Harmony Documentation

Cameron Kuchta

December 7, 2018

1 Introduction

Summit, the newest supercomputer at Oak Ridge National Laboratory, is currently being readied for users. In order to deploy the computer successfuly, it must undergo testing when running many different programs. These programs need to output the correct result and have good performance and finish reasonably quickly. To run these tests, a test harness has already been created that also records information from the test outputs. This harness requires users to manually get information from each test instance separately or they need to create a temporary database each time. To ease the analysis of each of these tests, we have setup a system for monitoring tests while running, recording results to a continuous database, and reporting them to the user.

2 Monitoring Tests

We decide which tests to monitor according to an input file. To assert that each test is running, we read a status file found within each test directory. This file contains information on all of the test instances for it. We then check that each test has an instance that is currently in queue to be run or is running. New test instances should automatically be created after one instance is finished but if this does not occur, it is important to notify the user. When alerting the user, we use Slack and send messages to the OLCF System Test group.

The following are descriptions of each of the files used for monitoring tests.

2.1 `scripts/connect.py`

This file contains a single function that connects to LSF. There should be no reason to change this. When initiating the connection, there should be a queue with the correct name on LSF. To find a list of all queues on the machine, run `bqueues`. Choose one of these queues from the list and input it into the connection.

2.2 `scripts/job_status.py`

2.2.1 `Job`

This class contains the information for a single job that exists in LSF. The `possible_status` variable holds the keys that lead to the name of each status. If another status is needed, put it in that file. During initialization of the class, the status is assigned to the class. If a new status is needed, this will also need to be changed. If new information needs to be stored for a job, put it in this class.

2.2.2 `JobStatus`

This class deals with queries on LSF. The `in_queue` function tests that a job is in the queue so if a new status is added that represents a job in the queue, change this function.

2.3 `scripts/job_monitor.py`

2.3.1 `JobMonitor`

This class takes care of initializing monitors. It automatically limits the maximum number of jobs that are allowed to be monitored.

2.3.2 `remove_dead_threads`

This function removes all dead monitors so that new monitors can get started.

2.3.3 `monitor`

This function acts as a monitor for a single job on LSF. Once started, it will first notify the user that it has begun. It will then checks on the job after some interval depending on what is set in the config file. The monitor quits once the job enters one of the possible done status'. If a new status is added to the possible status' and it is an exit status, make sure to put it in here. The monitor automatically takes care of cleaning up after it wakes up to notify so that it does not take up much memory while sleeping.

2.4 `scripts/parse_file.py`

This file takes care of parsing all files needed for this set of programs. If a new file needs to be parsed, this is probably the best place to put it.

2.4.1 `ParseEvent`

This parser takes in event files that are created by a test instance. Each event file is created so that each key value pair is connected with an equals sign and each pair is separated by a space.

Example: `field1=val1 field2=val2 field3=val3`

2.4.2 `ParseRGTStatus`

This parser parses the `rgt_status.txt` file that contains the information for each test instance for some test. The `remove_header` function removes all lines that start with either `#` or are empty. When parsing the file, the expected columns are `harness_start`, `harness_uid`, `job_id`, `build_status`, `submit_status`, and `check_status` in that order. If the order changes or if there are new columns, make sure to add them here.

2.4.3 `ParseJobID`

The job id parser reads the LSF id from the file that the test instance creates. It expects that the first number is the id for the test instance.

2.4.4 `ParseRGTInput`

This parses the input file that the harness uses. All comments are removed from the file. If the comment begins in the middle of the line, the beginning of the line is preserved. Each line is then parsed to get the field and values from the line. If the line is incorrectly formatted, an error is thrown. We then assert that there exists a `path_to_tests` variable for finding where tests are located. The input file should also contain all tests that should be monitored. Any misspellings on the `path_to_tests` and `test` or lines that are not understood are thrown as errors. If a new line is needed in the input file, this function will need to be changed.

Example:
```
path_to_tests = /foo/bar/
test = app name
```

2.5 `scripts/test_status.py`

This file deals with checking that all tests that should be in the queue are. It can be run by itself to automatically send messages to Slack on which tests do not exist.

2.5.1 `slack_notify`

This is the notifier that the monitors use. It pulls all information from the config file and uses the slack app.

2.5.2 `get_test_directories`

This function gets the paths to all tests found in an input file. The application should be located in the `path_to_tests` directory. Under this should be the test name and then `Status/latest` which leads to the latest instance. If this path changes, this will also need to change.

2.5.3 `check_tests`

This function first parses the input file to get the tests to check. It then gets the paths to the most recent instance for each test. The path to the `job_id.txt` is expected to be `path_to_tests/app/testname/Status/latest/job_id.txt`. It then gets the job id for the most recent instances and checks that it is in the queue or is currently running. If the test is not in the queue or something in the file path does not exist, a message is sent to Slack with this information. If this message needs to be changed, this is where to do so.

2.6 `scripts/notifications/slack_app/slack_application.py`

This file takes care of running everything for the Slack application. It does not deal with the commands that the application uses.

2.6.1 `SlackApp`

The application should be initialized off of the config file.

