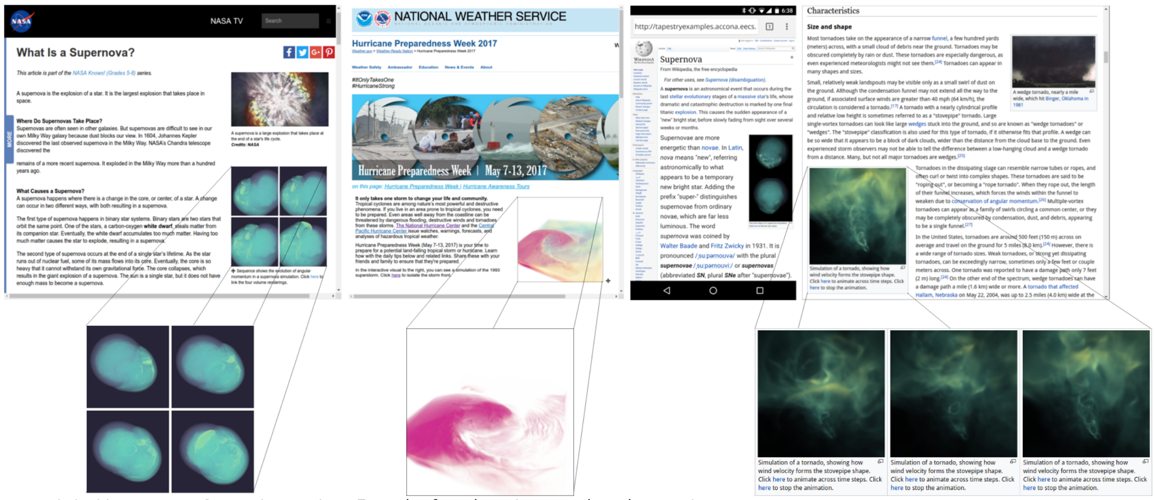
Scalable Web-Embedded Volume Rendering

Mohammad Raji *†
University of Tennessee

Alok Hota *†
University of Tennessee

Jian Huang †
University of Tennessee



Four linked hyperimages for synchronized inspection across time steps of the turbulent progression of a supernova. Data from DOE SciDAC Terascale Initiative.

Example of one hyperimage and one hyperaction for scripted interaction. The application developer can choose to highlight regions within the same phenomenon. Data selected from a regional scale WRF superstorm simulation ensemble.

One hyperimage with one hyperaction playing a temporal animation of a forming tornado. Data from a CM1 tornado simulation on the Blue Waters supercomputer at NCSA.

Figure 1: Example webpages with embedded volume rendering. From left to right: NASA's education outreach page on supernovae, a National Weather Service page on hurricane preparedness, the Wikipedia page on supernovae (viewed on a mobile device), and the Wikipedia page on tornadoes. Each embedded volume render appears as a static image but also allows traditional 3D interactions such as rotating and zooming, as well transfer function modulation. Additional interactions, such as scripted animation and linking and unlinking multiple views are also supported.

Abstract

In this paper, we develop a method to encapsulate and embed interactive 3D volume rendering into the standard web Document Object Model (DOM). The package we implemented for this work is called Tapestry. Using Tapestry, data-intensive and interactive volume rendering can be easily incorporated into web pages. For example, we can enhance a Wikipedia page on supernova to contain several interactive 3D volume renderings of supernova volume data. There is no noticeable slowdown during the page load by the web browser. A user can choose to interact with any of the volume renderings of supernova at will. We refer to each embedded 3D visualization as a *hyperimage*. Hyperimages depend on scalable server-side support where volume rendering jobs are performed and managed elastically. We show the minimal code change required to embed hyperimages into previously static web pages. We also demonstrate the supporting Tapestry server's scalability along several dimensions: web page complexity, rendering complexity, frequency of rendering requests, and number of concurrent sessions. Using solely standard open-source components, this work proves that it is now feasible to make volume rendering a scalable web service that supports a diverse audience with varying use cases.

Keywords: visualization system, web applications, volume rendering

*These authors contributed equally to this work.
†e-mail: {mahmadza,ahota,huangj}@utk.edu

1 Introduction

Web browsers have gradually become a popular front-end for scientific visualization applications. Many systems exist, such as ParaViewWeb [16], ViSUS [24], and XML3D [30]. There are many reasons driving the trend of merging web technologies into scientific visualization delivery. Namely, the web browser is one of the most familiar interfaces for users today. It is also readily the most platform-agnostic client software in modern personal computing.

In traditional systems, the back-end visualization and the front-end interaction are typically synchronized and centered around one particular dataset or user at a time. We believe there is potential in going beyond traditional system designs, by enabling multiple users, multiple datasets, and simplified platform-agnostic adoption.

In this work, we present a system called "Tapestry", which explores the potential of embedding 3D visualizations in web pages as self-contained DOM elements and focuses on the "scale of audience."

Our driving use case stems from publicly accessible websites utilizing volume visualizations and scientific imagery. Consider the many *knowledge-oriented* websites, such as Wikipedia pages about supernovae and tornadoes, and *science-oriented* websites, such as NASA National Snow and Ice Data Center and the NOAA Storm Prediction Center. Volume visualizations on these websites are static images. Compared to static images or curated videos, interactive 3D visualizations can be more engaging for visitors as dissemination media. They can be educational and help build a stronger bridge between scientific teams and the public. In addition, our system can concurrently handle multiple users and datasets. Therefore, it can also be used for scientific teams to document, share, and discuss interactive simulation results on the web.

This need is novel, because the set of performance metrics to consider are not typical of traditional remote visualization research. Our primary goals are therefore to (i) significantly increase the front-end simplicity of embedding an interactive visualization into a web document, and (ii) ensure that the server-side support of the front-end 3D visualizations can be available on-demand. In our work, we define front-end simplicity as both minimal code change for developers and minimal load time and runtime overhead for users, which is crucial in the web ecosystem. We believe the simplicity encourages adoption by web developers. This simplicity, along with the support of an unknown and highly varying volume of users, requires on-demand availability of the supporting back-end. This means that the server must be able to transparently scale based on the number of users and datasets.

Our contribution is twofold. First, we provide the mechanism to maximize front-end simplicity. Second, we develop a scalable server-side architecture that ensures on-demand availability for multiple simultaneous datasets and users. In results, we demonstrate the efficacy of our method by showing various embedded visualizations and their capabilities in different web pages. We also show the performance and scaling of our system under a stress test.

