Browser-Based Distributed Computing

CMPSC 450 - Mini Project

Yuya Jeremy Ong
yjo5006@psu.com

1 INTRODUCTION

Recently, high performance systems have been utilized in various applications within domains of scientific computing, processing of complex simulations, solving optimization problems, and most recently areas in machine learning and deep learning. The majority of these high-performance systems today are compiled and built on a centralized and monolithic infrastructure where machines are typically based on server-grade level hardware designed specifically for high-performance scientific computing. They have become one of the primary drivers for breakthrough innovation and development to be delivered within the scientific community.

Although, many of these state-of-the-art systems are very powerful and are capable of helping scientists, researchers, and engineers to help perform complex tasks, the development, maintenance, and especially the cost for these systems are particularly expensive. Much of the cost of these systems are rooted at the purchasing of expensive systems as well as the cost of maintaining them.

Recently, with the ever growing number of powerful commodity based hardware, such as desktops and laptops equipped with Core i7 Intel processors, or mobile phones with improved processors for computation, we find that these systems independently are capable of running very complex computational problems even on such small devices. However, many of the applications that are often utilized by these consumers do not take advantage of the full hardware capability, leaving many people with often times idle CPU cycles. Majority of the consumers make use of the use cases for these commodity machines often use them for web browsing - using browsers such as Google Chrome and Firefox to visit various resources and content online. Over the course of their usage habits, average users of such devices mostly uses computers to browse the web, which often doesn not incur many CPU cycles (unless many tabs were present).

According to a recent study conducted by the Miniwatts Marketing Group [7], globally there are approximately close to 4,156,932,140 Internet-connected devices. If we assume that the average computer (having an Intel i7 920 2.8 GHz processor as our "average" device) clocks in around 6.3×10^10 FLOPS, the theoretical upper bound for the amount of computational power in total would amount to approximately 2.7×10^20 FLOPS, which would be 0.27 ZFLOPS.

In short, this opens a window of opportunity for researchers and engineers who can make use of these idle CPU cycles and aggregate them together into a network of one cohesive architecture to perform various computational problems within a distributed setting. By making using a browser as an endpoint for computation, this allows a much more open, dynamic, and mostly platform agnostic system which can take advantage of the ubiquity of today's general computing infrastructure.

In this paper, we explore a novel distributed computational platform called Queen [3], which makes use of the front-end browser computational power powered by Javascript to perform various computational processes. The library includes several different APIs and functionalities which helps developers and users to facilitate message passing between each of the worker browser nodes and the master node. Specifically we will first explain and dive into the general system architecture of how Queen was developed and is operated. Then we will look into the typical development process of how one would build a program using the Queen platform. Furthermore, we evaluate a key experiment with the . Finally, we end our paper with an analysis of some of the comparisons to existing state-of-the-art distributed computing systems such as MPI, and point out some key merits and demerits of utilizing such an infrastructure for scientific computing applications.

2 QUEEN PLATFORM

In this section, we describe the Queen platform and some of the key features and system architecture of the system. This section will provide some context and background into which how the system is built and how it works.

2.1 Introduction to System

The Queen platform is based on an open-sourced project by @unsetbit, which has been first publicly available around 5 years ago [4]. According to the introduction README, it claims that it is a "server which is capable of brokering socketed communication between browsers which are connected to it and other applications or scripts. You can think of the Queen Server as a pool of browsers which you can execute code on. Taking the abstraction further, you can think of Queen Server as distributed execution platform using browsers as computation nodes." In essence, Queen acts as a browser pooling and resource orchestration platform which relies on the connection of browsers providing the computational resources which they are sent back to a centralized server node.

2.2 Features

We now describe some of the key features which the Queen platform offers as a distributed browser-based computational pooling resource platform. Here, we describe some of the highlighted features and also interject how they may be useful in certain scenarios or particular use cases.

First, Queen being a platform for orchestrating computational resources and operations make use of socket.io [6], a framework for developers to help writing applications which requires asynchronous communication over a TCP network. The API provides an interface which allows bidirectional communication between the worker providers, worker instances, and the queen server. As this project makes use of the Socket.io interfaces, it is also possible to facilitate inter and intra node communication between other

workers or worker providers, which can facilitate similar forms of communication like the MPI protocol does.