2.6.1.1 `get_my_mention_token`

This function gets the token that our app looks for when people are mentioning the app's bot. In essence, it pings Slack and pulls from the `user_id` from the response. If the response from the server changes, this will also need to change.

2.6.1.2 `get_my_mentions`

We pull messages from Slack by using `rtm_read()`. The client holds messages as a queue and when pulling, only one message is read. We limit the total number of times we pull according to the `max_reads` variable found in the configuration file.

2.6.1.3 `allowable_message`

Each message pulled is sent through a validation process to make sure that it does not contain malicious data. If there are further safety checks for a mention to undergo, put it here.

2.6.1.4 `search_messages`

Messages are then checked that they are messages. If so, we check that the mention token exists in the message and if so, prepare to respond.

2.6.1.5 `remove_dead_messengers`

Messages are sent via separate threads so that they do not block up the application if connection is slow. We also want to limit the maximum number of messengers we have open and thus need to check on and remove dead ones. The upper limit on the number of possible messengers is set by `max_messengers`.

2.6.1.6 `message_responder`

If this function, we first parse the message to find the command that was called and return the requisit message. When sending messages, a new thread is started using the `send_message` function.

2.6.1.7 `send_message`

Messages are sent using the `slackclient` package. Each message is first split and then sent separately to the channel of choice.

2.6.1.8 `split_message_to_send`

When sending messages to Slack, there is an upper limit to the message size. Thus, messages are automatically split if they are too large. The split length is set by `max_message_length`.

2.6.1.9 `main`

This function starts and runs the application. Once connected to Slack, the messages are checked periodically depending on `watch_time` and responds when valid messages with commands are recieved.

2.7 `scripts/notifications/slack_app/slack_commands.py`

This file contains all the commands that the application can respond to. If new commands are needed, this is where to put them. Remember to preserve security and only allow commands that do reads on Summit.

2.7.1 `docstring_parameter`

This function acts as a property on other functions and lets variables be entered into their docstrings before being shown. Commands are shown to the user as that functions docstring so passing variables is important for descriptions.

2.7.2 `is_command`

This function is also a property and labels functions so that they are presented to the Slack user as commands. Both of these properties are needed when creating a new command.

2.7.3 `get_functions`

This gets all functions in a class that are labeled `is_command`.

2.7.4 `make_columns`

This takes in a list of tuples and formats them as columns so that messeages sent to Slack are formatted nicely. To change the sizes of these columns, change the values wherever `make_columns` is called.

2.7.5 `MessageParser`

The `MessageParser` class takes care of parsing any message sent to our app from Slack.

2.7.5.1 `parse_message`

This takes care of parsing what command the user wants to run. If a new command is added, make sure to add how it should be called here.

2.7.5.2 `slack_help`

Get a message with information about all of the commands that the app can run.

2.7.5.3 `my_jobs`

Get a message containing all jobs that a user currently has in LSF.

2.7.5.4 `check_tests`

Check that all tests in some `rgt_input.txt` are in the queue or are running.

2.7.5.5 `monitor_job`

Start monitoring a job and get notified whenever it's status changes.

2.7.5.6 `all_jobs`

Get all jobs in LSF whether they are in the queue, currently running, or are already done. LSF removes old jobs after `CLEAN_PERIOD` with is found in the `lsb.params` file. This is the file that contains the parameters for LSF and is found in the machine, not our code.

3 Recording Results

Test instances are recorded in a database for easy access. The input to start the database is the input file for which tests to check. We then periodically check on these tests and update the database with new information.

3.1 scripts/database/connect_database.py

This file takes care of connecting to the database when new information needs to be added.

3.1.1 DatabaseConnector

This class holds the information needed for connecting to the database. It currently holds the `host`, `user`, `password`, `database` name, and the `port`. If new information is needed to connect to a database, it should be added here.

3.1.1.1 connect

Whenever something is needed from the database, a new connection is established to run the code and then disconnected. This is so that the database does not get blocked up.

3.2 scripts/database/create_database.py

There are some tables into the database that are initialized and never change. To initiate these we have created input MySQL files that can be run to set the initial values. These are located in the `db` directory. To initialize other tables with values, enter the new MySQL files there. When starting the database we also need to run some sql to do so.

3.2.1 execute_sql_file

This function runs an sql file on whichever database is specified. All other functions just run this one while pointing to a specific file.

3.2.2 insert_default

If you are adding a new table that needs initial values, make sure to add that file to this function.

3.3 scripts/database/update_database.py

The `update_database.py` file takes care of updating the database whenever called.

3.3.1 get_event_types

This function gets all the different event types that a test instance might output.

3.3.2 execute_sql

This function takes care of writing sql to the database. This is a good example of how to write querries so if you are changing this file, write the sql update similar to this one.

3.3.3 `UpdateDatabase`

This class takes care of updating the database from the file system. It does not automatically repeat itself and needs to be called from elsewhere. This class should be initialized by values in the config file. The expected fields found in the `rgt_status.txt` are `harness_start`, `harness_uid`, `job_id`, `build_status`, `submit_status`, and `check_status` in that order. Changing order or adding more fields should be taken care of here. When initializing the class, we all set which types of output files we allow, namely `build`, `submit`, `check`, and `report`.

3.3.3.1 `get_test_dirs_from_rgt`