

Website Background Processing for Large Scale Distributed Parallelism

Simon Newby

August 24, 2018

Abstract

This project investigates a new form of distributed computing where the modern web browser is used to enable large scale, volunteer computing. A framework called Panther is developed which allow distributed computing tasks to be completed in the background while users view a client webpage. Assessment of the framework under real world web browsing conditions highlights optimal task scheduling policies in this volatile environment, and also shows performance differences between several different web communication protocols.

Contents

1	Introduction.....	4
2	Background.....	6
	2.1 Berkley Open Infrastructure for Network Computing	6
	2.2 Web Browsing Behaviour and Distributed Computing via Web Browsers	7
	2.3 Web Technology.....	10
3	Panther: A Framework for Distributed Computing via Web Browsers	16
4	Testing Strategy	45
5	Testing and Evaluation	53
	5.1 Dartboard PI Task.....	53
	5.2 Mandelbrot Image Task Description	65
6	Conclusion and Future Work	68
7	References.....	69

1 Introduction

The ability for internet connected devices to communicate via the World Wide Web has transformed society in the last several decades, bringing social media, alternative news media, and data privacy issues to the forefront of national debate. The internet is the largest group of networked computers in history, with over eight billion connected devices, and this number increases with each passing moment. Consumer internet access speeds are increasing with the roll out of fibre optic cables and other technologies, and the processing power of internet enabled devices continually increases, with the modern internet-enabled smart phone now factors more powerful than early generation desktop computers.

These advancements could now allow an entirely new form of distributed computing to take place: where processing power is crowd-sourced from website visitors, allowing access to extreme processing power – all enabled via the standard web browser with no additional plugins required. These web browser based volunteers could be used to expand the volunteer base for existing volunteer computing projects, such as *Rosetta@Home*, which is dedicated to protein structure prediction, or they could be used for the processing of entirely new distributed computing projects.

As ascertained during the project preparation phase, using the web browser for distributed computing provides some intrinsic benefits for volunteer computing. The ubiquitous nature of the web browser provides an extremely prevalent platform for standard computer users to become volunteers. Additionally, having a task execute in JavaScript within a web browser resolves any cross-platform deployment issues, allowing one codebase to target all available platforms.

However, using the web browser as a conduit for distributed computing introduces a new key challenge: volunteers using a web browser are available for computation for an unknown and uncontrollable amount of time. Due to the intrinsic security settings enforced by web browsers, any inflight computation associated with each webpage is immediately terminated as users navigate away from each page.

The behaviour introduces the concept of a new parallel design pattern: the Master and Volatile Worker. While existing volunteer computing frameworks operate with some level of volatile workers, the level of volatility is much higher for web browser based workers, as in line with web browsing habits, a webpage viewing session may only last from several seconds to several minutes, meaning each worker is only available for a short period of time. If resources are committed to delivering a task to a volatile worker for processing, but no completed task is returned, then processing cost has committed for no actual return. If this lack of return is great enough, potentially the entire feasibility of distributed computing via web browsers is removed.

The aim of this project is to create and evaluate a framework that allows distributed computing via web browsers to take place under real world web browsing conditions. This project aims to advance research in this area by focusing on the following areas:

- Creating a framework that can act as expansion to existing volunteer computing tools and not as a replacement. Most recent research has focused on the creation of standalone frameworks that do not expand existing volunteer computing tools.
- Conduct live deployment and live testing of the framework under real world web browsing habits. Previous research has only simulated approximated performance under real world condition and therefore factors such as message passing latency may not have been accurately modelled.
- Assess if alternative task distribution policies enable a higher return in completed tasks and therefore provide more utilisation of volunteers' computing resources.
- Assess any performance differences between the *XMLHttpRequest* and *WebSockets* communication protocols, and assess if one may have the advantage over the other for completing distributed computing via web browsers.

In **Chapter 2** background information on this research topic is given. **Chapter 3** contains an explanation of the created framework that enabled distributed computing via web browsers to take place. **Chapter 4** describes the test setup that was developed in order to test the framework in an automated matter, at scale and under real world browsing conditions. **Chapter 5** contains the testing results for two distributed computing tasks that were mounted on the framework. Finally, **Chapter 6** contains the final conclusions for the project and recommendations for further work.

2 Background

In this chapter, the current state of volunteer computing will be described, followed by a relevant previous research into web browsing behaviour, and distributed computing via the web browsers. Finally, the key web technologies that enabled the subsequent development of the framework will be described.

2.1 Berkley Open Infrastructure for Network Computing

The majority of volunteer computing projects are hosted on the open-source *Berkley Open Infrastructure for Network Computing* framework [1], most commonly known as *BOINC*. The framework was initially released in 2002 and since then has had numerous major updates that have been managed via a community development and governance process.

While *BOINC* is the most mature and prevalent framework for volunteer computing, the usage statistics show a fixed or even declining number of project volunteers. For example, the *SETI@Home* project has 1.7 million volunteers with the *BOINC* user client installed, but only around 170 thousand users who are actively contributing CPU cycles to complete *SETI@home* tasks [2]. As outlined in the project preparation phase, this stagnation is likely explained by the initial overhead of downloading the *BOINC* client software and opting into a project before any device can be utilised. Additionally, the *BOINC* client is mainly targeted at desktop computer, while modern user behaviour has moved away to more portable devices for day-to-day activities, including browsing the web.

Unlike the aim of this project, where a volunteer does not need to download any additional software other than the standard web browser, *BOINC* requires a volunteer to download and install *BOINC* client software specific to their computing platform. The client runs as a constant background process on the user's operating system, so whenever their computer is switched on, the *BOINC* client will request and process tasks when free system resources are available.

As discussed in the introduction, this does create some volatility in volunteer availability, as volunteers will stop processing tasks when their device is switched off or fully utilised by other user processing. However, volunteer sessions are still likely to last from minutes to hours, and since the *BOINC* client is running as a local process, it can access the local file system and persist in progress tasks between processing windows. In line with this functionality, project owners using the *BOINC* framework create tasks that can take several hours to fully process, and multiple tasks can be downloaded and buffered in the client before any processing begins. The risk of a

volunteer not returning to continue an in-progress task is also lowered, as this would require the user to uninstall or disabled the *BOINC* client.

Tasks are allocated to *BOINC* clients via a central *BOINC* server, and each *BOINC* client is identified via a unique identification string. The task allocation works under the master-worker principle: a *BOINC* client, the worker, signifies its ability to process a task to the server, the master. The master monitors completion of tasks, and can allocate a task to a different worker if one worker has failed to return the task in a sufficient timescale.

Under the *BOINC* framework, all tasks must be entirely self-contained and isolated from each other. There is no client-to-client communication, which aligns with distributed computing via web browsers, as browser-to-browser communication cannot occur directly but must be facilitated through a central web server.

2.2 Web Browsing Behaviour and Distributed Computing via Web Browsers

In this section, the key findings from two previous research papers are discussed. These papers were reviewed in regards to understanding real world web browsing behaviour and potential methods of task scheduling that could reliably work under these conditions.

2.2.1 Understanding Web Browsing Behaviours through Weibull Analysis of Dwell Time

The Weibull probability distribution has been used extensively in the engineering field to help predict the lifespan and failure rate of mechanical and electronical components. The research team for *Understanding Web Browsing Behaviours through Weibull Analysis of Dwell Time* established that the Weibull distribution could also accurately model the dwell time of website visitors on individual webpages [3].

A single dwell time observation is the time a single website visitor spent on a webpage before navigating away from the page or closing the browsing tab. In the study, the research team obtained two-weeks of server logs for 205,873 web addresses, calculated the dwell time of users on each page, and then attempted to fit the Weibull distribution function to the dwell time observations using various input parameters to the Weibull function.

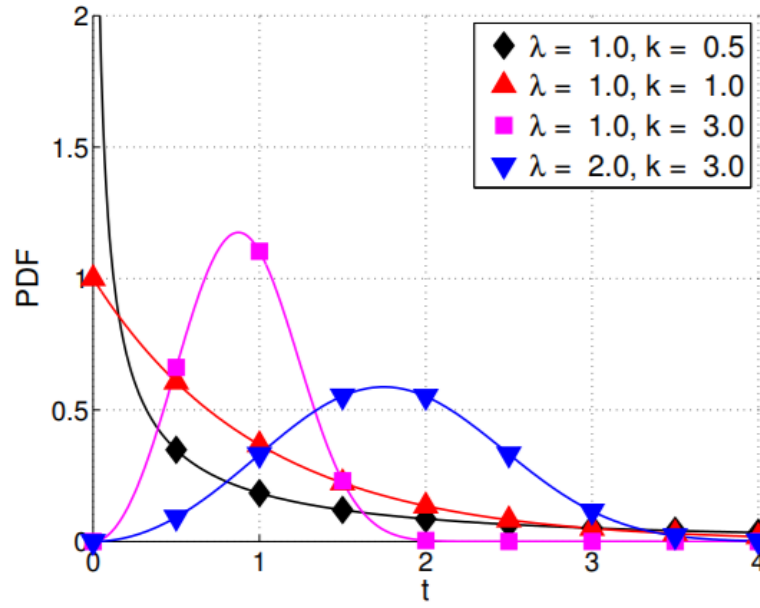


Figure 1: Example Weibull Distributions. Reproduced from [3].

The Weibull distribution function has two input parameters: shape and scale. As seen in **Figure 1**, the shape parameter controls the gradient of the curve, and the scale controls the total range of the values. For the Weibull probability plot, the X axis represents the time to failure of the observed entity, and the Y axis represents the probability of the observed entity failing at that point in time.

The key finding of the paper was that on 98.5% of the assessed webpages the fitted Weibull distribution function had a shape parameter of < 1 , which indicates a positive aging effect. In regards to component life time analysis, a positive aging affect is where the longer the component remains alive, the higher probably it has of remaining alive.

In regards to web browsing behaviour, this means that users are more likely to leave a webpage early on – for example, when quickly traversing through web pages to find interesting content to view – and more likely to remain on a page the longer they view it. For example, viewing the first ten seconds of a ten minute video and then deciding to stay longer, then after several minutes of viewing they are likely fully committed to watching until the end of the video.

In line with the positive aging Weibull distributions, real world webpage viewing habits are defined by many quick viewing sessions and fewer longer viewing sessions.

2.2.2 Gray Computing: A Framework for Computing with Background JavaScript Tasks

During the project preparation phase, several research papers were reviewed that investigated distributed computing via web browsers. The majority of these papers were published before the latest set of HTML5 web standards and concentrated on overcoming the technical limitations that were being imposed by web browsers to enable distributed computing. They did not evaluate the performance and feasibility of distributed computing in web browser under real world browsing habits.

Gray Computing: A Framework for Computing with Background JavaScript Tasks is the most recent paper published about this research topic [4]. The overall aim of the paper was to establish if it would be cost effective for website owners to offload computational tasks to their visitors in comparison to purchasing separate cloud resources to complete the same processing work. The impact of real world web browsing habits was assessed in regards to the potential adverse impact on cost effectiveness due to lost work.

In line with the behaviour shown by the Weibull distribution, the research team suggested a task scheduling policy where tasks are allocated to clients based on the clients current dwell time. In line with the dwell time, tasks would begin small and then increase in size as the client's dwell time increased until a threshold was reached. This scheduling policy was simulated in Matlab and it was concluded that it could provide up to 4% increase in completed task under real world browsing behaviour. While this suggests that new task scheduling policies could potentially provide a more reliable return of tasks in a Master and Volatile Worker environment, some questions about both the added complexity of this policy and the accuracy of the simulated testing remain:

- This task scheduling policy would require a distributed computing task that can scale in size, and potentially not all tasks can do this. Additionally, it would require project owners to create tasks of different sizes and ensure tasks of the appropriate size were constantly available to allow the scheduling policy to work.
- Smaller tasks sizes would increase the amount of messages between the server and clients. It is unclear if the latency for this increasing message passing has been factored into the calculations.
- The server, acting as the master, would now need to be able ascertain the size of each task and retrieve tasks based on task size – this could potentially slow down the fulfilment of each task allocation and put more processing load on the server.
- While the number of completed tasks has increased, it is unclear if the overall amount of completed processing has increased. The completion of many smaller tasks may not match the completion of less tasks of a larger size.

While the framework created by this project does not implement this task scheduling policy in regards to dynamic task sizes, the real world testing of the framework has helped answer some of the above questions.

2.3 Web Technology

In this section an explanation of the key technologies that are used for content delivery via the world-wide-web will be given, with focus on elements central to the understanding of the developed framework for distributed computing via web browsers

2.3.1 HTTP and HTTP Request Types

The *Hypertext Transfer Protocol* (HTTP) is the main method of communication between two internet connected devices [.

Request headers:	Response headers:
Host: www.wbpfserv.xyz User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:61.0) Gecko/20100101 Firefox/61.0 Accept: */* Accept-Language: en-GB,en;q=0.5 Accept-Encoding: gzip, deflate Referer: http://www.wbpfcient.xyz/ Origin: http://www.wbpfcient.xyz Connection: keep-alive	HTTP/1.1 200 OK Server: nginx/1.12.1 Date: Mon, 20 Aug 2018 13:40:37 GMT Content-Type: application/json; charset=utf-8 Transfer-Encoding: chunked Connection: keep-alive X-Powered-By: Express Access-Control-Allow-Origin: * ETag: W/"18-2K767IM4CQphWirHJNYntS7n9Eg" Content-Encoding: gzip

Figure 2: HTTP Request and Response Headers

A single HTTP transaction is formed of two parts: a request and a response. Each part is formed of request headers, which adhere to the HTTP protocol itself. In most instances, a client, such as a web browser on an internet connected device, will make a HTTP request for a specific resource identified by a Uniform Resource Locator (URL), which are more commonly known as web addresses. A web server that is responsible for this URL address will receive the request, parse the received request headers, create the appropriate HTTP response, and send it to the client, which in turn will parse the received request headers.

Various types of HTTP request can be made, each associated with a certain set of behaviour. **Figure 2** shows a HTTP GET request between a client and a server. Several HTTP request types include a message body, and this message body can contain many types of data, such as text or pictures. The message body is most widely used to deliver HTML, JavaScript and CSS code which forms a webpage.

For the development of the framework for distributed computing via web browsers, two types of HTTP request were used: GET and POST. A HTTP GET request is where the server sends a message body to the client in the HTTP response, and a HTTP POST request is where the client sends a message body to the server in the HTTP request.

These two HTTP methods were used to allow the passing of tasks, code files, and other elements between different components in the final architecture.

2.3.1 The Web Browser

Since HTTP is a text based protocol, it is entirely possible for internet communication to occur without using web browser. However, the web browser is the main platform for manual human interaction with the webpages. The web browser provides full multimedia platform, allowing pictures, sound, movies and user interaction. The combination of these elements has allowed the delivery of advanced content, such as massively multiplayer 3D video games played directly through the web browser.

The web browsing experience is made of three key technologies: HTML, JavaScript and CSS. Hypertext Markup Language (HTML) provides the content and structure of the webpage. JavaScript is a fully featured programming language interpreted by the web browser, allowing runtime, dynamic content to be delivered. Cascading Style Sheets (CSS) is used to style the look of the webpage.

While processing speed, interactivity and visual complexity of web browser based content has continued to advance with the new standards of HTML, JavaScript and CSS, some limitations in functionality have remained in place for security reasons. For example, the Web Browser cannot access files on the user's hard-drive unless specific user interaction sequence is followed. This is to stop malicious agents attempting to embed JavaScript code that reads, deletes, moves or copies files on a user's hard disk.

One of the key limitations in functionality imposed by the web browser in regards to distributed computing via web browsers is the inability to persist JavaScript computation between different browsing sessions. If a web browsing tab is closed, or the user navigates to a new URL in the same tab, the JavaScript computation will stop almost immediately. This is to stop malicious users trying to embed JavaScript code on one website, such as a key logger, and then continue its execution as the user navigates to other web pages. This web browser behaviour is the driver of the Master and Volatile Worker pattern, as volunteer clients can arbitrarily terminate in-flight computation with no notification given to the master.

2.3.2 JavaScript

The following sub-sections explain some of the key features of the JavaScript programming language in context of the development of the framework for distributed computing via web browsers.

2.3.2.1 JSON

JavaScript Object Notation (JSON) is the native serialization format for JavaScript data. Serialization is used to enable communication between computers with different application platforms, including different web browsers.

In the case of JSON, an in-memory JavaScript object can be parsed into a string via the `JSON.stringify()` function. This string could then be attached to the body of a HTTP request and sent to another machine.

A JSON string can be returned to an in-memory JavaScript object via the `JSON.parse()` function. Once parsed into a JavaScript object, the variables within the object can once again be accessed via JavaScript code.

With JavaScript and JSON, the receiver of the JSON string does not need to know the structure of the data within string. When parsing back into a JavaScript objects, numbers, strings, and JavaScript intrinsic objects, such as Arrays, are parsed back in the correct types.

When the `stringify()` is used on a JavaScript object that contains a reference to another JavaScript object, the referenced object will also be written out to the string by value. When parsed back into JavaScript both objects will be created with the in-memory reference restored. This functionality allows object-within-object serialization.

2.3.2.2 WebWorker API

Unlike many fully-featured programming languages, native JavaScript is a single-threaded programming language and does not allow any user controlled multi-threaded. This provides advantages and disadvantages.

For server-side programming, the single-threaded completion of client requests mitigates the need to manage data-race conditions between different threads of execution, and can enable a simpler codebase. However, for both server-side and client-side programming, areas of code with heavy computation or delays, will block all further processing until complete.

In the case heavy computation being done in the browser for the processing of a distributed computing task, this would block the processing of the rest of the website's code. For example, heavy computation could stop code related to user graphical user interface (GUI) interaction occurring and therefore create an unresponsive website.

As discussed in the Project Preparation phase, early attempts to complete distributed computing via web browsers attempted to overcome this JavaScript behaviour by creating code with set break intervals – where the task processing could be exited to allow other operations on the website to occur before returning for more task processing.

In effort to allow non-blocking computation in the web browser, the HTML5 standard introduced the WebWorker API. The WebWorker API allows a JavaScript code file to begin execution on a separate thread of execution managed by the web browser, and the API allows the newly created thread to communicate with the parent thread via a set of defined function calls.

