Real-time Web-based Remote Interaction with Active HPC Applications

Tim Dykes[1,4], Ugo Varetto[2], Claudio Gheller[3] and Mel Krokos[4]
[1]*HPE HPC/AI EMEA Research Lab, Bristol, U.K.*
[2]*Pawsey Supercomputing Centre, Kensington, Perth, WA, Australia*
[3]*Institute of Radioastronomy, INAF. Via P. Gobetti, 101 40129 Bologna, Italy*
[4]*School of Creative Technologies, University of Portsmouth, Eldon Building, Winston Churchill Avenue, Portsmouth, U.K.*

Keywords: Web-based Scientific Visualization, Remote Interaction, High Performance Computing.

Abstract: Real-time interaction is a necessary part of the modern high performance computing (HPC) environment, used for tasks such as development, debugging, visualization, and experimentation. However, HPC systems are remote by nature, and current solutions for remote user interaction generally rely on remote desktop software or bespoke client-server implementations combined with an existing user interface. This can be an inhibiting factor for a domain scientist looking to incorporate simple remote interaction to their research software. Furthermore, there are very few solutions that allow the user to interact via the web, which is fast becoming a crucial platform for accessible scientific HPC software. To address this, we present a framework to support remote interaction with HPC software through web-based technologies. This lightweight framework is intended to allow HPC developers to expose remote procedure calls and data streaming to application users through a web browser, and allow real-time interaction with the application while executing on a HPC system. We present a classification scheme for remote applications, detail our framework, and present an example use case within a HPC visualization application and real world performance for remote interaction with a HPC system over a Wide Area Network.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

In the modern Internet era, much of the information we need throughout our working day is available on the web. Over the past few decades communications infrastructure, user hardware, web browsers, and associated tools and libraries have evolved to such an extent that many user applications that once would have been installed locally are now run entirely remotely. This new, remote, web-based, paradigm means that a modern Internet user can do much more than type emails and view static pages in their browser, as was the case in the early days of the Internet. They can now type documents through online word processors, edit photos with image manipulation tools, compete in online 3D games via real-time 3D rendering and streaming, design engineering solutions with web CAD applications, and more, all from the comfort of a laptop web browser. This paradigm has led to a user expectation of powerful web services, exploiting *cloud computing* (Patidar et al., 2012), that are more accessible, portable, and simpler to use, than traditional applications. Web browsers typically do not require expensive high powered hardware, and are pre-installed on most personal computing devices. Users can access services built and hosted on the other side of the world to communicate, run applications, and store and retrieve data in real-time, without prior installation or configuration on their local machine. With the rise of low cost, on-demand, cloud-based web infrastructure, for example Amazon Web Services (AWS) and Microsoft Azure, and fast Internet streaming speeds, it is becoming commonplace for fully featured Software As A Service (SaaS) applications to be available in-browser (e.g. (Miller, 2009)).

Conversely, High Performance Computing (HPC) applications are typically executed remotely by exploiting computing clusters accessible by terminal or remote desktop protocol (e.g. Virtual Network Computing (VNC) client or X Forwarding protocol). User applications are run via job requests submitted to a resource management system, which are queued and executed when the required resources become available. This approach is effective for many traditional HPC applications (e.g. modelling and simulation); however, due to the paradigm of web and cloud computing, the modern day research scientist (a typical HPC user) has access to a much broader array of tools

Dykes, T., Varetto, U., Gheller, C. and Krokos, M.
Real-time Web-based Remote Interaction with Active HPC Applications.
DOI: 10.5220/0010260300880100
In *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2021) - Volume 3: IVAPP*, pages 88-100
ISBN: 978-989-758-488-6
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

that may benefit from, or even require, high performance computing, from interactive computing with Python-based tools to workflow software for coupling and/or chaining multiple large scale parallel applications.

Notably, there are a variety of HPC applications that necessitate a Human-In-The-Loop (HITL) (Nunes et al., 2015) with real-time user interaction; for example, visualization software, monitoring and inspection utilities, debugging tools, and computational steering software. This type of application requires some form of user interface, which is typically achieved through a client-server remote software approach where a local user application communicates with a remote HPC application; for example, a local graphical debugging client connected to a debug server running on a HPC system. However, these approaches are often tailored to a specific application and require a significant amount of setup (detailed further in Section 2). This class of application could benefit from the accessibility, portability, and simplicity of cloud-like web-based software.

There are existing efforts to provide cloud-like services for HPC, from creating HPC-like environments or running traditional HPC applications on cloud-computing infrastructures (Canon et al., 2010; Church et al., 2015; Younge et al., 2017), to building cloud-like infrastructures on HPC systems (Mauch et al., 2013). This extends to HPC applications exposed to users through the web via workflow management tools (Goecks et al., 2010; Brown et al., 2015) and Science Gateways (Gesing et al., 2015), and public web platforms supported by HPC infrastructures such as the Theoretical Astrophysical Observatory (TAO) (Bernyk et al., 2016) and similar efforts in other scientific communities. However, for developers to build HPC applications requiring real-time interaction, interoperability between web and HPC technologies is a necessity, particularly in terms of auxiliary software libraries. This is challenging especially due to the significantly disparate environments, both software and hardware, of web and HPC. A domain scientist developing high performance research software is unlikely to also be an expert in web development or even familiar with the languages and tools common in the field, and vice versa. This difficulty is compounded when there is a requirement for real-time interaction, necessitating high performance implementations on both sides. There are only few examples of HPC applications and web applications interacting in real-time (see Section 3); whilst this is partially due to the infancy of a HPC-Web convergence, the problem is exacerbated by a lack of general purpose tools to facilitate interoperability.

This challenge of interoperability has been the focus of existing efforts in the HPC community, for example with REST APIs for web-based interaction with resource management systems (Cholia et al., 2010; Cruz and Martinasso, 2019), environments such as EnginFrame[1] and Bridges (Nystrom et al., 2015) that are designed to support web portals and non-traditional HPC workloads, and containerized solutions packaging HPC software for general portable usage such as those available on NVIDIA's GPU-Cloud[2]. It is clear there is an emerging paradigm shift in HPC from the traditional command line batch scheduled job execution to a more accessible and user-friendly experience motivated by web, cloud, and interactive technologies. However, there is still a long distance to go, particularly for applications requiring real-time user interaction.

In this paper, to help bridge the gap between HPC and Web applications and address the lack of general purpose interoperability tools, we introduce WSRTI: a WebSocket (Fette and Melnikov, 2011) based framework for fast data streaming and simple, real-time, user interaction with HPC applications. The core principle of this framework is to allow HPC specialists and research scientists to quickly and easily create web interfaces to monitor and interact with active HPC applications in real-time. We provide a lightweight mechanism for a headless HPC application to expose a Remote Procedure Call (RPC) interface automatically linked to a web based graphical user interface. This is complimented by data streaming support to allow the user to transport data to and from the application independently of the RPC mechanism.

