Community Market Dispatcher System

By Michael Jolson & John Tantalo

Abstract

Every day, countless computer processor cycles go to waste as machines are left idle by their users. This is a waste of the community's money and resources, as fragmentation requires many more computers to be purchased than are actually required to fulfill everyone's computation needs. One way of addressing this problem is to treat computation as a commodity, maintaining a set of servers in a warehouse, and selling computation time on the servers. However, with things as they are now, there would still be plenty of server and desktop machines distributed throughout the community, spending plenty of time idle. We chose to solve this problem by creating an application which would allow people to enter into an open batch computation market, selling spare cycles on their machines and purchasing processing time on other people's machines as needed. In this way, new machines would only need to be purchased when the entire community needs more processing power, as opposed to when a single individual or group needs more processing power.

Introduction

In our project, we created a community market dispatching system. The purpose of the system is to accept batch jobs from community members, store each job's execution state, and then dispatch the jobs to idle machines owned by community members. A job is the execution of a program – analogous to the execution of an independent process within a normal computer. In our system, a job is the execution of a Python function stored in a module. Each machine will run a job for a certain amount of time before packaging the new state up and committing it to the dispatch server. Job scheduling is performed as a function of how much a community member is willing to pay for job execution, with high-paying jobs being given preference for executing on faster machines. In this way, there is only incentive for community members to purchase new machines when the cost of executing a job on the open market exceeds the cost of buying a new computer (which is an indication that the community needs more processing power).

To make our lives easier, and to ensure that each job could be executed on any arbitrary platform, we chose to limit the jobs we would accept and only take Python programs. Python is a cross-platform scripting language which can easily be used to perform complex computations. To ensure that we can execute each job on any arbitrary platform at any time, we use the Stackless Python implementation (www.stackless.com) for program execution. Stackless Python uses its own cross-platform stack and allows seamless interruption and resumption of program execution through tasklets. Each tasklet is an independent thread which can be scheduled, executed, and interrupted at any given point in time.

In our project, we provide a prototype community market dispatching system. We provide four different parts of the project: the Dispatch Server, Computational Clients, a Web Interface, and a Standalone Application Layer. We have implemented Dispatch Server and Computational Client software in the Python scripting language, Web Interface software in the PHP scripting language, and a proof-of-concept standalone application in the Perl scripting language.

Related Work

Special-Purpose Computing

Distributed computing projects in the past have typically focused on a single goal, with all clients performing the same functions on elements of a very large dataset. These projects usually have volunteer clients, as their aims are scientific in nature. Clients run software with built-in functions, such as data mining or bioinformatics methods, on data downloaded from a central server. Once the client has completed its computation, the result of the analysis is sent back to the server.

One of the most well-known distributed computing projects is SETI@home, operated by the University of California, Berkeley, in which clients scan radio telescope data for signals of extraterrestrial intelligence. SETI@home is part of the Berkeley Open Infrastructure for Network Computing (BOINC), which coordinates many volunteer distributed computing projects. Other projects include Einstein@home (LIGO, astrophysics), Grid.org (United Devices, medicine), Folding@home (Stanford, biology), and ClimatePrediction.net (Oxford, climate change).

General-Purpose Computing

The interest in volunteer special-purpose distributed computing projects has inspired projects aimed at commoditizing general-purpose distributed computing. One such project is CPUShare (www.CPUShare.com), which supports all operating systems and architectures. Clients running CPUShare are credited for units of computation, and in turn CPUShare takes a commission. CPUShare has been in development since 2004, and has recently began accepting jobs from "CPU buyers."

Application

Requirements

To implement a community computation market, the following conditions must be met:

1. A central dispatch system shall store a list of batch jobs and computational clients which can process the jobs.
    a. Jobs shall be stoppable; a computation client running a job must be able to stop a job at any point in time and submit it back to the central dispatch system.
    b. Each job shall be executable on each computational client - i.e. jobs shall be cross platform and shall be able to be executed on one platform, stopped, serialized, and then executed on a totally different platform.
    c. Computation client software shall be available for many different platforms, using different system architectures and/or operating systems.
    d. The central dispatch system shall use a database for storing all system information.
    e. The central dispatch system shall maintain a ready-list of jobs waiting for execution.
2. Jobs shall have prices associated with them which shall be paid to the clients that execute them. The price of a job is a constant set by the job's owner.
3. The central dispatch system shall measure the speeds of clients at regular intervals.
4. The central dispatch system shall dispatch ready jobs to clients for execution.