2.3.2.3 Asynchronous Background Communication

As the web browsing experience and web browsers have become more advanced, the way in which data is transferred to the server to the browser has also changed.

Early web sites were mainly a static experience. The user would input a URL address into the web browser and receive all of the required content, images and data during the initial communication phase. In order to receive more data, the user would then have to follow an embedded URL hyper-link within the current webpage in order to trigger a full URL refresh and begin the process of loading another static webpage.

Many modern websites have moved towards a Single Page Application (SPA) approach. This is where a single initial URL address is used to load the webpage, and then to access more content, background asynchronous communication requests are made to the server, with the content of the current webpage being updated with the responses. This update method facilitates much of modern web browsing experience, such as real time chat windows, where each message is passed between the users and servers via background asynchronous communication. Under this method, the user does not have to navigate between URL addresses in their web browser in order to receive more content.

To achieve this asynchronous message passing, two native communication objects are provided in the current HTML5 and JavaScript standard for web browsers.

The **XMLHttpRequest** communication object allows fully fledged HTTP communication to be made without refreshing the browser URL address. As a request over the network is classed a slow operation, the request is done asynchronously. The HTTP request is sent by the client web browser, however program execution continues without waiting for the equivalent HTTP response message.

A call back function is registered with the XMLHttpRequest object, which will be called when the HTTP response has been received from the server. The ability to continue execution between the HTTP request and response provides the possibility of task processing and message passing occurring in parallel.

Each end-to-end HTTP request is fully isolated from other HTTP requests made between the same client and server. One factor to consider when using HTTP is the potential overhead this form of messaging passing could introduce if done at a high frequency, as the full HTTP processing, including generation and parsing of request headers, will need to be completed for every message. Additionally, the HTTP request must always be initiated by the web browser client, which enforces a master-worker pattern where the master is reliant on notification from workers to understand the current status of the task computation.

WebSockets is a communication protocol added after XMLHttpRequest protocol. In contrast to the previous protocol, WebSockets opens a continuous connection between the client and the server and does not use the HTTP protocol. In this way, the server can now initiate messages to the client. Additionally, the server can ping the client at set intervals to establish the client is still active.

Since a continuous connection is made, a smaller amount of meta-data is needed for each communication request compared to HTTP. Potentially this could allow faster message passing than the XMLHttpRequest method.

2.3.2.4 Cross-Origin Communication

By default, a web server will only respond to XMLHttpRequest or WebSocket communication requests from the same origin as the web server. This means that the communication request must be coming from a webpage that was served by the same server. For example, an XMLHttpRequest communication object could not make a HTTP request from webpage www.site-one.com to www.site-two.com. The server for the second site would reject the request with the appropriate error code provided in the HTTP response.

The blocking of cross-origin communication requests is for security reasons. Cross-Site Request Forgery (CSRF) is where a cross-origin request is made to target website that has some trust in the web browser's credentials. For example, a malicious agent could embed code in a webpage which makes a cross-origin request to a webstore that already classes the user as logged in due to unique cookie data present in the web browser's local cache.

As the web browsing experience has become more advanced, however, the desire to embed multimedia content from other website providers has increased. For example, a web video streaming service may want to allow users to stream a video from their website on a different webpage. With this desire, later iterations of HTTP standards have allowed cross-origin requests to occur, and the ability for these requests to be accepted is controlled by the receiving server.

2.3.2.5 URL Query Strings

When requesting a resource from a URL via HTTP, either by updating the main browser URL address or via a XMLHttpRequest, additional information can be provided to the receiving server by using query string values:

`http://www.server.com/resource?length=100&height=150`

As seen in the above example, a ? symbol can be used after the URL address itself to begin inserting query string values. A value, either string or number, can be associated with a variable name, and multiple variables can be separated using the & symbol. For the subsequent development of the framework, query strings were used to allow communication between separately deployed components.

2.3.2.6 Node.js

Node.js is a JavaScript interpreter that is mainly used for in the development of web servers. Node.js was chosen for the development of the framework's master server for performance reasons. Node.js uses the Google V8 engine and has an established reputation for speed in message when using the WebSockets protocol. Due to the single-threading nature of the Node.js runtime, it can be powered by relatively low-powered hardware, as there is no benefit added by the use of additional CPU cores. This could allow a positive impact on the total running costs of the framework.

3.3.3 Summary

While web technology is a broad field, the previously explained technologies are the key components that were used for the development of the framework for distributing computer via web browsers.

3 Panther: A Framework for Distributed Computing via Web Browsers

3.1 Introduction

In this section, an explanation of developed framework for distributed computing via web browsers will take place. The final framework is called Panther, and the Panther framework consists of three key components:

- a Node.js master server for centralized task management.
- a JavaScript API code-bundle that is sent to volunteers' web browsers to enable communication with the master server.
- HTML and JavaScript code embedded in a client webpage that initiates the Panther task distribution process.

Three extra components were also developed and deployed, and these extra components allowed testing of the Panther framework under live conditions: where multiple concurrent visitors to a client webpage take part in the completion of a distributed task hosted on the Panther framework under real world browsing habits.

In the first section, a summary of the key factors that influenced the final architecture of the Panther framework will take place. This is followed by a high-level explanation of each developed component, followed by a more detailed explanation of each component. Finally, the end-to-end life-cycle of a volunteer browsing session under the Panther framework will be described.

3.2 Framework Development Goals

For the development of the Panther framework, eight key goals development goals were present:

1. Create the framework so it can be either can be **deployed independently** or act as an **expansion** to existing volunteer computing tools, leveraging existing functionality that has already been developed by the volunteer computing community.
2. Allow switching between the **XmlHttpRequest** and **WebSocket** protocols when the master server communicates with web browser workers.
3. Allow **switching** between different distributed computing tasks without having to modify or redeploy code.
4. Place **minimal constraints** on how project owners structure their task data and task processing code.
5. Log sufficient **data** to allow analysis of the framework's performance and the end-to-end task processing lifecycle.
6. Prioritise **speed** so tasks are allocated to workers in timely manner by the master.

7. Allow workers to **checkpoint** incomplete tasks and return them to the server in their current state, allowing the task to be continued by another worker.
8. Create no negative impact on **user experience** when tasks are being processed in their web browser.

How development goals 2-8 were achieved will be explained in the upcoming explanation of the Panther framework's design. For development goal one, and the ability for the Panther framework to work as an addition to the volunteer computing community, it required a review of the workings of *BOINC* in line with software development principles.

As previously discussed in the background chapter, the majority of volunteer computing projects are hosted on the open-source *BOINC* framework, a mature toolset with an active development community. For existing volunteer projects, the move to an entirely new framework that allows web browser based clients would create an unwelcome overhead, as potentially two separate back-end server stacks would have to be paid for and maintained in unison. Project owners would also have to ensure that tasks are not duplicated between the two frameworks in order avoid unnecessary duplication of work by project volunteers.

Modifying the code base of *BOINC* framework is an undesired outcome. Forked and modified versions of *BOINC* framework have been created in the past, but there has been little uptake of these versions by volunteer projects, showing that project owners prefer to utilize the core build of *BOINC*, which is supported by the community governance process.

In regards to the above, the code base of existing volunteer frameworks was viewed as immutable in regards to the addition of web browser based volunteers. The architectural aim of Panther was to create an independently deployable framework that could i) interface with existing volunteer computing frameworks and leverage their functionality without the existing frameworks having to be modified or redeployed ii) also be used as a separately maintained component for entirely new developments in the volunteer computing community.

A key enabler of this goal was the transfer method between the *BOINC* server and *BOINC* clients. Clients request and return tasks from the *BOINC* server via HTTP requests.

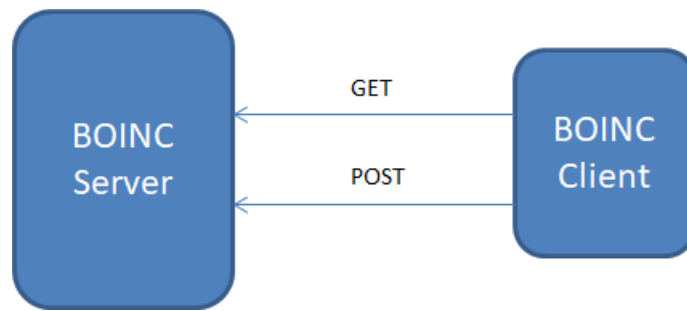


Figure 3: BOINC to Client Communication

A non-*BOINC* program could communicate with a *BOINC* server if the correct format was used for the HTTP request header and message body. This introduces the idea of a custom-client that could request tasks from a *BOINC* server, complete these tasks, and then return them to the server, plugging seamlessly into the existing *BOINC* server architecture.

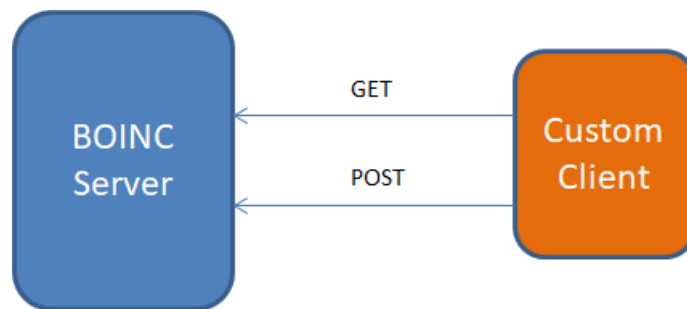


Figure 4: BOINC with Custom Client

However, if the custom-client was directly created in web browser code, it is likely that some difficulties would occur in direct communication between the web browser and the *BOINC* server:

- The web browser may not be able to locally store or interact with the files that are sent by the *BOINC* server.
- Server to browser communication protocols would be limited to what *BOINC* already uses to transfer data.
- As *BOINC* is a framework that was designed around persistent background processing, it is unlikely that requests for new tasks by the client have been optimized low-latency response as required by short web browsing sessions.

The above could be resolved by adding a layer of indirection between the web browser and the *BOINC* server, introducing a proxy-server between the *BOINC* servers and the browser-based client.

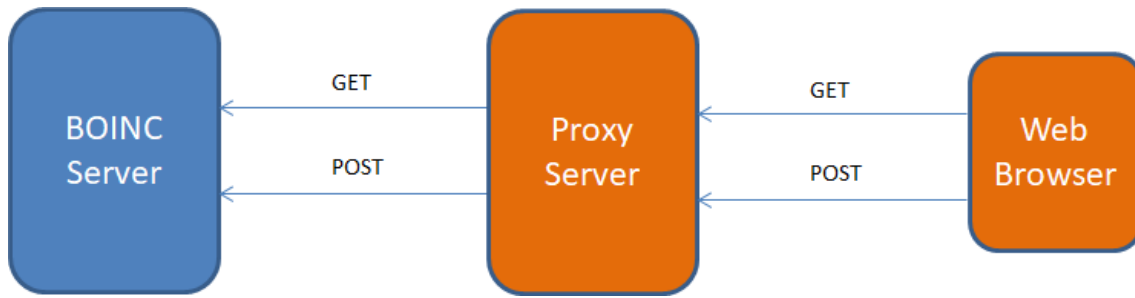


Figure 5: BOINC to Web Browser Client via Proxy Server

The proxy server could interact with the *BOINC* server as a custom-client, complete any modification activities required for the *BOINC* tasks to be delivered and handled by a web browser client, interact with web browser clients under its own task allocation and scheduling policies, and then return completed tasks to the *BOINC* server in the correct format.

For any current or future volunteer computing frameworks with defined HTTP or FTP client communication interfaces, the proxy-server could act independently deployable component that allows a conduit between the framework itself and web browser based volunteers. The fact existing frameworks will not have to modify their code or be dependent on the code of the new framework aligns with good software development principles.

It is likely that specific web browser only tasks would need to be generated for web browsing based processing, and this can already be handled by the *BOINC* framework. Under the *BOINC* framework, tasks are associated with meta-data that informs the *BOINC* server what application platform the task should be run on. When a client makes a task request to the *BOINC* server, it identifies what application platform it is running on. This allows project owners to create tasks that run on specific hardware and allocate them only to clients that are running that hardware. A task coded in CUDA, for example, will only be sent to volunteers who have a CUDA-compliant NVIDIA GPU. The *BOINC* meta-data could be updated to have a new application platform type for web browser only tasks, and the proxy server could identify itself to the *BOINC* server as only processing these types of tasks.

The designed framework for distributed computing via web browsers should also have the potential to scale to extreme users volumes, in line with the framework being deployed on a high-traffic website. When handling high volumes of requests, computer architects have two options to increase processing power in order to provide quick response times to client requests:

Vertical Scaling is when web servers processing power (CPU, Memory, etc.) is increased to allow quicker fulfilment of each client request.

Horizontal Scaling is where web server code is replicated into new machines and client requests distributed across the multiple servers. Client requests are first sent to a load balancer, which then routes the request to one of the back-end servers based on the desired scheduling policy.

While both methods can be used in unison, when dealing with extreme user volumes, horizontal scaling becomes mandatory as vertical scaling is ultimately limited by the maximum computing power that can be achieved in a shared memory server node.

The BOINC framework is already designed to handle tasks allocation to millions of unique volunteer clients. This could be exploited when horizontally scaling the framework, with each new server instance identifying itself as a new client to the *BOINC* servers. A load balancer could be configured to ensure volunteer client requests are routed to the same server to ensure each completed task is passed back to its original origin server.

As discussed in the Project Preparation phase, the proxy server code, which provides tasks to web browser clients, could be hosted in several places.

Option One: The proxy-server code could be hosted on the same server that provides the webpage to the client. In essence, the client webpage server would also fulfil an extra role as the task manager for its visiting clients.

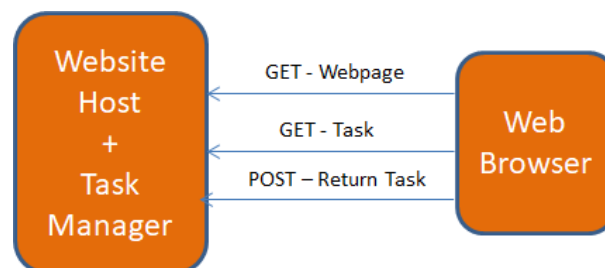


Figure 6: Webpage Server acting as Framework Host

Option Two: The proxy-server and task manager is hosted on its own server, and the website visitor communicates with this server via cross-origin HTTP requests. The client webpage (delivered by the original server) will still have to embed a small amount of additional code in order to initiate this communication.

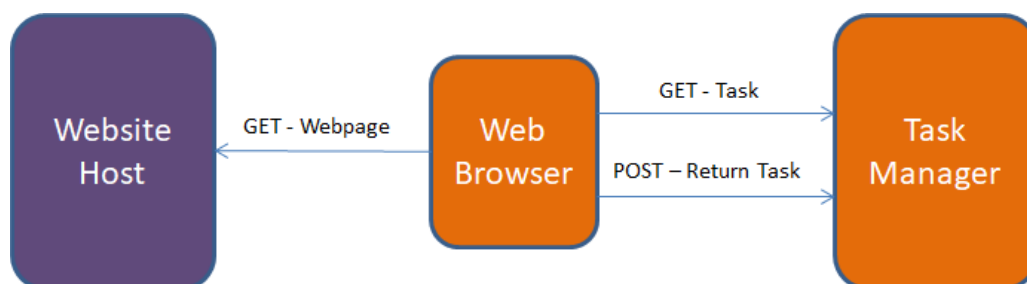


Figure 7: Framework hosted on separate server

It was decided that option two, hosting the proxy-server on its own server was the most optimal choice for the following reasons:

- There are many server-side programming languages and frameworks, and to enable full utilisation of these servers it would mean coding a version of the PANTHER framework for each platform: JavaScript, C#, Python, PHP, etc.
- While site owners may be willing to let their site visitors become project volunteers, they may be unwilling to take on additional server running costs themselves. Running the task manager code on the same server as the website server will likely affect its performance and the site owners may need to increase the power of their servers or deploy additional servers.
- A proxy server that is hosted on its own server would allow it to service visitors from multiple different websites via cross-origin HTTP requests. A webpage server would only service visitors from its own domain.

3.3 Panther Framework Architecture

The following figure shows the high-level view of the Panther framework and supporting components that enabled live testing and evaluation.

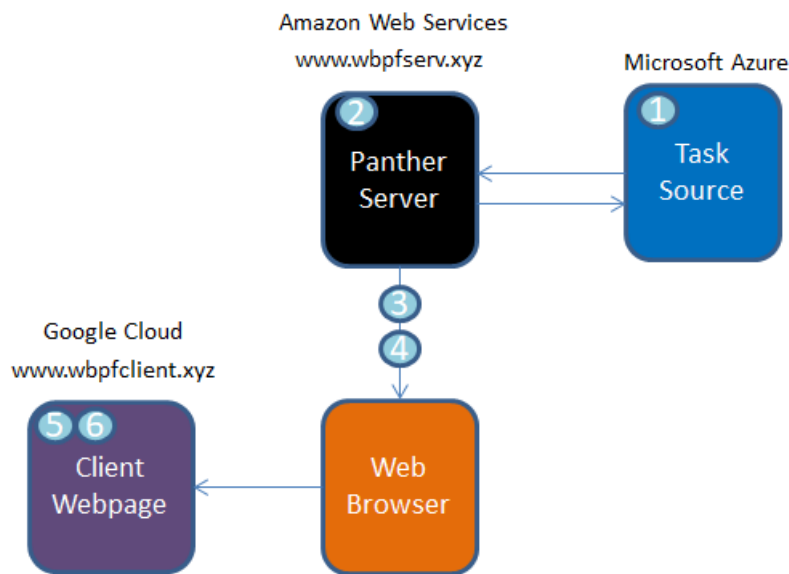


Figure 8: Panther Components

To ensure that testing of the framework happened under as close real world conditions as possible. As seen in Figure 8, each component has been hosted on separate cloud computing providers. This is in effort to replicate real world latency for communications between each component. Large cloud computing providers have their own data centres, connected by their own network cables, and are optimized for speed. If all components were hosting on the same provider, there is a risk that message passing between components could be unfairly optimized.

