CPU Torrent

Final Report

Aaron Fernandes
Shruti Gandhi
Bhavesh Patira
Swapneel Sheth

Table of Contents

Abstract

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 presents 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.

Introduction

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 a service to offload subjobs and submit them on different servers.

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, we schedule utility job together with proprietary job.

Current Genetegrate System

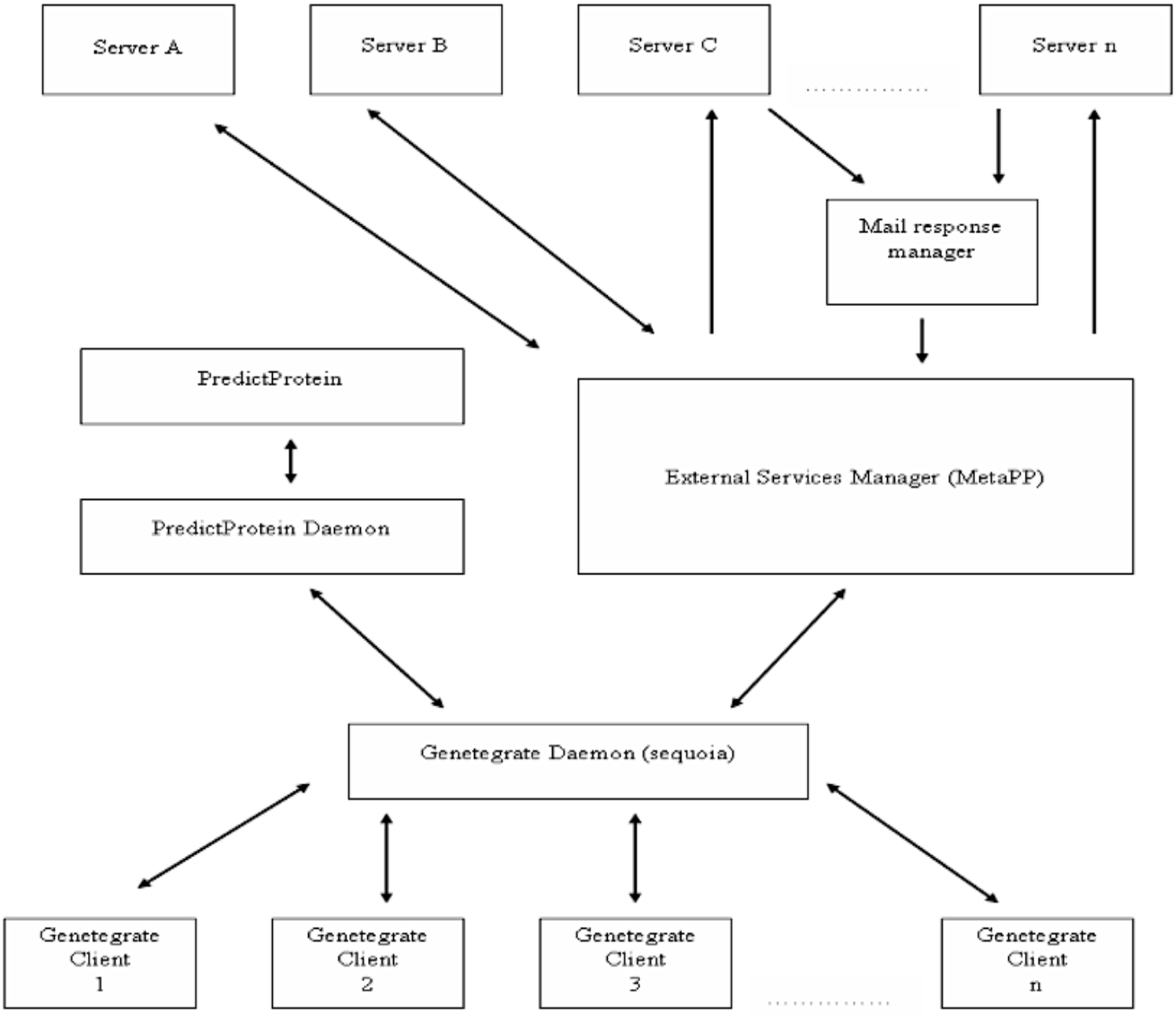

Illustration 1: Current Genetegrate Architecture

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. 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.

