Autonomics System Documentation

Table of Contents

I. Installation

Before you begin an installation, it is highly recommended that you read the system overview section and familiarize yourself with the various system processes that work together to provide Autonomics's major functionality.

If you are comfortable with how the system works, you will find detailed installation instructions below. Please be advised that the system is currently in development, and that the software is EXPERIMENTAL, so these installations may be subject to drastic change.

Hardware Requirements

Hardware requirements depend upon the jobs that will be run locally with respects to the manager.py process. Any locally-run jobs will require resources on the machine running manager.py, while remotely executed analyses will require resources on the remote machines.

In this initial release, Autonomics has classes to run the following analyses remotely:

- BLAST
- Quantification

Conversely, these analyses are run locally:

- Adapter/Quality Trimming
- Read Normalization
- *de novo* Transcript Assembly
- GO/KEGG annotation

Read Normalization and *de novo* Assembly Routinely take upwards of 10GB of memory, with trimming and GO/KEGG annotation taking much less (on the order of megabytes). Typically, we dedicate 20 cpus for each *de novo* assembly, between 100-200 cores for each BLAST annotation job, and 150 cores for each Pfam annotation job. The remaining analyses are currently single-threaded.

A future release of the system will include remote analysis classes for all jobs, and users are free to develop their own classes to shift these analyses to a remote resource.

Software Requirements

- Python 2.7.x
- MySQL 5+
- redis server
- SSH server
- Additional python modules:
  biopython
  MySQL-python
  paramiko
  pg8000
  sqlalchemy

Installation Scenarios

In the system's current implementation, the manager.py process will schedule large *de novo* assembly jobs and read normalization jobs locally. These processes require large amounts of RAM (20GB+ for Illumina HiSeq), and thus you may choose to run the manager and dispatcher processes on a machine with these resources.

The data_gremlin process must run on a machine with access to all of your data sources via a network connection. If you do not have a high-memory machine inside your local network, this may lead to the scenario where your data_gremlin runs on a different machine than your dispatcher and manager.

If your large-memory machine has SSH access to all of your intended data sources, then the manager, dispatcher, and data_gremlin can all be run on the same machine. In the installation instructions that follow, we will provide instructions for both the single-machine and two-machine installation scenarios.

Note: In future releases, all job types will have a remote execution class, removing the need to run dispatcher and manager on a high-memory machine.

Getting the Source Code

Regardless if this is a one system or two system installation, the first step is to download the source code for the system from our online repository. Please note, if you are performing a two-system installation, the source code will need to be downloaded on both machines involved in the installation.

The Autonomics source code (in Python) can be downloaded from the moroz lab public GitHub account: https://github.com/morozlab/autonomics

The Autonomics source should be either cloned (linux command line: git clone https://github.com/morozlab/autonomics) or downloaded and unzipped into a folder owned by a secure account, as the source directory will be the eventual home of credentials files storing connection details for data sources and remote computational resources.

Be sure to append the path to the cloned or unzipped repository to your PYTHONPATH environment variable.

Setting Up the MySQL Database

Autonomics requires a MySQL relational database system to hold system and project state. The documentation for the database schema can be found in the docs directory of the autonomics distribution. To set up the MySQL database for use with the pipeline, first install the latest version of the MySQL, available from http://dev.mysql.com/downloads/.

Once MySQL is installed, create a database named zero_click on the MySQL server. With the database created, use the two SQL script files, <zeroclick.sql.tables> and <zeroclick.data.for.sql.tables> in the docs directory. Using the root user for your database installation, load both of the SQL scripts, starting with the script containing only the structure of the database tables.

A SQL script can be loaded into the MySQL database with:

```
mysql  -u user -p database_name <script_name>
```

After loading the tables-only file, load the SQL script containing the necessary setup data.

TWO-MACHINE INSTALLATION

Repeat the above installation steps on the high-memory computer running the dispatcher and manager processes.

Installing the Redis Keystore

Autonomics utilizes a redis keystore to speed up the annotation of Gene Ontology and Kyoto Encyclopedia of Genes and Genomes terms.

First, download and install the redis keystore from http://redis.io/

Then download dump.rdb from our public ftp site.

Stop your redis keystore if it is running, transfer the dump file to your redis directory, and restart the redis server. The server should read in the dump file and populate the key store with the necessary annotation mappings.