Six components make up the live environment:

Component One: Task Source

The Task Source (c1) is the origination point for unprocessed tasks, and the long-term storage point for completed tasks. This component could be fulfilled by existing framework, such as BOINC, or could be newly created.

Component Two: Panther Server

The Panther Server acts as an intermediary between tasks originating from the Task Source (c1) and visitors to the Client Webpage (c5), who will process the tasks. It requests tasks from the Tasks Source, caches them in memory, and then allocates them to Client Webpage (c5) viewers under various communication and scheduling policies. After a task is completed by a visitor, the server returns it back to the Task Source (c1).

Component Three: Volunteer Task Code

This is JavaScript code which executes the task in the webpage visitor's web browser. The code is stored on the Panther Server (c2) and delivered to the visitor's web browser during the initial connection between the two components. The task must be mounted via the Panther Client API (c4) in order for it to be run on the Panther framework.

Component Four: Panther Client API

A client-side JavaScript API that project owners must use in order for their Task Code (c3) to correctly interface and communicate with the Panther Server (c2). Using an API allows multiple different types of tasks to be delivered via the Panther framework without the project owners having to understand the specific implementation details of the framework.

Component Five: Client Webpage

A webpage which has opted in to the Panther framework: when opening the webpage in a webpage, visitors will begin processing tasks sent to them via the Panther Server (c2) using provided code (c3 and c4).

Component Six: Client Webpage Embedded Code

A small amount of HTML and JavaScript code that the Client Webpage (c5) owners must embed on webpages they want visitors to process tasks on. This initiates the initial connection between the Client Webpage (c5) and the Panther Server (c2), and begins the task processing on a background thread of execution.

Each component will now be described in detail.

3.3.1 Component One: Task Source

While the Panther framework has been designed to interface with existing volunteer computing frameworks, such as BOINC, for the live deployment and test setup, a new set of code has been created to fulfil the purpose of the Task Source component.

This choice was made for the following reasons:

- The BOINC development community advise that approximately a three month period is required to setup a new BOINC-based volunteer project. While this initial overhead is acceptable for a long-term volunteer computing project, it is too long for this project's development timescale.
- In reality, BOINC is made up of multiple servers each fulfilling different roles in the volunteer computing process. Having to host multiple servers, would add to overall project costs. As seen in the final component, the Panther framework has the option of interfacing with a newly created component that has allowed newer cloud computing technologies to be used to create a lower-cost task source.
- As testing the same task at different sizes is a key component of the evaluation, creating a new component has allowed specific functionality to be added to allow quick configuration of different test cases. The new component can dynamically generate task data based on submitted arguments, allowing tasks of different sizes (both data size and computation time) to be created as required. While different size tasks could be inserted into the BOINC framework, the client does not have full control over which tasks are allocated to it, potentially leading to an issue where BOINC allocates the wrong tasks for the desired test setup.

The Task Source component has been created on the Microsoft Azure cloud computing platform, using server-less functions to provide and receive task data via HTTP calls. Server less functions are a service offered by most large cloud computing providers and are a way for users to avoid having to pay for the 24/7 running cost of a web server that only provides services that are needed sporadically. Instead of paying for hourly hosting of the server, the user pays a small amount for each call of the function.

Functions are provided their own unique URL, can be accessed via all HTTP request types, and can interact with other services provided by cloud computing providers. Since functions are accessed via HTTP requests, URL query strings can be used to provide parameters to the function call, and responses can be given via the HTTP standard.

Three functions have been created to act as the Task Source. When called with a HTTP GET request, one function returns a Dartboard PI task data in JSON format, and one function returns Mandelbrot task data in JSON format. These tasks will be discussed further in the Testing and Evaluation phase.

The Mandelbrot function has been created to generate the task data during its execution and not obtain the data from a persistent data store. This method allows URL query strings to be used as function parameters to shape the generated data.

<https://generatetask.azurewebsites.net/api/MandelbrotTask?width=300&height=200&iter=1000>

For example, calling the Mandelbrot function at the above URL with the query string parameters will create a task which requires a Mandelbrot image to be generated with a width of 300, a height of 200, and 1000 iterations per pixel. While a real distributed computing task would likely have unique packets of data to be allocated to clients and obtain them from a persistent data store, as seen in the following components, this functionality allows different test cases to be generated without any redeployment of servers or manual generation of data.

One function is used to return completed tasks to the Task Source. It is called via a HTTP POST request, with a message body in the JSON format. A NoSQL Document Database is also hosted on Microsoft Azure cloud platform, and the function inserts the completed task data into this database. Once in the database, the Microsoft Azure cloud console can be used to query the results, and this has been used to ensure the correct result is being returned for all tasks after each testing run.

3.3.2 Component Two: Panther Server

The Panther Server is the most complex component with the largest codebase. It is a Node.js server hosted on Amazon Web Services (AWS). The Express framework has been used on top of Node.js for basic handling of HTTP requests for various URL routes, and multiple JavaScript modules have then been created to manage the end-to-end task processing lifecycle.

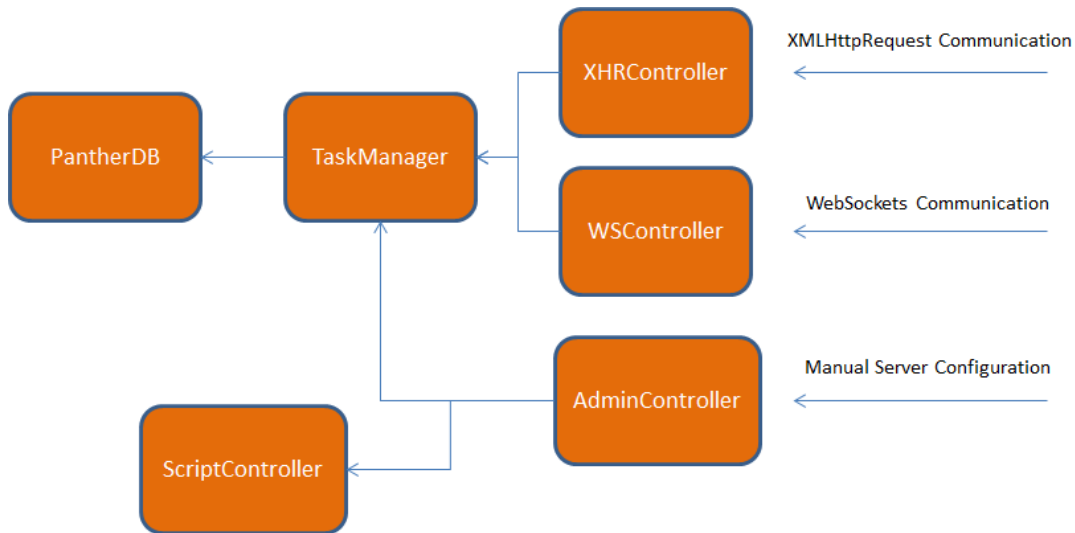


Figure 9: Panther Server Module Structure

Figure 9 shows the module structure. The arrows show the direction of the dependency between the modules. For example, the **XHRController** module is dependent on the **TaskManager** module, while the **TaskManager** has no dependency on, or knowledge of, the **XHRController**.

A key driver for the module structure was enabling the Panther Server to communicate with client browsers under several different communication protocols in an interchangeable way. For example, the task scheduling functionality remaining entirely encapsulated from the communication protocol, each with its own module, used to transfer the task to the client.

The **AdminController** module is responsible for rendering a HTML webpage to the Panther Server owner, where they can control the key functionality of the server.

The **TaskManager** is the module that controls task storage, allocation and scheduling.

The **PantherDB** module is responsible for opening a connection to a MySQL database and writing log entries via SQL queries.

The **WebSocketController** and **XHRController** modules are responsible for handling direct communication with web browser clients via the respective communication protocols.

The **ScriptController** module is responsible for sending the appropriate JavaScript task file to web browsers clients as the initial communication is established.

Each module will now be described in detail.

3.3.2.1 Module: Admin Controller

The server owner, or project owner, interacts with the Panther Server via the Admin Control webpage at the www.wbpfserv.xyz web address. The adminController.js module is responsible for generated the HTML webpage view for the user, receiving and parsing the returned user form, and implementing the contained instructions from the administrator.

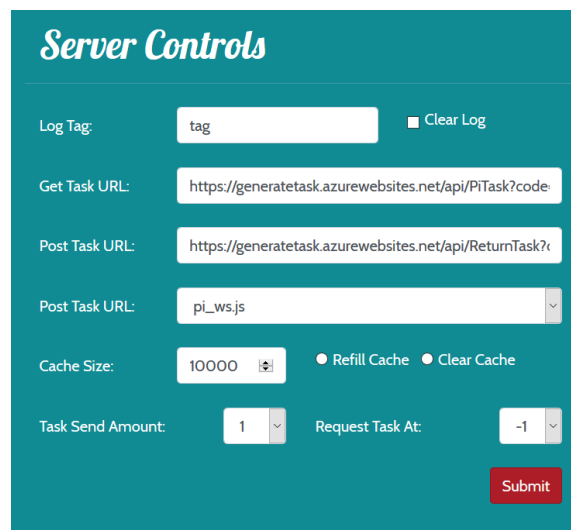


Figure 10: Panther Server Admin Controls

Figure 10 shows the control dashboard and the instructions that can be submitted by the server administrator. The instructions are as follows:

Log Tag

A separately hosted MySQL database is used by the Panther Server to store a log of task management activities undertaken by the server. This enables subsequent analysis of the performance of the Panther framework. The current Log Tag is written out with ever log entry, and being able to change this tag allows different test cases to be tagged with a unique tag for later retrieval and analysis.

Clear Log

Clears all log entries present in the MySQL database.

Get Task URL

The URL that the Panther Server will query and request tasks from. This is what connects the Panther Server to the Task Source, providing either one of the two task function addresses. As previously discussed, URL query string parameters can be included to set what size of the task should be generated by the Task Source, allowing the current task on the Panther Server to be amended without any redeployment of the server.

Updating this URL address will clear some of the state on the server, clearing any currently held tasks from local memory. This is to ensure that no tasks of different types or sizes are being allocated at the same time.

Post Task URL

The URL that the Panther Server will return completed tasks to in JSON format. Again, connecting the Panther Server to the Task Source, which will as previously discussed store the completed tasks in a document database.

Task Script

Sets which JavaScript code module will be sent to the client web browsers on the initial connection. Web browsers use the module to process tasks they receive and communicate with the Panther Server. How this code is generated and packaged is discussed in the upcoming sections.

Cache Size

This numeric input sets the maximum amount of tasks that the Panther Server will manage at one time.

Refill / Clear Cache

A toggle option that when set to Refill, will make the Panther Server requests tasks from the Task Source until Cache Size threshold it reached. When set to Clear, any tasks currently being held in the task cache will be cleared from the server.

Task Send Amount

When processing tasks, web browsers clients have several configuration options that allow different scheduling methods for requesting and receiving tasks. This dropdown option sets how many tasks that are sent to each client at each new task request. This ability is discussed further in the upcoming sections.

Request Task At

Due to the client being able to receive and buffer multiple tasks, the client also has ability to request a new task(s) when the amount of local remaining tasks is reached. For example, a web browser client could request three tasks at a time, and when one task is left remaining, request another batch of three tasks before beginning processing of the final task. This introduces the concept of asynchronous tasks requests, where tasks are processed in parallel to new tasks being sent by the Panther Server. By default, the web browser client does not make a task request until no more tasks are available, effectively synchronous task processing, but this option can be used to enable asynchronous message passing.

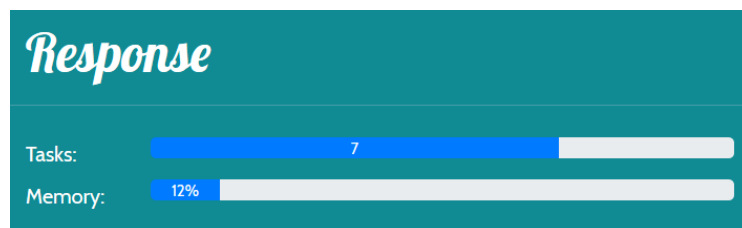


Figure 11: Admin Screen Real-time Metrics

Additionally, as seen in **Figure 11**, the administrator webpage also provides some real-time metrics on the condition of the server. Two progress bars show i) the current amount tasks remaining in the task cache ii) the current memory usage of the server. Both of these progress bars are updated every several seconds when viewing the admin controller webpage.

3.3.2.2 Module: Task Manager Module

The Task Manager module is the core module of the Panther server, and most other modules within the server have some form of dependency on its functionality. For example, the previously discussed Admin Controller module makes numerous function calls to the Task Manager when being instructed to fill, clear, or resize the Task Cache.

The Task Manager is responsible for obtaining tasks from the Task Source, storing tasks in local memory, managing the retrieval of the next task(s) for a client, monitoring the completion status of tasks, writing a log of actions to a log, and returning completed tasks to the Task Source. The Task Manager module contains a singleton object, which other modules can get a reference to by calling a public function from the module.

The key calling points for the module are the **FillCache()**, **GetNextTask()** and **HandleClientResponse()** functions, corresponding to three main routes of execution for the TaskManager: the initial request for new tasks from the Task Source component, the retrieval of the next task for processing by a web browser client, and the handling of a task returned from a web browser client.

FillCache() Function

The FillCache() function begins the process of the Panther Server requesting Tasks from the Task Source component. This process also demonstrates how the Task Manager module monitors the status of each task.

On initiation of this function, via the AdminController module, the Task Manager establishes how many tasks are needed to fill the task cache, and requests them, via individual requests, from the currently set Task Source URL. On receiving each response from the Task Source, the InsertTaskIntoCache() function is called with the Task Source's response message, which contains the task data in JSON format.

In line with overall architectural aim of the Panther Framework, being an independently deployable component with no dependencies on other existing frameworks, the Task Manager does not rely on any specific task meta-data being sent from the Task Source, but generates its own meta-data to manage the task lifecycle. Additionally, an existing framework, such as *BOINC*, is likely to have its own meta-data associated with the task, and this will need to be persisted through the Panther Framework's lifecycle and returned back to the Task Source. For example, the Task Source may have its own unique ID to identify each individual task, but the Panther framework itself will disregard this and generate its own unique ID for each task. The Task Manager does this by encapsulating each task in its own TaskContainer object with its own unique TaskID.

```
function CreateTaskContainer(taskID, task)
{
    return {
        tid: taskID,
        sid: "",
        sessionTaskNumber: 0,
        sessionDowntime: 0,
        status: "unprocessed",
        checkpointNumber: 0,
        timeTaken: 0,
        Task: task };
};
```

Figure 12: TaskContainer Creation Code

Figure 12 shows the meta data that the TaskManager associates with each task.

The **Task ID** is 16-digit random hash string calculated by using inbuilt cryptographic functions present in Node.js. At each generate of a new Task ID, a check is undertaken to ensure this ID is not already in use by another task present in the Task Manager.

The other meta-data variables are updated as the task is allocated to web browser clients, and are explained in the upcoming sections.

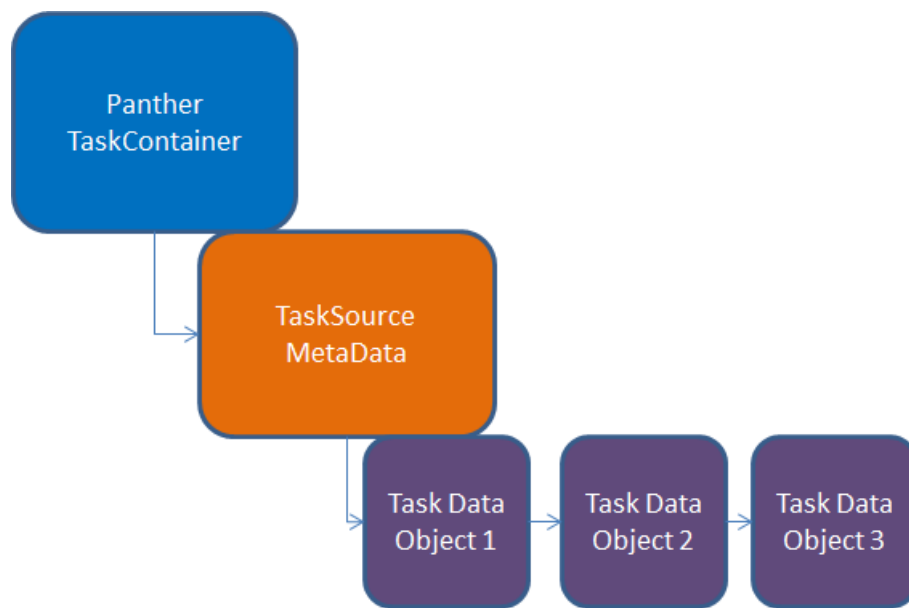


Figure 13: Example Task Structure within TaskContainer

The task data received from the Task Source component is parsed into a JavaScript object and is assigned as a variable within the TaskContainer that can be accessed like any other variable. Because of the native abilities of the JSON format, it is possible for multiple JavaScript objects encapsulated within each other to be parsed and unparsed. Therefore the actual task itself could be made up of any format. For example, multiple JavaScript objects linked in a graph like structure; or a single object containing an array of other objects. No task format is enforced by the Panther Framework on the project owner other than the need for a root JavaScript object which may be a standalone object or container for other objects. **Figure 13** shows an example of potential object structure where the TaskSource also has its own meta-data container.

Once the TaskContainer object has been created for the newly received task, the TaskContainer object is inserted in a JavaScript **Map()** object, which is a hash table. In this data structure, a value is associated with a key identifier, and values can be added and retrieved from the Map() structure in constant time operations if the key value is known. The Task ID functions as the key, and the TaskContainer object is treated as the value. With this data structure, the TaskManager can retrieve any TaskContainer and its encapsulated task in constant time using the corresponding Task ID (as long as sufficient hashing function has been implemented in Node.js). Additionally, there is no time and space complexity added when increasing the number of tasks being held by the Panther Server. This method of task storage and retrieval has been used to ensure quick response times by the server when a web browser client is requesting a task for processing.

Additionally, in aiming to provide quick response times for task retrieval, a key feature of the Panther Server is that it does not store the tasks in any kind of persistent storage, such as a hard disk. The tasks are stored in the JavaScript Map() object and nowhere else, meaning that they only exist in dynamic memory, which is faster to retrieve data from than a hard disk. The current memory usage bar on the Admin Control webpage can be used to ensure that the server memory does not overflow and begin paging onto the hard disk.