The remainder of the paper is structured as follows: Section 2 presents a novel classification scheme to identify and differentiate remote applications. With reference to this scheme we discuss related work in Section 3, before presenting the WSRTI framework in Section 4. This is followed by a practical explanation of the requirements for a HPC application to utilize WSRTI in Section 5, followed by a specific example use case within a high performance visualization application in Section 6. We briefly discuss performance in Section 7, and Section 8 concludes the work with a summary and future directions.

[1] https://www.nice-software.com/products/enginframe

[2] https://www.nvidia.com/en-us/gpu-cloud/

2 A CLASSIFICATION SCHEME FOR REMOTE APPLICATIONS

The inherently remote nature of working on a HPC system can discourage use of interactive applications. HPC systems are traditionally hosted in dedicated computing centers and on university campuses, and access typically requires a local workstation and terminal connection via Secure Shell (SSH) protocol. Once connected, the user can submit jobs through a resource management system such as SLURM (Yoo et al., 2003) or the Portable Batch System (PBS)[3] to request computing resources in either a *batch* or *interactive* manner. *Batch* submission inserts the job into a queue and schedules it to run when resources are available to be allocated, which allows efficient job management by the scheduler and maximum resource utilization. However, the job must be defined ahead of time and often failure will yield the allocation and require a new job to be created. *Interactive* submission indicates the resources are required immediately, providing the user with a shell where they can interactively launch applications for the duration of the allocation. While it is standard practice to support an interactive queue for applications that require it, interactive applications typically have a broader set of requirements, e.g. supporting user interaction.

An important characteristic of an interactive HPC application is the means by which user interaction is supported. This can be complicated by one or more layers of indirection that typically exist between the user and the compute nodes on which their software executes, illustrated by Figure 1. This indirection is necessary for security of the system, however it can introduce difficulties for the end user who is routinely also outside of a gateway firewall. Furthermore, compute nodes typically have a stripped down operating system with only high performance components, and as such may not include software for traditional desktop environments (e.g. an X11 server). Interaction with software running on the compute nodes must somehow address these layers of indirection between the user and the application. The most common approach is for the remote application to act as a server, and a local application on the user's machine to act as a client, coupled via a communication schema and forwarding mechanism such as SSH tunnelling. However, there are a variety of different approaches for this client-server scenario, from remote-desktop software to bespoke application frameworks.

To contextualise the work presented in this paper, we first critically review and classify the existing approaches for remote interaction with high performance applications, in terms of *direct* vs. *indirect* approaches, and *web* vs. *native* approaches. This classification scheme is outlined in Figure 2, in the context of a typical HPC setup. In this diagram, the high performance application consists of *Interface* and *Compute* components, which are exposed to the user via five different approaches with varying layers of indirection. Indirection of an approach refers to the use of auxiliary software (*Aux*) to support the remote access. Applications are connected via an *Application Layer Protocol* (ALP) (Zimmermann, 1980), and each *hop* refers to a jump between network layers that may need facilitating (e.g. by port forwarding).

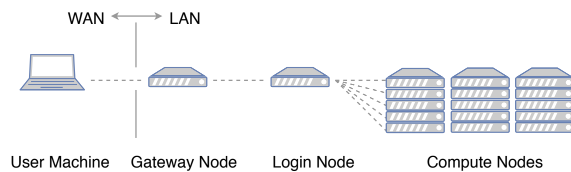

Figure 1: The typical layers of indirection between a user machine and the compute nodes of a HPC system.

Direct Remote Native (DRN): This classification refers to native applications that implement the full end to end client-server model, i.e. the application interface is installed on a users local machine, and the application server runs on the compute nodes of a HPC system. The connection is typically a custom messaging schema sent over a common application layer protocol (e.g. ZeroMQ[4], TCP sockets), and each hop is often enabled via SSH tunneling. This approach can perform well due to developer optimised messaging schemas tailored for the specific application, however relies on availability and installation of a client for the specific user machine. The increased performance is more effective for interactive applications, particularly those requiring real-time interaction. Examples of this class of application are high performance visualization software tools ParaView (Ahrens et al., 2005) and VisIt (Childs et al., 2011), and remote debugging software such as NVidia NSight Eclipse edition (NVIDIA Corporation, 2018a).

Indirect Remote Native (IRN): In this case native applications implement part of the client-server model, however they may rely on auxiliary software to extend fully to the user machine, or they may not directly connect to the remote application. There are various indirect approaches, and the location of the interface or auxiliary software does not always match exactly the two example scenarios shown in Figure 2. A typical example of the first scenario is a client

[3] e.g. http://www.pbspro.org/

[4] http://zeromq.org/

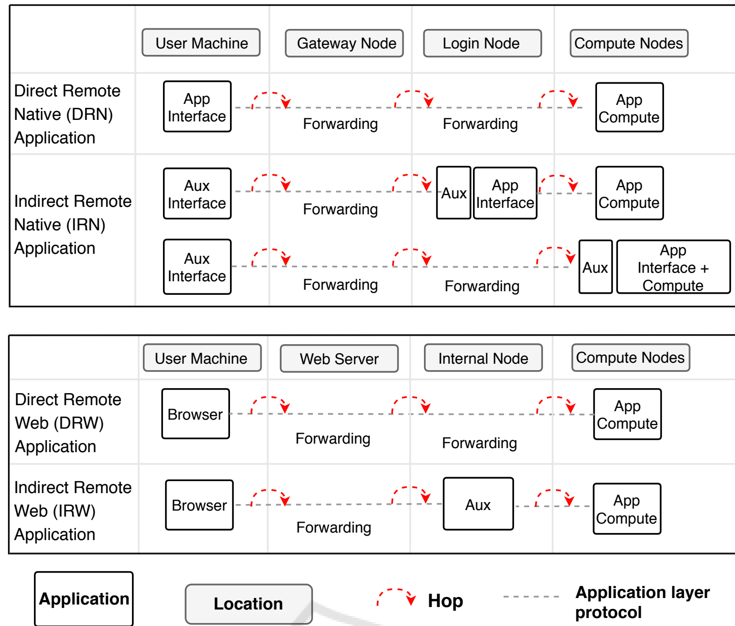

Figure 2: A classification scheme for remote applications in the context of HPC systems. *Top*: Native applications (DRN, IRN), *Bottom:* Web applications (DRW, IRW).

application which runs on the login node of the HPC system in order to interact with the resource management system; the client is then accessed locally via a remote desktop approach such as TurboVNC or X Forwarding. The second scenario requires the interface and auxiliary software to run directly on the compute nodes; this can be more difficult to achieve due to the lack of common desktop environment software on compute nodes. Both approaches can provide more portability when using auxiliary software. For example, a linux client can run on the HPC system whilst the user connects through remote desktop software on a Microsoft Windows-based PC. However, performance can degrade as auxiliary software is typically not able to optimise data transfers as effectively as custom approaches in a Direct Remote Application, and may use slow transfer protocols such as reliance on a shared filesystem. Examples of this type of application are remote profiling tools such as Intel VTune Amplifier (Intel Corporation, 2018) or the NVidia Visual Profiler (NVIDIA Corporation, 2018b). This approach is also commonly used to allow installing a variety of user software, which may in fact be direct remote applications, on a HPC system, such that the users of the system only need to install a remote desktop solution on their own machine.

