME>
   ```

   to your `.bashrc` or equivalent.
5. Now you can run:

   ```
   $ make push_docker
   ```

   which will upload the docker image to your Quay account. Take note of the image's tag for the next step.
6. Finally you will need to tell Toil from where to pull the Appliance image you've created (it uses the Toil release you have installed by default). To do this set the environment variable `TOIL_APPLIANCE_SELF` to the url of your image. For more info see *Environment variable options*.
7. Now you can launch your cluster! For more information see *Toil Provisioner*.

Running Cluster Locally

The Toil Appliance container can also be useful as a test environment since it can simulate a Toil cluster locally. An important caveat for this is autoscaling, since autoscaling will only work on an EC2 instance and cannot (at this time) be run on a local machine.

To spin up a local cluster, start by using the following Docker run command to launch a Toil leader container:

```
docker run --entrypoint=mesos-master --net=host -d --name=leader --volume=/home/
↪jobStoreParentDir:/jobStoreParentDir quay.io/ucsc_cgl/toil:3.6.0 --registry=in_
↪memory --ip=127.0.0.1 --port=5050 --allocation_interval=500ms
```

A couple notes on this command: the `-d` flag tells Docker to run in daemon mode so the container will run in the background. To verify that the container is running you can run `docker ps` to see all containers. If you want to run your own container rather than the official UCSC container you can simply replace the `quay.io/ucsc_cgl/toil:3.6.0` parameter with your own container name.

Also note that we are not mounting the job store directory itself, but rather the location where the job store will be written. Due to complications with running Docker on MacOS, I recommend only mounting directories within your home directory. The next command will launch the Toil worker container with similar parameters:

```
docker run --entrypoint=mesos-slave --net=host -d --name=worker --volume=/home/
↪jobStoreParentDir:/jobStoreParentDir quay.io/ucsc_cgl/toil:3.6.0 --work_dir=/var/
↪lib/mesos --master=127.0.0.1:5050 --ip=127.0.0.1 ---attributes=preemptable:False --
↪resources=cpus:2
```

Note here that we are specifying 2 CPUs and a non-preemptable worker. We can easily change either or both of these in a logical way. To change the number of cores we can change the 2 to whatever number you like, and to change the worker to be preemptable we change `preemptable:False` to `preemptable:True`. Also note that the same volume is mounted into the worker. This is needed since both the leader and worker write and read from the job store. Now that your cluster is running, you can run:

```
docker exec -it leader bash
```

to get a shell in your leader 'node'. You can also replace the `leader` parameter with `worker` to get shell access in your worker.

Docker-in-Docker issues

If you want to run Docker inside this Docker cluster (Dockerized tools, perhaps), you should also mount in the Docker socket via `-v /var/run/docker.sock:/var/run/docker.sock`. This will give the Docker client inside the Toil Appliance access to the Docker engine on the host. Client/engine version mismatches have been known to cause issues, so we recommend using Docker version 1.12.3 on the host to be compatible with the Docker client installed in the Appliance. Finally, be careful where you write files inside the Toil Appliance - 'child' Docker containers launched in the Appliance will actually be siblings to the Appliance since the Docker engine is located on the host. This means that the 'child' container can only mount in files from the Appliance if the files are located in a directory that was originally mounted into the Appliance from the host - that way the files are accessible to the sibling container. Note: if Docker can't find the file/directory on the host it will silently fail and mount in an empty directory.

Contributing

Maintainer's Guidelines

- We strive to never break the build on master.
- Pull requests should be used for any and all changes (except truly trivial ones).
- The commit message of direct commits to master must end in `(resolves #` followed by the issue number followed by `)`.

Naming conventions

- The **branch name** for a pull request starts with `issues/` followed by the issue number (or numbers, separated by a dash), followed by a short snake-case description of the change. (There can be many open pull requests with their associated branches at any given point in time and this convention ensures that we can easily identify branches.)
- The **commit message** of the first commit in a pull request needs to end in `(resolves #` followed by the issue number, followed by `)`. See here for details about writing properly-formatted and informative commit messages.

- The title of the **pull request** needs to have the same `(resolves #...)` suffix as the commit message. This lets Waffle stack the pull request and the associated issue. (Fortunately, Github automatically prepopulates the title of the PR with the message of the first commit in the PR, so this isn't any additional work.)

Say there is an issue numbered #123 titled *Foo does not work*. The branch name would be `issues/123-fix-foo` and the title of the commit would be *Fix foo in case of bar (resolves #123)*.

- Pull requests that address **multiple issues** use the `(resolves #602, resolves #214)` suffix in the request's title. These pull requests can and should contain multiple commits, with each commit message referencing the specific issue(s) it addresses. We may or may not squash the commits in those PRs.

Pull requests

- All pull requests must be reviewed by a person other than the request's author.
- Only the reviewer of a pull request can merge it.
- Until the pull request is merged, it should be continually rebased by the author on top of master.
- Pull requests are built automatically by Jenkins and won't be merged unless all tests pass.
- Ideally, a pull request should contain a single commit that addresses a single, specific issue. Rebasing and squashing can be used to achieve that goal (see *Multi-author pull requests*).

Multi-author pull requests

- A pull request starts off as single-author and can be changed to multi-author upon request via comment (typically by the reviewer) in the PR. The author of a single-author PR has to explicitly grant the request.
- Multi-author pull requests can have more than one commit. They must *not* be rebased as doing so would create havoc for other contributors.
- To keep a multi-author pull request up to date with master, merge from master instead of rebasing on top of master.
- Before the PR is merged, it may transition back to single-author mode, again via comment request in the PR. Every contributor to the PR has to acknowledge the request after making sure they don't have any unpushed changes they care about. This is necessary because a single-author PR can be reabsed and rebasing would make it hard to integrate these pushed commits.

CHAPTER 5

Indices and tables

- genindex
- search

Index

Symbols

A

C

D

E

S

T

U

W