a. The central dispatch system shall match jobs to clients such that faster clients receive higher-paying jobs.
b. Clients shall negotiate a time period to execute the jobs with the central dispatch server.
c. Clients shall execute a job, stop the job, serialize the job state, and then submit the new job state back to the server in no more than the negotiated amount of time. If the server does not receive job state within the negotiated amount of time, the job dispatch is terminated, the job state is rolled back to the previous state, and the job is added back to the ready-list of jobs waiting to execute.
d. When the central dispatch system receives a new job state within the negotiated period of time, the system shall commit that job state to the database and add the job back to the ready-list.

5. A web interface shall exist for performing administrative functions.
    a. The web interface shall allow the registering of new computation clients. Registering shall allow the system to determine who owns a client and should receive payment for jobs executed.
    b. The web interface shall allow submission of new jobs.
    c. The web interface shall allow changing the price of each job a user owns.
    d. The web interface shall allow users to see the progress of their jobs.
    e. The web interface shall allow users to download, in XML format, a report about the current state of the system.
        i. This report shall conform to an XML schema document.
        ii. This report shall provide details about computation clients.
        iii. This report shall provide details about jobs.
        iv. This report shall provide details about job dispatches to clients.

Specifications

1. The central server shall be implemented in Python 2.4.3.
    a. Client requests, job submission, etc. shall be handled via XML-RPC, a standard remote procedure interface.
    b. The server shall interface with the database using MySQLdb, a standard python library.
    c. A separate process running Stackless Python 2.4.3 shall generate serialized states for test jobs. This test state shall be in the same format as the job states submitted by job owners.
    d. A pooling algorithm shall collect client requests and periodically assign jobs to clients.
        i. The pooling algorithm shall form a collection of clients requesting jobs to execute.
        ii. After a specified amount of time has passed, jobs waiting to be executed shall be matched to the clients waiting for jobs. Matching shall be done with the highest-paying job going to the fastest client, the second highest-paying job going to the second fastest client, etc…

    e. The server shall issue each client a test job after every set interval of time.
        i. The client's speed shall be viewed as the speed in which it performed its last test job.
2. The client interface shall be implemented in Stackless Python 2.4.3.
    a. The client process shall download serialized states from the central server, resume the states, and send a new serialized state back to the server after a threshold amount of time has elapsed.
    b. The job submission process shall load a Python module into memory and create a Stackless Python tasklet, which wraps a given function in a Stackless Python execution frame. This frame shall then be initialized, interrupted, serialized, and sent to the server. Each job is associated with name and price arguments given as command line arguments.
3. Community members shall possess accounts with the system.
    a. Each client and each non-test job shall be linked with an account.
    b. Charging money for job execution and paying money for using machines shall be handled on an account basis.
    c. Only the community member who owns an account shall be able to view the results of a job linked to that account.
4. The database shall be implemented in MySQL with transaction support.
    a. The database shall contain a table containing community member account information.
    b. The database shall contain a table containing job information, including the source code of each job, and which account the job belongs to.
    c. The database shall contain a table of job state information linking each job state to each job, preserving which client is responsible for executing each job state, and maintaining when each job state was checked out and checked back in.
    d. The database shall contain a table of clients, linking each client to an account.
    e. The database shall contain a table of measured client speeds and the times they were measured at.
5. The web interface shall be implemented in PHP 5.0.
    a. The web interface shall provide a page for requesting new accounts.
        i. New accounts shall be confirmed by following a hyperlink within an email automatically sent to the requestor, following the request for a new account.
    b. The web interface shall provide an interface for logging into and out of accounts.
    c. The web interface shall provide a page for viewing job statuses, as well as changing the price offered for each job.
    d. The web interface shall provide a page for registering a new client.
    e. The web interface shall provide a page for changing an account password.
    f. The web interface shall provide a page for downloading an XML report of the current system state, as deemed relevant by the system.

i. The XML report shall conform to an XML schema document posted on the website.

ii. The XML report shall preserve user privacy by stripping out information linking jobs or clients to any owner but the requesting owner.

iii. The XML report shall provide enough information so that standalone applications can generate relevant results from the document.