Direct Remote Web (DRW): A web application running in the users browser is directly connected to the application server running on compute nodes. The direct approach requires exposing a public web server that also has internal access to the compute nodes, and directly connecting the user web application to the server via a web-capable ALP such as WebSockets (the direct connection may include intervening router software). This approach is most flexible and convenient for the user, however introduces significant security concerns if the web-server is to be made public. A simple work-around for an internal application is to run the web server on an internal node, and use SSH Tunneling to forward ports from the user machine to the internal node, thus securing the user connection via SSH. Some examples of these approaches are the ParaView Web Interface and JupyterHub (Jupyter Hub Development Team, 2018), detailed further in Section 3.

Indirect Remote Web (IRW): A web application running in the users browser is connected via a web server to auxiliary software (beyond simple routing software), which in turn is connected to the application server. This approach allows for the auxiliary software to provide a layer of security between the public web server and the application, however can introduce difficulties for interactivity. For example, common techniques are to use a shared filesystem or database polling as an intermediary step to pass requests from the user to the server, which is sufficient for data access but introduces additional latency and constraints not amenable to real-time interaction with the application. This approach is typical for public facing web portals supported by HPC resources, for example TAO (Bernyk et al., 2016) and the Cosmological Web Portal (Ragagnin et al., 2016).

This four-part classification scheme effectively describes the ways in which applications are run remotely on HPC systems. Some may support more than one mode of execution, and so can fit into multiple categories of this scheme; for instance extensive visualization software ParaView can be used with various types of interface that can be classified in each of these categories.

3 RELATED WORK

With reference to the classification scheme of Section 2, there are many remote applications that can be placed into one or more of these categories. The scope of the work presented in this paper is limited to those that fit into the Direct or Indirect Remote Web categories and can support real-time interactivity. The related tools we have identified that can be classified as DRW or IRW are listed below.

ParaView (Ahrens et al., 2005) is a large scale parallel visualization software, designed for effective exploitation of HPC systems. A web enabled version ParaViewWeb[5], can act as a Direct Remote Web Application by allowing the user to remotely connect via web browser to a ParaView server running on a HPC system. The connection is enabled via custom library *wslink*[6] that connects JavaScript web clients to a Python web server through ALP WebSockets. Furthermore, the in-situ library Paraview Catalyst (Ayachit et al., 2015) allows users to instrument their application for in-situ analysis, visualization, and computational steering purposes.

The Cactus computational framework (Goodale et al., 2003) supports a web browser interface for insitu visualization and steering tasks. The user can instrument existing HPC applications with the Cactus API, and perform interactive steering tasks and view visualization outputs through a web browser. The standard implementation utilises a HTTPD [7] web server, and forwards ports to the user for remote access.

The Jupyter Notebook[8] is a web application allowing users to interactively write and execute code, and transform, analyse, and visualize data using a variety of languages. The Notebook can be set up manually on a HPC system, exposing a web interface via the built in Notebook python web server that can be accessed by browser from a user machine via port forwarding. Furthermore, it is possible to deploy JupyterHub as a multi-user hub to access over HTTP, an approach also viable on HPC systems (Milligan, 2017).

The WebVis framework (Zhou et al., 2013) is a multi-user, client-server, visualization system with a web-based client that can interact with a cluster-based visualization server. The client is connected to the server via a back-end service built using the Google Web Toolkit and a Java web server, which communicate with an institutional web service that forwards events and images to and from the internal render cluster (i.e. an IRW approach). Client GUI interactions are forwarded to the server, and images returned, through an EventBus using the *HTTP server push* paradigm.

The commercial remote desktop software FastX[9] enables users to use existing desktop interfaces via the web with a WebAssembly module for a uniquely efficient high performance remote desktop service in the browser. This option is an effective solution for enabling access via the web for applications with existing interfaces, but does require a FastX license. A similar approach can be taken with other remote desktop protocols, such as VNC through e.g. TightVNC[10]. However, compute nodes of HPC systems commonly have a stripped down version of Linux that does not support GUI applications, so in most cases the application must already support remote execution from a login node. This software can enable Direct and Indirect Remote Applications to be used as Indirect Web Applications.

Remote frameworks that support web such as FastX are seeing some success, for example the web visualization portal at the Texas Advanced Computing Centre supports web-based usage of Paraview through a web-based VNC session, however such approaches do require the user to already have a direct/indirect remote application. Other solutions are built upon bespoke frameworks which are effective for a single application but less useful for the general case of a HPC application requiring remote interaction or monitoring. The development of WSRTI, described in the next section, is intended to address the lack of general purpose tools for interoperability, in order to support the development of more applications using the DRW and IRW approach for remote interactivity.

WSRTI, the development of which is described in the next section, is intended to address this lack of general purpose tools for interoperability, in order to support the development of more applications using the DRW and IRW approach for remote interactivity.

[5]https://kitware.github.io/paraviewweb
[6]https://github.com/kitware/wslink
[7]https://httpd.apache.org/
[8]http://jupyter.org/
[9]https://www.starnet.com/fastx/
[10]https://www.tightvnc.com/

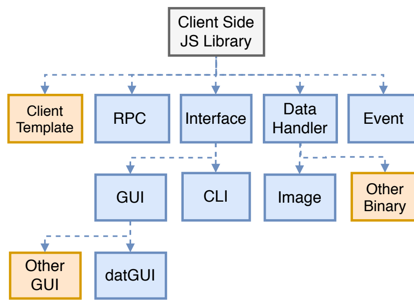

Figure 3: Structure of the client-side JavaScript library.

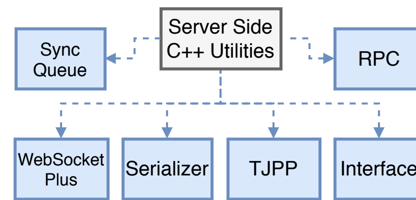

Figure 4: The collection of C++ utilities for HPC applications.