We will discuss the overall background in Section 2, Tapestry's web enabled architecture of scientific visualization application in Section 3, example ways to use Tapestry in existing web pages and new web applications in Section 4, and results in Section 5. We conclude and discuss future work in Section 6.

2 Background

The advent of D3 [7] has made JavaScript and the Document Object Model (DOM) the new standard environment for information visualization. Since D3, some previous works in scientific visualization have also looked at the possibility of web-based visualizations. Most notably, Visualizer and LightViz use the ParaViewWeb API [16] and enable exploratory visualization by sending requests to a remote ParaView process. ParaViewWeb was also successfully integrated with data management systems, such as MIDAS, to allow for a faster and more efficient analysis of massive datasets [15]. ViSUS also enables web-based renderings using a browser plugin backed by a heavy server, such as the IBM BlueGene [24]. Tamm and Slusallek used XML3D backed by a cluster to enable realtime ray-tracing in the browser without a need for a plugin [29, 30]. We have observed two main points regarding these systems.

First, most systems in the past have relied on browser plugins or WebGL on the client side. Such solutions come with a long delay for data transfer during load time, and heavy CPU and memory footprints during runtime. In addition, for user interactivity, they have to rely on the user's own client-side resources. These resources can range from a workstation to a laptop or even a tablet. Some of these devices will not have enough resources for rendering.

Second, most traditional visualization systems see the browser as an OS-independent platform, but still adhere to a one-to-one mapping between a browser and a dedicated server instance. In the past, this has been somewhat inevitable due to slower rendering speeds. With the advent of faster ray-tracers, such as OSPRay [32], volume rendering can now focus on the "scale of audience" and benefit from a web ecosystem similar to D3.

With Tapestry, we aim to test and validate that a system can be engineered such that performance is not limited by client-side resources. Instead, all constraints are compartmentalized on an external server, which can be more easily scaled up in terms of hardware and high performance algorithms. The topic of how to best decompose a visualization pipeline in a remote visualization setting has been well studied [27, 34]. We contribute a new approach that allows for an *m*-to-*n* relationship, where multiple users can concurrently view any number of datasets using the same server instance, in contrast to traditional 1-to-1 user to server correspondence.

The tight coupling of client and server in existing remote visualization systems is in part due to not distinguishing the differing roles in state management between client and server. Typical implementations in past and current remote visualization systems essentially replicate the application states on both the client and server. Instead, we use loose coupling by separating the application space from the system space. The application space maintains the application states. The system space answers requests from the application space, and remains stateless. This "separation of concerns" applies well to creating data visualization as a shared web service. We can hence provide a simplified front-end that does not depend on plugins.

One work that comes closer to meeting our goal is VTK.js [19]. VTK.js is the browser-based version of the Visualization Toolkit [26] and only requires WebGL for volume rendering. Although the problem of "scale of audience" does not apply to VTK.js, it is impractical for large datasets with high quality renderings, because the full dataset must be downloaded to the client's computer before rendering. The problem is exacerbated when there are many linked datasets or interactive time series volumes on a page.

Our server-side leverages the following components and design principles from existing literature.

Volume rendering is well understood from an algorithm perspective [22]. Highly efficient implementations using many-core processors, either GPU or CPU, are available as community-maintained open-source renderers [2, 4, 8, 32]. In this work, we use OSPRay [32] because of its rendering performance. Additionally, its software-only nature makes it easier to manage in a typical cloud-managed container. A GPU-based renderer that exhibits similar throughput to OSPRay can also be used. We chose to encode all OSPRay-rendered framebuffers as PNG images, typically just a few kilobytes per image in size at good compression quality. Of course, other image formats like JPG can also be used.

Level-of-detail is a proven approach to manage the trade-off between speed and quality for time-critical visualization [5, 20, 37]. Tapestry uses a similar approach. When a user interacts with the 3D visualization in the web document, rendering requests are made at a lower resolution. After a user pauses, rendering requests are made at a higher resolution. As our own results show, a 432^3 volume of supernova can be requested, rendered, encoded, and received in 0.34 seconds at 512×512 resolution and 0.05 seconds at 64×64 resolution. 3D interaction with embedded volume visualizations can sustain fully interactive frame rates.

Parallel visualization generally takes three approaches: data-parallel, task-parallel, and a hybrid of the two [12, 36]. Our primary concern is how many rendering requests our system can handle (i.e. requests/sec). Given the efficient rendering performance, we chose the task-parallel approach to process rendering requests in parallel. As is commonly done [11], we group worker processes into a two-level hierarchy: the top level is the computing cluster as a whole, the second level is on each computing node. All worker processes within the same node share volume datasets via memory-mapped regions of disk. Following known practices to resolve I/O bottlenecks of volume visualization [18], our implementation includes a dedicated I/O layer as the data manager on each node to manage prefetching.

3 System Architecture

As described above, Tapestry avoids the traditional tight coupling between client and server. Instead, we chose to decouple the application space from the system space (system diagram in Figure 2).

The application space maintains all of the *dynamic states* related to rendering and 3D interaction. The system space is dedicated to answering rendering requests and stays stateless without maintaining any application state information. This design facilitates the transition to *m*-to-*n* mapping from the standard 1-to-1 mapping in typical client-server remote visualization. In essence, the stateless server provides visualization as a web service that answers rendering

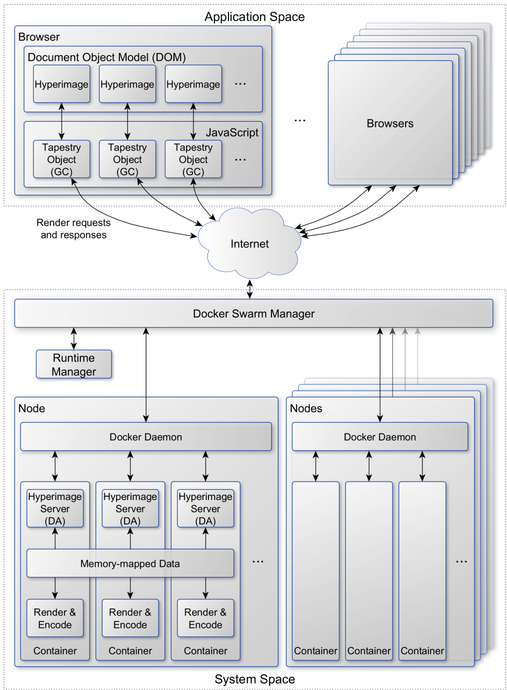


Figure 2: The Tapestry system architecture, which separates the application space and system space. The application space can have many instances, one per active browser. The system space has one instance comprised of many computing nodes. The system space has a unified interface towards all of the application instances.

requests coming from multiple users concurrently. From the server's perspective, there is no dependency between the rendering requests, even for sequential rendering requests from the same user.