Software Architecture and Components

Our software consists of several different components. The Dispatch Server component resides on a single machine and handles the submission of new jobs, tests the speeds of various client machines in the community, and dispatches jobs to client machines for execution. The Computation Client component resides on every machine in the community. It requests jobs from the Dispatch Server, receives them, executes them for a time, and then uploads them back to the Dispatch Server. The next component is a Web Interface for communicating with the Dispatch Server. From the website, you can upload jobs, determine the progress of jobs, change how much you are paying for a job, register a new Computation Client, and download an XML report about the status of the community. The next component is our standalone application layer, where we have applications which read the XML reports from the website and then calculate meaningful results from them. In our project, we have implemented a single example standalone application which will calculate the optimal price to pay for a job to get it to execute at a desired speed. The final component is a back-end database, which stores system-state; each other component communicates with the database to obtain necessary data.

The Dispatch Server architecture is broken down into several subsystems: the job submission subsystem, the dispatcher subsystem, and the speed testing subsystem. The job submission and dispatcher subsystems reside within the same set of Python scripts, which are executed in standard Python. The job submission subsystem is implemented as a server which waits for a remote procedure call, using XML-RPC. When a request for job submission is received, it will attempt to create the job in the database and will then return whether or not the job submission was successful. The dispatcher subsystem is further broken down into a listening server and a dispatcher. The listening server waits for clients to submit either a request for a job to execute or the results of a job execution. If the listening server receives the result of a job execution, it is uploaded to the database (assuming it was submitted in time, otherwise the submission is rejected). If the listening server receives a request for a job to execute, it submits the requesting Computation Client's ID to the dispatcher, which will pool the client with all other requesting Computation Clients, match a job to it, and then return that job's code and execution state to the Computation Client. The speed testing subsystem resides within its own Python script, executed in Stackless Python. The speed testing subsystem creates jobs at fixed intervals that take a standardized time to complete. In this way, these jobs can be submitted to Computation Clients to gage their speeds.

The Computation Client is fairly simple and consists of a single Python script – executed in Stackless Python – that does one thing: it requests jobs from the Dispatch Server executes them for a specified amount of time, and then returns the new job state to the Dispatch Server.

The Web Interface is broken down into two subsystems. The first subsystem is a PHP interface which the community member communicates with when viewing the website. This subsystem is responsible for handling all account actions. These include logging in, logging out, changing an account password, registering a new account, submitting a job, viewing the status of an accounts jobs, registering a new Computation Client, and getting an XML report of the system's state. The second subsystem is responsible for directly communicating with the Dispatch Server to create a new job. It is

a Python script – executed in standard Python – which uses XML-RPC to communicate the new job's information to the Dispatch Server. The PHP subsystem calls this subsystem by executing it at the shell.

The standalone application for calculating optimal job price is composed of a single component. It is implemented as a Perl script that will read in an XML report generated by the system's Web Interface and then use XPath to extract the relevant job and Computation Client information from it. The script then uses the information to calculate the optimal price a job should be set at to get it to execute at a given speed.

Functionality

Implemented and Is Available

We have fully implemented our dispatch server, our web interface, our client process, and a standalone application for determining optimal job pricing. The client process is able to communicate with the dispatch server to negotiate for job assignment. The web interface accepts job submissions and reports job results.

Partially Implemented

We have not thoroughly tested our system to ensure that the bug count is low. We have done thorough testing in the area of core functionality which would result in system inconsistency – allowing inconsistencies in the system would not be allowable – however we felt it was more important to take the time to complete the desired functionality set rather than to make sure that the application was as error-free as we could make it.