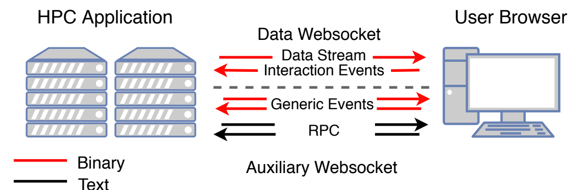

Figure 5: The data flow between a HPC application and WSRTI. Data is split across two sockets, one for dedicated binary streaming and a second for binary event and text RPC messages. Variation of image also presented in (Dykes et al., 2018) (Figure 5).

4 A FRAMEWORK FOR CONNECTING REAL TIME HPC APPLICATIONS TO THE WEB

The WSRTI library[11] is built to facilitate interoperability of interactive web and HPC technologies. We aim to reduce as much as possible the burden on a research scientist to understand web technologies, and allow them to link their application to a web interface automatically from application-side specifications. The key features we support are data streaming, RPC and event forwarding, and dynamic interface generation. The following subsections discuss the technical details of the library and illustrate how each of the key features is supported.

4.1 Overview

WSRTI is conceptually split into two components, a client side web library and a set of application-side C++ utilities to assist HPC developers in exposing RPC functionality and application data. The two components are connected via a WebSocket communication scheme discussed in Sections 4.2.

The client-side JavaScript library is structured as illustrated in Figure 3. The Data Handler, Event, RPC, and Interface modules are described in the following sections, and integrated into an included client template consisting of a web client capable of receiving and displaying images with a console input and debug log.

Figure 4 illustrates the server-side utilities, consisting of modules for binary serialization, generation of JSON interface objects, RPC, and asynchronous queues, along with convenience wrappers for image compression and WebSocket servers. These modules can be used by the HPC application developer to export data and interface descriptions, while sending and receiving RPCs and events. All modules of the library are documented with Doxygen[12], and contain unit tests that also act as example usage.

4.2 Communication: Data and Event Streaming

Client and application side utilities enable the user to package and asynchronously stream data back and forth between the active HPC application and web client; this includes generic binary data, formatted binary messages, and strings. In order to facilitate fast and interactive web communication, WSRTI exploits the WebSocket protocol. The full-duplex nature of a WebSocket connection is ideal for interactivity; messages can be sent and responses received without polling, allowing streaming services to be built in a simple and efficient manner. Two sockets are created to connect to the application, a dedicated *Data Stream* WebSocket for streaming binary application data and user interaction events (mouse and keyboard input), and an Auxiliary WebSocket for string-based RPC commands and generic application events, as illustrated in Figure 5.

The user can exploit the dedicated *Data Stream* to forward application data to the client. The client data handler module consists of set of data receivers, one of which monitors the Data Stream at any one time. Data receivers are simply implemented, and typically assume the message they receive is of an expected for-

[11] https://github.com/RemoteRTI/WSRTI

[12] http://www.stack.nl/ dimitri/doxygen/

mat. The client and application should both agree on the type of data expected on the Data Stream (this can be agreed, for example, via an event on the Auxiliary stream). Messages are received in the form of a JavaScript *Blob* and forwarded to the current active data processing module, which is by default an image processor. Keyboard and mouse events can also be automatically streamed to from the client to the application on the Data Stream.

The user can exploit the dedicated *Auxiliary Stream* via the client event module to send and receive formatted binary and string messages (distinguishable in the WebSockets layer). Formatted binary events are stored as JavaScript *TypedArrays* with a preceding integer identifier, using a schema agreed upon by both client and application. This identifier is inspected and the event is forwarded to the appropriate event handler, for example this may be a non-standard data message (e.g. a one-time downloadable file) or metadata for analysis. String messages are inspected for valid JSON-RPC formatting and forwarded to the RPC module (Section 4.3).

The use of two dedicated sockets allows WSRTI to make assumptions about the type of message received and optimize client-side de-serialization. This approach is used for data streaming from application to client, which may potentially trigger messages in high frequency and volume. For example during image streaming the client data handler can choose to interpret all messages received as JPEG compressed images and directly update the display at high frame rate. This avoids the need to de-serialize and check a message identifier which can have a noticeable impact on performance in the JavaScript data handling module.

The application-side *serializer* utility is a C++ header-only library to assist with serializing C++ objects and structures to binary messages, for example to create events or serialize application data. The *tjpp* convenience library assists with compressing images to JPEG via lib-jpeg-turbo[13]. These can then be passed to a WebSockets server (*WebSocketPlus* utility) via a synchronized queue (*SyncQueue* utility) to be asynchronously sent to the client.

4.3 Bi-directional RPC

The client and the application can exchange RPC requests in both directions. To avoid the user implementing application specific RPC handling in both the client and application the application developer can specify a function name and list of arguments as part of the interface generation process (see Section 4.4). Interface elements such as a button or input field on the client can be bound to a procedure call on the server at application run-time. This allows the application RPC interface to be extended or updated without changing the client code, and avoids the need for specific RPC functions and arguments to be known in advance,

Internally, the RPC module can send and receive JSON-RPC[14] formatted requests. The JSON-RPC format is a simple specification that covers the necessary feature set for RPC in this context. JSON (JavaScript Object Notation) is essentially a subset of JavaScript, and natively supported by the language. Bi-directional RPC allows the server to also trigger client-side operations if supported, such as handling a dynamic interface descriptor or accepting a file download. This model of RPC allows more advanced clients to be implemented, for example facilitating a peer-to-peer model where a client can trigger an action that will be forwarded to multiple other clients. This feature would support an advanced implementor in creating a collaborative environment through indirect messaging between clients.

4.4 Dynamic Interface Generation

The library follows an application-centric design, meaning that the majority of the client functionality is defined by the application through dynamic interface generation and RPC linkage. This allows the developer to focus on their HPC application and the data they want to expose, rather than on building a web GUI. The HPC application at runtime can define and modify the look and feel of the web client user interface, for example adding or removing buttons and sliders and specifying RPC calls and arguments that should be linked to client actions.

The interface generation mechanism depends on the construction of an *interface descriptor*, typically on the application side, which is forwarded to the client on connection. This can then be updated via client-targeted or broadcast RPC messages from the application, or upon request by the client. The descriptor is a hierarchical structure, reflecting the typical conceptual hierarchy of a user interface, that describes a series of buttons, sliders, input fields and display fields.

Figure 6 demonstrates the application-side process to create an interface descriptor. Groups of menu elements are of type *group*, while elements have types such as *Button* or *TextInput* which are used by the client to generate web interface elements. The *args* object is used to specify arguments to the RPC call,

[13]https://libjpeg-turbo.org/

[14]http://www.jsonrpc.org/specification

which are linked to data objects contained in the interface descriptor and can be generated via menu interactions or manually. The JSON descriptor is generated using a language specific JSON library, such as RapidJSON[15] for C++. A C++ *Interface* header included with the framework provides utility functions to generate JSON menu elements such as text input boxes, numerical sliders, and buttons, whilst an *RPC* header supports creation of RPC messages.