The application and system spaces have different life cycles. The system space stays up as long as the web service is up. The application space exists as individual instances, with one instance per each session when a user accesses a web page with embedded 3D visualizations.The application space can have many instances, e.g. a browser tab on a laptop. The system space is a single entity shared by all instances of the application space.

In the application space, an interactive 3D visualization appears as a hyperimage element. Multiple users may independently view a web page with one or more interactive hyperimage elements. Each hyperimage is controlled by an attached Tapestry object in JavaScript, which presents the 3D interaction expected in current scientific visualization. Scripted interactions for the 3D visualization can be included as well. Details are in Section 3.1.

The system space is hosted on a cluster of nodes. These nodes comprise a Docker Swarm, which abstracts the system into a unified web endpoint for all users. The rendering and image encoding functionalities are managed by containers, which the swarm manages altogether as a collection. The system also includes other functionalities such as elastic task handling and automatic resource scaling. More details of the system space are in Section 3.2.

3.1 Application Space

3.1.1 Design Choices

Tapestry's application space resides in the web browser, both for desktop and mobile devices. To embed a visualization in the browser, we could consider using SVG, HTML5 canvas or the 3D-enhanced WebGL canvas [10]. However, we instead chose to use the image tag (`<img>`) for the following reasons.

First, as a result of separating rendering service from interaction, the responses from the rendering service are images. Using an `<img>` tag to display the results is natural, especially when users will likely use the web document for browsing first, and as a web application second, at their discretion. SVG is inappropriate because it is primarily for vector graphics, and better suited for plots and information visualization.

Second, the `<img>` tag is in ubiquitous use by web pages. If it can be made interactive, adopting embeddable 3D volume rendering for presentation visualization would become considerably easier.

Lastly, HTML5 and WebGL canvases are heavyweight elements. The initialization cost can cause performance issues when a web page uses multiple 3D visualizations. The loading time for a hyperimage, however, is equal to the time it takes to load a single image, as on any other website. The runtime cost of WebGL solutions depends on the user's hardware, and may slow down viewing the web document, especially on mobile devices. In contrast, the runtime cost of a hyperimage element is the same as making a web request in a typical AJAX call. AJAX calls are widely-used, standard asynchronous calls for data, for example used for auto-complete in Google's search bar.

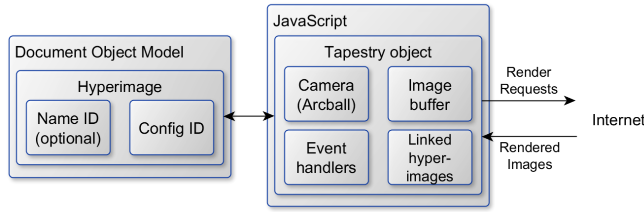


Figure 3: In the application space, each hyperimage element is paired with a Tapestry object. Hyperimages are DOM elements. Tapestry objects handle user interaction and communicate with Tapestry server.

3.1.2 Embedding Interactive Renders in the DOM

Figure 3 shows a closeup of the application space architecture. In this example, we show a single hyperimage in the DOM, but multiple may be present. A hyperimage is a simple `<img>` tag with extended capabilities. As the user interacts with a hyperimage, rendering requests are sent to the system space, and the image content is continuously updated with new renders. Hyperimages provide a simple way to embed interactive volume renders in web pages.

Each hyperimage is controlled by an attached Tapestry object in the `tapestry.js` JavaScript code. The Tapestry object contains a simple graphics context: camera management through arcball, an image buffer for received images, event handlers and a list of other hyperimages that may be linked to the object. Tapestry objects provide interaction logic for hyperimages and are explained in more detail in Section 3.1.3.

Interaction with hyperimages can be through mouse or touch gestures. We also allow additional interaction through *hyperactions*. Any DOM element (e.g. hyperlinks) can become hyperaction-enabled. When clicked on, these elements then perform an action on one or more hyperimages. Hyperactions provide a simple connection between textual content and volume renderings.

Listing 1: Sample code for adding a hyperimage into a webpage

```
<script> $(".hyperimage").tapestry({}); </script>
<img class="hyperimage" data-dataset="supernova"/>
```

Listing 1 shows the full HTML code needed to include a 3D visualization on a page. The second line of Listing 1 shows a simple example of a hyperimage. The `class` attribute identifies the tag as a hyperimage, and the dataset being rendered is added in the `data-dataset` attribute. Note, `data-*` is the standard prefix for custom attributes in HTML5 [31]. Hyperimages become interactive by replacing the source attribute of the tag. When the user is not interacting, a hyperimage is effectively a simple image.

For time series data, a hyperimage can take an optional `data-timerange` attribute. The value of this attribute represents the time step range through which the volume can animate. This range is formatted as <integer>..<integer>. For example, a value of 5..15 would mean that the hyperimage cycles through time steps 5 to 15 when animated.

Hyperimages can also have an optional `id` attribute. The `id` is used to connect hyperimages to hyperactions. JavaScript developers commonly use this attribute to find elements in the DOM.

3.1.3 Interaction Management

All interactions with hyperimages are managed through their corresponding Tapestry objects. A sample initialization of Tapestry objects is shown in the first line of Listing 1. Aside from including our Javascript library and its dependencies, Listing 1 shows the minimal code needed to setup hyperimage support in a web page. Optional settings such as initial camera position can be sent to the Tapestry constructor if needed. Examples with extra settings are shown in Section 4.2.

When a web page loads, Tapestry objects create event handlers to handle mouse and touch gestures on hyperimages. The initialization also creates an arcball object. The arcball is used to translate user gestures to a transformation matrix. Typically in 3D applications, the transformation matrix is used to alter the model-view matrix in the graphics pipeline. However in our case, the 3D scene does not exist in the application space. Therefore, when the user moves the mouse for rotation, we multiply a virtual camera position by the inverse of the transformation matrix and send the updated position to the server in order to obtain a new render. This way, the volume stays in the center of the world space while the camera moves around.