Future Desirable Functionalities

There are many functionalities that we would like to stick into future versions of the system. Our current matching algorithm for jobs and Computation Clients leaves much to be desired. Currently, we just take a specified amount of time to create a pool of Computation Clients that are waiting on jobs to execute. Then we pick from the list of jobs, which are waiting to execute, assigning the highest paying job to the fastest client in the pool and then the second highest paying job to the second fastest Computation Client in the pool and so on. This doesn't seem to be the best algorithm to us, allowing for many situations where the highest paying job is not matched with the highest paying Computation Client. There are also synchronization issues, where jobs may only be executable by a certain limited subset of Computation Clients since, for efficiency issues, we make the pooling time much smaller than the negotiated time that each Client executes a job for.

Another piece of functionality that we would like to add in a future design is the possibility for inter-process communication. Currently, we have it setup so that jobs are single-threaded and cannot communicate with other processes executing within the community. We avoided putting this functionality in because we decided that it would not be feasible in the timescale of the project. But setting up a method for distributed inter-process communication would be most desirable. Putting such

functionality into the system would require enabling processes to block and submit messages to each other. Messages between running processes would be required to be directly communicated between the Computation Clients running them, over the Internet. This would require each machine being informed, by the server, the IP addresses and ports of the processes they are communicating with. It would also require functionality for rolling back all communicating job states in the case that any of the job states got rolled back during execution. This would require a complex dependency-tracking system and we felt it was beyond the scope of our project.

Yet another bit of functionality that we would like to see in the future is the ability for people to receive billing statements via email. The goal of the project is to actually trade computation cycles for money, so it would make sense for people to actually receive statements to see how much they are paying/earning each week or month. However, we felt that this wasn't inherently critical to the project and was more icing than anything else.

The Computation Client may also be improved. One issue is that it has no GUI which makes it difficult for a community member to use. Another thing it should have is an installation script which will make it easier to install, rather than requiring the manual installation of Stackless Python and then the Computation Client. Two last bits of functionality that would need to go into a final product is functionality to ensure it only requests a job when there are idle CPU cycles and functionality to ensure that a certain percentage of CPU goes into execution of any jobs the client requests.

One feature we did not have time to implement was passing in data files as arguments to jobs. We believe this functionality could be easily implemented in further extensions of this work.

Application Achievements, Successes, Failures

We had many successes and failures during the course of our project. Most of the failures involved either trying to accomplish too much or software errors that resulted from using different platforms. At first, we sought to accomplish an awful lot. We wanted to make the project so that jobs could be multi-threaded, with different threads executing on different machines. However, we had it pointed out to us that this was probably outside our grasp for the scope of the project.

We also had a lot of cross-platform difficulties. We were originally having errors with state serialization and de-serialization. We were testing the system using three different Computational Clients: one on an 80x86, one on a PowerPC, and one on a SPARC processor. We found that states were interchangeable between the PowerPC and the SPARC processors but not so with the 80x86 and any of the other two processors. We initially concluded that this was an endian-ness issue. However, after much debugging, we concluded that we were running a different version of Stackless Python on the 80x86 platform. Upgrading the version fixed the problem. We then later ran into a bug where states serialized on a SPARC machine would not de-serialize at all. This bug is still open and we are talking to the Stackless Python developers in the hope that it can be resolved.

Application Implementation Details

The central server exposes the following methods to XML-RPC:

1. job(accountid, module, price, name) – This method is called by the job submission process (either by a client or through the web interface) with the id number of the owner's account, the serialized module, a price, and name for the job. These values are stored in a record of the *jobs* table. This function returns the jobid corresponding to this new record.
2. init(jobid, state) – This method is also called by the job submission process with the initial

serialized state of the job. This state represents the job as it is fully loaded into memory, but no significant computation has begun. This method is divorced from the job method above because the state depends on the jobid, which is the return value of the job method (see the result method below).

3. request(clientid) – This method is called by the client process when the client is seeking a job to work on. The return value is a stateid, state, module, and a timeout. The server then expects the client to call the state method with the next state of the job within the window specified by the timeout (in seconds). Unless this method returns a test job, it will block as the client is matched to an available job based on its speed. See the description of the scheduler, below.
4. state(stateid, state) – This method is called by the client process when the client needs to check in a state. If the client has not exceeded the timeout, then the state is saved in the database as 'accepted', otherwise the state is rejected.
5. result(jobid, result) – This method is called by the job (running on the client) when the job is finished. When a job or test job is encapsulated as a Stackless tasklet, this call is included, and gives the result of the job's function as the argument. For this reason, the job needs to know its own jobid.