The memory only approach to task storage does limit the total amount of tasks that the Panther Server could store compared to the potential of hard disk storage, and an additional risk is that the failure of a server will mean the loss of all task data on that server, however the decision to have reduced task storage space and data redundancy has deliberately been made to focus on the speed of the server when trying to fulfil client task requests as quickly as possible. Additionally, if being used with a framework like BOINC, any tasks lost due to a server crash will simply be rescheduled to other clients.

After the TaskContainer object, encapsulating the newly received task from the Task Source, has been added to the Map() data structure, the Task ID of that TaskContainer is used in a second data structure: a JavaScript **Queue()** object that is used to schedule the next task to be processed. The Task ID of TaskContainer is added onto the back of the queue, using the enqueue() function. At this stage, the processing of a new task from the Task Source component is complete.

GetNextTask() Function

The WebSocketsController and XHRControllers are the modules that directly communicate with client web browsers. When the request for a new task for a web browser client is received, they call the **GetNextTask()** function on the TaskManager module, providing an argument of how many tasks are required.

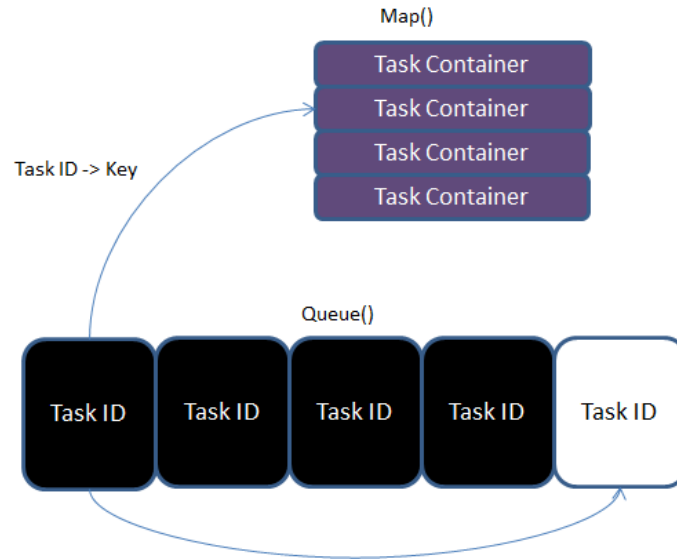


Figure 14: Task Allocation Process

Figure 14 shows the task retrieval process. When calling the `GetNextTask()` function in the `TaskManager` object, the first `Task ID` present in the `Queue()` object is removed using the `dequeue()` function. This `Task ID` is then used as the key to retrieve the equivalent `TaskContainer` from the JavaScript `Set()` object. Again, both the `enqueue()` and `dequeue()` functions are constant time operations no matter how large the queue grows in size. This combined with constant time retrieval of the task from the JavaScript `Map()` object means that the retrieval of a task is always a constant time operation.

Unlike a task work-queue where reliable workers are present and the task may now be fully removed from the work-queue, in the Panther Server the process is different. Since the workers are unreliable and volatile, there is no guarantee that an allocated task will be completed, so the task must still be able to be allocated again in the future. In light of this, the `Task ID` that was removed from the front of the queue is also added to the back of the queue, creating a cycling effect where the task will eventually return to the front of the queue for allocation again, and this process may be repeated multiple times before a task is successfully completed. If a `Task ID` is removed from the queue and it is not present in the `Map()` object, then the task has already been completed, and the further entries are removed from the queue until an uncompleted task is found. Since Node.js is a single threaded environment, there is no risk of data race conditions when accessing both data structures. When multiple tasks are requested, the above process is repeated for the required amount of times.

The cycling of tasks within the queue also ensures that when the task cache is approaching empty status, there is no delay in completing the remaining tasks. If tasks were allocated and a fixed time was waited before reallocating that task - waiting to see

if the task has been successfully completed - then potentially a large delay could occur as the final tasks might need to be allocated several times before successful completion. Under this method, as the queue size shrinks, tasks will cycle to the front of the queue quicker. If the queue size is small enough and number of client browser sessions is high enough, the task could be allocated to two different clients in parallel, increasing the chance of its completion.

HandleClientResponse() Function

When the WebSocketsController or XHRCController, the modules that directly communicate with client web browsers, receives a returned task from a web browser client, they pass the returned TaskContainer object to the TaskManager via the **HandleClientResponse()** function.

At this stage, the TaskManager reviews the meta-data within the received TaskContainer object. If the status variable in the TaskContainer is set as 'complete', then the client has fully processed the task. The TaskManager returns the completed task, without its associated meta-data added by Panther, to the Task Source component via the URL set on the Admin Controller webpage. The Panther Server also removes the associated TaskContainer from local memory.

If the status variable of the TaskContainer is set to 'checkpoint', then the client has returned a partially completed task to the Panther Server. At this point, the Panther Server retrieves the equivalent TaskContainer from the JavaScript Map() object in local memory. A side-by-side comparison of the local TaskContainer and the client's returned TaskContainer now takes place. If the checkpointNumber numeric variable is higher on the client's returned TaskContainer than the server-side TaskContainer, then the client has advanced the processing of the task. The server-side TaskContainer is overwritten by the newly received TaskContainer. If the checkpointNumber variable on the received TaskContainer is smaller than or equal to the server-side version, then no further advancement of the task has taken place, and the returned TaskContainer is discarded with no further action.

When task check pointing is in use, the same task scheduling policy is still used, however check-pointing provides the potential for a web browser client to receive a task that has already been partially processed, and multiple different browsing sessions may work towards the completion of a single task.

3.3.2.3 Module: PantherDB

The PantherDB module is the enabling of logging of Panther Server actions and subsequent analysis of performance and task handling. The module provides two public functions. One called writeLog(), which is predominately called by the TaskManager module whenever an action related to task allocation occurs.

On calling the writeLog() function, the PantherDB opens a connection to an external MySQL database, hosting on Amazon Web Services, and writes a single row of output to a table. More advanced types of SQL queries – such as multi-row inserts or table joins – have been avoided at this stage in order to maintain fast performance.

The Task Manager writes to the PantherDB at the following stages:

1. When a new connection to a web browser client is started
2. When a task is requested by a web browser client for processing
3. When a task is returned from a client that has a status of completed
4. When a task is returned from a client that has a status of checkpointed
5. When a completed or check-pointed task is returned from a client that is a duplicate of already completed work.

The fields are populated via the meta-data contained in the TaskContainer that is being handled at that stage of the request. While the output data is transactional in nature, logging into the database with MySQL database with a front-end GUI application allows more complex SQL queries to be performed to aggregate the data and understand the full end-to-end lifecycle of each webpage viewing session. Key data features, such as how many tasks each individual viewing session completed, or the average time to complete the task across all website visitors, can be calculated.

3.3.2.4 Modules: WSController and XHRController

A separate module has been created to directly handle the requests from client web browsers for each communication protocol used by the Panther Server.

The XHRController uses the Express framework to handle the XmlHttpRequests HTTP requests made to several server URLs.

The WebSockets protocol does not use HTTP requests, so to understand what type of request is being made by the client, a separate msgType tag is used for each communicate request between the client and the server.

Both modules are primarily dedicated to handling the specific issues raised by each communication protocol but delegate most of the requests to other modules on the server. For example, both modules will call the GetNextTask() function of the TaskManager module whenever a client requests a new task for processing.

The modules are also responsible for decompressing and JSON parsing of the received messages from the client before passing the data to other modules.

3.3.2.5 Module: Script Controller

The ScriptController module is responsible for sending the JavaScript code to be executed by the client web browser. Each code file is the combination of two separate components: the Task code, and the Panther Client API code, which are explained later in this chapter.

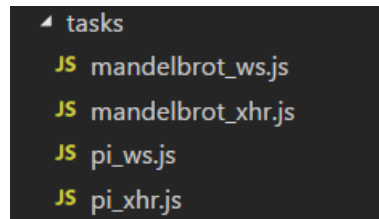


Figure 15: Bundled Task Scripts with Panther Client API

The code files are placed in a folder named *tasks* on the Panther Server, and the code file that is sent to the client can be set on the Admin Controller webpage, which interfaces with the ScriptController module.

When the task code is under development, the Panther Client API code is imported as an external module. However, multiple JavaScript code modules become an issue when executed in the browser as a WebWorker initiated script, as WebWorkers can only be initiated with a single JavaScript file.

To overcome this, a program called Webpack was used. Webpack traces the function dependencies between code modules and merges them all into a single, deployable JavaScript file. An additional benefit of web pack, and a technique used in many production websites, is JavaScript minimization. Minimization is the process of removing all blank spaces within the JavaScript file and reducing its overall file-size.

While it is possible that the created JavaScript code file could be provided website owners to embed directly in their webpage code in a WebWorker, this would limit the client webpage to one specific task. Having the code delivered from the Panther Server does increase the number of server operations required at the initiation of the client connection, but this does allow a client webpage hosting to the framework to process different tasks without any code-changes, fulfilling one of the framework design requirements

This completes the explanation of the Panther Server component.

3.3.3 Component Three: Volunteer Task Code

There are several constraints placed on a project owner who is developing code to execute their task in the web browser via the Panther framework:

- The code must be written in JavaScript
- A entry function must be defined in the following format: function(task)

While the first constraint is imposed the web browser itself, the second constraint is imposed by the Panther framework. A function of this structure must be registered with the framework which will become the initiation point of the task processing. However, once this point is reached, the project owner is free to structure the code as they need, including further function calls, or the inclusion of other modules that have been consolidated into the one file via Webpack. Once the initiation function exits, control return to the Panther API, which will class the task processing as completed.

```
const pi = function(task)
{
    let x_coord, y_coord;

    while(task.current_throw < task.total_throws)
    {
        // Generate random number for x and y coordinates
        x_coord = Math.random();
        y_coord = Math.random();

        if (((x_coord * x_coord) + (y_coord * y_coord)) <= 1.0)
        {
            task.current_hits++;
        }

        task.current_throw++;

        // Checkpoint every 100 million
        if(task.current_throw != 0
            && task.current_throw != task.total_throws
            && task.current_throw % 50000000 == 0) panther.CheckpointTask();
    }

    task.pi_estimate = 4.0 * (task.current_hits/task.total_throws);
};
```

Figure 16: PI Task Function Code

The task variable that the initiation function will receive as its single argument is the task data in its original format and already parsed into a JavaScript object. While Panther task meta-data does exist in the client code, it is not accessible by the task code itself, and the task data encapsulated in the TaskContainer gets passed to the initiation function. This can be seen in **Figure 16**, which shows the Dartboard PI task function.

A certain programming paradigm is required when using the check-pointing functionality of the Panther Framework. As also seen in **Figure 16**, the task can be checkpointed and return to the server in its current state by using the `panther.CheckpointTask()` function call. However, the programmer must ensure that the code execution from the initiation function is sufficient to allow partially completed tasks to resume processing at the correct stage. As seen in the Dartboard PI code, this is achieved by making the loop iteration variable, `current_throw`, part of the task data itself, so on the resumption of processing the while loop begins at the correct iteration.

3.3.4 Component Four: Panther Client API

The Panther Client API is a JavaScript module that task code must import and interface with in order for the task to be run on the Panther framework.

```
const p = require('../tasks_dev/panther');
let panther = new p.PantherXHR();
panther.SessionURL = "http://www.wbpfserv.xyz/task/session";
panther.GetURL = "http://www.wbpfserv.xyz/task/get";
panther.PostURL = "http://www.wbpfserv.xyz/task/post";
panther.ReceiveTaskFunction = pi;
panther.Begin();
```

Figure 17: Mounting Task Component via Panther Client API

Figure 17 shows the process for initiating the client side Panther code, using XMLHttpRequest as the communication protocol with the server.

The first line of code is the import of the Panther module, which also allows Webpack to generate the consolidated JavaScript file for deployment.

The second line instantiates a Panther object which will use the XHR communication protocol to communicate with the Panther Server.

Lines 3-5 are setting the URLs of the Panther Server will used to make requests to the Server – this is a specific action for the PantherXHR object.

Line 6 shows the task processing function being registered with the Panther object. This function, as seen in **Figure 16**, is what will be called by Panther API to process a received task.

The `panther.Begin()` passes control to the Panther object and begins the process of requesting and processing tasks from the Panther Server. The `Begin()` function will also check that appropriate variables, such as the task processing function, have been set before execution begins.

The Panther Client Module is made up of three classes: a base class called Panther, and two child classes, PantherXHR and PantherWS, which inherit functionality from the Panther base class.

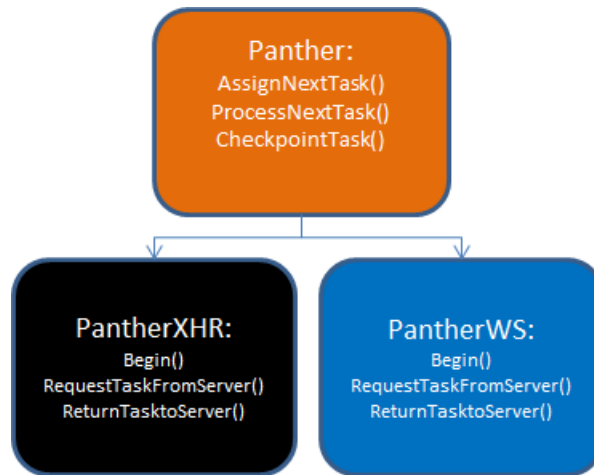


Figure 18: Panther Client API Class Hierarchy

The Panther class contains functionality which is entirely independent from the communication protocol used to contact the Panther Server, such as the task processing loop. The PantherXHR and PantherWS override the communication functions of Panther object and implement their respective communication protocols to contact the server. As seen in the description of the Panther Server, specific modules exist on the Panther server to handle communications under the respective communication protocol.

Task Buffering and Task Request Scheduling

As seen in the Admin Controller webpage, the project owner can set several task scheduling policies to be used by the client web browsers.

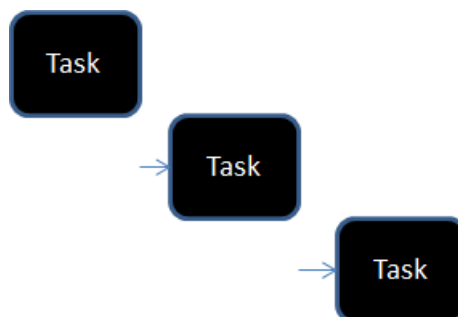


Figure 19: Standard Task Scheduling Policy

The default setting is a standard master-worker approach, where the client requests a single task from the server, processes the task once it has been received, and then requests another task. This can be viewed as synchronous task requests, where the

client will have to wait for a new task to be delivered from the Panther Server between the processing of a task.

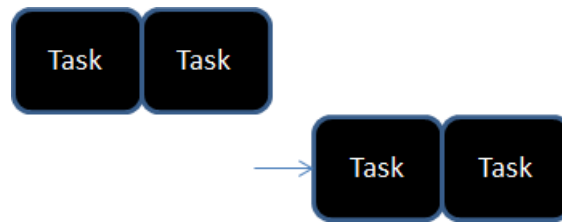


Figure 20: Batches of Tasks

However, the server admin can configure the server to send batches of tasks to the client. On receiving multiple tasks, the Panther object will store the received tasks in a local array buffer. When a new task is required, the buffer will be checked first for available tasks before sending a request to the Panther Server. While this is still synchronous messaging between the client and the server, it does reduce the amount of client downtime waiting on tasks to be received.

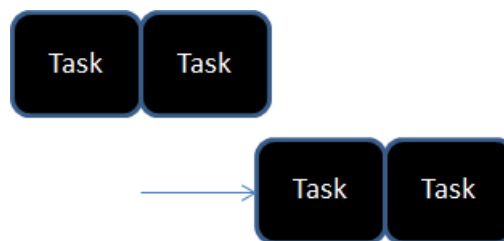


Figure 21: Asynchronous Task Messaging

To introduce the asynchronous message passing between the client and the server, and the ability for the client to process a task while additional tasks are being requested and received from the server in parallel, the server admin can set a threshold for when the next task request should be made depending on the amount of remaining tasks in the local client buffer array.

As seen in Figure 21, combining the two scheduling options could allow the client to request two tasks at a time from the server, and then request another batch when one task still remains for local processing. If the time taken to receive the new batch of tasks is less than the time taken to process the remaining task, potentially all communication downtime could be eliminated. The potential benefit of this process is assessed in Testing and Evaluation chapter.

Task Checkpointing

As seen in the previous description of the Panther Server and Task Code components, the project owner has the ability to checkpoint the current state of the task data, and by doing so return the task to the server, potentially allowing for an entirely new browsing session to continue the processing of that task, or a further checkpoint from the current session to override the latest value.

Since the Checkpoint() function can be called at any point in the project owner's task code, flexibility is provided in the checkpointing strategy. The task could be checkpointed at fixed intervals during a loop, or could be checkpoint at uneven iterations. For example, checkpointing could occur at small intervals at the beginning of a task, and then eventually revert to larger intervals. The project owner can decide.

Under the Panther framework, however, one constraint is imposed on checkpointing: the Checkpoint() function should not be used behind a conditional statement. At each checkpoint, a counter variable called checkpointNumber, which is stored in the TaskContainer that encompasses the task, is incremented by one, and this value is used by the Panther Server to establish if a returned task is further along in processing than the one currently stored on the server. A checkpoint function call behind a conditional statement that does not occur will divert in the checkpointNumber than one where the checkpoint does.

Client Side Performance Monitoring and Meta-Data

The Panther Client Module is responsible for updating many of the meta-data variables contained in the TaskContainer object that encompasses the task data itself, such as the previously discussed checkpointNumber.

The client side Panther object adds its unique Session ID to the sid variable in the TaskContainer before returning it to the Panther Server.

The Client Side Panther module also contains that Timer object. This object is used to record the amount of milliseconds that pass at different stages of the task processing process.

The timeTaken variable contains the amount of time taken for the client to complete the task or the time taken up to the current task checkpoint.

The sessionDowntime variable is how long the session had to wait to receive more tasks when no tasks were available to take from the local task buffer.