All Tapestry objects communicate with a common host address. When the user interacts with a hyperimage, the attached Tapestry object continuously sends new requests to the server-side and asks for updated renders. During interaction (e.g. when rotating), the object requests low resolution images (64x64 by default) to allow for smoother transitions. When interaction stops, the object requests a high resolution image.

Tapestry objects use the HTTP GET method for requests. As a result, renderings can be saved or shared after interaction just like any image with a valid address. A rendering request takes the form shown in Listing 2. The `DATASET` parameter denotes which configured dataset should be rendered. The camera position is given by <POS_X, POS_Y, POS_Z>, and the up vector is given by <UP_X, UP_Y, UP_Z>. The `QUALITY` denotes whether or not to do a high-resolution render. Finally, additional optional parameters can be added as a comma separated string of key-value pairs. For example, to specify which time step in a time-varying series.

Listing 2: Two example rendering requests sent from the application space. The supernova dataset is requested in both. The following six values represent the camera position and up vector and the last value represents image size. The second request also contains an optional time step parameter.

```
http://host.com/supernova/128.0/-256.0/500.0/0.707/0.0/0.707/256
http://host.com/supernova/128.0/-256.0/500.0/0.707/0.0/0.707/256/
    timestep=5
```

A rendering request can become obsolete if its response arrives after the next rendering request. Network latency, rendering speed, and the user's interaction speed are typical causes of this. Such requests are typically detected and automatically canceled by browsers. The cancellation means that the TCP connection for the request is dropped and the HTTP server would not be able to send a response when the rendering is done. In such cases, HTTP servers wait for a specific timeout period before dropping the connection from their end. If too many of these timeouts accumulate, the server becomes unresponsive. This is due to the fixed size of the response queue that it holds for each client. One way to overcome this issue is to reduce the timeout period on the server-side. Picking an appropriate and universal timeout period, however, can be a challenge. An alternative is to buffer image requests on the client-side to avoid cancellation by the browser. In our system, each Tapestry object buffers the last 512 image requests and their responses in an image object array, oblivious to the user. Note that this buffer resides on the client-side for each user.

Table 1: Supported hyperactions

Action	Description
`position(x, y, z)`	Sets the position of the camera
`rotate(angle, axis)`	Rotates the camera `angle` degrees about the given axis
`zoom(z)`	Sets the relative camera Z position
`link(id1, ...)`	Links the viewpoint of other hyperimages to the current hyperimage's camera
`unlink(id1, ...)`	Unlinks the viewpoint of other hyperimages
`play()`	Animates the time steps of a time series dataset
`stop()`	Stops the time series animation
`switch_config(name)`	Switches to a new hyperimage configuration

Tapestry objects also manage the speed of interaction. The user's mouse movements can cause greater than 30 requests per second. We limit this by only allowing every fifth request to be sent to the server. By measuring the mouse speed of users, we found that this limit results in at most 25 requests per second, suggesting that their maximum speed originally caused 125 requests per second. This is a good trade-off between seamless interaction and low server demand.

In addition to mouse and hand gestures, Tapestry allows another type of interaction: hyperactions. Hyperactions provide a way for the DOM to manipulate a hyperimage without user intervention. A simple use case of a hyperaction is a hyperlink in a text that rotates a hyperimage to a specific viewpoint. Any standard DOM element can be converted to a hyperaction by adding three attributes: the class `hyperaction`, a `for` attribute that denotes which hyperimage should be associated with the action, and a `data-action` attribute describing the action itself. For example, a hyperlink that sets the camera position of a hyperimage is shown in Listing 3. When clicked on, this hyperaction sets the camera position of the hyperimage with the `id` of teapot1 to $(10, 15, 100)$. A full list of supported actions and their syntax is shown in Table 1.

Listing 3: An example hyperaction that sets the camera position to the given position for the teapot dataset.

```
<a class="hyperaction" for="teapot1" data-action="position
    =10,15,100">a new viewpoint</a>
```

3.2 System Space

Figure 4 shows a close-up of the system space architecture within a physical node. In this section, we make a distinction between a *physical node*, a *Docker container*, and a *hyperimage server instance*. A physical node refers to the real machine on which multiple containers may be launched. There may be multiple physical nodes. A Docker container is an in-memory virtual operating system. A single Docker container is shown in Figure 4. Multiple containers may coexist within a single physical node. These virtual systems run a single hyperimage server instance.

A hyperimage server instance runs a web server that manages attributes of given datasets, and handles any rendering requests it receives in sequence. The server instances are elastic, and any available server can handle a request from any user. Due to the containerized nature of the hyperimage servers, the system can automatically scale resource allocation.

3.2.1 Container-Based Rendering Services

Virtualization and containerization are classic concepts in software architecture [23]. Open-source software container platforms have become very powerful lately, as exemplified by the growing popularity of Docker [1] among cloud hosting services. In fact, in the

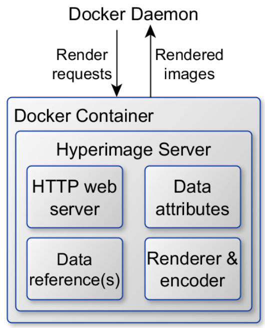


Figure 4: A container is the basic processing unit in Tapestrys system space. Each container runs an instance of the hyperimage server, which contains a web server to receive rendering requests, a renderer/encoder unit to process and answer rendering requests, a reference to the datasets in the shared memory of the computing node, and metadata (data attributes). Containers on the same node are managed by a Docker daemon. Rendering request and response routing is handled by Docker.

scientific computing community, developers of scientific gateways have started to adopt Docker as well [28].

We also chose Docker for its power and simple interface. It provides the ability to create, execute and manage lightweight software containers which are similar to virtual machines. Each container includes a small, stripped-down version of an operating system as well as all the dependencies needed to run an application independently. Multiple Docker containers can run in parallel on the same node.

In a cluster setting, each physical node runs a local Docker daemon, which manages all running containers on that node. Across nodes, we use Docker Swarm as another layer of abstraction on top of a collection of physical nodes, allowing a pool of Docker containers to appear as a unified system with a single entry point, which makes it simple for clients to access.

The Docker Swarm Manager is responsible for monitoring and managing the underlying containers. The Swarm also routes incoming requests to available containers and load balances them. Load balancing is done using an internal Ingress load balancer [17].

In Tapestry, each Docker container runs a Ubuntu instance (488 MB in memory). Inside the virtual systems, we run a hyperimage server instance that handles incoming rendering requests.

