CPU Torrent

Final Report

Aaron Fernandes
Shruti Gandhi
Bhavesh Patira
Swapneel Sheth

Introduction

Sequence analysis jobs often consist of invoking a *utility* service like blast prior to executing a *proprietary* service like prof. These jobs can often be split to execute the utility service remotely and then execute the proprietary service locally using the utility results, while waiting in the local queue for an available host. There are many potential providers for the utility services. This project present an architecture that enables cpu cycle *sharing* (or offloading) to reduce both user wait time and service-provider computation requirements. We do that by leveraging existing throughput-improving installations.

Background

PredictProtein and other protein sequence analysis systems provide services that are often computation-intensive with high cpu costs. Some organizations provide their analysis systems for public use at little or no charge, but there may be *long queues* waiting for cpu cycles to become available. Other organizations might be encouraged to make their systems available, if the cpu costs they would then bear could be reduced. CPU Torrent is service to offload subjobs and submit them on different servers.

Approach

We implemented CPU Torrent by modifying sequence analysis queuing as follows:

- When requests arrive, we split the jobs into utility vs. proprietary jobs
- We send utility job to CPU Torrent while proprietary job waits
- If CPU Torrent eventually returns utility results, we use that as input to proprietary service and schedule for next available local host
- If CPU Torrent returns failure/timeout, schedule utility together with proprietary service