The client interface module (Figure 3) is split between a CLI component and a GUI component. In the template example. the CLI component is linked to an input box on the web page, and parses the input string for internal commands (such as a request to print the help message to the log), distinguished by a preceding '.' character, or commands to the server which are forwarded as RPC calls. This is helpful for simple text interfaces and debugging.

The GUI component generates an interface based on the interface descriptor. By default, the interface is generated via lightweight graphical interface library datGUI[16], however this can be replaced by a different interface library (e.g. React, jQuery) by overriding a series of interface generation functions. This allows users with web-experience to exploit more extensive interface libraries to add widgets and other advanced interface elements. Whilst the client typically parses the interface descriptor from the application and dynamically generates a user interface, it is also possible to manually construct this on the client-side if preferred.

5 EXPOSING A REMOTE APPLICATION FOR INTERACTION VIA WSRTI

In order to interact with an active remote application, there is a set of requirements the application should satisfy. As interaction is typically a necessary part of computational steering, the following requirements take inspiration from those for steering libraries (Brooke et al., 2003). However, WSRTI is intended to apply more generally to remote interaction with applications, as opposed to steering of numerical simulations, as such the requirements are generalised to represent the minimal requirements for user interaction. At least one of these requirements must be met in order to enable remote interaction: **(1)** expose a representation of application state, **(2)** expose a representation of application data, **(3)** expose an RPC interface, **(4)** accept input from standard Human Interface Devices (HIDs).

[15] http://rapidjson.org/

[16] https://github.com/dataarts/dat.gui

```
// JSON Value objects, a top level descriptor, a group
      representing a subfolder and its contents.
Value dsc(kObjectType), user_grp(kObjectType),
      user_content(kObjectType);
string current_input_file = "";
// A text box for an input file name and a 'load' button
user_content.AddMember("Input File",
      ui_text_input(current_input_file));
// Describe the argument to the 'Load' function: 1
      string, the 'Input File' text box content
vector<string> args = { "User Settings/Input File" };
user_content.AddMember("Load", ui_button("cmd_load",
      args));
// Add the contents to the 'User Settings' group
user_grp.AddMember("type", "Group");
user_grp.AddMember( "contents", user_content);
// Add 'User Settings' group to the interface descriptor
dsc.AddMember("User Settings", user_grp);
```

Figure 6: Creating an interface descriptor with one menu called "User Settings", containing a text input box and a button to load an input file by name. RapidJSON custom memory allocator arguments are removed for brevity.

Supporting **(1)** is a minimal requirement allowing a web interface to display the current application state. This could be, for example, whether the application is still running and some indicator of algorithmic progress, e.g. the current time in a computational simulation. **(2)** enables an interface to display a representation of the applications working data. This could be a subset of simulation data for analysis, or pre-generated analyses such streaming 3D visualization or graph plots. **(3)** enables linking web interactions to an RPC interface to trigger application mechanisms such as modifying variables or changing state. **(4)** relates to input from HIDs, for example linking keyboard and mouse interactions to actions within the application such as controlling a virtual camera during visualization or stepping through program execution in a debugging tool.

In order to support one or more of these requirements, the application must contain a control loop or checkpoint, in which point actions **(1)** and/or **(2)** can be performed, or input from the remote client can be accepted and handled (Figure 7, left). A typical case of control loop is iterating over the time domain during a simulation (or time stepping), in which case WSRTI could be used for computational steering. These requirements may be satisfied by *instrumenting* user code with a set of additional functions. Figure 7, right, demonstrates the minimum necessities to add interactivity to a generic application for WSRTI. *setup()* is responsible for initialising the WebSocket servers and providing callbacks for event handling,

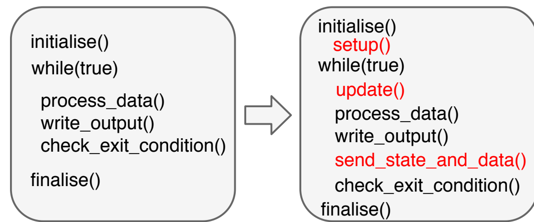

Figure 7: The generic case for instrumenting a high performance application for remote observation or interaction.

along with constructing the JSON interface descriptor which is sent to the remote client. *update()* receives events and RPC requests from the client and processes them accordingly, potentially modifying the user code parameters based on the events received. *send_state_and_data()* outputs application state and data for the client to process, either in the form of interface updates or streaming data.

To demonstrate the applicability of WSRTI to real-time applications, we use as an example case a high performance batch visualization code. This HPC application is instrumented to accept input, RPC and image streaming to become a web-accessible real-time large scale visualization server for HPC systems.

6 USE CASE: SCIENTIFIC VISUALIZATION

Splotch (Dolag et al., 2008) is a scientific visualization package designed to run on HPC systems and process large scientific datasets. Splotch is open source, written entirely in C++, with minimal dependencies beyond those for parallel models and specific file I/O. Splotch can exploit OpenMP for shared memory parallelism, MPI for distributed memory parallelism, and is also able to exploit heterogeneous machines with computational accelerators (Jin et al., 2010; Rivi et al., 2014; Dykes et al., 2017).

Initially designed to run in batch processing mode, Splotch reads a file on start-up describing input data and visualization parameters. After loading the input data and setting up the visualization scene, an iterative render loop performs the visualization, creating and writing a series of one or more images along a pre-defined camera path. We detail the steps taken to instrument our code to communicate with the WSRTI library and fulfil the requirements in Section 5, extending Splotch from a batch visualization software to a visualization server capable of real-time interaction through a web interface.

The Splotch algorithm is presented in Figure 8, with additions for WSRTI marked in red. The key al-

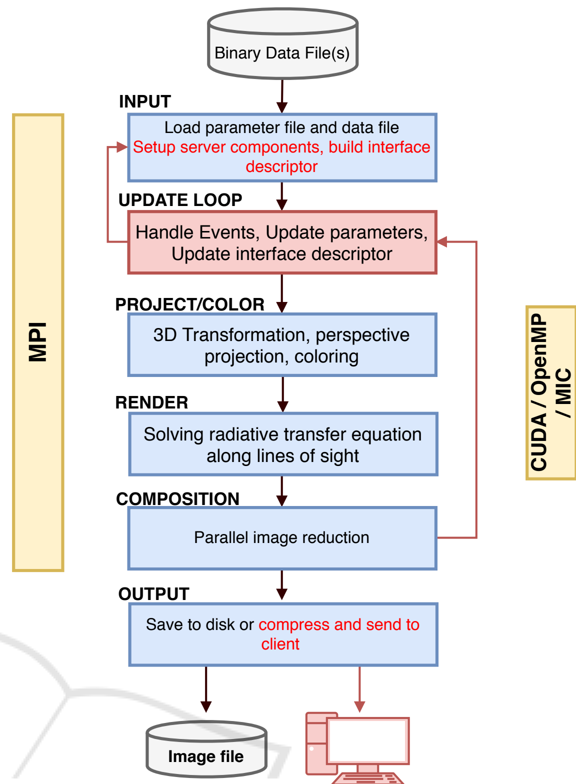

Figure 8: The Splotch algorithm, with changes for WSRTI interactivity highlighted in red. Image also presented in (Dykes et al., 2018) (Figure 4).

gorithmic change involved the addition of an update function with loop back mechanism to move from algorithm completion back to the update, followed by a series of additional functions to perform tasks as detailed below.