3.2.2 Hyperimage Server and Data Attributes

A hyperimage server is initialized once and lives for the lifetime of the web service. A hyperimage server takes a configuration directory during initialization. All valid configuration files – properly formatted JSON files – within this directory are used to provide data attributes for the server instance. These configuration files are easy to set up, and provide basic information about a dataset. An example configuration file is shown in Listing 4.

Listing 4: Example JSON configuration file providing data attributes

```
{
    "filename" : "/path/to/data/magnetic.bin",
    "dimensions" : [512, 512, 512],
    "colorMap" : "cool to warm",
    "opacityAttenuation" : 0.5,
    "backgroundColor" : [38, 34, 56],
    "capImageSize" : [1024, 1024],
}
```

The configuration files consist of a list of key-value pairs. Valid keys and possible values for configuration files are shown in Table 2. These parameters are standard visualization data attributes. The developer is expected to know basic information about the dataset, such as `filename` and `dimensions`, but most others are optional and can revert to default values. `capImageSize` sets the hyperimage's high-resolution size, when interaction is not occurring.

Table 2: Table of key-value pair data attributes available to developers when configuring a hyperimage server. The first three parameters are required, while the rest are optional. Bold indicates default values when an optional parameter is not provided.

Key	Possible values
`filename`	Any valid path to a single file, or a path with wildcards for multiple volumes
`dimensions`	3-element array describing the dataset's extent
`maxImageSize`	2-element array for the rendered image's maximum size
`dataVariable`	A valid variable name for the dataset (for NetCDF files)
`colorMap`	**`grayscale`**, or any provided map
`opacityMap`	**`ramp`**, any provided map, or a custom array of floats
`opacityAttenuation`	Single value $\in [0,1]$ for opacity map dampening, or **1.0**
`backgroundColor`	3-element array describing a color, or **[0, 0, 0]**
`samplesPerPixel`	Single value describing the rendering sample rate, or **1**
`cameraPosition`	3-element array describing 3D coordinates, or **[0, 0, 0]**
`cameraUpVector`	3-element array describing a 3D vector, or **[0, 1, 0]**

Currently the server is able to manage binary and NetCDF files, both very common formats for scientific data. The filename provided may be a path to a single file, i.e. a steady volume, or a path with wildcard characters to describe multiple volumes, i.e. an unsteady time-varying series. Examples `filenames` for a time-varying series could be: `"~/supernova/*.bin"` for all available time steps or `"~/supernova/time_[5-10].bin"` for 5 specific time steps.

During initialization, the datasets referred to by the configurations are loaded. Since each physical node may run multiple server instances, we allow datasets to be memory-mapped when loaded. This allows the physical node's host operating system to maintain an in-memory map of a file that can be given to each server instance. With memory-mapping enabled, all server instances within the same physical node effectively share the data in memory. This reduces data loading costs and allows using multiple configuration files to reference the same dataset without additional overhead.

Attributes about the dataset from the configuration, such as transfer function or data variable, are kept alongside the reference to the data. Multiple configuration files may reference the same dataset, for example, to have different transfer functions applied to the same dataset. This allows for transfer function selection via hyperactions.

3.2.3 Rendering Request Handling

After routing from a common endpoint to a specific Docker container, a rendering request is handled by a hyperimage server. Rendering requests from the client actually ask for an image URL in which various options are embedded. Image requests are processed by the C++ web server, built with the Pistache library [21], by first parsing the options and then rendering the requested image.

Each incoming rendering request contains the dataset, camera position, up vector, and a flag indicating low or high resolution. Low-resolution renders are performed while a user is interacting with a hyperimage, as is done with time-critical visualization. Camera and renderer settings are updated accordingly. The request may also contain parameters like which time step in a time-varying series.

We then render an image of the corresponding volume from the given camera position using the OSPRay renderer. The lifecycle of the OSPRay rendering objects in each server are equal to that of the program itself. Data and rendering attributes are pre-configured per volume during hyperimage server initialization. When the render is complete, we composite the OSPRay framebuffer onto the appropriate background color and encode the image to PNG format using the LodePNG library. We use PNG encoding for its good compression rate and quality.

There is no need to store the image to disk on the server, so the encoding is done to a byte stream in memory. At this point, all information about the camera position and other dynamic state parameters are no longer needed nor held.

The web server receives the PNG byte stream from the rendering module. The byte stream itself is sent as a response with the `image/png` MIME type denoting it as a PNG image. The response takes the reverse path of the incoming request. The Docker Swarm Manager, which routed the request to this container, handles responding to the appropriate user. *The hyperimage server itself remains oblivious to whom it has communicated with.*

3.2.4 Job Assignment

All Tapestry objects from the application space send rendering requests to a common endpoint, where it can be relayed to any available Docker container. The container then passes the request into its hyperimage server instance. In the case of a single container and single hyperimage server, any requests from n users will be queued up by the web server. Each request will block until rendering and network transfer of the image is complete. The low render and encode time makes serialization acceptable for low resolution images. However, larger resolution images may cause noticeable delays.

With multiple containers, any container available across the physical nodes may be selected for any given request. Sequential requests from a single user may also be routed to different containers on different physical nodes. This has several benefits. First, new render requests can be processed while other requests are blocked for I/O, network transfer, or rendering. Second, this is also more efficient for context-switching the CPU resources on the physical nodes. Finally, the elastic routing provides fault tolerance in case of a hyperimage server or physical node going down. Such elasticity is possible due to the back end remaining independent of dynamic states.

3.2.5 Automatic Resource Scaling for Variable Demand

The number of users making requests will vary over time and is hard to predict. To meet the variable demand, we monitor the current load on all containers and scale the number of containers up or down accordingly, through the runtime manager (RM) shown in Figure 2.

On cloud platforms such Amazon EC2 and Google Cloud [3, 13], the RM checks CPU usage across all containers at regular time intervals (e.g. $t_i = 5$ seconds). If the average CPU usage stays above a threshold CPU_{max} for consecutive intervals (e.g. $n_i = 10$), the RM launches a new container. CPU_{max} threshold is often set to 80%. If CPU usage stays below a threshold (CPU_{min}) for consecutive intervals, the RM will close a container. CPU_{min} is typically 10%. After starting or closing a container, the RM will take no more actions for a period of time. These parameters are configurable.