PredictProtein daemon runs on a cluster called *Sequoia*, which has 23 nodes. Each node has two CPUs, thus allowing a maximum of 46 jobs to be processed simultaneously. Incoming jobs are processed as follows:

1. A new job is put on a queue.

2. When a node becomes available, the PredictProtein daemon takes a job from the head of the queue, and dispatches it to the node for processing.

3. PredictProtein periodically checks a shared location for the results of each dispatched job.

4. When results become available, they are sent back to the requester (GeneTegrate or individual users).

In order to process a job, the input protein sequence is usually first run through a utility job (*blast*, *clustal*, etc.) and then the actual requested job is executed. As jobs have to wait on the queue for a node to become free, the waiting time on the queue is large. Utility jobs like blast and clustal are offered by various institutions. The idle time spent on the queue can actually be used to process the utility job elsewhere.

CPU Torrent enhances the Predict Protein component in the current architecture (Illustration 1) by adding new components to better handle load on PredictProtein by offloading utility jobs.

CPU Torrent

1. Alternative Approaches

There were two alternative approaches to solve the problem. They are as follows:

1. **"Split Only If Needed" Approach**

   In this approach, whenever a new job is submitted, we first check if there is a free node available. If there is one, the job is sent to that node directly. If not, the job is put on the queue and the utility part of the job is offloaded to be processed elsewhere.

2. **"Split All" Approach**

   In this approach, all incoming jobs are split and the utility part of the job is processed separately. Once this is done, the job is added to the queue for processing and sent to a node when available.

We decided to use the "Split All" approach, as it would help in reducing the load on the Local Servers.