This level of detail may likely not be needed in a production deployment of the Panther framework, and having clients update this data potentially opens a window for

malicious tampering of results. But for evaluation purposes, these values are needed to understand the full life cycle of each browsing session – total amount of tasks processed, amount of time waiting for tasks, etc. – and they used by the server-side TaskManager when writing details to the database log on the receipt of each task from a client.

Task Size and Task Compression

As described in the Background Reading section, JSON is the standard serialization format for the JavaScript language, allowing platform independent communication.

Most web server frameworks, including the ones used in this project, use compression techniques on JSON strings before sending them out to web browser clients. As seen in Figure 2, information on compression format of the HTTP body is included as a request header in both the HTTP request and response.

During initial testing of tasks with larger data sizes, tasks were both failing to be returned to the Panther Server, or the Panther server itself was crashing. Investigation showed that when tasks were being returned to the server, the task size had greatly increased in size: in some cases, increasing to over 100 megabytes in size.

When being returned to the server, the JavaScript task object was being converted to a JSON string, and by default this does having a bloating effect on the task size. For example, a 64-bit floating point number, made up of eight bytes, would require one byte per numeric value within the variable in the JSON string format. For smaller numeric values of less than eight characters, space may be saved, but for larger values, more space will be required.

Additionally, while data compression is native for communication from web servers to web browsers, there is no native compression method for returning data from web browsers to the web server. Both of these factors were causing tasks sizes to get so large as to be unmanageable by a high response web server.

To overcome this, additional development time was required to implement manual client side compression of tasks in the web browser, and also decompression of task once it was received by the web server. This was done by importing a module that provided a pure JavaScript implementation of the gzip compression method. One downside of this approach was the increasing of the size of the JavaScript task bundle that is sent to client web browsers. Overall, the need to compress task data before it to the server is likely to have an impact on the overall performance of the task processing. A possible alternative to this is explained in the further work section.

3.3.5 Component Five and Six: Client Webpage and Embedded Panther Code

The Client Webpage component is hosted on the Google cloud platform, and consists of a single web address of www.webpfclient.xyz. When visiting this page, the background task processing will begin. Within the HTML code of this webpage is the third Panther framework component, which is the code that initiates the connection to the Panther Server and begins the task processing process.

```
<script>

    var oReq = new XMLHttpRequest();

    oReq.addEventListener('load', function() {
        var worker = new Worker(window.URL.createObjectURL(new Blob([this.responseText])));
    });

    oReq.open("get", "http://www.wbpfserv.xyz/script");
    oReq.send();

</script>
```

Figure 22: Client Webpage Panther Initiation Code

As seen in Figure 22, a website only has to embed less than ten lines of code within their webpage to activate it with the Panther framework. In line with the design goals, no code changes or redeployment of the client webpage is needed to change the task that is processed. This is all handled by the Panther Server.

An XMLHttpRequest is used to request the task code from the Panther server. This is the JavaScript code bundle containing components 4 and 5, and is handled by the ScriptController module on the Panther Server. Once received, the code module begins processing on a separate thread of execution via the WebWorker API. This ensures that the Panther code does not block any other code related to the webpage's execution, fulfilling another design goal.

3.3.6 End-to-End Viewing Session

A full viewing session consist of a web browser navigating to the client webpage at www.wbpfclient.xyz and then at a future point, navigating away from the URL and stopping all computation associated with it. The below shows the full life cycle of viewing session from the perspective of the Panther framework:

Action Number	Action Description	Action Owner
1	Web Browser Navigates to Client Webpage and receives webpage structure and content	Client Webpage Server
2	XHR Request made to Panther Server for Task Script	Embedded Panther code in Client Webpage
3	Task Script XHR request received and current Task Script returned to requestor	Panther Server (ScriptController)
4	Task Script initiated in WebWorker within the web browser	Embedded Panther code in Client Webpage
5	PantherXHR or PantherWS object instantiated	Task Script
6	Task processing function registered with Panther Object	Task Script
7	Panther object set to take control of execution via Begin()	Task Script
8	Request made to Panther Server for task(s)	PantherXHR or PantherWS Object
9	Request received and next task(s) returned to Requestor	Panther Server (XHR or WS Controller and TaskManager)
10	Tasks received, parsed into JavaScript objects, and added to local work buffer	Panther Object
11	Next task in work buffer passed to task processing function	Panther Object
12	Task processed	Task Script (task processing function)
13	Completed Task is compressed	Panther Object
14	Compressed Task Returned to Panther Server	PantherXHR or PantherWS Object
15	Next task taken from local buffer (go to step 11). If no tasks available (go to step 8).	Panther Object

3.3.7 Framework Running Costs

The Panther Server was deployed on an AWS t2.large instance, which is made up of 2 vCPUs and 8 gigabytes of ram, costing \$0.1056 per hour in the EU (London) region. The MySQL database used for the PantherDB logging was deployed on an AWS db.t2.large instance, costing \$0.153 per hour.

Overall, the running cost of the Panther framework with these specifications is \$188 per month. It is likely with a production deployment of the Panther framework, the advanced logging of server transactions would not be required. Without the use of the MySQL database, the cost would reduce to \$77 per month.

The Panther Server was deployed to a relatively high power virtual instance to ensure any hardware bottlenecks did not produce any adverse results. It is likely with tuning that the cost of running a Panther Server could be reduced even by running a low memory server and ensuring the number of tasks present in the in-memory task cache does not exceed the available memory.

For example, running a t2.medium instance, which has the same processing power as the current instance but uses four gigabytes of ram instead of eight, would reduce the monthly cost to \$40 per month. As the Panther Server does not use the hard disk for task storage, no additional hourly costs must be paid for additional hard disk space.

3.3.8 Framework Development Timescales and Issues

The development of the Panther framework proceeded mostly in line with the planned timescales. However, two additional days were spent on development in order to understand the client side compression issue and implement a solution, which delayed the start of the testing and evaluation.

While the Panther framework can switch between the XMLHttpRequest and WebSockets communication protocols, implementing a TaskManager module that could service both methods has restricted the exploration of some of the server side features of the WebSockets protocol. For example, the WebSocket protocol can be used to establish how many active connections the server has, and also initiate messages to connections. This will be discussed further in the further work chapter.

3.3.9 Summary

In this section, an explanation of the Panther has been given. Overall six components were developed to enable live deployment of the framework at www.wbpfsever.yyx and a client webpage at www.wbpfclient.xyz for testing of the framework at scale and under real world browsing conditions.

4 Testing Strategy

The objectives of the testing and evaluation phase were as follows:

- Assess the time-to-completion of two different distributed computing tasks processed on the deployed Panther framework, where user behaviour on the client webpage aligns with real-world web browsing behaviour in regards to dwell time.
- Understand the performance difference between a traditional distributed computing setup and the Panther Framework.
- Establish if different task sizes and scheduling policies can enable faster completion of tasks under real-world browsing conditions, and assess if these methods are cost-effective from a server utilisation perspective.
- Extrapolate the performance implications of using web browsers for distributed computing at a large scale.

Retrospectively, it took more than a week longer than anticipated to create a test setup that could fulfil the above objectives, and this did reduce the amount of time available for the testing of different task scheduling policies. In this Chapter, an explanation of how the testing setup was created that allowed testing of the Panther framework at scale and in a fair and repeatable manner.

4.1 Test Automation

A key to accurate performance measurement between different test setups is the reproducibility of the exact same testing conditions. While a web browser can be manually opened and used to view the client webpage, beginning the task processing, and multiple browser tabs could be opened to simulate multiple users, several problems arise:

- Once the number of browsing windows exceeds the number of CPU cores on the local computer, the Operating System will begin cycling the threads execution for the browsing windows between the available CPU cores. Task processing would not truly be running in parallel as a single CPU can only scale so far in regards to parallel thread execution.
- A human cannot reliably reproduce the opening and closing of multiple browsing windows at the set time intervals, which would need to be done in effort reproduce multiple different users navigating to and leaving the client webpage. Again, if this was attempted, a human could only manually manage a small amount of browser windows, and even if multiple humans were used, the reliability would still be factor.

To mitigate these issues and to operate the testing at scale, two things need to occur for the test setup:

- i) The testing needed to be able to run in a distributed fashion: where the number of browsing sessions does not exceed the number of CPU cores available on each machine.
- ii) The opening of closing of browsing windows at set time intervals must be fully automated to ensure the exact same browsing behaviour is repeated.

How this was achieved is described in the upcoming sections.

4.3.4 User Browsing Behaviour Data

To key metric for distributed computing via web browsers is user dwell time: which is the time the user spent viewing a specific URL address before navigating to a new URL address or closing the web browser. In regards to distributed computing via web browsers and the master-volatile-worker pattern, a single user dwell time observation can be seen as the total time of availability of single worker before the immediate termination of that worker.

As discussed in the project preparation phase, the initial aim was to obtain real-world user dwell time transactional data from published server logs. However, when reviewing previous academic papers and searching web resources, no sufficient existing data sets containing the dwell time metric could be found.

Many webpage analytics providers, such as Google Analytics, provide summary statistics for webpages, including dwell time, but they are aggregates and averages, and not suitable to understand the individual dwell time transactions that overall form real-world browsing habits.

The Machine Learning Repository had two data-sets of interest. On review, on however, the data-sets were found to be dated from over a decade ago. With the introduction of ever increasing speeds of consumer internet connections via broadband and fibre-optic cables, new methods of web browsing via mobile devices, and the move to single-page web applications, it is likely that user browsing habits in regards to webpage dwell time have changed since then.

Additionally, both data-sets were made up of basic transactional HTTP request data. The dwell time attribute is not a native field for HTTP transactions, and in many instances analytics providers only provide an estimate for this metric. After a web-page is served to a client after a HTTP request, the server has no knowledge of how long the client stays on that page unless additional tracking communications are made between the client and server.

To calculate dwell-time from the available data-sets, different HTTP requests from the each unique browsing session would need to be chained together to calculate the time between each request. Both data-sets were large in size, and transforming the data into required format would take too long within the given project timescales.

In light of the above, the decision was made to use the findings of the previously discussed *Understanding Web Browsing Behaviour through Weibull Analysis of Dwell Time* and generate randomized test data that matches various Weibull probability distributions.

As previously discussed, the main finding of the research paper was that holistic dwell time of webpage visitors matches Weibull distributions with scale parameters of less than zero. When scale parameter of the Weibull function is less than zero, a probability distribution occurs with a positive aging effect: the longer the live span of the observed entity, the higher probability of the entity continues to remain alive.

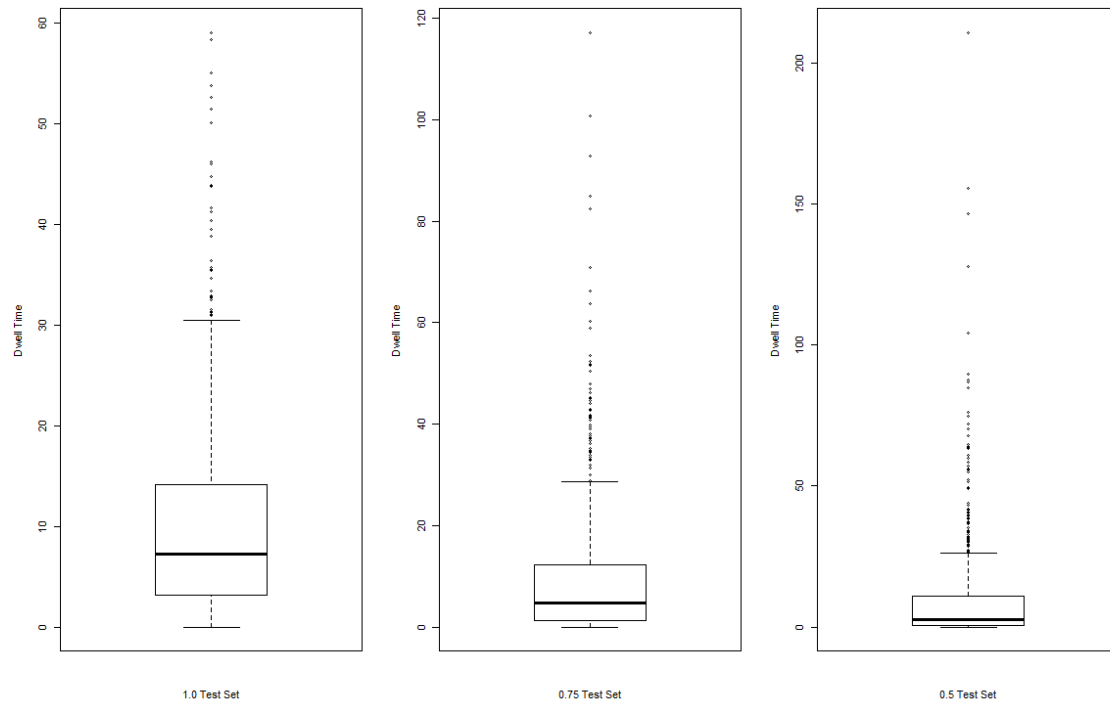


Figure 23: Boxplot of real world browsing habits test sets

In regards to user dwell time on webpages, the Weibull distribution can be viewed as majority of visitors to a webpage leaving it in a short period of time (due to the short dwell time leading to a lower probability of remaining any longer), and a few visitors staying for a much longer period (due to the probability of them remaining on the page increasing as their dwell time increases and increases).

The R statistical software package was used to generate three sets of randomized page view times that matched the Weibull distribution with three different shape parameters: 1.0, 0.75, and 0.5. The distribution of the view times can be seen in **Figure 23**.

Each set was made up of 720 separate view times, which during testing would translate into 720 separate visits to the client webpage at www.wbpfclient.xyz. The total view time in each set amounted to approximately 107 minutes.

4.3.5 Website Testing Tools

To implement the viewing of the client webpage across multiple web browsers in parallel, existing website testing tools were reviewed for suitability. Many websites offer testing tools with the ability to script automated test cases. Website testing falls into two broad categories:

- i) **Load testing:** where numerous HTTP requests are made to a web server over a sustained period in parallel to assess the performance of the web server under extreme load.
- ii) **Functional testing:** where a specific functional requirements of the webpage, such as a user login process, are tested by following a fixed set of interaction steps with the webpage.

A key requirement for the testing of client webpage and the Panther framework was visiting the client webpage in multiple web browsers, as it is the web browser itself that executes the JavaScript code that has been provided in the message body of the HTTP message. With load testing providers, web browsers are not used and HTTP messages are sent and received via specialist programs. With functional testing, only one web browser is used to confirm the required functionality is present.

This placed the testing of Panther framework between the two categories, requiring a mixture of load testing (multiple visits in parallel) and functional testing (interaction with the web browser), which made all reviewed testing product unsuitable.

Additionally, the hardware availability and implantation is not known for the off-the-shelf testing tools, so if implemented in a cloud environment, it is possible a different amount of processing power could be provided for each testing run. Not knowing the in-use hardware will also make it difficult to make a comparison between the performance differences between traditional distributed computing and distributed computing via web browsers.

In context of the above, it was decided to create a custom test setup that allow full control over hardware usage and work distribution.

4.3.6 Test Setup

The final test setup consisting of six virtual machines hosted on the Google Cloud platform. Each virtual machine contained 4 CPUs, which allowed 24 web browsers to be run in parallel. The test setup could not only be used for testing of the Panther framework, but also allow distribution of any executable file to run in parallel, and the ability for all 24 processes to communicate with each other.

24 CPUs was the maximum amount of CPUs that could be instantiated on the Google Cloud platform with a standard account. Attempts were made to use the computing facilities provided by Edinburgh University, however the back-end compute nodes on Cirrus and Archer are unable to send external internet messages, so HTTPs request to the client webpage cannot be made.

In the following sections, a description of the key components that form the test setup are described, follow by a final summary.

4.3.6.1 Virtual Machine Specification

To achieve 24 available CPUs, six virtual machines were instantiated. All six virtual machines were an exact replica of each other, being instantiated from an instance template which has been developed to house all the required programs and data. Each machine had 15 gigabytes of RAM and four CPUS, each CPU being an Intel Xeon @ 2.20 GHz.

4.3.6.2 Selenium

Selenium is a code library designed for the automated testing of webpages. It has bindings for multiple languages, including JavaScript.

```
function PageView(instance, driver, viewTime)
{
    let startTime = new Date();

    driver.get('http://www.wbpfcclient.xyz');

    console.log("Instance " + instance + ": Page Opened...");

    driver.sleep(viewTime).then(function() {

        let endTime = new Date();
        endTime = (endTime - startTime) / 1000;

        console.log("Instance " + instance + ": Page Closed (" + endTime + "s)...");

        InitNextView(instance, driver);

    });
}
```

Figure 24: Selenium Code Completing One Viewing Session

Selenium allows full control over the web browser via the functions provided in its API. **Figure 22** shows the Selenium code used to automate the process of navigating to the client webpage and then waiting for a set amount of time before beginning the process of navigating away. While most of the features of Selenium are not required, it was used to automate the browser on each CPU in order for it to open and close the client webpage, following dwell times that were generated for each test case.

4.3.6.3 Node.js

While Node.js is mostly known for its web server capabilities, it also can be used as a local process to run JavaScript code that can access the local filesystem and other operating system fulfilled commands. Node.js was installed on the instance template and one Node.js process was used to run the Selenium testing code on each CPU core.

4.3.6.4 Headless Firefox and GeckoDriver

Web browsers are graphical applications and normally cannot be run from a command line interface, which do not have advanced graphical capabilities. This could cause issues with running web browsers on the cloud instances which are interfaced with via the command line. However, in response for the need automated testing of web applications, most modern web browsers allow them to be run in headless mode, where the full processing of the web browser takes place, including JavaScript execution, but no graphical GUI is displayed.

Firefox was used with Selenium to allow full automated control over the browser in headless mode. GeckoDriver is an executable file created by Mozilla, the creator of Firefox, which serves as an interface between the Selenium API and the Firefox browser.

4.3.6.5 MPI and Data Splitting

The Message Passing Interface (MPI) is a framework used for distributed computing. It allows executable files to be initiated in unison on a group of networked PCs, and allows communication between the running processes. For the testing, one virtual machine was logged into via SSH, where MPI was used to initiate the Node.js processes, running the Selenium testing code, on all 24 CPU cores at once.