Current GeneTegrate Architecture

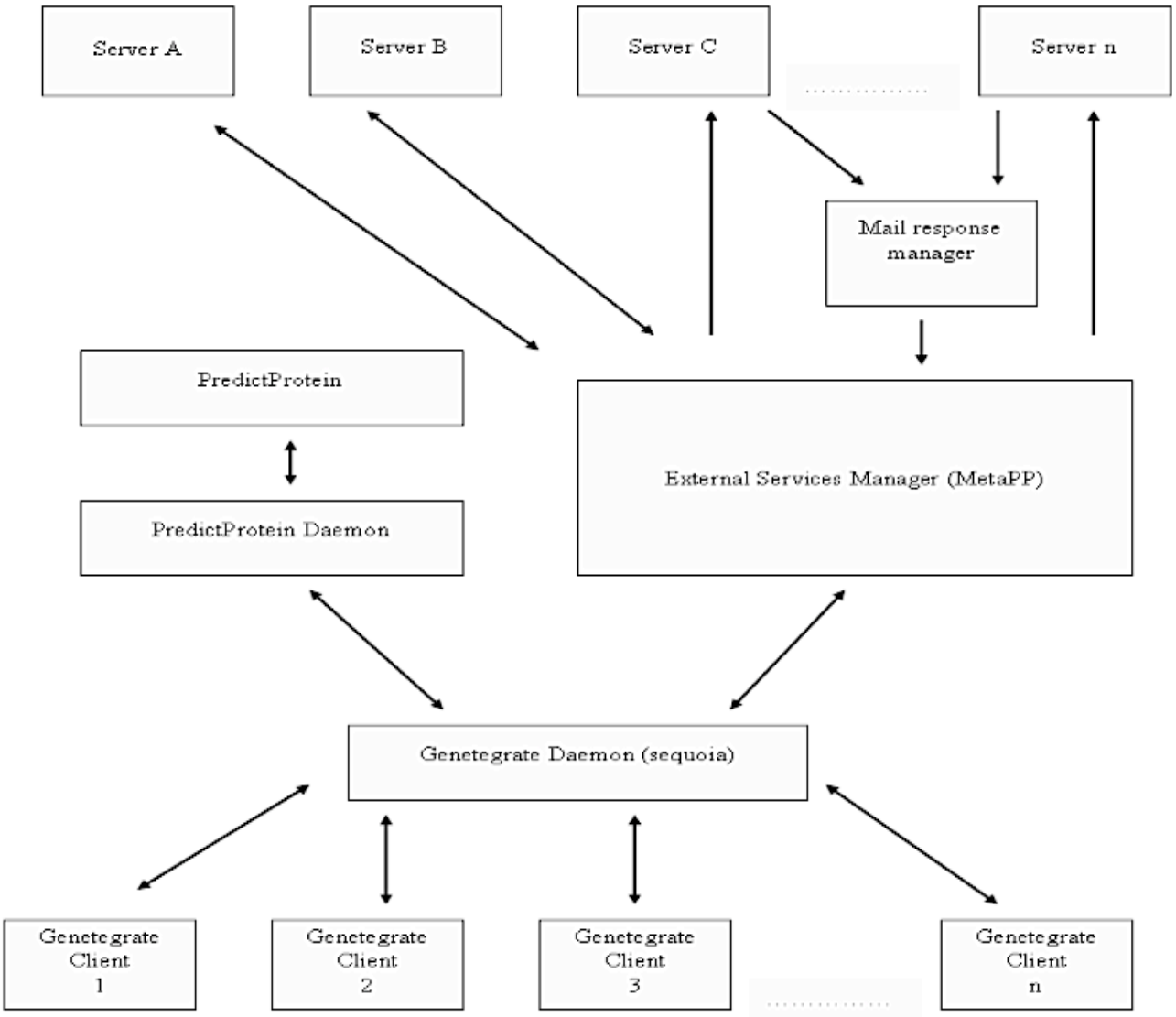

Illustration 1: Current GeneTegrate Architecture

The above diagram depicts how a client accesses a service via GeneTegrate. Only the GeneTegrate Daemon interacts with the GeneTegrate Clients running on individual user systems. PredictProtein is a set of local services that run in the GeneTegrate lab. All requests for services belonging to PredictProtein are directly routed to the PredictProtein Daemon. Requests for third party services are handled by the External Services Manager (MetaPP). It contacts the individual servers where the requested services reside. For those services that return results via email, we have the Mail Response Manager which in-turn forwards the results to the GeneTegrate client.

CPU Torrent enhances the Predict Protein component in the current architecture (figure 1) to figure 2. We introduced new components to better handle load on Predict protein and offload all utility jobs.

Modified Architecture

make caption same as the title "modified arch"