For Tapestry, however, we found that we have to use unconventional settings. Instead of using 80% for CPU_{max} and 10% for CPU_{min}, we should use 10% as CPU_{max} and 5% as CPU_{min}. This is very counter intuitive. The reason in the end was due to how new high performance software, such as OSPRay, use many-core CPUs. OSPRay uses all cores available, even running through Docker containers. When we start with just one container on a physical node, seeing CPU usage go from 5% to 10% means a doubling of workload, which in turn is a more reliable metric to trigger adding a new container. Lastly, we must note, as the throughput test shows (Section 5.2.2), small image size rendering tasks cannot fully saturate hyperimage servers. For larger rendering tasks (e.g. larger image size, larger volume), automatic resource scaling is still beneficial.

4 Application Development

4.1 Integration into Static Web Pages

Hyperimages can be easily added to a web page using HTML tags and a short Javascript function call. To integrate hyperimages into a page, the developer must include the tapestry.js file as well as its dependencies, namely arcball.js, sylvester.js, math.js and jQuery.js. Then, one line of JavaScript needs to be called to initialize all hyperimages, shown in Listing 5. This call creates a Tapestry object per hyperimage tag. Default parameters such as the size of the hyperimage and the initial position of the camera can be sent to the object through the constructor.

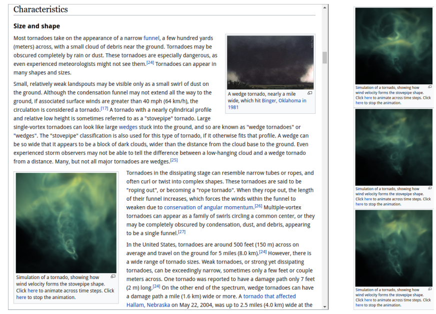


Figure 5: Left: embedded volume render of a tornado simulation (dataset details in Table 3) in a Wikipedia page on tornadoes. The user can start and stop an animated temporal sequence. Right column: example of three time steps. Previously, the page held a static image showcasing the shape of a stovepipe tornado. Now users can interactively see the temporal progression of the natural phenomenon.

Listing 5: The one-line call to create a Tapestry object for every hyperimage on the page.

```
$(".hyperimage").tapestry({});
```

The size of our JavaScript libraries and their dependencies is 148KB, of which 96KB are for jQuery (standard in many websites).

4.2 Example Web Applications

We now go over several application scenarios. In each scenario, we have added hyperimages to an existing web page.

4.2.1 Time-Varying Data Animation (Wikipedia Example)

Many Wikipedia pages can benefit from real interactive volume renderings to help with scientific explanations and engage users. Listing 6 shows the changes needed to include a timeseries hyperimage into a Wikipedia page.

Listing 6: Relevant code for adding an animated timeseries hyperimage to the Wikipedia tornado page.

```
$(".hyperimage").tapestry({
    "host": "http://host.com:port/",
    "width": 256,
    "height": 256,
    "zoom": 300,
    "n_timesteps": 20
});
<img id="timeseries" class="hyperimage" data-volume="tornado" data
    -timerange="0..20"/>
<a class="hyperaction" for="timeseries" data-action="play()"></a>
<a class="hyperaction" for="timeseries" data-action="stop()"></a>
```

Figure 5 shows the Wikipedia page on tornadoes after the modification. The page includes a hyperimage linked to a series of time steps from a tornado simulation dataset. Users can click a hyperaction to play or stop the animation, while still having the ability for 3D interaction with the volume rendering.

4.2.2 Multiple Linked Views (NASA Example)

Here we show a NASA educational outreach page explaining supernovae. The relevant code changes are in Listing 7.

The modified page is shown in Figure 6. The page now contains four hyperimages showing consecutive time steps of a supernova simulation dataset. The views can be linked and unlinked with the hyperaction in the caption. When linked, all four hyperimages rotate or zoom together when a user interacts with any one of them.

Listing 7: Relevant code needed to insert the four linkable hyperimages and hyperaction into NASA's supernova web page

```
<script>
$(".hyperimage").tapestry({
```

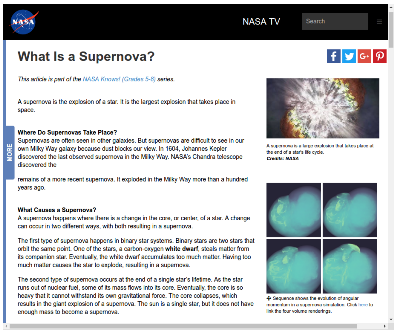


Figure 6: Embedding four consecutive time steps of a supernova simulation into a NASA educational web page (dataset details in Table 3). The four hyperimages (bottom right) can be linked or unlinked using the hyperaction in the caption below it. Previously, the page had only a static figure (top right) showing an artist's rendition. Now users can also interactively explore how a supernova evolves over time.

```
  "host": "http://host.com:port/",
  "width": 128,
  "height": 128,
  "zoom": 300
});
</script>
<img id="s1" class="hyperimage" data-dataset="nova1" />
<img id="s2" class="hyperimage" data-dataset="nova2" />
<img id="s3" class="hyperimage" data-dataset="nova3" />
<img id="s4" class="hyperimage" data-dataset="nova4" />
<a class="hyperaction" for="s1" data-action="link(s2,s3,s4)"></a>
```

4.2.3 Changing Transfer Functions (NOAA/NWS Example)

NOAA and NWS provide vital information regarding climate and weather. Herein, we enhance their storm preparedness web page to show actual WRF modeled storm data using code in Listing 8.

Listing 8: Code needed to insert a hyperimage and hyperaction into the NWS hurrican preparedness page. The hyperaction performs a configuration switch to change transfer function

```
<script>
$(".hyperimage").tapestry({
  "host": "http://host.com:port/",
  "width": 256,
  "height": 256,
  "zoom": 300,
});
</script>
<img id="storm" class="hyperimage" data-volume="superstorm"/>
<a class="hyperaction" for="storm" data-action="switch_config(
      storm_front)"></a>
```

This informational page, enhanced with a hyperimage showing storm features, is shown in Figure 7. In this example, users can interact with a superstorm dataset. Upon clicking a hyperaction, the configuration file is switched. This changes the transfer function to highlight the storm front.

5 Results and Discussion

Here we show the performance of the runtime components of a hyperimage server: the C++ HTTP web server handling requests and responses, and the renderer and encoder. Each instance of the hyperimage server runs in a container on a node with 48 cores (dual-socket Xeon E5-2650 v4, 2.9 GHz) and 128 GB memory.