One challenge posed by doing the testing distributed manner was ensuring that each CPU conducted a unique section of the 720 testing sessions. The full testing datasets were present on each machine, and Node process had to take 30 of these session timings without overlapping with any other processes to fully decompose the file across the testing instances.

When running under MPI, each process is given a unique rank number. MPI has library binding for C, Fortran, and other languages, and these bindings can be used in the

program code to establish each processes unique rank via a function call, and use this unique rank to decompose the test data between instances.

JavaScript does not have any MPI language bindings, so therefore the MPI rank of each process cannot be establish via a function call. However, this was overcome by extracting the MPI rank from the operating system's environment variables.

Using the MPI rank, each Selenium test process was able to parse through the test data in csv format and extract an even chunk of the data. For example, rank 0 would take the first 30 csv rows, and rank 1 would take the following 30, and so on.

4.3.6.6 Browser Caching

By default, browsers attempt to locally cache resources they come receive in order for them to be delivered quicker on the next visit to a webpage. In the case of the test setup, since the single Firefox web browser operating on one of the CPUs would simulate multiple unique visits to the client webpage, any caching of resources on the first visit, such as the JavaScript task code delivered from the Panther server, would optimize the following visits. To ensure this would not occur, JavaScript and HTML code was updated to instruct the browser not to cache any of the resources that were received on each visit.

4.3.6.7 C Programs

To ascertain the performance differences between running the task through interpreted JavaScript in a web browser and compiled machine-code, C version were of both tasks were created. These were distributed across the CPU cores by also using MPI.

4.3.7 Final Test Architecture

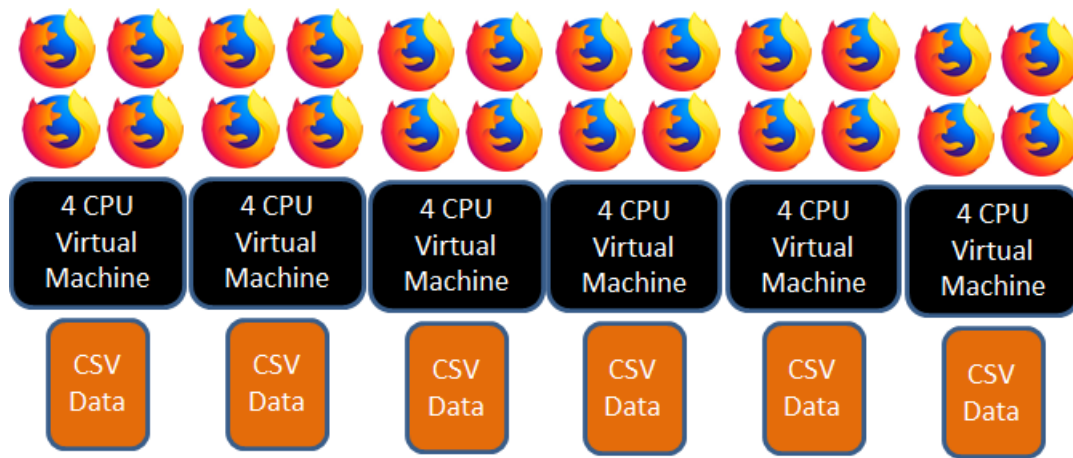


Figure 25: Test Setup

Figure 25 shows the final test architecture, which allowed 24 web browsers to visit the client webpage in parallel, following real world browsing habits, with each test case being able to be repeated as required.

```
mpirun --mca plm_rsh_no_tree_spawn 1 --bynode -hostfile hostfile -np 24 node test.js 75 30
```

The above command line shows the initiation process for a single test run. This would begin the testing for the 0.75 shape dataset, with 24 processors, and each processor completed 30 sessions from the test data set.

5 Testing and Evaluation

5.1 Dartboard PI Task

Dartboard PI is a method of estimated PI through the generation of random numbers. During an iteration of the method, two random numbers between 0 and 1 are generated. These number represent X and Y coordinates on a 1 by 1 square grid. After throwing a dart, it is assessed if the dart landing with a central circle within the square, representing the dartboard, or outside of the circle. After multiple dart throws, the resulting total of hits and misses of the dartboard can be used to estimate the value of PI.

Key feature of this method in regards to processing are:

- The processing time can be increased or decreased via increasing or decreasing the number of iterations, or throws, used to estimate PI.
- The data-size is fixed: changing the number of iterations does not change the size of the task data.
- The computation can be decomposed into different tasks. Two tasks could complete separate iterations and then the resulting total hits and misses from both tasks could be used to estimate on PI value.

For the testing, the test data sets with 720 viewing sessions each were distributed between each of the 24 CPU cores, meaning each core and its attached web browser would be responsible for simulating 30 of the webpage visits. Due to the differences in the individual view times, this means the total view time of each browser is not evenly distributed, and some cores may finish earlier than others. Overall, for each of the data-sets that simulate real world browsing habits, each test run would result in 720 unique visits to the client webpage over a period of 4-5 minutes, with 107 minutes of total viewing time.

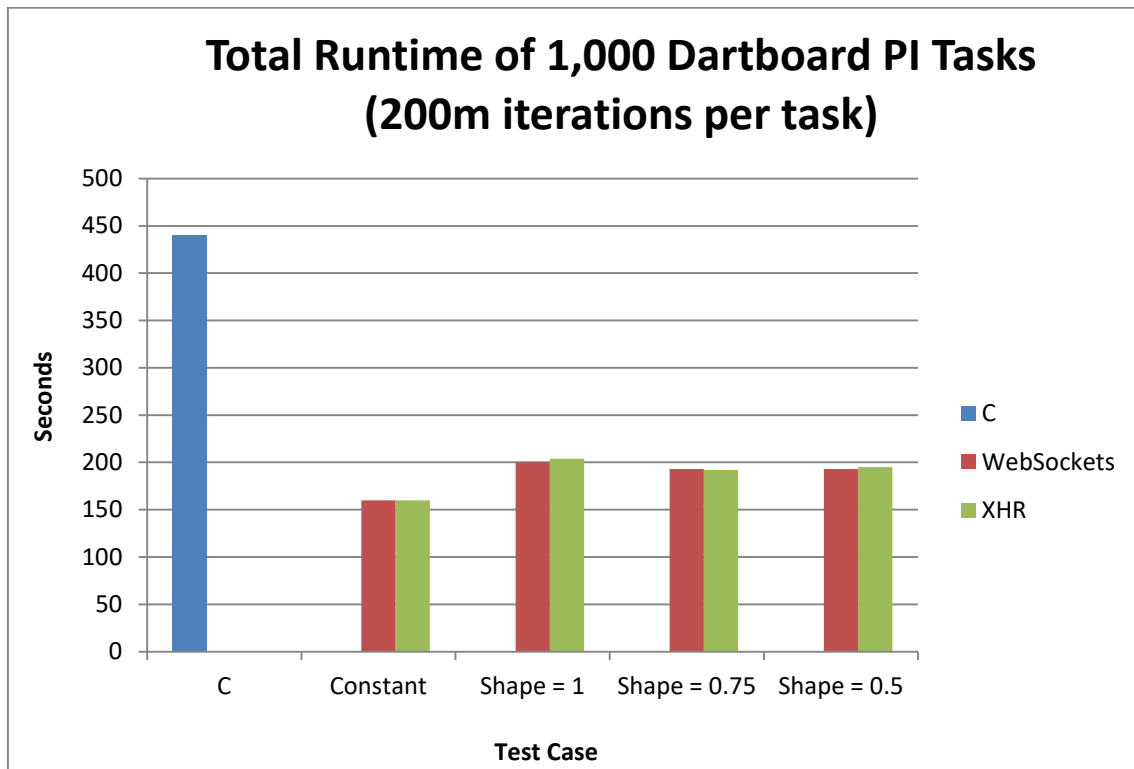


Figure 26: Total Runtime of 1,000 Dartboard PI Tasks of 200m Iterations

The first test was establishing the total runtime taken to complete 1,000 Dartboard PI tasks when the number of dart throws, or iterations, set to 200 million. The results are shown in **Figure 26**. Standard task scheduling was used, where a web browser client would process one task in full before requesting another task from the Panther Server.

The first result, against expectation, is that the C version of the Dartboard PI task has a longer runtime than the all test cases that were run on the Panther framework. The C program has been compiled with all compiler optimization options enabled, but runs 260 seconds slower than when the tasks are complete via constant webpage viewing via Panther.

The Dartboard PI task is heavily reliant on random number generation. During each iteration of the task, two random numbers are generated to establish the X and Y coordinates of the thrown dart. It is likely that the standard library C random number generator function is slower than the standard JavaScript version. Additionally, extra operations are required to get a result between 0 and 1 for the C program, while the JavaScript random by default provides a value in this range. This suggests that while the consensus is that machine-code compiled programs will outperform interpreted programs, such as JavaScript, it should not be treated as a given. Overall, specific function calls and operations between the languages should be considered in terms of their impact on performance.

The results show that there is no large difference in performance between the XMLHttpRequest and WebSocket protocols. When the constant web page viewing test cased is used, both protocols took 160 seconds to complete the required computation. For the test cases using real world browsing habits, there was at maximum a 1% difference in performance between the protocols. No communication protocol is the fastest processor through all three test cases, with WebSockets being marginally faster at shape 1 and 0.5, and XMLHttpRequest being marginally faster at shape 0.75. This suggests that the dynamic routing and latency of HTTP requests can explain the performance differences at this stage.

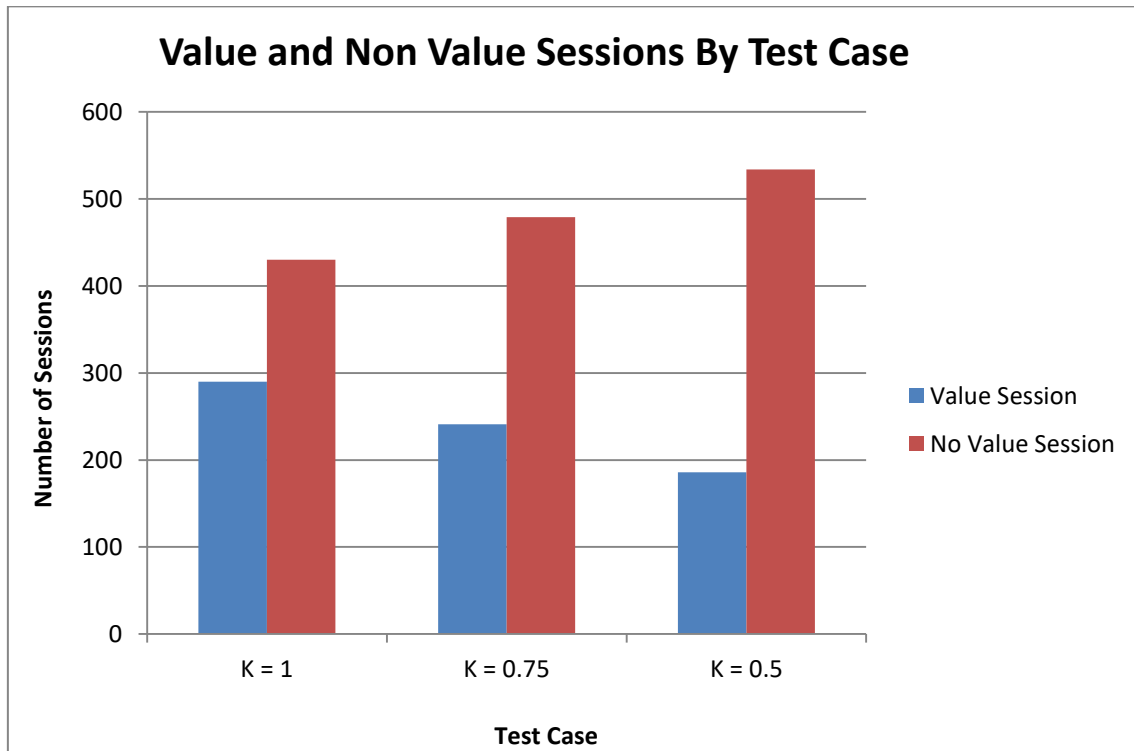


Figure 27: Value and Non Value Sessions

Under the simulated browsing habits test cases, the fastest completion time measured was 192 seconds, and this was produced XMLHttpRequest protocol at a shape of 0.75. Only marginal performance differences occurred between shapes 0.75 and 0.5. However, both these distributions performed noticeably faster than the 1.0 distribution, which is an unexpected result.

Figure 27 shows a comparison of the value session and non-value viewing sessions for the simulated test cases. A value session is where the web browser has navigated to the client webpage and completed and returned at least one task before navigating away from the page. A non-value is where the web browser has navigated to the client webpage but not completed any tasks before navigating away from the page. For the non-value sessions, the server is committing resources, such as sending out the JavaScript code bundle and the task data, but getting no return for it.

For the simulated test cases, the best case value-session result is only 40% of the total sessions. As expected with the Weibull distribution, the numbers of value-session decreases with the lowering of the shape value. The processing of the 0.75 and 0.5 test sets results in total value sessions of 33% and 25%. Overall, under all test cases, the majority of visits to the client webpage do not complete and return a single task, and each one of these sessions could cost to the server up to three operations without any return. Using the log data, the average processing time of the Dartboard PI Task at this size is 3.75 seconds, suggesting that the majority of sessions are not remaining active long enough to complete the initiation process with the Panther server and then process a task of this size.

Reviewing the runtime results in Figure 26 against the total value-session results in Figure 27 shows an unexpected result. The test scenario of shape 1.0 has the highest amount of value-sessions during the test run, but produced the slowest runtime compared to the other real-world scenarios with a lower amount of value-sessions.

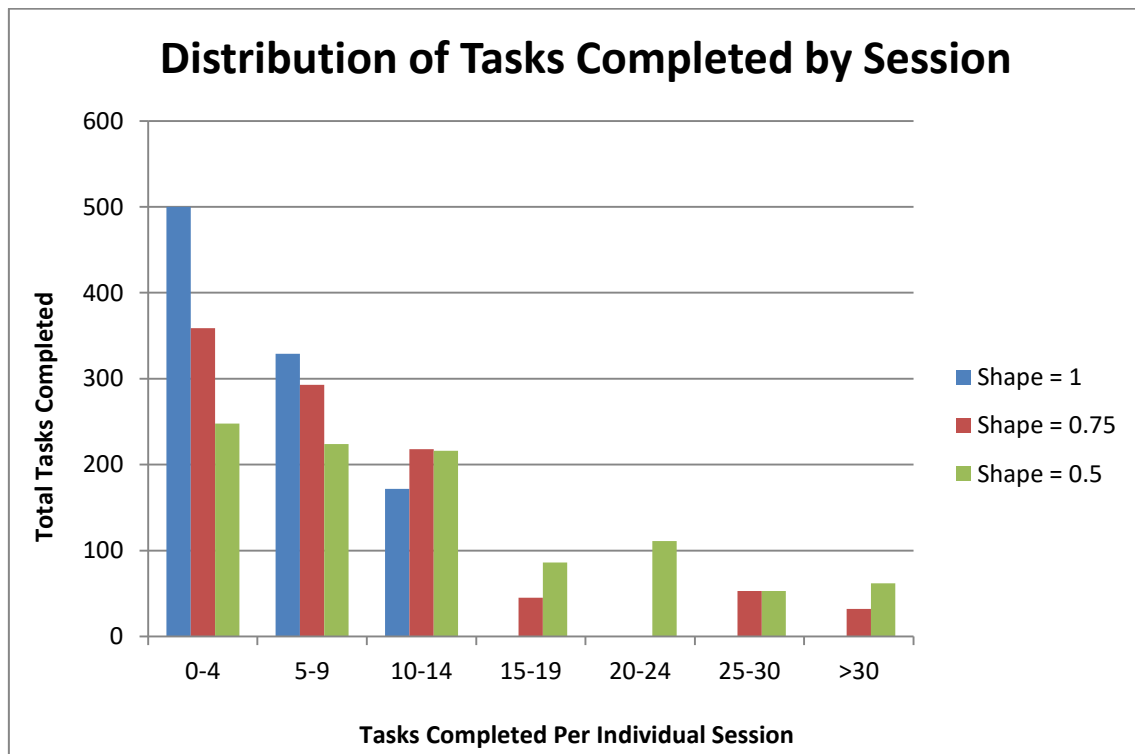


Figure 28: Tasks completed per individual session

Figure 28 shows the distribution of task completion between sessions by the overall amount of tasks completed by each session. As can be see, for the shape = 1 scenario, no sessions completed 15 or more tasks. However with shape values of 0.75 and 0.5, some single sessions could complete up to 30 or more tasks. In the case of 0.75, one single session completed 64 tasks, which is 6.4% of the total task population.

The 0.75 and 0.5 simulation sets completed the processing faster due to the availability of longer sessions, where a large amount of uninterrupted task processing could take place. For shape = 0.75, the fastest of the run times, 64% of the task processing was completed by only 28% of the total value sessions, and each of these sessions completed at least 5 or more tasks, so based on the average task completion time, these sessions had a lifetime of at least 23 seconds. 36% of the task processing was completed by 72% of the value sessions, and these sessions completed 4 or less tasks. In regards to all sessions, whether they provided any value or not, this means 64% of the task processing was completed by only 9.3% of the sessions that visited the client webpage.

For the simulated real-world tests, the total downtime across all sessions ranged from 7.2 to 9.9 seconds, with no observable differences between the WebSockets and XMLHttpRequest communication protocols. The downtime is the amount of time a session was idle waiting for the Panther server to send a task for processing with no other locally buffered tasks available for the session to process. The total processing time across all sessions ranged from 3,211 to 3,224 seconds. The processing time is the amount of time the session spent actively processing tasks from start to completion. In light of this, only 0.23% of potential processing time is being lost on sessions waiting for tasks from the Panther Server.

Overall, there is very little scope to increase performance by reducing this latency. In regards to reduce the processing time of the same amount processing work, the next step is to try and convert non-value sessions into value sessions.

Improving Performance via Reducing Task Processing Time

To initially investigate if reducing the processing time of each task helped increase the amount of value sessions, and therefore reduce the overall processing time, the Dartboard PI task was tested on the shape = 1 test set, the real world browsing simulation with the longest runtime in the previous section, with smaller and smaller tasks size.