An example of this procedure can be found at: http://goo.gl/Db68H (stackoverflow)

TWO-MACHINE INSTALLATION

The redis keystore must only be installed on the machine running the manager process.

Modifying settings.py

This module contains system-wide settings for a number of important data paths, wait timers, and account details. Here is a list of variables defined in this module, and how they should be set:

INSTALL_DIR:

This directory is the path to where the Autonomics source code is installed. In the two-machine installation scenario this must be set on both machines.

home_dir:

This is the root directory for all projects. Manager and dispatcher will look for data files here, and in the two-machine installation scenario the data_gremlin will move new run data to this directory. Each system project will have its own sub-folder underneath this directory.

remote_dir:

This is the directory on the remote cluster where temporary files will be created during execution. This variable will be deprecated when the new location system is rolled out in a future release.

ZC_HOST:

This is the IP address/machine name of the server that will be running the manager and dispatcher code. In the single-machine installation scenario, this should be set to localhost.

ZC_USER:

This is the user name that the data_gremlin uses to connect to the server running the dispatcher and manager processes.

ZC_PASSWD:

This is the password data_gremlin uses to authenticate the ZC_USER at ZC_HOST.

ZC_DB_NAME:

The name of the database set up on the Autonomics MySQL server.

Registering Data Sources

Data sources are registered by adding a row to the mysql data_sources table for the source, as well as creating a credentials file in the credentials directory.

Credentials Files

Two example files (pipeline_mail_account & 10.41.128.44) in credentials directory:

```
cat   pipeline_mail_account
host:pop.gmail.com
user:morozhpc
passwd:XXXXXXXX

cat   10.41.128.44
host:10.41.128.44
user:ssh_user
passwd:XXXXX
```

Starting Services

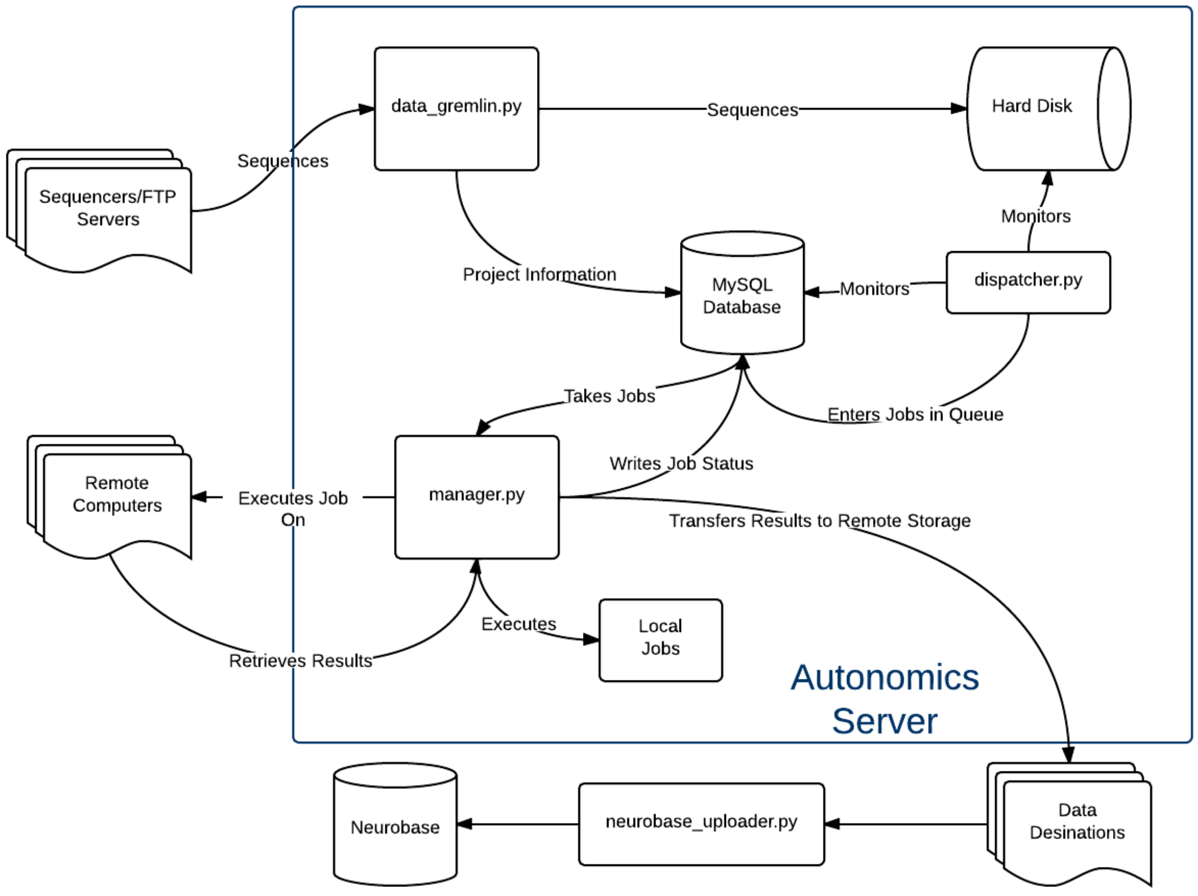
Three services need to be started: manager, dispatcher and data_gremlin
All three jobs run forever.