2. Description

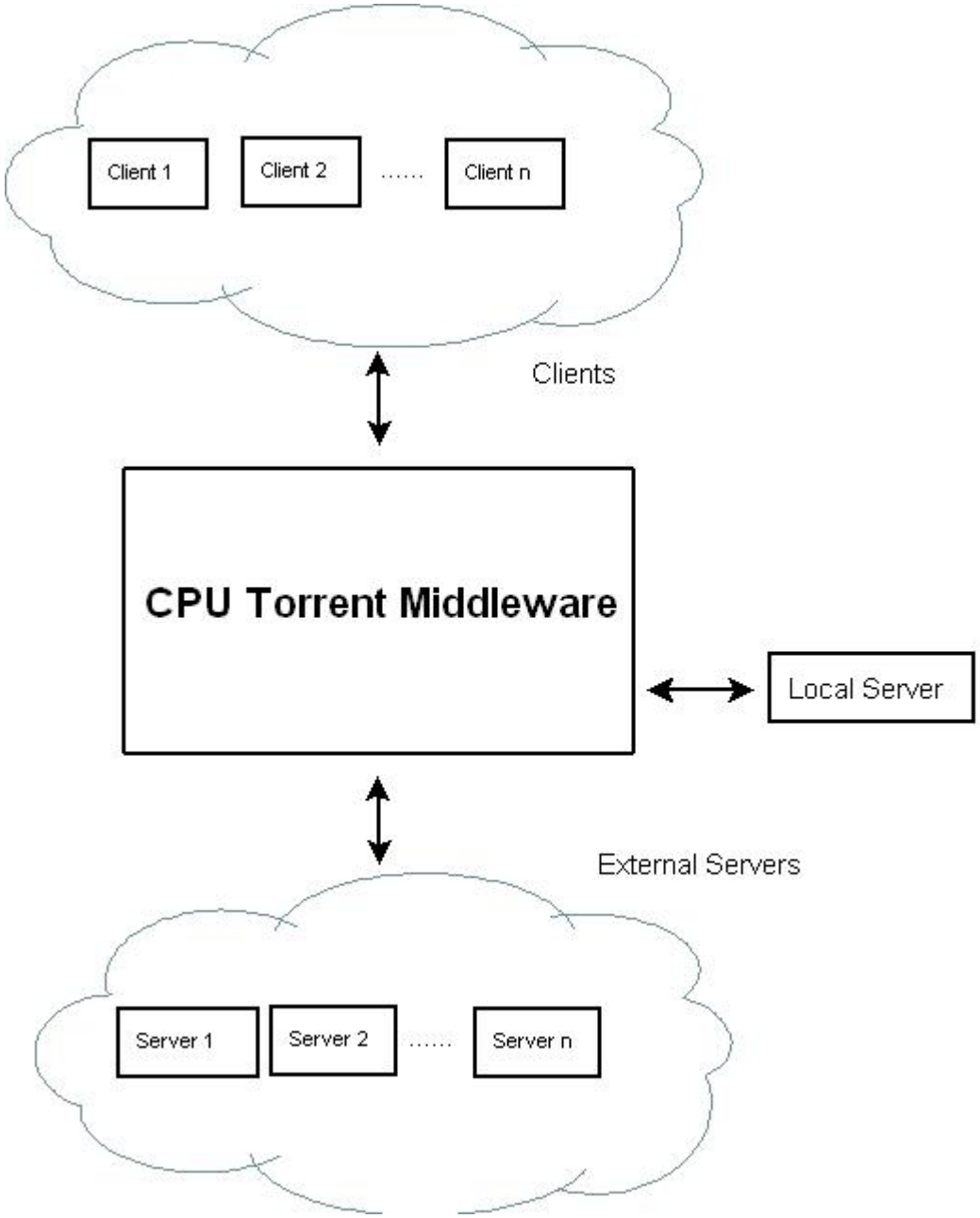

Illustration 2: CPU Torrent and a Generic System

The above diagram shows the general structure of the CPU Torrent system. The clients submit jobs to the middleware. The middleware is responsible for choosing sites to process the jobs – either on the Local Server or by offloading them to the External Servers.

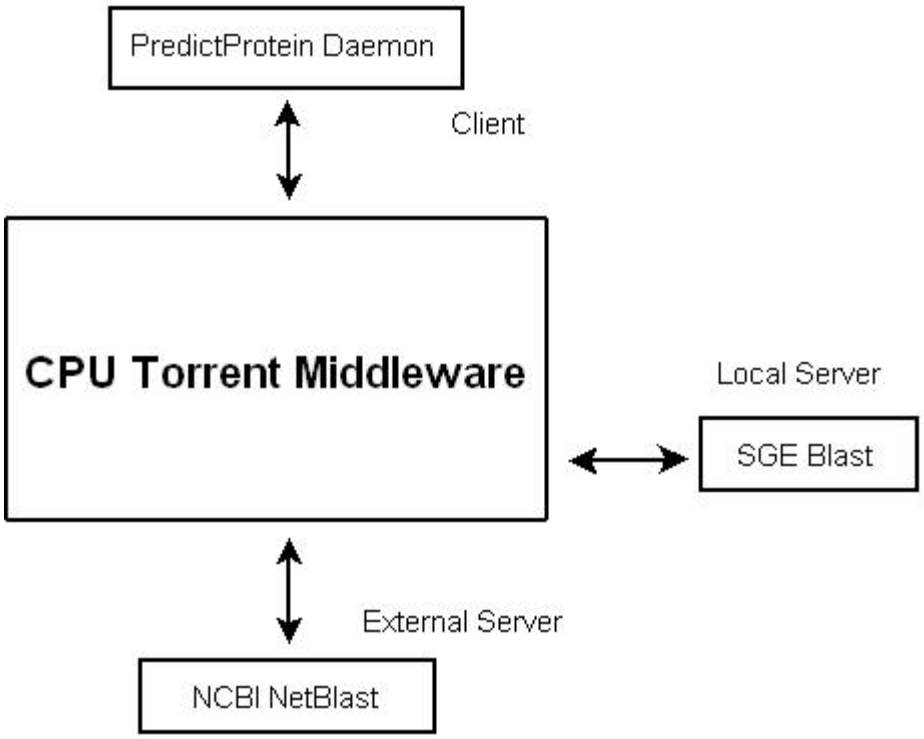

Illustration 3: CPU Torrent and PredictProtein

The above diagram shows the general structure of the CPU Torrent system with PredictProtein. The PredictProtein daemon is the client. It submits jobs to the CPU Torrent middleware. The utility job in this case, is *blast*. There are two possible options where it can run:

1. SGE Blast (Local Server)

   This is the local installation of *blast* on the Sequoia/Gaia clusters which is accessed via the Sun Grid Engine (SGE)

2. NCBI Netblast (External Server)

   This uses NCBI's *netblast* program to submit jobs to the remote NIH servers and get the results back.