In order to make the comparison fair, as the task size was reduced, the number of overall tasks requiring completing was increased. The same amount of processing was being completed, but the amount of tasks to do it varied. For example, 1000 tasks were completed when 200 million iterations per task, and when the task size was reduced to by half to 100 million iterations, the task amount was doubled to 2000.

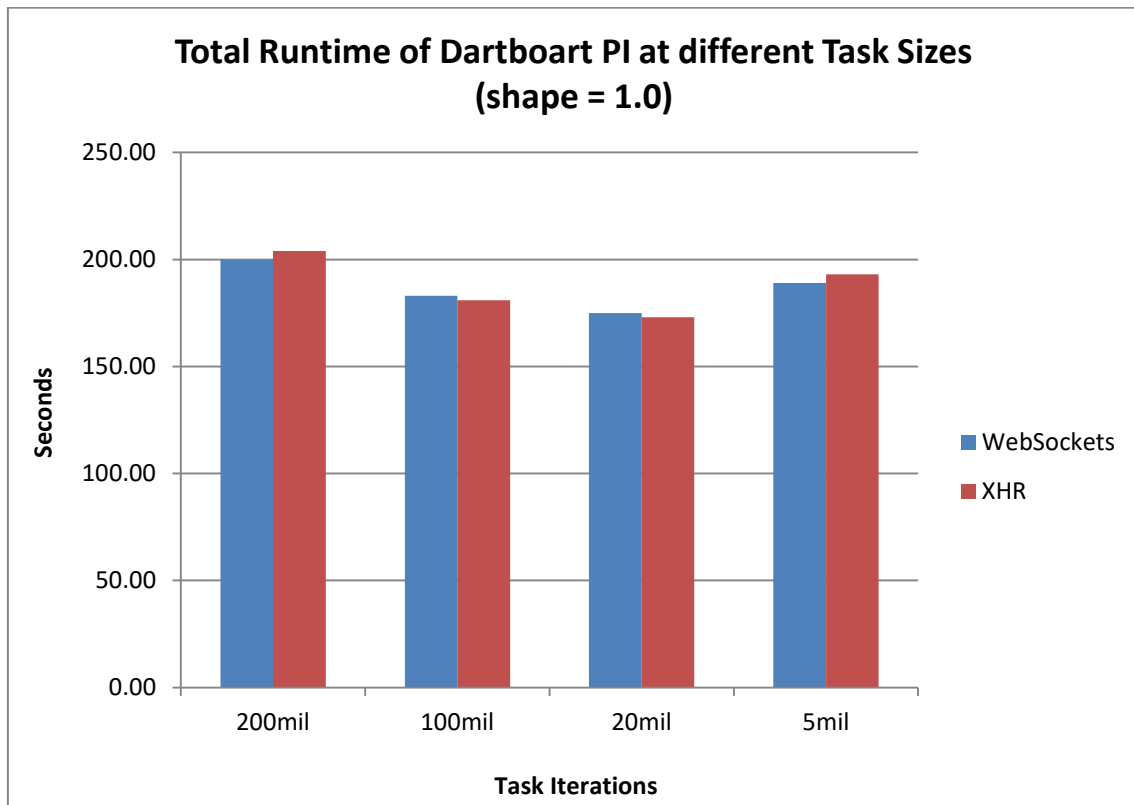


Figure 29: Runtime of different Dartboard PI Task Processing Times

Figure 29 shows the result of shrinking the task size on the total runtime of the task processing. The results show that shrinking the task size has initially had a positive result on the reducing the total runtime of the task processing.

When initially halving the task size to 100 million iterations, and making the average task processing time 1.87 seconds, the overall task run-time was reduced by 11.27% compared to the run-time for the task size at 200 million iterations.

When reducing the task size even further, to 20 million iterations, and making the average task processing time 0.38 seconds, the overall task run-time was reduced by 15.19% compared to the run-time for task size at 200 million iterations.

However, at the smallest tested task size, of 5 million iterations, where the average task processing time was 0.09 seconds, the run-time speed up began regressing. At this size, a task processing speed up of only 5.39% was achieved compared to the run-time for task size at 200 million iterations.

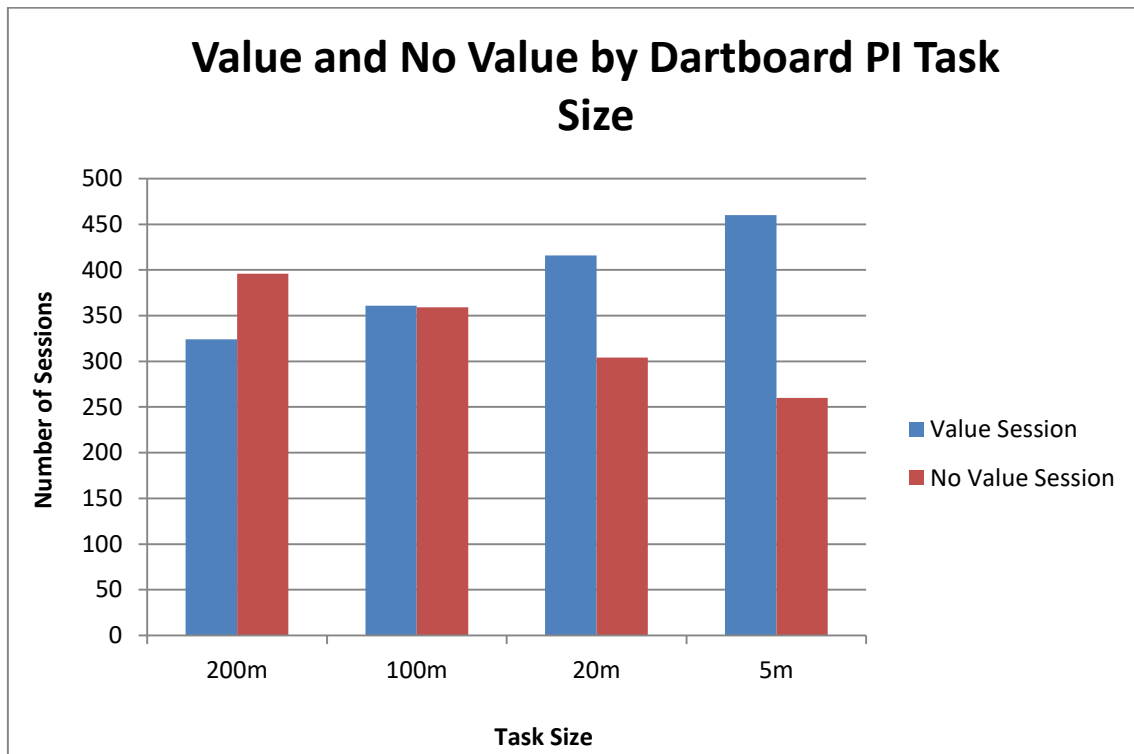


Figure 30: Value and Non Value Sessions

Figure 30 shows the total value and non-sessions achieved via shrinking the task size. It can be seen that tasks that can be processed quicker can convert non-value sessions into value sessions. In the case of task sizes at 20 million and 5 million iterations, more value sessions are now achieved than non-value sessions.

However, when comparing **Figure 29** and **Figure 30**, it can be seen that the smallest task size, 5 million iterations, while achieving the most value sessions, has provided a slower runtime than the two larger task sizes of 20 million and 100 million iterations. This raises the question of why the result with the highest number of value sessions has not translated into the fastest runtime.

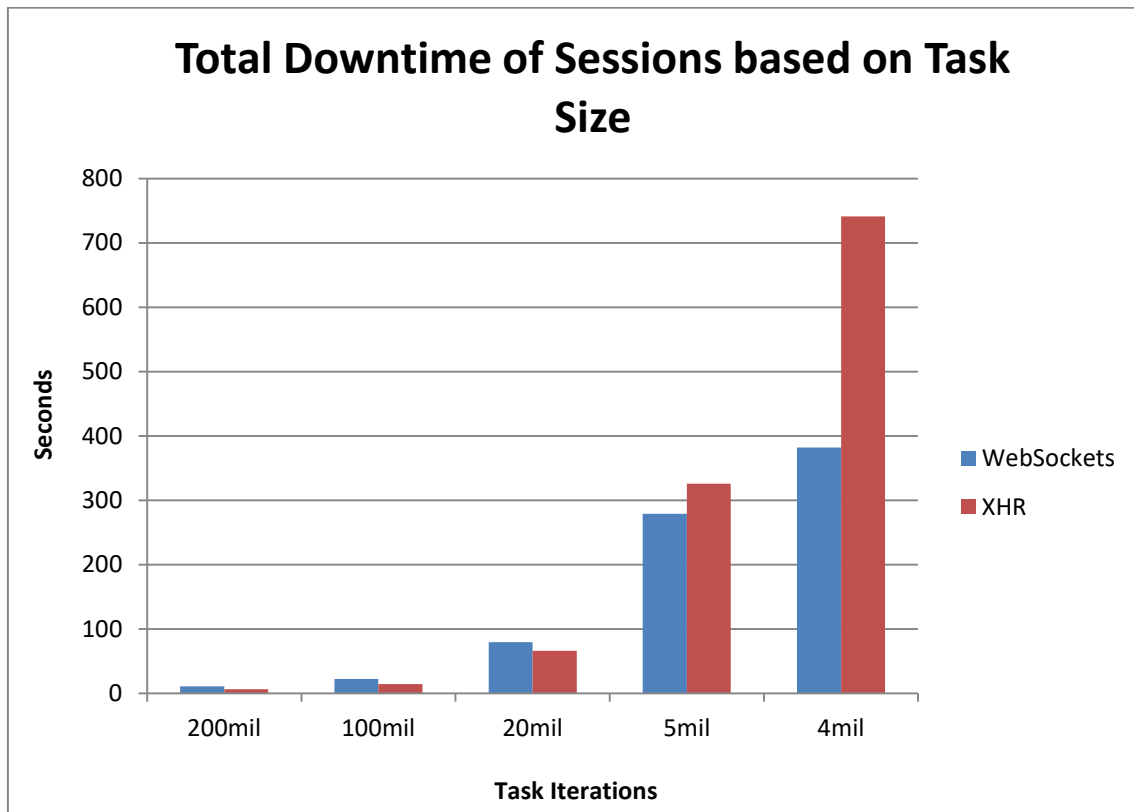


Figure 31: Total Downtime based on Task Size

Figure 31 shows the total downtime across all sessions for the testing of different size tasks. The results show that as the task size decreases, the downtime, which is sessions waiting for work to be delivered by the Panther Server, increases. The fine graining of the tasks had increased the message passing overhead as more messages, in form of smaller tasks, need to be communicated between the Panther Server and clients.

The result for task sizes at 5 million iterations started to show a diversion between the WebSocket and XMLHttpRequest protocols, with the WebSocket protocol having 46.66 less seconds downtime than XHR. The lower meta-data overhead of WebSocket communications could potentially begin to have a performance benefit when faster message passing occurs due to lower task sizes.

To test if this was the case, several more test for even smaller task sizes took place. As seen in **Figure 31**, at a task size of 4 million iterations, the difference in downtime between the two protocols widened, with WebSockets having 64.93% less downtime than the XMLHttpRequest protocol.

A test at 2 million iterations per task also took place. At this level of fine-graining, the actual task workload failed to complete in the provided test timescale, with approximately 50% of the required 50,000 tasks being completed during the 720 sessions. This shows that the performance loss due to task fine-graining and the increasing message passing overhead will eventually result in extreme performance loss.

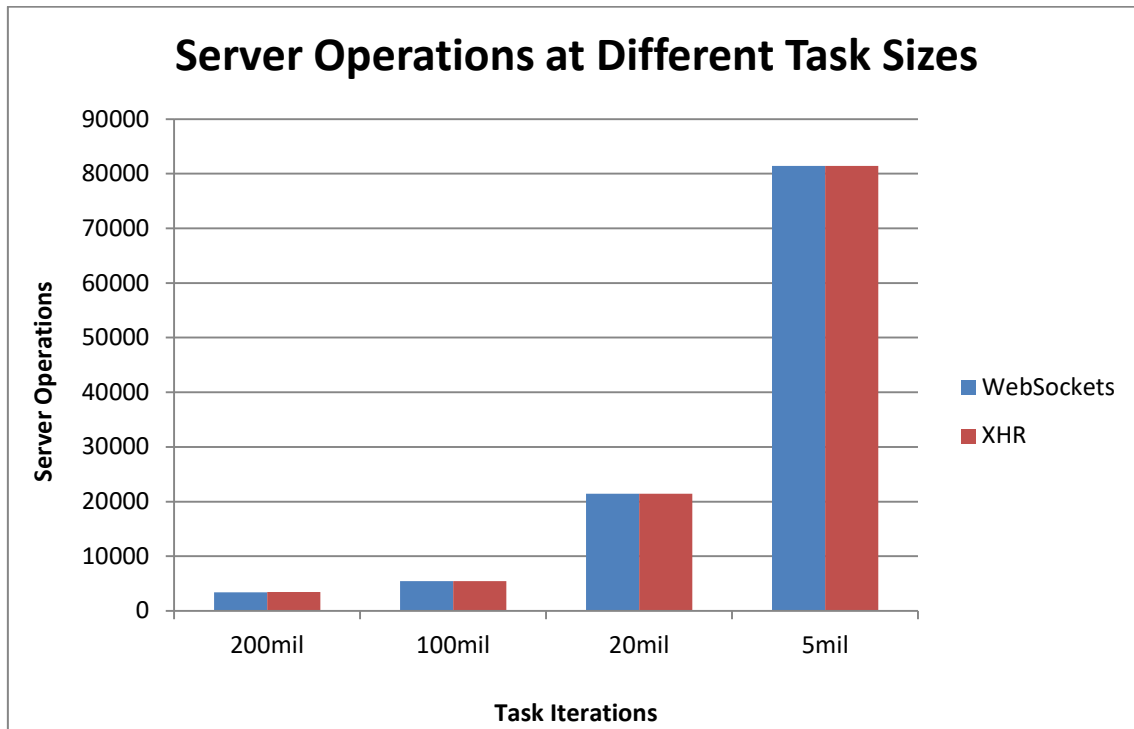


Figure 32: Server Operations Based on Task Size

While a Dartboard PI at a size of 20 million iterations provided the fastest runtime for the completion of the required workload, other factors also need to be considered.

Figure 32 shows the amount of server operations that were required during the task processing at different task sizes. A server operation is any HTTP or communication action needed to be undertaken by the Panther server to handle a client request, such the initial assigning of a Session Id, the sending of a task, or the receiving of task.

While the reduction of the task size from 200m iterations to 20 million iterations decreased the total runtime by 15.19%, the amount of server operations required increased by a factor of five. This aligns with the fact that a task size of 20 million iterations is five times smaller than a task size of 200 million iterations. Overall, the performance increase gained by fine-graining of tasks is much lower than the linear increase in server utilisation by the increased message passing for smaller tasks.

Task Scheduling Policies

The next phase of the evaluation focused on if new task scheduling policies could allow the continued fine graining of tasks, therefore increasing the value sessions, but also offset the increase in client downtime and server utilisation that was mitigating continued performance gains.

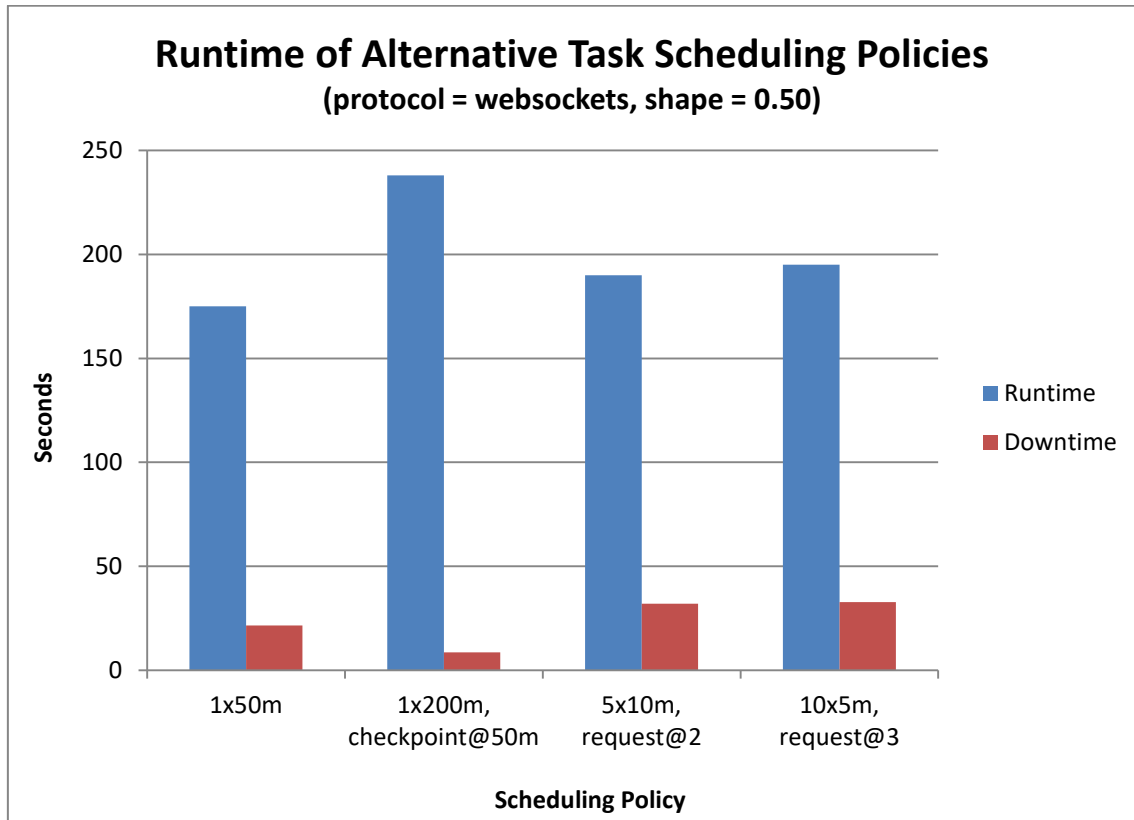


Figure 33: Total Runtime of Dartboard PI Task by Scheduling Policy

A task size of 50 million iterations became the focus point for evaluating alternative task scheduling policies. Three alternative task scheduling policies were testing on the Websockets protocol on the 0.50 dataset. The three policies were tested against a standard task scheduling approach, which was used in the previous testing: the sending of one task at a time to client, each task containing 50 million iterations, and the client only requesting after the completion of the current task.