Another feature which is provided by the Queen platform is their ease of integration with existing libraries, packages, and any projects which makes use of the JavaScript language. Modern JavaScript has evolved in many ways to the point where there have been active development of packages and libraries people can utilize for specific applications, such as linear algebra, signal processing, and most recently Deep Learning with the release of TensorFlow.js from Google. This makes it also another highly competitive platform against some of the other architectures where dependencies and infrastructure setup is a very tedious process.

As the particular infrastructure that we our working is comprised of a heterogeneous set of hardware and browser with various specifications and configurations, the challenge of matching hardware constraints is a very challenging task at hand. However, by leveraging various meta-data provided by the browser's User-Agent information or frameworks such as Modernizr [2] which offers rich understanding of hardware metadata and browser specifications. One of the major bottlenecks to many of the designs of supercomputing infrastructure relies on the unified nature of the hardware and requires to often have similar specifications across each of the nodes. In many distributed systems today, the transfer of knowing the hardware specification is also not provided, which makes also makes such distributed system much more intelligent than some of the traditionally existing architectures that are out there today.

One of the main key features which makes Queen a very robust distributed system is having the ability to automatically connect to browsers using various emulation tools such as Selenium [5]. Making use of such systems would allow as a better sandboxed environment for browsers to perform computation on - which can mitigate potential security risks and compatibility issues which may arise due to browser version differences. This helps to normalize version controls across different dependencies certain which programs may use depending on the environment the developers originally developed it for.

The Queen infrastructure also provides a very good robust control mechanism which allows you to utilize their Queen thin-client which enables users to run scripts remotely to these clusters. Furthermore, the Queen provides mechanisms for error handling and unresponsive node endpoints. The platform enables workers that are unresponsive to automatically recover and resumes the jobs that have been provided to them immediately.

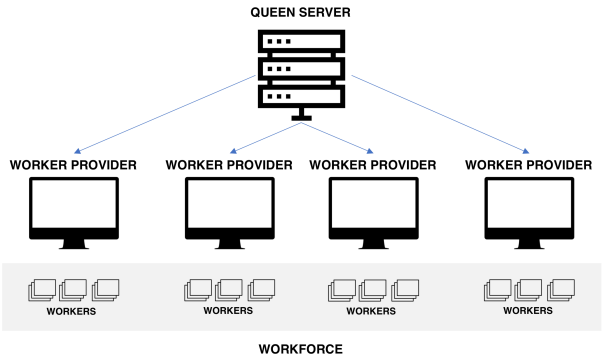
2.3 System Architecture

The architecture of Queen, to some extent is very reminiscent of a grid-computing infrastructure similar to that of the SETI@HOME project which makes use of the BIONIC [8] computational platform. The BIONIC platform was developed in part of a large peer-to-peer public-resource computing system. The BIONIC system works through first establishing a master node whose responsibility is to supplement the corresponding data and code to the worker nodes. The worker nodes would then download the corresponding client on the node and complete small distributed tasks assigned by the master node and send back the computed results, which the whole process is then repeated again for any incoming new set of data.

The only major difference between the BIONIC architecture and the Queen architecture is how the system makes use of a browser based browser based daemon, instead of a dedicated program which has to be setup and compiled locally on the client's machine. This provides for a higher advantage with regards to on-boarding as majority of systems today already are equipped with a web browser capable of executing pieces of code which can be provided by the master node.

Finally, the entire development ecosystem is surrounded based on JavaScript and HTML, which makes for the entry to barrier in terms of using the system very accessible. With a solid package management system offered by npm (a library package management system for the JavaScript environment), users can quickly iterate and develop complex programs building on top of wealth of various code bases from existing projects and scaling them accordingly.

Figure 1: Queen System Architecture



Looking at **Figure 1**, we can see the general architecture of the system and the various interactions between the various components which handles the computation to the resource orchestration process. The *Queen Server* is a system which helps to aggregate all of the worker nodes into one central location. This acts as the primary hub for where all of the computational resources are centralized at a single point. The essential role of this server is to create what is known as the *Workforce*, which is a collection of *Worker Providers*, which are the individual computational systems (i.e. desktops, laptops, mobile phones, etc.). Within each of the Worker Providers, or a browser page, the browser would execute multiple small iframes known as *Workers* which contains a sandboxed environment for specific tasks that the computational node would perform at the front end.