5.1 Automated Script-Driven Test Setup

To test our system from the client-side in a repeatable way, we simulated user interaction. We used "monkey testing", a standard approach to stress-test web pages. Monkey testing involves simulating clicks, touches, and drags across elements of the page. We used this on hyperimages to simulate user interaction.

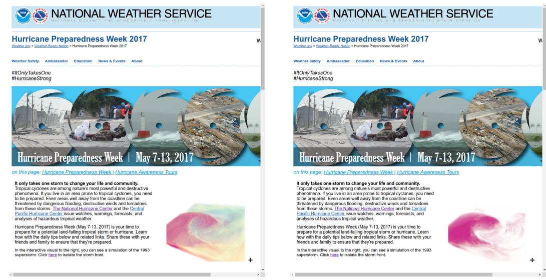


Figure 7: This National Weather Service page provides vital information to those living in hurricane-afflicted regions. Here, users can interact with a model from a WRF ensemble simulation (dataset details in Table 3). A hyperaction switches configurations to guide a viewer to see certain features, using a different transfer function to highlight the storm front. This embedded hyperimage replaced a recorded video on this web page.

Figure 8: We use monkey testing for repeatable and realistic system performance testing. Shown is a screenshot during monkey testing. Using spline paths, we emulate typical user interaction. Each emulated mouse event appears as a red circle.

We used the gremlins.js JavaScript library to control the monkey testing. The test web page contained five hyperimages (for isotropic turbulence, supernova, magnetic reconnection, jet flames, and the Boston teapot). A single hyperimage was randomly chosen to interact with.

To simulate realistic user interaction, we generated splines centered on the hyperimage with anchor points at random positions within the bounds of the hyperimage. We used the smooth.js JavaScript library for spline creation.

The monkey testing simulated mouse-down, mouse-move, and mouse-up events. Mouse-down events were followed by repeated mouse-move events. A mouse-up event could only occur after a mouse-move with a 1% probability.

Each event occurred along the spline. The delay between events was 8 milliseconds, or a total of 125 events per second. Because only every fifth event is processed as a request, this resulted in $125/5 = 25$ requests per second. This request rate is on par with the target expected user interaction speed.

This process simulated a smooth, curved click-and-drag interaction typical with 3D interaction. Figure 8 shows an example of the monkey testing on the magnetic reconnection dataset. Note that the render is low-resolution because the test is mid-interaction.

All testing in this work used monkey testing to perform repeatable tests, while still mimicking realistic user interaction scenarios.

Table 3: The datasets used for our tests. For time-varying data, varying time steps were used during testing.

Dataset	Size per Volume	Spatial Resolution	Time Steps
Boston teapot with lobster	45 MB	$356 \times 256 \times 178$	1
Isotropic turbulence [9]	64 MB	$256 \times 256 \times 256$	1
Jet flames [35]	159 MB	$480 \times 720 \times 120$	122
Superstorm [25] (1 run)	201 MB	$254 \times 254 \times 37$	49
Tornado [33] (wind velocity)	257 MB	$480 \times 480 \times 290$	600
Supernova [6]	308 MB	$432 \times 432 \times 432$	60
Magnetic reconnection [14]	512 MB	$512 \times 512 \times 512$	1

Table 4: Benchmarking results for rendering requests. The round-trip time for each request includes render, encoding and transfer time to and from the server. Based on these results, we render a 64×64 image during interaction. The system renders idle images at the size provided in the configuration, but capped at 1024×1024 to avoid bottlenecks the Docker swarm.

Dataset	Image size	Rendering time (s)	Encoding time (s)	Round-trip time (s)
Turbulence	64×64	0.009	0.010	0.042
	128×128	0.017	0.019	0.058
	256×256	0.037	0.048	0.108
	512×512	0.110	0.140	0.260
	1024×1024	0.300	0.540	0.807
	2048×2048	0.939	1.813	2.517
Supernova	64×64	0.012	0.013	0.048
	128×128	0.023	0.023	0.066
	256×256	0.049	0.054	0.196
	512×512	0.133	0.163	0.338
	1024×1024	0.373	0.592	1.228
	2048×2048	1.412	2.142	3.108
Magnetic	64×64	0.018	0.018	0.064
	128×128	0.036	0.031	0.106
	256×256	0.073	0.081	0.250
	512×512	0.186	0.210	0.596
	1024×1024	0.649	0.798	1.870
	2048×2048	2.170	2.885	6.928

5.2 Hyperimage Server Response Time

We benchmarked the rendering and encoding process using three variables that affect render time: image size, level of attenuation of a ramp opacity map, and number of samples per pixel. We tested 6 image sizes (64^2, 128^2, 256^2, 512^2, 1024^2, and 2048^2), 4 attenuation values (1.0, 0.5, 0.1, and 0.01), and 4 sampling rates (1, 2, 4, and 8). We tested each combination of these parameters, resulting in 96 test cases. We repeated each of the 96 cases 10 times with the camera at a randomized position to simulate the effects of the volume being at different distances and angles. We took the average time taken for 10 renders as the result for a given test case. To see the effect of image sizes, we then averaged the times for each image size. This simulates possible variation in image quality within same-sized images.

We tested using three datasets: supernova, isotropic turbulence, and magnetic reconnection. The datasets are described in Table 3. All three datasets are structured grids of floating point values. These datasets were chosen because they have different extents and sizes, but also because they each contain layers of fine detail that may be challenging to render. To test rendering time, each image was rendered to OSPRay's internal framebuffer and was then discarded to avoid buffer copy or encoding time. We then tested the encoding time (without saving to disk) separate from render time. Results are shown in Table 4.

The fastest rendering case was unsurprisingly 64×64 image size with the turbulence dataset, the smallest dataset chosen. Within the test cases that used a 64×64 image, attenuation of 0.1 and sample rate of 1 resulted in the fastest renders at 0.003 seconds, over 300 frames per second. At such a small size, the render time has very little deviation from the mean. On the other hand, the slowest renders occurred with 2048×2048 images with the magnetic dataset, the largest chosen dataset. The slowest recorded average render time was 6.394 seconds, with 0.1 attenuation and 8 samples per pixel. Combining rendering and PNG encoding, total computing time approaches 5 seconds on average.