The scheduling algorithm is simple. A time window of constant size collects all clients requests made within the window. When the window expires or a threshold number of requests are collected, then the requests are assigned to jobs by assigning the fastest clients to the most expensive jobs. The goal of this process is to ensure that fast clients are matched to high-paying jobs, and thus the total value of the system is maximized.

Client speed is measured by test jobs, which are implemented as simple cubic-time algorithms (nested loops). Test jobs are created dynamically for each client every thirty minutes. Test jobs are created by an independent process running on the server as a Stackless Python XML-RPC server.

The standalone application takes, as input, an XML report that conforms to the XML schema given on the website and a speed at which the user would like a job to be run. The application then uses XPath queries to extract a set of clients and their speeds and job prices from the report. These XML nodes are then processed to obtain an authoritative list of job prices and client speeds. The number of clients whose speed exceeds the desired job speed is then calculated – let this be denoted by n. Then, the list of job speeds is sorted and the nth highest job price is found – let it be denoted as p. The application outputs p as the minimum price to pay for a job so that it will be executed at the given speed. Since the nth highest paying job should be matched to the nth fastest machine (which we know to have a speed equal to or greater than the desired speed) we can conclude that p is the minimum price at which a job should be matched to that speed.

Application Evaluation Details

We have tested the system with multiple clients and jobs running on distinct hardware (Mac, Linux, Intel, PowerPC). In our tests, these architectures and operating systems have performed uniformly and correctly. The only problem we encountered while evaluated the system was related to SPARC machines. Under this architecture, Stackless Python failed to interrupt running tasklets correctly.

In our presentation, we intend to present more detailed experiments to evaluate our system.

User Interface

Current Interface

Our current user interface can be broken down into a different part for each subsystem. Neither the Dispatch Server nor the Computation Client has much of a user interface at the moment. The Dispatch Server interface is a set of Python scripts which must be manually started by an administrator. After the Dispatch Server is started up, it just needs to run; there is nothing really to see so it doesn't really display anything to screen except for debugging information (which would be disabled in a production version). It is terminated either by signal or by killing the spawning shell.

The Computation Client is also run as a Python script. It is given the domain name and port number of the Dispatch Server, as parameters, and then runs without displaying anything other than debugging information. It too can only be killed by signal or by killing the spawning shell. There is no functionality for freezing and submitting a job, currently in execution, to the Dispatch Server before quitting.

The Web Interface is the only part of the system which currently has much of a user interface. We use PHP, XHTML, and CSS to provide a consistent look and feel to the website. We use HTML forms to communicate with the user. Several screen shots of the website are provided below.

The standalone application has only a shell script interface, as well. To run the application, one runs the script, providing an XML report and a desired speed to run a job at as command line arguments. The application will then either print the optimal price for the job, to achieve the speed, or "NotPossible," if the desired speed is faster than any of the community machined.

Possible Improvements

The user interface for our project could be improved in a lot of areas. While we believe that the Dispatch Server, being an administrative process, could just be left as a series of scripts without a GUI, it makes sense to provide a parent script which can be invoked by itself to customize and execute all of the server processes. This would make the life of the server administrator easier. There are also a lot of tasks that need to be performed to set up the Dispatch Server; it depends upon the installation of

MySQL and SendMail, and requires information about which domain name and port number to uses. All of these setup tasks should probably all be inserted into a single installation script.

The Computation Client could use a better user interface as well. As mentioned above, a GUI would be very useful since community members, who may have little or no shell experience, will need to run this application. The GUI should, at the very least, contain settings for changing program settings – such as how much CPU it uses – and for submitting the job in progress and exiting the program. It should also provide some sort of system tray feature (if the environment it's in allows it) and simply display what it's doing.

The Web Interface could use some improvement, as well. For one thing, Vinay's suggestion made about adding an RSS feed to determine the state of one's jobs seemed quite meritorious and should definitely go into the website in the future. Also, since the website's designer was learning XHTML and CSS at the same time as implementing the website, it was designed in a very minimalist fashion, without any fancy code. Additionally, some of the pages do not display information in the most appealing fashion; however we decided that the Web Interface looked good enough and that we should work on adding new functionality to the project rather than optimizing the website's design.