1) python dispatcher.py &
2) python data_gremlin.py &
3) python manager.py &

II. System Overview

The Autonomics system is separated into a number of system modules written in Python, based upon their function. Autonomics uses a MySQL database as a backend to hold system state, although other database systems should work – as long as they have the appropriate database/table schema.

Schematic of the Autonomics System



A) The data_gremlin process monitors registered data sources (FTP servers and sequencers) for newly completed sequence data. New data is transferred to the autonomics server and analysis jobs are automatically configured. B) The dispatcher.py process detects eligible analysis jobs and schedules them in one of several queues, implemented in the MySQL database. C) The manager.py process removes jobs from the queue and executes them on the designated computational resource (a resource can be a registered compute cluster, server, or the local machine). D) Upon job competition, manager.py collects output files and transmits them to associated data destinations.

Data Detection

Data detection is performed by the data_gremlin process. Once every SLEEP_INTERVAL seconds, the data_gremlin wakes up and checks all data sources registered with the Autonomics system. Data sources are registered by adding a row in the data_sources table for the source, as well as creating a credentials file under the credentials directory.

New data is detected on each data source in a platform-dependent manner. For details on the detection method for each supported data source, please see the source code of data_gremlin.py.

Project Configuration

When the data_gremlin detects a new sequencing run on a data source, it will automatically configure an annotation project for that data with the default configuration settings.

III. data_gremlin

Data_gremlin is responsible for the following tasks:

1) Setting initial configurations in the MySQL database for newly discovered sequencing runs.
2) During configuration, determining the sequencing type (if possible) and setting the appropriate pre-assembly and assembly tasks to be run
3) Moving data from the sequence sources to wherever the manager and dispatcher processes are running when the data is ready.
4) Pushing configuration data from the web server to the compute server (if they are not the same machine)
5) Setting new runs as configured, downloaded, etc

This module configures new runs with the following logic:

- check runname_to_pid to see if the run_name already exists in the table. If so, the run is configured.
- check processed_runs to see if a (source_id, run_name) tuple exists in the table. If so, the run was configured at one point, and someone removed it from the system on purpose.

At this point, we've determined the run is ok to configure.

- enter a project_name for the run using run_name in pn_mapping, returning a project_id

- use the project_id to add jobs for each job in the default_configuration table. Get a list of job_ids for those jobs.
- use the list of job_ids and project_ids to enter a entry for the project in the configuration table for each job in default_configuration.
- determine if the run is paired end, set the appropriate flags for all jobs in the args table
- determine which assembly pipeline to use, and configure the pre_assemble jobs accordingly.

Data is moved with the move_<source_type>_data family of methods.

Configuration is pushed from the web server to the zeroclick compute server with push_configuration.

Configuration data includes (but is not limited to):

- project_name
- job_ids for all configured jobs
- custom arguments for configured jobs (pipeline_args, process_args, etc)
- whether the job is paired-end, has been configured, or downloaded

Configuration data is pushed to the Autonomics compute server so that the MySQL instance there can be maintained as a mirror of the database instance on the web server in the presence of firewalls. Without the firewall, the Autonomics server would access the web server MySQL instance directly.