3. Internal Details

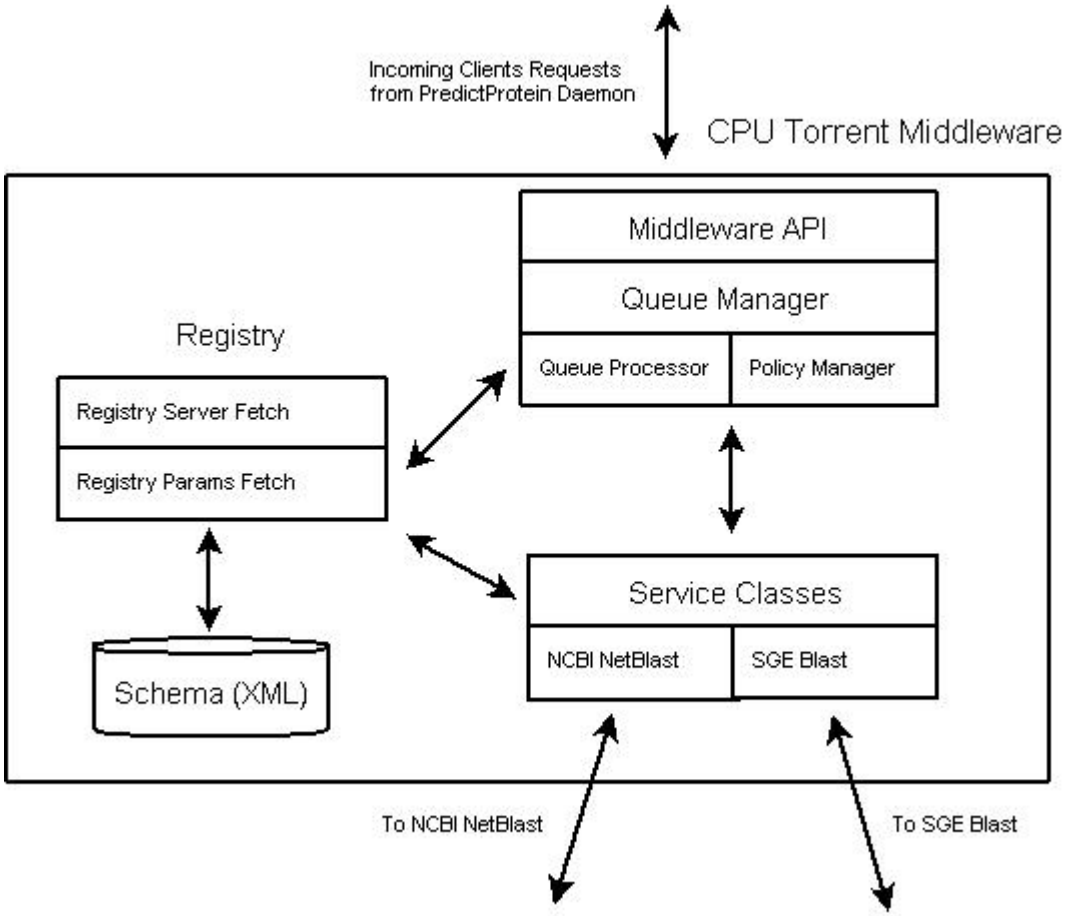

Illustration 4: CPU Torrent Middleware Internal Details

The above diagram shows the internal details of the CPU Torrent middleware. The various components are:

a) Middleware API

The Middleware API is the interface that PredictProtein uses to interact with the CPU Torrent Middleware. It is implemented using Java's Remote Method Invocation (RMI) and supports the following methods:

i. Submit a Job

```
String submit(byte[] fastaFile)
```

This method takes a Fasta Sequence as Input (as a Byte Stream) and returns a String which represents the Job ID. This ID serves as a handle to this job.

ii. Query the Status of a Job

```
int getStatus(String jobId)
```

This method takes a Job ID as Input and returns an integer representing the status of the job. The various status codes are shown below:

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

Table 1: Job Status Codes

iii. Get the Results

```
byte[] getResult(String jobId)
```

This method takes a Job ID an input and returns a Byte Stream containing the results of the job, which could be one of the following:

1. The Actual Results, if the job completed successfully.
2. "Job not found", if an invalid Job ID was passed.
3. "Failed", if the job did not complete for some reason.
4. "Processing", if the job is still being processed.

b) Registry

The Registry keeps track of the various Servers registered with the middleware and what services each of these servers provide. These details are maintained in the form of two XML files. For more details about the specific contents, see Appendix 1.d.ii

The component ServerFetch is used to retrieve a list of servers for a given service. It uses the Services.xml file for this. The component ParamsFetch is used to retrieve the various parameters needed to invoke the service on a particular server. It uses Servers.xml file for this.

c) Queue Manager

The Queue Manager provides basic features for managing the middleware queue – jobs are added to and removed from the Queue via the Queue Manager. The results of a job are also retrieved via the Queue Manager. It is implemented as a Hashtable.

d) Queue Processor

The Queue Processor processes the Job Queue. Monitors activate the Queue Processor when new jobs are added to the queue. The Queue Processor uses the Policy Manager (see below) to decide which server to send a job to.

e) Policy Manager

The policy manager uses a configuration file to determine the 'best' server to which a job can be sent. For more details, see Appendix 1.d.i

f) Service Wrappers

For each (service, invocation mechanism) pair, we define a Wrapper, that handles the specifics of how a job is submitted and how results are retrieved. For more details, see Appendix 3

4. Detailed Data Flow

a) Flow during CPUTorrent Middleware Startup

1. Execution begins when the RMI Server is started and begins listening at port 1099. The Server instantiates the Queue Manager.

2. Queue Manager in turn instantiates the Queue Processor and the Policy Manager. The Queue Processor starts in a separate thread, and will wait to be notified by the Queue Manager when a new Job request comes in.

b) Data/Functional Flow

1. A client contacts the Middleware (using the Middleware API) to run a job. The client provides a Fasta file sequence, and gets a Job Id.
2. The RMI Server inserts the new Job in the queue and notifies the Queue Processor.
3. Queue Processor uses Policy Manager to get the appropriate server based on the specified policies. Policy Manager returns a server ID for one of the servers from SGE, NCBI, etc to run the job.
4. The Queue Processor uses the Server ID to obtain the list of required and optional parameters (using *ParamsFetch*).
5. The Queue Processor then invokes the corresponding wrapper in a separate thread. It also updates the status of the job on the queue to indicate that it is being processed.
6. When the Job is completed, the Wrapper sets the Job's status to “Job Completed”.
7. When the client asks for the result of the Job, the job results are sent back through the RMI Server, and the job is removed from the queue by the Queue Manager.
8. If the Job failed, it is retried for a specific number of times after which a “Job Failed” message is returned to the Client.

Conclusion

We successfully implemented the CPU Torrent middleware that offloads *BLAST* jobs to *NCBI's netblast*. This was done using a probability-based model, where the probability of sending the job to NCBI could be set using a configuration file.

Future Scope

1. End to End Integration with PredictProtein.
2. Improve heuristics to take into account load on various machines, network overhead, etc. in order to improve how jobs are processed.
3. Modify the components to support multiple services (current system supports various services, but only one type of service for a given installation of the middleware).

Appendix

1. Installation of CPU Torrent

a) Getting the Code

i. Log into the *gaia* cluster from your host (e.g. maple, adgate, etc.)

```
ssh <username>@gaia
```

ii. Copy the tarball to your directory

```
scp ~ssheth/Code/cpu-torrent.tar.gz ./
```

Alternatively, the code can also be downloaded via the 2 methods shown below:

1. Via Windows Remote Desktop Connection

   Connect to `beach.cs.columbia.edu`. The tarball is located at `C:\CPU-Torrent-Backup\src\cpu-torrent.tar.gz`

2. Via CVS

   The CVS Repository is located on `york2.cs.columbia.edu`. The CVSROOT is `/proj/psl-cvs` and the module to check out is `Cpu-Torrent/src`.