The Queen's primary server is written in Node.js, and can be executed through a command line interface which is then run as a daemon process that initializes two server applications. The first one is a browser based web application which runs on one port - this is the front end in which the worker provider will access and use to communicate with the queen server. Furthermore, another port is opened to be used internally to bind the capture server to. The page browser acts as merely an interface for signaling a websocket connection between the server and the client nodes, where further communication is then facilitated.

3 DEVELOPMENT PROCESS

In this section, we outline the development methodology for a typical program which would leverage this type of platform. In particular, we look at the two key components which Queen uses for it's distributed program.

3.1 Program Structure

To execute a job in Queen, we develop what is referred to the developers of the platform as "Queen Scripts". For every Queen application, there are essentially two key components which developers must build out: the queen server and the client worker programs. The queen server's program is responsible for managing the orchestration of how the data and operations are passed around. The other program component necessary for the Queen Script requires a client end program, for which each node on the client would run a particular type of job.

Although, the level of abstraction provided by the Queen platform is not as sophisticated as that of the MPI interface, it still provides a powerful channel and number flexibility to define routines which allows you to take advantage of multiple machines simultaneously.

3.2 Server Script

The server side script within the Queen platform simply is based on a configuration file which the user can edit. Within this script, the user must provide an endpoint for where the client-side script resides (as a another web-resource on a separate web server). Furthermore, the configuration provides options as to how the workers should operate the script execution process and whether to continue with a new batch or simply end execution with the corresponding worker node. Finally, the user will have to define a specific type of event handler which is executed prior to the initialization of the program on the worker client. This routine is meant as a method for passing any sort of relevant dataset or pieces of instructions or key parameters the program would need prior to executing the script. The Server API provides access to all components of the Queen Platform including, but not limited to the Queen instance, Workforce, Worker, and WorkProvider instances.

3.3 Worker Script

The worker client script that the developer must write for the Queen script is simply a message handler for the onHandle message for the queenSocket which is injected within the iframe worker instance during runtime of the application. Through this code injection, the front end of the given worker can use this as a method to perform the given task that we assigned it by treating the new incoming data signal as a new task is provided by the server side script.

4 NETWORK LATENCY EVALUATION

In this section we outline an experimentation process to evaluate the performance of the Queen distributed computational platform. In particular, one of the most obvious bottlenecks we can find from such a platform is the amount of network latency such system can potentially pose for data communication between the queen server and the worker nodes.

The experiment we conducted in this project was based on a simple data transfer ping-pong routine. We attempted to measure the average time to measure an array of integers being transfered from one node to another. We evaluate the performance by varying the number of integers to send and compute the average time it takes for each node to completely transfer the data back and forth. We believe that this experiment is significant in looking into, as network transfer is one of the most expensive bottlenecks to consider when developing such applications. Therefore, we are interested in seeing how such system would work in a realistic desktop environment where we also take into account various factors such as external network traffic and real-world network latency factors.

4.1 Evaluation Environment

In this section we describe the experimental cluster environment that we have used to obtain our measurements. For our Queen server, we have deployed the daemon on the aci-i cluster running on the port 9300. However, since the aci-i node does not enable users to open ports externally outside of the Penn State firewall, we used ngrok [1] to expose the corresponding port environment outside of the aci-i cluster environment so that people externally can access the client-end site.

Then, we needed another server to host our client-end worker task JavaScript file. For this, we utilized a XAMPP Apache based sever on a Mac Book Pro, also using ngrok to expose a port outside to enable external compute nodes to access the source code they would be processing on. In this case, the hosting of the worker node source code could have been merged together with the aci-i node resources, however, we made the assumption that some of the source would be served on another server or a CDN based system independent of the Queen server.

For our worker providers, we utilized several different desktop workstations in the Pattee Libary Media Commons at Penn State University - logging into approximately 10 machines in the middle of the night around 3:00 AM. Each workstation tower is based on a Dell OptiPlex 7050 mini tower which is powered by an Intel Core i7-6700 @ 3.4 GHz with 16 GB of RAM running Windows 10. For each computer, we opened a Google Chrome browser which opened a page pointing to the worker provider page served by the aci-i Queen server.