IV. dispatcher.py

This module runs as a daemon and is responsible for monitoring new project data uploaded to the zeroclick server by the data_gremlin module. New projects found are added to the pn_mapping, run_stats and init_projects tables; all required jobs for a project are added to the jn_mapping & jid_dependencies tables. In addition, projects are checked that all of its jobs are completed

1) Loops continuously, performing the following tasks, sleeping for settings.DISPATCHER_SLEEP_INTERVAL between loops.

2) All jobs in jn_mapping table that (a) are not in queue table, and (b) are not marked started, and (c) have all dependencies satisfied, are inserted into queue table and jn_mapping table set to queued = 'Y'

3) Iterates over all directories found in settings.home_dir, treating each directory as a project_name.

3) For each project_name, checks if the directory for that project (settings.home_dir + project_name) contains SRC_UPLOADED, which marks that data_gremlin.py has finished its upload. If so, does the following for each project.

4) Checks pn_mapping table to see if project_name has been assigned a project_id. (project_id needed to access all other tables). If project in pn_mapping table either: (1) project already initiated or (2) project configuration was submitted via web interface which makes entries in pn_mapping, jn_mapping,

config & args tables (we call these custom projects as opposed to those that use the default job configuration; configuration refers to what jobs to run for a project and which args to use for a job.)

5) If project in pn_mapping table and not in init_projects table then its a custom project and we get a list of the jobs to run from the configuration table. Dependencies for each job are computed from the dependency table and job_ids for each dependency are entered into the jid_dependencies table. The project_id is inserted into the init_projects table.

6) If project is not a custom project we do the same procedure as in the above step except we use the delfault_configuration table. In addition, the project is inserted into the pn_mapping table and the jobs are inserted into the jn_mapping table.

7) Finally, projects that have been initiated but not completed are checked to see if all jobs have finished, if so project is added to completed projects table. If project has an 'upload' job_type entry in the jn_mapping table, this entry is set to started and finished and the time stamps are set.

V. manager.py

The manager process is responsible for taking jobs from the queue of eligible jobs, executing those jobs, monitoring job status, and retrieving and parsing job output. The manager is also responsible for delivering the data generated during job execution to the data destination for the project.

VI. Other System Processes

config_loader.py

This module monitors the directory proj_config/ located in the system's main data directory for configuration files pushed there by data_gremlin.py. This module is necessary to maintain that the Autonomics systems MySQL database contains every project configured on the web server.

This module is run as a daemon and performs the following tasks:

1) Get a list of configuration files stored in settings.home_dir/proj_config/ - configuration files for a project are named [project_name, project_name_global]; see the documentation for push_configuration() in the data_gremlin module for the format of these files.

2) Check for, and open, the global configuration file containing project and run information. If the project exists in the system, update any project-wide settings. Otherwise create a new project in the system and add the system-wide settings. System-wide settings include whether or not the project includes paired-end data, was configured on the web-server, and if it has been downloaded yet.

3) Open the job-specific configuration file. This file contains a line for each configured job for the project. If the jobs exist in the MySQL database on the Autonomics server, update job settings. Otherwise add the jobs and set the job settings.

Note: Please see the Autonomics database documentation for the runname_to_pid, pn_mapping, jn_mapping, args, and configuration tables for detailed descriptions of the various project and job settings.

This process must only be started in the two-machine installation scenario.

neurobase_uploader.py

This module loads data into the comparative Neurogenomics database, as such data arrives, if all dependencies are satisfied. The uploader is a wrapper around the upload.py script, and automatically checks a specified data directory for new analysis data to load. When the data is detected, the uploader checks the file's md5sum against a checksum provided by the autonomics system. If the two checksums match, the file is considered completely transferred and the uploader loads the data file.

Neurobase requires that the assembly sequences are loaded first, followed by quantification for the assembled transcripts. Any data file can then be loaded.

```
usage: neurobase_uploader.py [-h] [--sleep-interval SLEEP_INTERVAL]
                             [--dbhost DB_HOST] [--dbuser DB_USER]
                             [--dbpass DB_PASSWD]
                             [--monitor-directory MONITORED_DIR]

optional arguments:
  -h, --help            show this help message and exit
  --sleep-interval SLEEP_INTERVAL
  --dbhost DB_HOST
  --dbuser DB_USER
  --dbpass DB_PASSWD
  --monitor-directory MONITORED_DIR
```

Neurobase_uploader only needs to be started on a machine hosting a Neurobase instance.