Distributed Processing Made Easy: Proposal for the Creation of a Runtime Environment to Assist in Running Programs Across Multiple Systems

Colin Pronovost

2014-10-11
Abstract

I will produce a runtime environment that allows for distributed processing over a network connection to be achieved with ease. I intend to accomplish this by creating a daemon that will respond to commands sent from a controller and run code that the controller requests it to run. It will also provide a library of functions that programs on different systems can easily use to communicate and coordinate.
Introduction

Distributed processing is useful for a simple reason: many hands make light work. In other words, having more computers work on a problem reduces the amount of work each individual computer has to do, thus the process as a whole takes less time.

The aim of this project is to produce a runtime environment that will make the implementation of programs to perform distributed processing as easy as possible, therefore reducing the development time of such programs. It will accomplish this by creating a daemon that will run on the remote systems (the clients), listening for and responding to commands sent from a controller. The protocol is detailed in Appendix A. The client will accept jobs from the controller that it is able to perform, and, as soon as it has accomplished one, start another, if another exists. The daemon will run the programs it receives from the controller by calling a function in the supplied program with arguments sent from the controller. Enabling the controller to execute all or part of a program on several systems simultaneously would allow for programs to be parallelized without extensive rewriting, for instance by specifying that each individual program work on part of a larger problem, then send the program’s output back to the controller for integration into the overall solution. Dynamically loading programs to the clients will also allow for the programs to be installed on them simply by issuing a command from the controller, thereby easing the process of installing and updating the program.

The structure and properties of the runtime (in particular the dynamic loading) will allow for easy experimentation and benchmarking in order to determine if the process of parallelization is yielding significant gains. Ease of experimentation will allow it to be used as a tool for further research into the advantages of distributed processing and into which problems have parallel algorithms that compute the problem’s solution faster than their sequential counterparts.

A Note About Security

At this point, you are probably concerned about the prospect of having a daemon that downloads and installs programs over a network without administrative intervention. To address this concern, communication will be encrypted and signed using a PGP-like system detailed in Appendix B.

You may also be concerned about this daemon hogging system resources. In order to prevent this, the daemon will actively adjust its scheduling priority and limit memory usage in order to ensure that the clients’ users are not hindered by the daemon from using the system.
Prior Work

There are many libraries that attempt to assist in the creation of programs that take advantage of distributed processing. I will now take a brief moment to describe them and show how my proposed system is different.

**Message Passing Interface (MPI)**

MPI allows for interhost communication in a distributed processing environment, similar to the system of communication I have described. It even allows for programs to be run on multiple systems over a network (Message Passing Interface, 2014) (mpiexec(1) Man Page, 2014). However, the complexity of performing some operations in MPI increases the workload of production significantly (Message Passing Interface, 2014). The system I propose will have a simple, easy-to-use, and non-intrusive interface in order to ease production and experimentation.

**OpenMP**

OpenMP provides support for parallelizing loops and sections of code through preprocessor directives, allowing for code to be written to run in parallel with very little effort on the part of the programmer. However, since the platform is based on sharing memory across threads, it has virtually no support for any sort of network-based parallelization, unless shared memory across the network is implemented, for instance, in the kernel, so that the distribution over the network is transparent to user-space programs (OpenMP, 2014).

**Apache Spark**

Spark has an extensive library and claims to be easy to use (Apache Spark, 2014). However, it lacks a native interface and does not support dynamic loading, requiring the user to SSH into all cluster systems to update programs (Apache Spark, 2014) (Running Spark on EC2, 2014). This makes it not well suited to applications that require running several different programs or rapidly changing programs.

**BOINC**

BOINC is a framework that provides a lot of the same features that I have proposed. It allows for a form of dynamic program loading, remote function calls, and has a task-based framework (How BOINC Works, 2014). The key difference is that this project is focused on a system for doing research with the capabilities of distributed computing while BOINC is designed more for using volunteer CPU cycles to do big data calculations (How BOINC Works, 2014).
Goals

I will create a runtime environment that allows for programs to be uploaded to and executed on multiple systems and will allow for to be parallel programs to be written easily. In some cases, mainly ones where the program is already structured in a way where the task is internally subdivided into independent parts\footnote{For example, consider a matrix multiplication algorithm, in which each cell in the product matrix is independent of the others.}, parallelization could be accomplished without requiring major rewrites to the program. The controller will be able to call the programs’ functions, with parameters, receive their return values, receive the program’s stdout and stderr, and send to its stdin. The controller and clients will be able to post tasks that clients will then be able to check out and complete. It will also provide a library that will allow programs to easily communicate and coordinate with each other in order to facilitate the writing of parallel programs.

The library will provide functions that will allow programs to pass messages through a file descriptor representing an internet socket connected to the controller, which will copy the message, if necessary, to the other clients. Messages will be categorized, so clients can choose which messages they would like to receive. Library functions are detailed in Appendix C.

The user interface will provide features for collecting statistics on the running times of the program as a whole and its individual parts, how many parts are created, etc. in order to assist in experimentation with parallel algorithms.

As this is primarily a research project, I will also create some test programs that will be used for benchmarking to show the decrease in time that comes from using multiple systems.
Method

The controller will establish connection with each client, after which it will be able to send it messages and commands through a network socket. The client will receive and process the commands. These commands are detailed in Appendix A, but a few important details are presented here:

Programs will be transferred in ELF format, and a message will be sent back to the controller if any shared libraries are missing, the architecture is wrong, or there are other issues, which the controller will attempt to correct. The program and its libraries will then be loaded into a new memory page(s) and said page(s) will be marked executable.