Initialisation. After file input, we initialise the WSRTI server side components and create an interface descriptor as demonstrated in Figure. 6, adding text input and number sliders, adding RPC linkage for various modifiable visualization parameters.

Update. At the beginning of the control loop the update function performs various tasks. Event handling processes both events and RPCs received since the last update. Local parameters are updated, and updates are reflected in the interface descriptor. In the case of Splotch, we support HID events to control the visualization camera (mouse for rotation, keys for movement), and RPC messages to update visualization parameters such as the active dataset, color map, and graphical variables.

Output Data. Our output data, an uncompressed TGA formatted image, is passed through a JPEG image compressor and added to the message queue (using WSRTI syncqueue utility), which is asynchronously extracted and passed to the WebSocket-

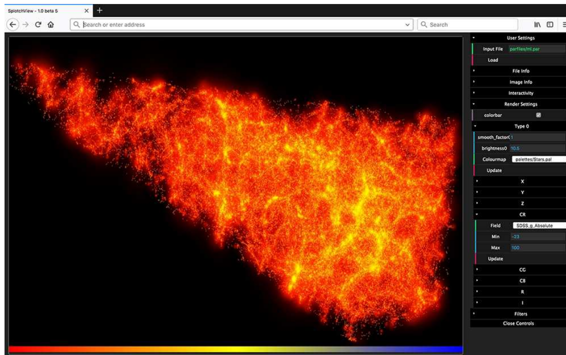

Figure 9: Splotch running on a HPC system visualizing an astronomical light-cone in real-time, streaming to web browser via the WSRTI framework. Dynamic interface is seen on the right hand side using the default dat.GUI interface generator. Image also presented in (Dykes et al., 2018) (Figure 11), see reference for further details.

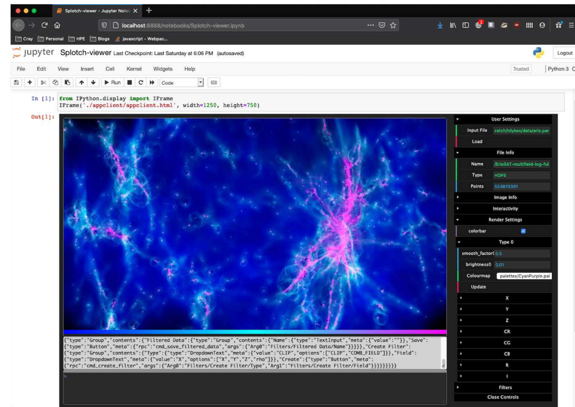

Figure 10: A Jupyter Notebook showing interactive visualization via the Splotch code running on a remote HPC system, visualizing the results of state of the art galaxy formation simulation GigaERIS.

Plus utility for sending to the client. In Splotch, the limiting factor on framerate is the data size; for large datasets (hundreds of Gigabytes and more), it is expected to run at a reduced frame-rate (e.g. 10fps) as compared to other real-time rendering. For this reason we use only frame-by-frame compression, rather than video compression which would increase the latency for user interaction at low frame rates.

Various application-specific optimisations were implemented to achieve a fast frame rate for real-time visualization, outside of the scope of WSRTI. These focused predominantly on reduced memory allocation and reuse of data buffers, and may or may not be necessary for a more general research application.

The initial instrumentation for interactivity is lightweight, and so a researcher looking for low-impact monitoring can be up and running quickly, however further work is required to create a fully featured interactive application. The interactive build of Splotch exploiting WSRTI is illustrated in Figure 9, and for more extensive detail on the feature-set and usage of interactive Splotch for astronomical visualization, the reader is referred to (Dykes et al., 2018).

Instrumentation via WSRTI also allows a deeper integration with web-based tools for scientific analysis, beyond the simple client shown in Figure 9. To demonstrate this, in Figure 10 we embed our web client within a Jupyter notebook, and connect it to the Splotch visualisation server running on a high performance computing system (test system as detailed in Section 7).

In this case, we are visualising a set of snapshots of a large state of the art galaxy formation simulation GigaERIS (Tamfal, Mayer, et al. in prep.), a higher resolution follow-up to the successful Eris simulation suite (Guedes et al., 2011). The full evolution of this simulation produces hundreds of snapshots, each of which consists of 1.1 billion particles, with 10-12 data fields per particle, resulting in approximately 50GB disk space required per snapshot. There are three types of particle, dark matter, stars, and gas; the visualisation consists of just the stars and gas (between 500 and 600 million particles for a single snapshot), with gas coloured by temperature and stars coloured by age. A volume rendering of such data is typically beyond the capabilities of a web browser, due to the *size* (too large to download and process locally), *location* (source data is often stored on the parallel file system of a supercomputer), and *complexity* (can require domain specific tools and high performance resources to visualize).

Whilst this example is simple and demonstratory, a full integration could benefit from Python to JavaScript bindings to allow full control of the Splotch server through the Notebook. For example, interesting subsets of data may be identified and extracted via the visual interface, and statistical analysis may then be applied using common Python-based analysis tools, either via downloading to the local machine, or via a hosted Jupyter Notebook instance on the HPC system.

7 REAL WORLD PERFORMANCE

HPC systems are typically accessed remotely, from elsewhere in the same building to continents on the other side of the world. For this reason, we run a number of tests spanning a wide physical area to demonstrate the extent of support for event and data streaming through WebSockets for WSRTI.

For many network-based tools, performance is significantly dependent on the performance of the network, as such the most significant factor in this case is the variable Internet connection between the user and the HPC system. Real-world performance for WSRTI is dependent on the user and their environment as well as the typical load on the network at any one time. This section describes the performance for one such environment, which represents a typical user scenario. A user laptop is set up as a client in a U.K. University laboratory, whilst a server application is executed on a HPC system at a remote computing facility abroad.