i. Compilation and Environment Setup

i. Untar the tarball (assuming that you are in the <dest> directory)

```
tar -xvzf cpu-torrent.tar.gz
```

ii. Change to the Code Directory

```
cd Code
```

iii. Set the CLASSPATH variable to include the Code Directory

```
export CLASSPATH=<path_to_dest>/Code/:$CLASSPATH
```

iv. Compile the Source Code

```
javac -d . source/*.java
```

```
javac -d . middleware/*.java
```

```
javac -d . services/*.java
```

v. Create RMI stub for the server

```
rmic middleware.CPUTorrentRMIServer
```

b) Deployment

i. Copy the server stub to the client code directory (assuming the client code is in the *client* directory)

```
cp middleware/CPUTorrentRMIServer_Stub.class client/
```

ii. Start the RMI registry (background if necessary)

```
rmiregistry
```

iii. Start the Middleware server (background if necessary)

```
java middleware.CPUTorrentRMIServer
```

c) System Configuration

Note: all paths are relative to the Code directory

i. middleware/Simple.conf

Simple.conf contains one row per server. The fields are in the format

```
<serverid0>:<percentage0>
<serverid1>:<percentage1>
```

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

A sample file is shown below:

```
0:0.5
1:0.2
2:0.3
```

ii. services/Servers.xml

This xml file contains the details regarding how a particular service can be invoked on a particular server. It has the following schema:

```
<servers>

    <server id="server_id" name="server_name">

        <parameters>
```

```
            <parameter required="true" name="parameter_name">
                parameter_value
            </parameter>
            <parameter required="false" name="parameter_name">
                parameter_value
            </parameter>
        </parameters>
    </server>
</servers>
```

The various parameters areas follows:

1. "server_id" is to identify a particular server. It corresponds to the server id in Simple.conf and Services.xml file.
2. "server_name" is the name of the server.
3. In parameter, *required* indicates if the parameter is required or optional.
4. "parameter_name" is the name of the parameter.
5. "parameter_value" is the value for that parameter.

A sample file is shown below:

```
<servers>
    <server id="0" name="ncbi netblast">
        <parameters>
            <parameter required="true" name="classname">
                InvokeNetBlast
            </parameter>
            <parameter required="true" name="command">
                C:\\netblast\\blastcl3
            </parameter>
            <parameter required="false" name="program">
```

```
                blastp
            </parameter>
        </parameters>
    </server>
    <server id="1" name="SGE Blast">
        <parameters>
            <parameter required="true" name="classname">
                InvokeSGEBlast
            </parameter>
            <parameter required="true" name="command">
                /usr/pub/molbio/perl/blastpgp.pl
            </parameter>
            <parameter required="false" name="maxali">
                3000
            </parameter>
            <parameter required="true" name="program">
                /opt/sge/bin/lx26-amd64/qsub
            </parameter>
            <parameter required="false" name="priority">
                0.01
            </parameter>
        </parameters>
    </server>
</servers>
```

NOTE: In the actual Servers.xml file, an entry for each server must be on the same line, i.e., without any newlines.

iii. services/Services.xml

This xml file maps one or more servers to a particular service. It is not used in the current distribution of the middleware. It has the following schema:

```
<services>
    <service name="service_name">
        <server>
            <id>
                server_id
            </id>
            <load>
                server_load
            </load>
        </server>
    </service>
</services>
```

The various parameters are as follows:

1. “service_name” is the service name.
2. "server_id" identifies a particular server.
3. "load" could be one of the following:
    a) A command used to find out the current load on the server
    b) The probability with which to send jobs to this server

NOTE: In the actual Services.xml file, an entry for each service must be on the same line, i.e., without any newlines.

iv. Other Configuration Parameters

- Number of retries per Job

  This is set in the middleware/QueueManager.java (line 18). The default value is 5.

- Timeouts

  Each Service Wrapper defines a timeout value before which a retry is triggered. For example, in source/InvokeNetBlast.java, the time is set to 120 seconds (line 25)

2. Testing CPU Torrent

The tarball contains a test client in the client directory. This client can be run as follows:

1. Compile the client code

   ```
   javac -d . client/*.java
   ```

2. Start the client

   ```
   java client.SubmitJob
   ```

On running the test client, we see the status codes for the jobs submitted. The explanation of these status codes can be found in Job Status Codes. Once, the client completes execution, the output is found in clientcopy<jobid>.blastpgp files (where jobid is the ID for the job submitted by the client; in this example, jobid = 0, 1, 2, 3, 4). A sample Fasta Input File is shown below:

```
>1tof_
GGSVIVIDSKAAWDAQLAKGKEEHKPIVVDFTATWCG
PCKMIAPLFETLSNDYAGKVIFLKVDVDAVAAVAEAA
GITAMPTFHVYKDGVKADDLVGASQDKLKALVAKHAA
```

A snippet of the output returned by running blast on the above file is shown below:

```
BLASTP 2.2.15 [Oct-15-2006]


Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.

Query= 1tof_
         (111 letters)

Database: All non-redundant GenBank CDS
translations+PDB+SwissProt+PIR+PRF excluding environmental samples
from WGS projects
           4,904,269 sequences; 1,693,139,330 total letters



                                                                 Score    E
Sequences producing significant alignments:                      (bits) Value

sp|P80028|TRXH_CHLRE Thioredoxin H-type (TRX-H) (Thioredoxin CH1...   196   2e-049
pdb|1TOF|  Chain  , Thioredoxin H (Oxidized Form), Nmr, 23 Struc...   196   2e-049
pdb|1EP8|A Chain A, Crystal Structure Of A Mutated Thioredoxin, ...   193   2e-048
gb|AAU93947.1| thioredoxin H [Helicosporidium sp. ex Simulium jo...   108   6e-023
gb|AAK72483.1| thioredoxin [Branchiostoma belcheri]                  100   3e-020
emb|CAI64401.1| thioredoxin h2 protein [Zea mays]                     95   1e-018
gb|AAM67018.1| thioredoxin [Arabidopsis thaliana]                     94   2e-018
emb|CAA84610.1| thioredoxin [Arabidopsis thaliana]                    94   3e-018
ref|NP_173403.1| ATTRX4 (thioredoxin H-type 4); thiol-disulfide ...    94   3e-018
pdb|1WM7|A Chain A, Solution Structure Of Thioredoxin Type H Fro...    93   4e-018
```

Text 1: Snippet of Output after running Blast

While a job is being processed, the client periodically queries the middleware for the status. A sample of the status on the console is shown below:

```
Status of job 1 is: 1
```

When the job completed, the following status message is printed on the console:

```
-----------------------------
job 1 completed with status 2
```

Meanwhile, at the middleware console, the following gets printed:

```
RMI Server: Server Registered
```

Here, the middleware server gets registered with the RMI Server.

```
Q Manager: Job 1 added
```

Job 1 is received by the Queue Manager and added to the queue.

```
Q Manager: size of queue:2
```

The size of the queue is shown, after being incremented.

```
Queue processor: sending job to server: 1
```

The Queue Processor sends the received job to Server with ID = 1.

```
SGEBlast: writing temp fasta input file: SGE1.f
```

The input Fasta sequence is written to a temporary file (SGE1.f).

```
Executing /usr/pub/molbio/perl/blastpgp.pl ./Code/SGE1.f
     maxAli=3000 fileOut=./Code/1.blastpgpabout to call blast
```

The job is submitted to the Sun Grid Engine using the above command.

```
Executing /opt/sge/bin/lx26-amd64/qsub -p 0.01 ./SGE1.sh
```

The status of the job is checked using the above command.

```
Checking SGEBlast
Looking for ./1.blastpgp
**********SGEBlst: I'm done***********
```

Here, we check if the output file is written to the filesystem and if it is, the job is complete.

3. Service Wrappers

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

1. Implements the Invoker.java interface and is stored under *Code/source* directory. This is required as the Middleware uses Java Reflection to invoke the Wrapper class.

2. Extends the Java Thread class, and implements the run() method to run each job in a new thread, thus making it multi-threaded.

3. Takes as first parameter, a *Job* object that contains a variety of job specific information.

4. Takes as a second parameter, a Hashtable that contains the *required* parameters.

5. Takes as a third parameter, a Hashtable that contains the *optional* parameters.

6. Generates the output file with the name *<job_id>.blastpgp*

7. Implements some timeout mechanism.

8. Updates the status of the job in the *Job* object depending on whether the job succeeded or failed.

List of References

1. GeneTegrate
   http://wiki.c2b2.columbiaedu/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
   http://en.wikipedia.org/wiki/BitTorrent