Figure 33 shows the results of the new scheduling policies against the standard approach. The results showed that none of the alternative policies provided a slower runtime than the standard approach as seen in the first column of the chart.

The first of the alternative policies, shown in the second column of Figure 33, used the inbuilt task checkpointing functionality of the Panther framework. A task size of 200 million iterations was sent to the client, and at each 50 million iterations the task was checkpointed and returned to the server in its current state. This produced the longest runtime, 30.5% slower than the standard scheduling policy. This is an unexpectedly poor result and does suggest the current implementation of the checkpointing procedure has introduced an inefficient overhead. One potential explanation is that the task may need to be duplicated within memory to allow the asynchronous sending of it back to the server, while also allowing a separate copy of the task to be continued to be processed in the client.

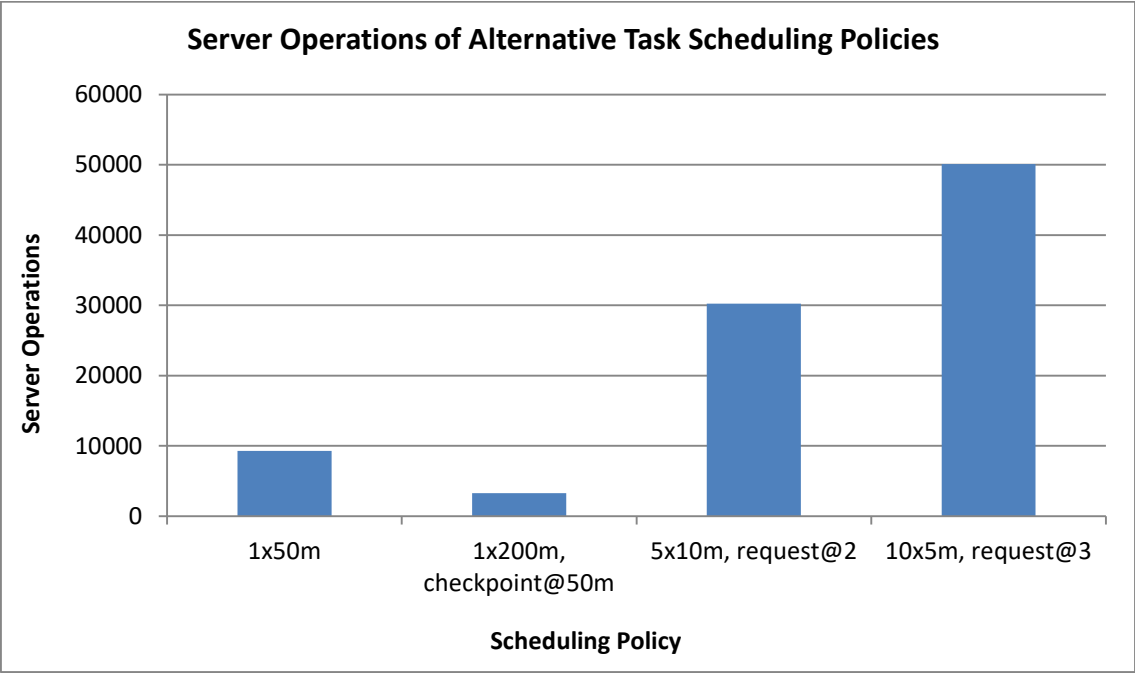


Figure 34: Total Servers of Dartboard PI Task by Scheduling Policy

However, this policy did achieve several other aims. As shown in Figure 34, the required server operations needed to process the required work load was the lowest of all the policies. The server only having to send out one task and its associated meta-data to the get a more fine-grained return and this reduced the overall number of operations required. Additionally, the overall downtime of the sessions was the lowest recorded – again explained by the reduced amount of messaging passing required.

The two other schedules related to the bundling of the multiple tasks and the requesting of additional tasks before all local task processing was complete. For the first policy, a group of five tasks of 10 million iterations in size were sent to clients at each task request. When the client has two local tasks remaining, another set of five tasks would be requested. For the second policy, a group of ten tasks of 5 million iterations in size were sent to clients at each task request. When the client has three local tasks remaining, another set of ten tasks would be requested.

The aim of these policies was enabled continued fine-graining of task sizes while reducing the message passing overhead and downtime, aiming to achieve this by grouping tasks when sent out from the server in order to reduce the overall number of messages passed. However, tasks would be returned in fine-grained sizes, allowing potential utilisation of short browsing sessions.

In all areas, these two policies performed worse than the standard message passing policy. As seen in **Figures 33** and **34**, a much higher cost in server operations has provided a slower runtime and also a higher downtime.

Reviewing these results as a whole, the overall conclusion is that the more attempts that are made shrink task time in order to gain more value sessions from shorter page viewing times, the more overhead from message passing occurs, and this overhead has an impact on all viewing sessions. The previously identified longer viewing sessions that are doing the bulk of the task processing will now doing less processing due to having to send and receive more messages during their lifespan, and the gain of value sessions from the smaller viewing sessions is not enough to make up for this loss.

This suggests that the task scheduling policy suggesting in the *Gray Computing* paper may not actually provide an increased return of computational processing when used in a live environment. Under the increasing task size policy, the longer viewing sessions will have to complete more messaging with the server to accommodate the smaller viewing sessions: again, potentially mitigating any gains made.

Additionally, while efforts can be made to reduce session downtime by pre-requesting work or decreasing the amount of message passing, only very few sessions will gain any benefit from this action. A session must save enough downtime to process another task in full to provide any extra value. Trying to reduce task processing times to do this is counterproductive as the act itself will increase the session downtime.

5.2 Mandelbrot Image Task Description

The Mandelbrot set is a group of complex numbers that continue remain bounded below a threshold value after repeated iterations of the same function.

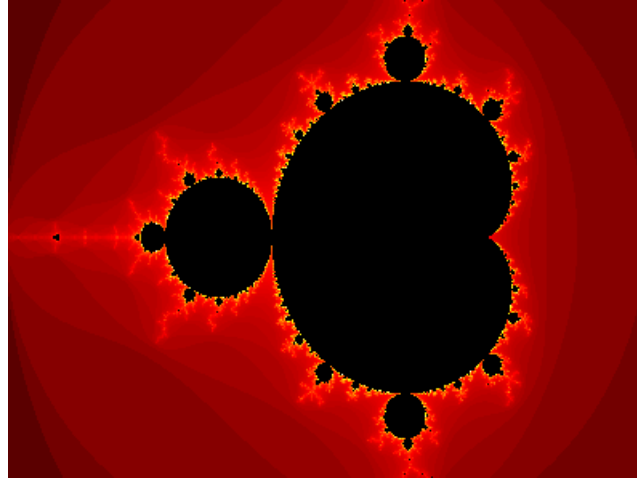


Figure 35: Mandelbrot Image created via Panther framework

The number of iterations that took place before the value either escaped the threshold value can be used to generate Mandelbrot images. **Figure 35** shows 400x300 pixel Mandelbrot image that was generated on the Panther framework. Black pixels indicate values that did not escape the threshold value after the desired numbers that took place, while the colour of the non-black pixels is determined by the number of iterations that took place before the value escaped the threshold.

The key feature of this task are:

- The task data size scales with the size of the Mandelbrot image being calculated. In the case of the Mandelbrot image created by this task, each new pixel added to the calculation require three extra numeric values to store the RGB colour values of the image.
- A Mandelbrot image can be decomposed into smaller tasks by splitting sections of the overall image into separate, smaller tasks. After processing, these tasks would then need to be merged together to create the overall image.
- The processing time can be modified by increased or decreasing the number of iterations that take place on assessing if a complex number has escaped the threshold. However, the impact on overall processing time is unpredictable, as the number of iterations per complex number cannot be predicted as a value may escape the threshold value at any point.

The key feature of interest for the Mandelbrot image task was the effect of data size on the message passing between the Panther server and worker web browsers.

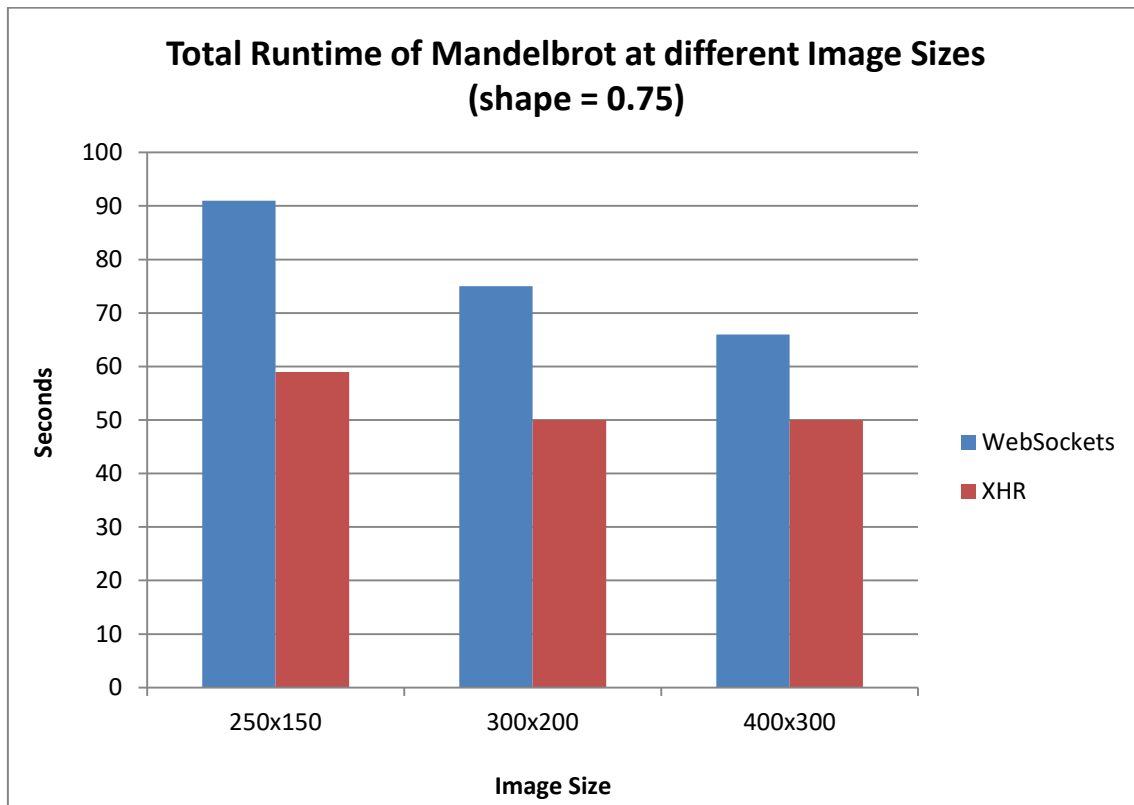


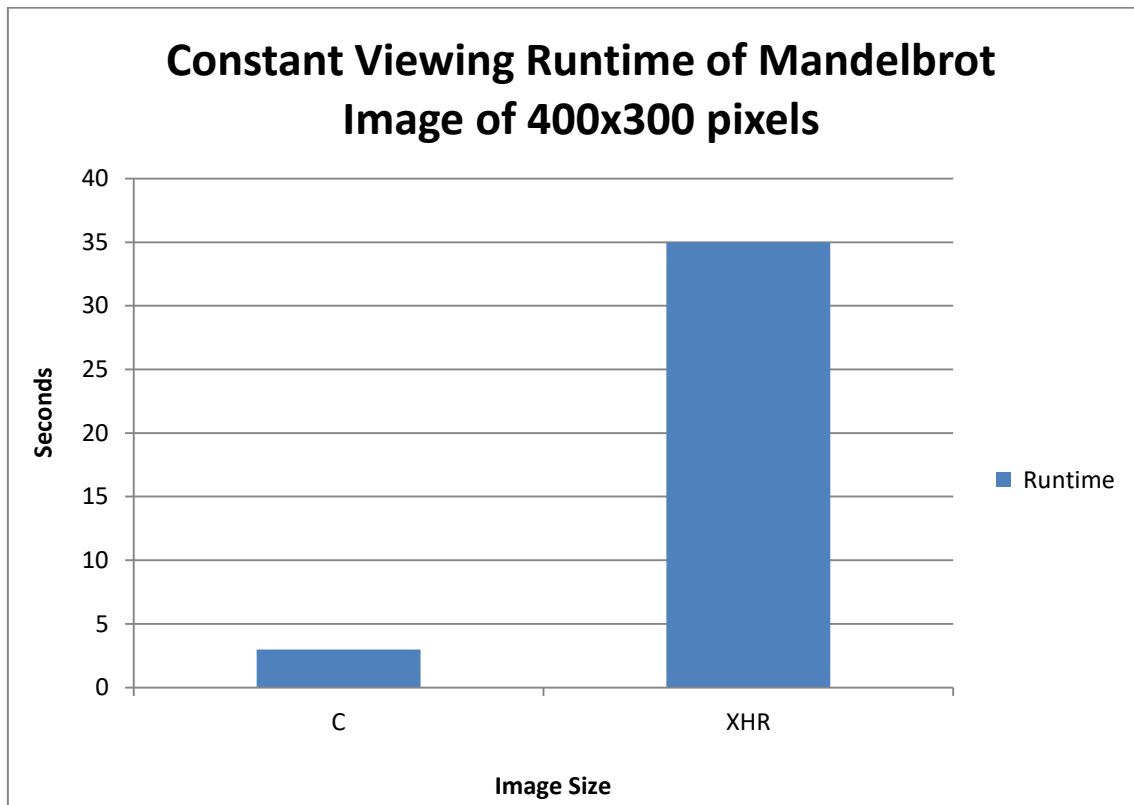
Figure 36: Total Runtime by Mandelbrot Image Size

Figure 36 shows the runtime of different Mandelbrot image sizes, once again scaling the number of tasks completed to ensure the same overall amount of image pixels were completed in each testing run remained the same.

Due to the unpredictable number of iterations for a complex number in the Mandelbrot set, comparing the runtime between the different image sizes is not a fair comparison.

However, the results show a clear difference in the total runtime between the XMLHttpRequest and WebSockets protocols, with the XMLHttpRequest protocol having a faster runtime in all test cases. In the case of the largest image size of 250x150 pixels, the XMLHttpRequest protocol ran 42% faster than the WebSockets protocol.

This suggests that the XMLHttpRequest protocol is a better when handling larger data transfers. At a size of 250x150 pixels, the returned data size is 30.40 kilobytes after the client side compression algorithm.



When not simulating real world web browsing habits and running 24 constant sessions within the browser compared to the C version of the Mandelbrot code, the C program greatly out performs the web browser code, running 1,070% faster than the JavaScript code.

Based on the previous attempts to optimize the runtime of tasks on the Panther framework, it is clear that attempts to reduce the task size or use alternative task scheduling policy will not decrease the runtime of web browser based anywhere near to the runtime of the C version. The only way to decrease the runtime of the JavaScript version at this stage to match the C version, would be to massively increase the level of parallel page viewing sessions beyond what is available in the current test setup.

6 Conclusion and Future Work

The goal of this project was to create a framework that could enable large-scale distributed computing via web browsers. The created Panther framework demonstrates a scalable architecture that enables website owners to let their webpage visitors partake in volunteer computing with only a few lines of code. The framework also provides a potential platform of expansion for exiting volunteer projects to access web browser based volunteers.

Simulation of real web browsing habits showed that an effective return on computation can be gained when processed on the Panther framework. While performance between the XMLHttpRequest and WebSockets communication protocols closely matched in most cases, certain scenarios did show an advantage for one protocol over the other, and this should be considered by project owners considering the use of web browsers for distributed computing.

Investigation into task allocation procedures showed that while overall task processing speed-up can be achieved by reducing individual task processing time, it is rarely cost effective to do so. To gain maximum utilisation of the Panther server and webpage viewers, every effort should be made to select a cost effective task size with a standard task allocation policy for the client. From there, efforts should be made to gain as many parallel viewing sessions as possible.

6.1 Future Work

Future development should concentrate on developing the Panther framework to allow a production deployment and distribution of a real volunteer computing project. While the current version of Panther is sufficient for testing and evaluation, if it was deployed for real usage, several new development would be would be required:

- For a long running distributed task management, the server should automatically refill its task cache once a certain threshold of remaining tasks has been reached.
- Password login and session management would need to be added to ensure only project owners can access the Admin control screen.
- Working at scale would require multiple instances of the Panther server to active with a correctly configured load balancer.
- The ability to interface with the BOINC framework should be implemented.
- To avoid the size increases due to JSON parsing, alternative task storage method, such as JavaScript binary ArrayBuffers() could be investigated.
- Investigate some of the server side capabilities of the WebSockets protocol in regards to task scheduling.

Methods should be investigated to allow testing at a greater scale of web browsers, possibly using the Panther framework on a website visited by real users. Finally,

alternative methods of task scheduling could still be explored to establish if any cost-effective improvements can be found for task distribution.

7 References

- [1] *BOINC Framework*. Online at <https://boinc.berkeley.edu/> (referenced 24/08/2018).
- [2] *BOINC Usage Stats*. Online at <https://boincstats.com/en/stats/projectStatsInfo> (referenced 24/08/2018).
- [3] Chao Liu. 2010 *Understanding Web Browsing Behaviours through Weibull Analysis of Dwell Time* SIGIR 10: Proceedings of the 33rd internal ACM SIGIR conference on Research and development in information retrieval. pp 379-386.
- [4] Yao Pan. 2017 *Gray Computing: A framework or Distributed Computing with Background JavaScript Tasks* IEEE Transactions on Software Engineering.
- [5] *HTTP overview*. Online at <https://developer.mozilla/kab/docs/Web/HTTP> (referenced 24/08/2018).
- [6] *Machine Learning Repository*. Online at <https://archive.ics.edu/ml/index.ph> (referenced 24/08/2018)