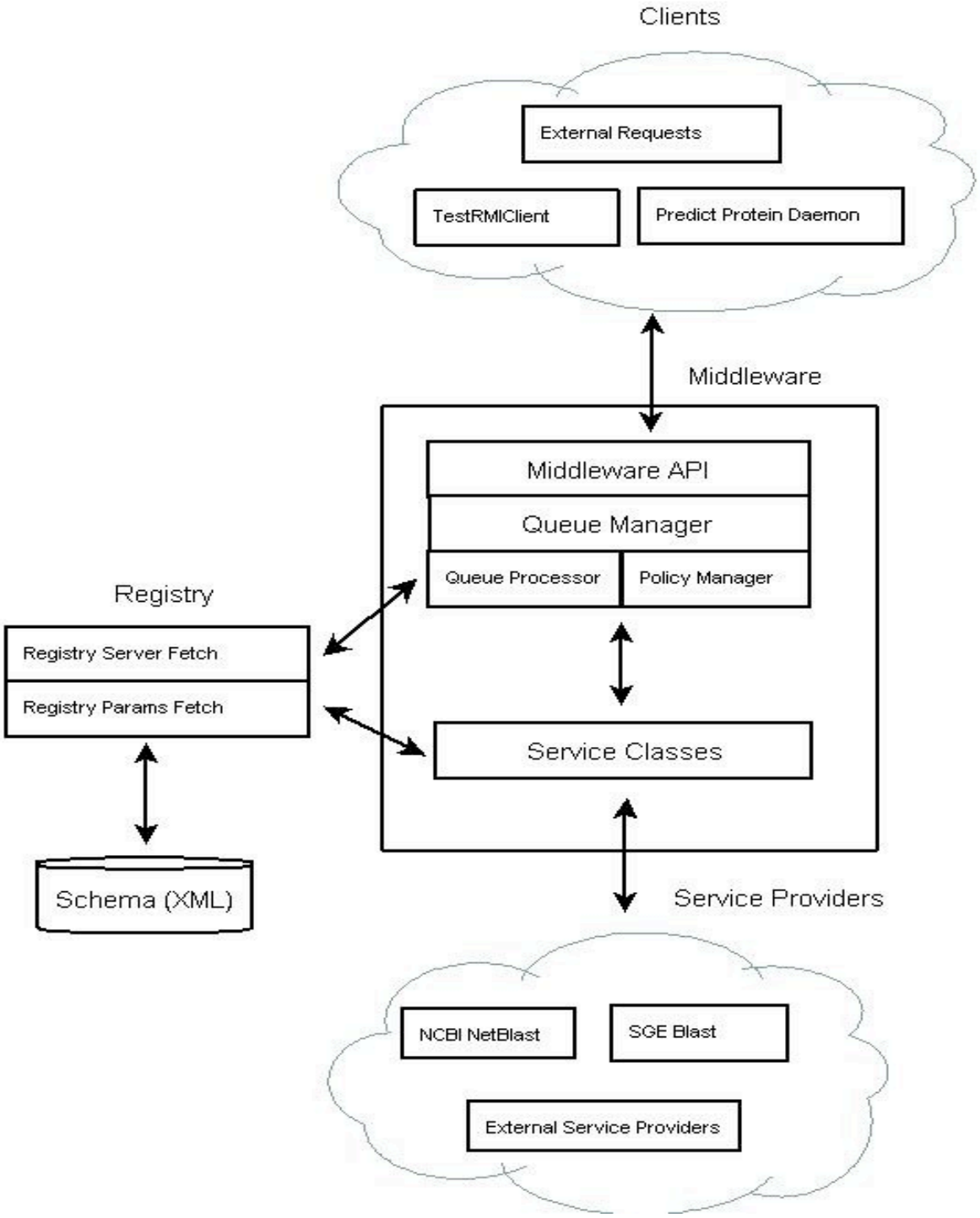

Figure 2. Component Architecture

make 3 diagrams – one for pp and cpu torrent, one for generic system and cpu torrent and one for internal details of cpu torrent blowup

redo components table, remove scanner_pp, etc

insert blurb for each box

location on beach will change, prolly fine

Components

Component	Description	Location on gaia (~ = /nfs/home1/ssheth)	Location on beach (~ = C:\geneWkspace\Genetegrate)
Clients	scanner_pp.pl, others	~/Code/clients	
Services get exact names!!!	Consists of two xml files, Params Fetch and Server Fetch	~/Code/services	
Middleware	Queueing Manager daemon, Queue Processor, Policy Manager	~/Code/middleware	
Source	Contains wrapper for different services like SGE, NCBI, Others	~/Code/source	

How do I get it?

1. ssh to maple.bioc.columbia.edu
   (eg: ssh <username>@maple.bioc.columbia.edu)
2. ssh to gaia.c2b2.columbia.edu
   (eg: ssh <username>@maple.bioc.columbia.edu)

   scp ~ssheth/.....
3. scp ~/Code/Genetegrate.tar.gz ./ (this will copy Genetegrate.tar.gz to the current directory of your local machine
4. There you go! Now you have it!

How do I install it?

replace <.+> with the actual thing

1. tar -xvzf Genetegrate.tar.gz (this will unpack the folder structure into your current directory.
2. cd Code
3. export CLASSPATH=/Code/ typo: =~/Code/
4. javac -d . source/*.java (To build all the service classes)
5. javac -d . middleware/*.java (To build the middelware)
6. javac -d . services/*.java (To build the registry classes)
7. rmic <middleware>.CPUTorrentRMIServer (To set up the RMI Interface. The Stub that is formed should be used by the clients who would like to invoke the Middleware to run thier jobs)

   explain the rmi process slightly here
8. rmiregistry & (To set up the RMI interface)
9. java <middleware>.CPUTorrentRMIServer (Starts the server)

Configuration files description

Note: all paths are relative to the Code directory

A) Setup middleware/Simple.conf

Simple.conf contains a row per service class. The fields are in the format
<serverid>:<percentage1>
<serverid1>:<percentage2>

make it consistent. server0 = percentage0 and so on

The percentages of all the servers should add upto 1. This percentage indicates the percent of total jobs that get routed to that corresponding server.

B) Setup services/Services.xml to include all the servers

<services>
<service name="xyz">
<server>
<id> server_id </id>

add proper eg here, some normalised load value, etc

<load> how_to_find_load </load>
</server>
</services>

The above xml should be modified to include all the servers that provide a service.

1. Replace "xyz" with the service name.
2. Replace "server_id" with an id you want to assign this server
3. Replace "how_to_find_load" with a command on how to find load for that server.

C) Setup services/Servers.xml to include all server parameters

<servers>
<server id="server_id" name="xyz.columbia.edu"><parameters><parameter required="true"
name="name_of_required_parameter1">param_value1</parameter>
<parameter required="false" name="name_of_optional_parameter2">
param_value2 </parameter></parameters></server>
</servers>

use real eg

The above xml should be modified to include all optional and required parameters for a server.

1. In server, replace "server_id" with the serverid for that server from Services.xml
2. In server, edit "name" to be some name for that server
3. In parameter, change "required" to "true" or "false" based on if the parameter is required or optional.
4. In parameter, change "name" to the name of the parameter
5. In parameter, change "value" to the value for that parameter.

How to test the Middleware?