The test system is Swan, a Cray XC50 located at the Cray computing facility in Wisconsin, USA. Each utilized node consists of 2 Broadwell 22-core Xeon CPUs clocked at 2.2Ghz. The web client runs on a Macbook Pro (early 2013 model) with 2.7 GHz Intel Core i7, and Mozilla Firefox 61.0.1, at the University of Portsmouth in the UK. The test environment on the HPC system is a C++ application (included with the library) that generates data buffers of varying size and streams them over WebSockets using the *SyncQueue* and *WebsocketPlus* utilities. It has the capacity to send without expecting reply, mimicking a data stream, or wait for replies and measure send and receive latencies mimicking events or RPC calls. On the user laptop the template web client in which we have implemented an additional generic client DataHandler (see 4.2), which can act as a binary data stream receiver or a ping-pong type application that will return each received message. The test system network is organised similarly to Figure 1, without the gateway node. A port is forwarded for each WebSocket via SSH tunnel, and the client web-page is hosted locally for the user. For a series of tests with varying packet size, we stream packets to the client and measure latency and bandwidth.

Figure 11 shows a series of bandwidth results for the described test setup. In this environment we reach a maximum sustained bandwidth for data streaming of ~9 MB/s, which is reached at packets of 2MB, with ideal bandwidth for packet ranges from 128KB to 2MB. As previously mentioned, test results are dependent on the current network load, and so may vary with time. Figure 12 shows data latency, for packets under 8KB we have a latency of under 100ms, which is typically acceptable for real-time interaction in a typical remote application. For packets up to 500KB latency remains in the 1-200ms range, steadily rising as packets increase in size beyond this.

Figures 11 and 12 show WSRTI performance when streaming from a HPC system roughly 4000 miles away, and typical users of HPC centers in their own country, or even their own institution, may obtain higher bandwidths. Conversely, users in remote locations may indeed obtain much lower bandwidths. This can affect both the number of data packets that we can receive per second, and the latency at which we receive them, as illustrated by inconsistent bandwidth measurements for larger data objects in Figure 11. In our Splotch use case, JPEG compressed images are streamed from the application to the client and user interaction events (mouse and keyboard) and RPC messages are returned. Figure 13 illustrates the compressed binary size of a typical 1920x1080 pixel High Definition (HD) Splotch image at varying compression qualities. Referring to Figure 11, we are able to achieve maximum bandwidth in the mid to high quality ranges (JPEG 60-95 range of Figure 13), and >10 FPS on all compressed data sizes.

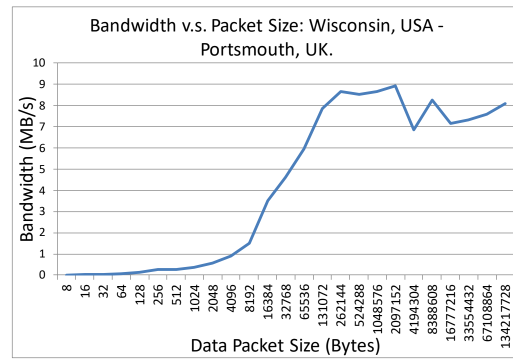

Figure 11: Bandwidth for data streaming from a WSRTI-enabled synthetic remote application.

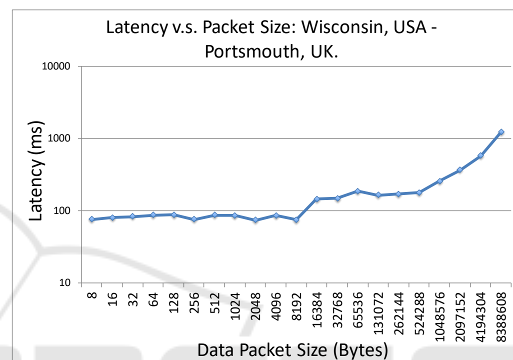

Figure 12: Packet latency for data streaming from a WSRTI-enabled synthetic remote application.

The size of binary events and RPC calls typically range from 32 to 128 bytes, potentially extending to a few kilobytes and above if a data packet is attached to an RPC call. As demonstrated by the data in Figures 11 and 12, for real-time interaction with a HPC application WSRTI can perform sufficiently to stream user interaction events and RPC calls between the web client and application. Whilst bandwidth may be low for small size packets case, the low latency means

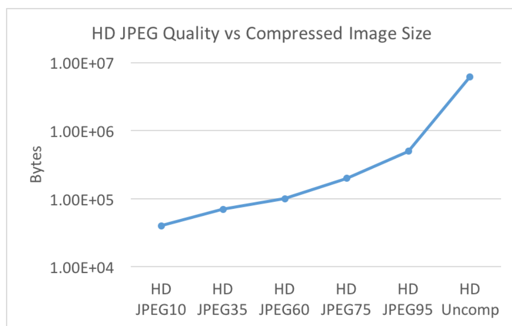

Figure 13: The relationship of JPEG compression quality factor and image byte-size for a typical High Definition (1920x1080 pixel) Splotch visualization image. *Uncomp* indicates the size of an uncompressed output.

interaction is effective and responsive.

For real-time data streaming, there are a few limitations on the web-client. Particularly, the web browser is less capable of processing large data (e.g. in the range of multiple Gigabytes). In our tests, web browsers typically have upper limits on single WebSocket messages starting around 256 MB, and will struggle computationally with buffers of multiple gigabytes. For larger data, it is recommended to perform computationally-intensive analysis on the HPC system and send small analysis datasets along with application status/interface and analysis results such as graph plots and visualization images to the client.

8 SUMMARY AND CONCLUSIONS

In this paper we introduce the WSRTI library, which aims to support integration between traditional HPC applications in languages such as C or C++, and the modern day Web environment. In particular, we aim to provide a lightweight solution to interact via the web with HPC applications in real-time.

We summarise and classify the techniques available for remote connections to HPC systems, explaining the process and benefits of each. We summarise existing tools supporting remote interaction via web, highlighting the lack of general purpose tools to assist HPC application developers in remotely interacting with their applications, and detail the features of WSRTI and how they can be used to support such activities, demonstrating usage by instrumenting a batch scientific visualisation software for real-time interaction. We expect future work to include streamlining our utilities to reduce the tax on HPC developers hoping to extend their HPC application for web connectivity, particularly reducing the necessity for boilerplate code. One of the avenues we are exploring is the use of a wrapper API to allow WSRTI to be used in a similar manner to popular in-situ visualization libraries such as Paraview Catalyst and VisIt Libsim. We also intend to further optimise the communication routines, especially for data streaming, considering features such as automatic compression factor adjustment and tiled compression for multiplexed image streams. This should be combined with an investigation into the optimal approach to set up a WebSocket between HPC applications and Web clients from a security perspective, whilst retaining interactivity. Finally, we are also considering interfaces for other languages in use in HPC, such as Fortran.