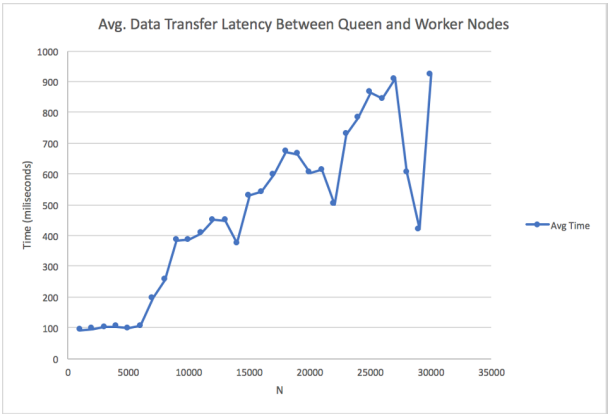
4.2 Results

We subsequently collected data of this ping-pong routine for 10 mini desktop tower nodes and computed the average runtime for each of the following nodes. We performed this analysis for each different values of N starting with 1000 and subsequently increasing each interval by 1000 up to 20,000 elements. For this analysis we have pushed only a small amount as we have tested for extreme values such as 1,000,000 elements - however the system refused to initialize or the worker nodes did not respond (and got killed off by the Queen server) due to the lack of memory support.

We plot our empirical observations based on the data we have collected and graphed, as shown on **Figure 2**. On the x-axis, we have the number of integer elements we transfered between the Queen node and the worker nodes. On the y-axis we show the

corresponding execution wall-time in milliseconds for the entire transfer operation of the data sending and receiving back.

Figure 2: Data Transfer Network Latency Results



Based on the figure we have generated, there are several key salient observations we can make and draw conclusions from. First, we can see that the firs 5000 elements has a constant transfer time of approximately 100 ms. However, beyond the point of N = 5000, we find that the time grows in a linear fashion as the data scales into a larger size. Another interesting point to note is the amount of variability in the measurements we found at approximately close to 30,000 elements. For these numbers we have performed multiple repeated sets of experiments on this particular interval of integers, however found the dip to still be present. We believe that this may have been caused due to either how Socket.io handles data packet transfers of a certain size or that there were external traffic amongst the other computers who were on the network also influencing our empirical observations.

5 DISCUSSION

In this paper, we have explored a the use of a novel and unorthodox architecture for a distributed computing platform which utilizes a different configuration for how computation is performed. Although, we find this architecture to be a very novel and interesting study, the practical use of the system is far from usable against some of the state-of-the-art systems which are present today.

As mentioned earlier and based on our empirical observations, the primary problem with such an architecture is based on the network latency of the system. As the hardware architecture is not based on a monolithic architecture, we find that the major weakness with a distributed system with such configuration is counted on attempting to reduce the overall amount of latency present in the system. However, considering the fact that this package was developed and last maintained about 5 years ago, we can make use of this similar architecture and improve the framework and library choices proposed by the original developers and instead opt for a WebRTC based method for inter-node communication. WebRTC utilizes a UDP based communication protocol instead of a TCP based method, which allows for ultra-low latency and fast communication speeds. It has been utilized in areas such as VoIP systems and has been fully implemented in video conferencing systems such as Google Hangouts. Furthermore, through the integration of other libraries and frameworks like WASM or Web-Assembly and WebCL, we can further optimize the code to take advantage of hardware-level optimizations to improve the speed of computation individually on the nodes themselves.

6 CONCLUSION

In our paper, we have introduced, evaluated, and experimented on the Queen Browser-Based Distributed Computational Platform. We have introduced the primary system architecture, development process, as well as some of the potential caveats it holds with regards to the network latency issues and how it can be potentially mitigated. We hope in the future, we can make use of this similar architecture to better improve on this system and actualize an improved version of the Browser-Based Distributed Computational System to make use of the commodity hardware at a global scale.

REFERENCES

[1] [n. d.]. ([n. d.]).
[2] [n. d.]. Modernizr. https://modernizr.com/. ([n. d.]).
[3] [n. d.]. Queen. http://unsetbit.com/queen/. ([n. d.]).
[4] [n. d.]. Queen - Github Repository. https://github.com/unsetbit/queen/. ([n. d.]).
[5] [n. d.]. SeleniumHQ. https://www.seleniumhq.org/. ([n. d.]).
[6] [n. d.]. Socket.IO. https://socket.io/. ([n. d.]).
[7] [n. d.]. World Internet Users and 2018 Population Stats. https://www.internetworldstats.com/stats.htm. ([n. d.]).
[8] David P Anderson. 2004. Boinc: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 4–10.