Our example standalone application could use a better user interface as well. It should have a GUI which will take a desired speed from the user, automatically download an XML report from the website, and then calculate the optimal price to pay for a job to achieve that speed.

Responses to Project Critiques

1. *How are you handling transfer overhead? –Aruno (1)*
   Any time spent transferring data between the client and server is time spent not doing work. We have attempted to minimize the effect of transfer overhead by giving clients large amounts of time to work on a job before sending it back, on the order of 10 times the transfer time.
2. *Who is handling each portion of the project? –Aruno (1)*
   Michael is the project manager and is handling the website, standalone application, and server maintenance. John is the developer, and is writing most of the code.
3. *Will the user interface feature RSS feeds of jobs? –Vinay (1)*
   We did not have enough time to implement RSS feeds, but this feature could be easily implemented in the website.
4. *Is Stackless Python generic enough for users? –Divya (1)*
   Stackless Python is not significantly different from standard Python. We chose Python partly because it is a well known cross-platform language and is easy to learn.
5. *How are you handling security/privacy? –En (1)*
   This project is merely a proof of concept, and we did not consider security or privacy in any respect. These considerations may be taken as avenues of future work.
6. *How is the speed of a client measured? –Dsouza (1)*
   Client speed is measured by a function that is expected to take between 30 and 120 seconds to complete. We estimate the client speed regularly by sending this test job to the client every thirty minutes and measuring the time to complete the job.
7. *What happens when an error occurs? –Steve (1)*
   We believe client-side error handling was feasible, but we did not choose to implement this as robustness was not a specific goal with respect to jobs.

8. *Will the scheduler be fair? –Steve (1)*
   Since we intended our scheduler to be market-based, naturally situations may occur where jobs either completely dominate the system or are completely dominated by other jobs.
9. *Is a web portal sufficient for submitting jobs? –Meng (1)*
   Since our project was a proof of concept, we narrowed the scope of jobs to small jobs that could be easily submitted through the web interface. For larger jobs, this interface may not suffice.
10. *May data be submitted along with jobs? –Meng (1)*
    At this time we did not implement a system to allow data to be passed as arguments to jobs, but this is certainly feasible.
11. *How is the database updated after a job is done? –Tran (1)*
    When a job completes, the return value of the job's function is serialized and sent to the server. The presence of this result value indicates that the job has finished.
12. *May different processes communicate with each other? –Xin (1)*
    We specifically assumed jobs are independent processes, since interprocess communication between asynchronous processes is a sizable challenge. Also, implementation of concurrency controls would be very difficult in such a short time frame.
13. *How would data be transferred to and from clients? –Xin(1)*
    If jobs could accept data, then the easiest method to store and transfer this data would be to serialize it in the same structure as the job's state, so the data portion of the job would be transferred roundtrip between client and server. This is obviously not preferred, but any other method may prove to be extremely difficult to implement.
14. *Does the system provide any debugging features for job owners? –Divya (2)*
    Our system assumes jobs run bug free. This is not unrealistic, since job owners may run their process locally before submitting them to the dispatch system. In other words, our system is a production environment, and is not intended for development and debugging.
15. *Do architecture differences matter? –Meng (2)*
    No, Stackless Python is intended to run identically in all platforms and architectures, although we did find a bug in Stackless that prohibited us from using SPARC machines. This is not a deficiency of our system, but rather a fixable bug of Stackless.
16. *Does the system have quality of service guarantees? –Hangwei (2)*
    Since it is impossible to decide when a given program will halt, quality of service guarantees are difficult or impossible to give to users.
17. *How are the parameters of the scheduler determined? –Nahal (2)*
    Our scheduler uses arbitrarily chosen, fixed parameters.
18. *Does your system support languages other than Python? –Tran (2)*
    Our system is deeply integrated with Python and Stackless Python. Supporting other languages would require a complete redesign of the client, server, and database components, although most of the concepts (e.g., jobs, market algorithm) may be reused.
19. *Does the system use any kind of sandboxing? –Vinay (2)*
    No, we assume that jobs do not attempt any system calls for input/output, although sandbox modules are available for Python.