The program’s stdin, stdout, and stderr will be connected to memory streams to that they can be used by the controller.
## Appendix A

### The Communication Protocol

The controller’s messages are the following:

<table>
<thead>
<tr>
<th>1 byte</th>
<th>2 bytes</th>
<th>3 bytes</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESSAGE_TYPE</td>
<td>MESSAGE_LENGTH</td>
<td>STATUS_CODE</td>
<td></td>
</tr>
</tbody>
</table>

and may contain extra data, the length of which is an unsigned 16 bit integer specified in the MESSAGE_LENGTH field. MESSAGE_TYPE is one of the following:

- **0 request status**
- **1 jobs published**
- **2 exec function**
- **3 pipe to stdin**
- **4 request stdout**
- **5 request stderr**
- **6 load program**
- **7 unload program**
- **8 load library**
- **9 terminate**
- **255 bad type**

which have the following meanings:

- **request status**: requests the client’s status, which is returned in STATUS_CODE.
- **jobs published**: informs clients that jobs have been published.
- **exec function**: requests that the client execute a function. The data contains the function’s name, arguments, and return type.
- **pipe to stdin**: requests that the client use the provided data as the stdin for the next invocation of the program.
- **request stdout**: informs the client that the controller is interested in the stdout of the program.
- **request stderr**: like request stdout.
- **load program**: requests that the client load the program contained in the data. The program is passed as an ELF file.
- **unload program**: requests that the client unload the currently loaded program.

---

1. Subject to change over the course of the project if the needs of the project mandate it.
* load library: requests that the client load the library contained in the data. Same format as load program, to be used when the client requests a library for the program.
* terminate: specifies that the client should clean up and exit.
* bad type: sent when the client sends a message with a bad MESSAGE_TYPE.

The client’s messages to the controller are the following:

<table>
<thead>
<tr>
<th>MESSAGE_TYPE</th>
<th>MESSAGE_LENGTH</th>
<th>STATUS_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>2 bytes</td>
<td>3 bytes</td>
</tr>
</tbody>
</table>

and may contain extra data, the length of which is an unsigned 16 bit integer specified in the MESSAGE_LENGTH field. MESSAGE_TYPE is one of the following:

0 request status
1 request job
2 function return
3 request stdin
4 send stdout
5 send stderr
6 program loaded
8 library loaded
9 terminated
255 bad type

which have the following meanings:

* request status: requests the controller’s status, which is returned in STATUS_CODE.
* request job: asks the controller for a job.
* function return: informs the controller that the function that it requested the client to execute has returned, and passes the controller the function’s return value if non-void.
* request stdin: requests that the controller give the client data to use as stdin for the program.
* send stdout: passes the program’s stdout to the controller.
* send stderr: like send stdout.
* program loaded: informs the controller that the program the controller requested the client to load has finished loading.
* library loaded: informs the controller that the library the controller requested the client to load has finished loading.
* terminated: informs the controller that the client is about to exit.
* bad type: sent when the controller sends a message with a bad MESSAGE_TYPE.

STATUS_CODE may have the following values:
0 IDLE
1 BUSY
Appendix B

The Security Protocol

All communication will be encrypted with RSA. Each message will be encrypted with the client’s public key, and a SHA-512 hash of the message will be encrypted with the controller’s private key, so that the client may verify the correctness and authenticity of the message.

\footnote{Again, this is subject to change if project needs mandate it.}
Appendix C

The Library Functions

The following functions will be provided on the client:

- `int getControllerSockFD()` returns the socket for communication with the controller.
- `void sendMessage(struct message* msg)` sends the controller a message. `message` contains 4 members: `uint8 message_type`; `uint16 message_length`; `uint8 status_code`; `void* data`.
- `struct message* recvMessage(struct message* msg)` receives a message from the controller. If `msg` is non-null, the message is stored in that address, which is returned. Otherwise, new memory is allocated and returned. The length of the data is checked to ensure that it is equal to the `message_length` field, otherwise the message is dropped.
- `void setMessageMask(int mask)` specifies which messages the client is interested in receiving.
- `void clearMessageMask(int mask)` clears the message mask so that all messages are received.

The following functions will be provided on the server:

- `int getNumClients()` returns the number of clients.
- `struct client data* getClientData(int client, struct client data* cldata)` retrieves data on the specified client, such as OS, architecture, clock speed, amount of RAM, and other useful information. If `cldata` is non-null, the data is stored in that address, which is returned. Otherwise, new memory is allocated and returned.
- `void sendMessage(struct controller_message* msg)` sends a client a message. `controller_message` is the same as for the client, except that an `int client` field is added.
- `void sendAllMessage(struct message* msg)` sends all clients a message. `message` is the same as for the client.
- `struct controller_message* recvMessage(struct controller_message *msg)` receives the next message. If `msg` is non-null, the message is stored in that address, which is returned. Otherwise, new memory is allocated and returned. The length of the data is checked to ensure that it is equal to the `message_length` field, otherwise the message is dropped.

\footnote{Again, these are subject to change based on the needs of the project}
References