ACKNOWLEDGMENT

We gratefully acknowledge Swinburne Centre for Astrophysics and Supercomputing for hosting author Tim Dykes while part of this work was completed. We thank Lucio Mayer, University of Zurich, for providing the GigaERIS data for visualization. Thanks to HPE & Cray UK for providing HPC resources. Mel Krokos acknowledges support by NEANIAS, funded by the EC Horizon 2020 research and innovation programme under grant agreement No. 863448.

REFERENCES

Ahrens, J., Geveci, B., and Law, C. (2005). 36 - paraview: An end-user tool for large-data visualization. In Hansen, C. D. and Johnson, C. R., editors, *Visualization Handbook*, pages 717 – 731. Butterworth-Heinemann, Burlington.

Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., and Mauldin, J. (2015). Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 25–29, New York, NY, USA. ACM.

Bernyk, M., Croton, D. J., Tonini, C., Hodkinson, L., Hassan, A. H., Garel, T., Duffy, A. R., Mutch, S. J., Poole, G. B., and Hegarty, S. (2016). The Theoretical Astrophysical Observatory: Cloud-based Mock Galaxy Catalogs. *The Astrophysical Journal Supplement Series*, 223:9.

Brooke, J., Coveney, P., Harting, J., Jha, S., Pickles, S., Pinning, R., and Porter, A. (2003). Computational steering in realitygrid. In Cox, S., editor, *Proceedings of the UK e-Science All Hands Meeting 2003, 2-4 September, Nottingham, UK*, pages 885–1/4. EPSRC.

Brown, D. K., Penkler, D. L., Musyoka, T. M., and Bishop, O. T. (2015). Jms: An open source workflow management system and web-based cluster front-

end for high performance computing. *PLOS ONE*, 10(8):e0134273.

Canon, S., Wright, N. J., Muriki, K., Ramakrishnan, L., Shalf, J., Wasserman, H. J., Cholia, S., and Jackson, K. R. (2010). Performance analysis of high performance computing applications on the amazon web services cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science(CLOUDCOM)*, volume 00, pages 159–168.

Childs, H. et al. (2011). Visit: An end-user tool for visualizing and analyzing very large data. In *In Proceedings of SciDAC*.

Cholia, S., Skinner, D., and Boverhof, J. (2010). Newt: A restful service for building high performance computing web applications. In *2010 Gateway Computing Environments Workshop (GCE)*, pages 1–11.

Church, P., Goscinski, A., and Lefèvre, C. (2015). Exposing hpc and sequential applications as services through the development and deployment of a saas cloud. *Future Generation Computer Systems*, 43-44:24 – 37.

Cruz, F. A. and Martinasso, M. (2019). Firecrest: Restful api on cray xc systems. In *Proceedings of the Cray User Group*.

Dolag, K., Reinecke, M., Gheller, C., and Imboden, S. (2008). Splotch: visualizing cosmological simulations. *New Journal of Physics*, 10(12):125006.

Dykes, T., Gheller, C., Rivi, M., and Krokos, M. (2017). Splotch: porting and optimizing for the xeon phi. *The International Journal of High Performance Computing Applications*, 31(6):550–563.

Dykes, T., Hassan, A., Gheller, C., Croton, D., and Krokos, M. (2018). Interactive 3D visualization for theoretical virtual observatories. *Monthly Notices of the Royal Astronomical Society*, 477:1495–1511.

Fette, I. and Melnikov, A. (2011). The websocket protocol. RFC 6455, RFC Editor. http://www.rfc-editor.org/rfc/rfc6455.txt.

Gesing, S., Dooley, R., Pierce, M., Krüger, J., Grunzke, R., Herres-Pawlis, S., and Hoffmann, A. (2015). Science gateways - leveraging modeling and simulations in hpc infrastructures via increased usability. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 19–26.

Goecks, J., Nekrutenko, A., Taylor, J., et al. (2010). Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86.

Goodale, T., Allen, G., Lanfermann, G., Massó, J., Radke, T., Seidel, E., and Shalf, J. (2003). The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin. Springer.

Guedes, J., Callegari, S., Madau, P., and Mayer, L. (2011). Forming Realistic Late-type Spirals in a ΛCDM Universe: The Eris Simulation. ApJ, 742(2):76.

Intel Corporation (2018). Intel vtune amplifier 2018 user's guide.

Jin, Z., Krokos, M., Rivi, M., Gheller, C., Dolag, K., and Reinecke, M. (2010). High-performance astrophysical visualization using Splotch. *ArXiv e-prints*.

Jupyter Hub Development Team (2018). Jupyterhub — jupyterhub 0.9.1 documentation.

Mauch, V., Kunze, M., and Hillenbrand, M. (2013). High performance cloud computing. *Future Generation Computer Systems*, 29(6):1408 – 1416.

Miller, M. (2009). *Cloud computing*. Que Publishing.

Milligan, M. (2017). Interactive hpc gateways with jupyter and jupyterhub. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 63:1–63:4, New York, NY, USA. ACM.

Nunes, D. S., Zhang, P., and Silva, J. S. (2015). A survey on human-in-the-loop applications towards an internet of all. *IEEE Communications Surveys Tutorials*, 17(2):944–965.

NVIDIA Corporation (2018a). Nsight eclipse edition cuda toolkit documentation.

NVIDIA Corporation (2018b). Nvidia cuda toolkit profiler user's guide.

Nystrom, N. A., Levine, M. J., Roskies, R. Z., and Scott, J. R. (2015). Bridges: A uniquely flexible hpc resource for new communities and data analytics. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 30:1–30:8, New York, NY, USA. ACM.

Patidar, S., Rane, D., and Jain, P. (2012). A survey paper on cloud computing. In *2012 Second International Conference on Advanced Computing Communication Technologies*, pages 394–398.

Ragagnin, A., Dolag, K., Biffi, V., Cadolle Bel, M., Hammer, N. J., Krukau, A., Petkova, M., and Steinborn, D. (2016). An online theoretical virtual observatory for hydrodynamical, cosmological simulations. *ArXiv e-prints*.

Rivi, M., Gheller, C., Dykes, T., Krokos, M., and Dolag, K. (2014). Gpu accelerated particle visualisation with splotch. *Astronomy and Computing*, 5:9–18. 12 months embargo.

Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In Feitelson, D., Rudolph, L., and Schwiegelshohn, U., editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg. Springer Berlin Heidelberg.

Younge, A. J., Pedretti, K., Grant, R. E., and Brightwell, R. (2017). A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 74–81.

Zhou, Y., Weiss, R. M., McArthur, E., Sanchez, D., Yao, X., Yuen, D., Knox, M. R., and Czech, W. W. (2013). *WebViz: A Web-Based Collaborative Interactive Visualization System for Large-Scale Data Sets*, pages 587–606. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zimmermann, H. (1980). Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432.