1. ssh to maple.bioc.columbia.edu
   (eg: ssh <username>@maple.bioc.columbia.edu)

2. ssh to gaia.c2b2.columbia.edu
   (eg: ssh <username>@maple.bioc.columbia.edu)

3. cd Code

4. javac -d . client/*.java (To build all the service classes)

5. java Client.TestRMIClient (have _stub.class with the client

6. The output is found in clientcopy<jobid>.blastpgp

explain the _stub.class part

typo: java client.

Description of the wrapper service class

The Wrapper Service class is called by the QueryProcessor, when it chooses to invoke that particular service. The Query Processor will use the Servers.xml to obtain the configuration parameters for that Server. The QueryProcessor will pass to the Service class constructor the Job object, the Hashtable of the required parameters for that service, and the Hashtable of the optional parameters for that service. The Wrapper Service class will:

1. Implement the Invoker.java interface and be stored under /source directory. This is required as the Middleware uses java reflection to invoke the Wrapper service class. It will also extend the Java Thread class, and implement the run() method to run each job in a new thread.

2. Take as a second parameter, a Hashtable that contains the required parameters.

3. Take as a third parameter, a Hashtable that contains the optional parameters.

4. Generates the output files with the name <jobid>.blastpgp

5. Have some timeout mechanism (preferably using monitors), to indicate if the job has been running for too long without any results

6. Update the status of the job in the Job object it has been passed. The status values are obtained from the Job java class.

Status Number	Description
0	New Job
1	Job Processing
2	Job Completed
3	Job Failed

Flow during Startup

1. RMIStub.java communicate with the CPUTorrentRMIServer, which in turn initiates QueueManager.java.

2. The queue which will hold all the incoming requests will also be instantiated as a Hashtable at this point. Also a notify counter is set up. This will be used to count the number of incoming notifications to the Queue Manager. When a new request comes in the QueueManager increments the notify counter.

3. QueueManager.java inturn instantiates QueueProcessor.java and PolicyManager.java. Note that the Queue Processor will be started on a seperate thread, and will wait to be notified by the QueueManager when a new Job request comes in. When the QueueProcessor is notified that a new job is waiting on the queue, it decrements the notify counter.

Data/Functional Flow

1. A client like scanner_pp.pl contacts the Middleware API interface (RMIStub.java) in our middleware to run a job. This is done through the submit method which can be invoked by the client using the RMI stub. The client provides a Fasta file sequence, and gets a Job Id, which can be used to query the status of the job and get the results of the Job.
2. The implementation of the RMI stub - the CPUTorrentRMIServer will invoke the addJob method of the QueueManager. This inserts the new Job in the queue, increments the notify counter and notifies the QueueProcessor about the new Job.
3. QueueProcessor.java contacts ServerFetch.java by providing it a service name. ServerFetch.java returns list of all servers that provide that service
4. When the QueueProcessor is notified, it extracts the job from the queue. QueueProcessor.java contacts PolicyManager.java to get the most readily available server based on the policies based on the heuristic.

PolicyManager.java return a serverid for one of the servers from SGE, NCBI, etc to run the job.

5. The Queue Processor then invokes the corresponding class for the service on a seperate thread. It also updates the status of the job on the queue to indicate that it is being processed. The particular service class contacts ParamsFetch.java in the registry with a serverid to get all required and optional parameters.
6. When the Job is completed, its status is set to completed. When the client asks for the result of the Job, the job result will be sent back and the job will be removed from the queue. This is done by reading the output file generated by the Wrapper service and writing back to the client as a byte array.
7. If the Job fails (due to an exception), the retry counter of the job is incremented, the job status is reset to new and the Query Processor is notified.

Future Scope

1. End to end integration with Predict Protein

2. Organize communities of utility and proprietary service providers and users

3. User jobs prioritized based on "karma rating" derived from their domain's (or other identifiable unit's) historic/recent contribution of "quality of service" (max cpu cycles, min queue wait time) to the community

4. Analogy to BitTorrent imperfect, since BitTorrent can work with ad hoc clients joining/leaving a distribution community on the fly since real-time rather than historic contribution used

5. Define policies to enhance heuristic.

List of References

1. GeneTegrate
   http://wiki.c2b2.columbia.edu/genetegrate/index.php/Genetegrate_Home

2. Predict Protein
   http://www.predictprotein.org/

3. Meta PP
   http://www.predictprotein.org/newwebsite/meta/submit3.php

4. BitTorrent
   ttp://en.wikipedia.org/wiki/BitTorrent