Based on these results, we recommend using 64×64 images during interactions. For the idle image, the system renders the image at the size as described in the DOM of the webpage, but cap the size at 1024×1024 (i.e. specified as the `capImageSize` in the data attribute configurations) to avoid tying up containers in the swarm. Note that 1024×1024 is quite large, even on modern screens. We also found very little difference between renders with more than 2 samples per pixel, and suggest using 1 or 2 samples during use.

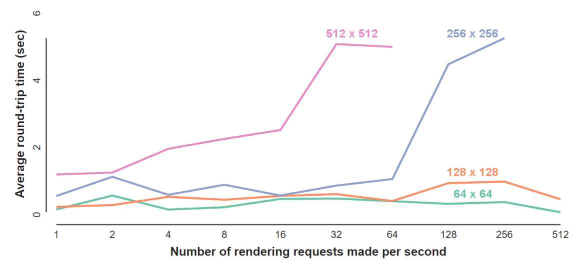


Figure 9: System throughput results showing request rate vs. response time for four different image sizes: 64×64 (green), 128×128 (orange), 256×256 (blue), and 512×512 (purple). For low resolutions, there is no degradation in response time even at high request rates. This led us to choose 64×64 as the default interaction resolution.

5.3 Hyperimage Server Throughput

We also tested the scalability of our system space design in regard to the system's overall throughput. The experiment is set up like a stress test, as commonly done in web environments.

We used `curl` to programmatically generate rendering requests and control how many requests are made per second. The test target is the full 3-node cluster: 20 containers per physical node, 60 containers in total. We used the same datasets as the previous test in Section 5.2. For each dataset, we generated rendering requests for four image sizes: 64^2, 128^2, 256^2, and 512^2.

The test started issuing one rendering request per second for 5 seconds, then doubled the request rate every 5 seconds. In total, we tested for request rates of: 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 requests/second. Our goal was to understand the relationship between request rate (throughput) and round-trip response time (performance).

For each test setting (i.e. image size and request rate), we average the response time collected from all three datasets together, in order to show an overall system throughput under a mixture of different sizes of rendering jobs. Figure 9 shows the scaling curves for the four image sizes: 64×64 (green), 128×128 (orange), 256×256 (blue), and 512×512.

As shown, when the image sizes are small (i.e. 64×64 (green), 128×128 (orange)), there is no degradation in response time as request rate increases. This solidifies our design choice of using 64×64 size for interactive image delivery, and only opt to a large size for idling images. For 256×256 image size, response time starts to degrade when the request rate reaches 64 requests/second. For 512×512, the degradation takes place sooner, around 16 requests/second.

5.4 Hyperimage server scaling

We also tested the scalability of the Tapestry architecture to show how the system space reacts to growing and shrinking audience size. The scalability curves are shown in Figure 10 over the course of 24 minutes and 15 seconds.

We created a test page containing five hyperimages of different sizes. Upon loading, one of the five hyperimages was chosen for interaction using monkey testing. Groups of two clients connected to the server separated by 30 seconds, as shown by the blue line in Figure 10. There were a maximum of 24 simultaneous clients.

As clients joined, the runtime manager launched new containers to handle the load, shown by the orange line. The number of containers plateaued at 13. Average CPU usage across Docker containers is shown by the green line. Usage stabilizes as containers are added. When too few containers are present for the given number of users, there is greater variability to the CPU usage.

A pattern noticed during the scalability test was that the number of containers was roughly half of the number of concurrent users. The plateau region shows this pattern well, as 13 containers were able to stably serve 24 users. These 13 containers were distributed across

Table 5: Summary of the pros and cons between client-side rendering, stateless, and stateful server-side rendering.

Architecture	Pros	Cons
Client only	Does not require external server, existing frameworks	Requires data transfer initial overhead, relies on potentially inadequate local resources, relies on approximated volume rendering techniques via WebGL
Client & Stateful server	Does not rely on client resources, no transfer time, low interaction overhead, dedicated server resources	Requires server-side setup, requires consistent connection to server, does not scale well for many users
Client & Stateless server	Does not rely on client resources, no transfer time, low interaction overhead, multi-user *m*-to-*n* mapping	Requires server-side setup, requires consistent connection to server

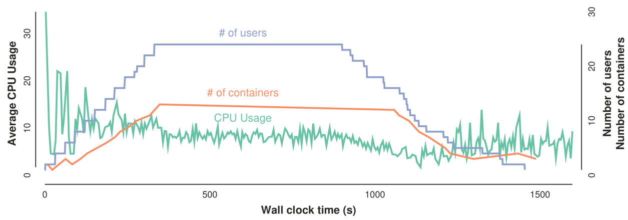


Figure 10: Example results of automatic resource scaling. The X-axis is wall clock time elapsed since the start of the test (seconds). The test started with two application instances, with two more added every 30 seconds. The first two instances have the longest life spans, and subsequent ones have linearly decreasing life spans (details in Section 5.4). (1) Green: average CPU usage of the swarm by percentage (left Y-axis); (2) Orange: the number of active containers according to varying demand (right Y-axis); (3) Blue: the number of concurrent users over time (right Y-axis).

three physical nodes. They served approximately 600 requests per second during the test.

5.5 Discussion

A visualization that allows real 3D interaction can achieve better user engagement and provide more information than a still image or video can provide. In this respect, Tapestry helps make 3D visualization more accessible. The model used by Tapestry also simplifies how a visualization can be hosted as a web service using open-source industry standards, such as Docker, jQuery, and OSPRay.

Comparison to VTK.js. As previously mentioned, client-side systems such as VTK.js have limitations on dataset size and render quality. They also rely on potentially inadequate local resources. Additionally, client-side solutions have significant load time and runtime overheads. For example, a 308 MB supernova volume would need to be pushed to each user. If the user is on a mobile device, this is infeasible. Render performance would be slow on a mobile device as well, leading to an unresponsive web page. Table 5 summarizes the pros and cons between client-side rendering versus the stateless remote rendering in Tapestry.

Adoption. Tapestry's server can be hosted on Amazon AWS, Google Cloud, or other similar platforms. As is, hyperimage-enabled web pages can be viewed on a desktop or mobile browser. However, the application space is not restricted to residing inside a browser. Because the web endpoint served by the Docker swarm follows standard HTTP protocols, a mobile app or desktop application can use the service as long as the application can communicate with the remote rendering service (e.g. via Linux's `curl`).

Data Sources. Hyperimage elements provide a concise abstraction that helps make data-intensive volume rendering transparent to the application developer. However, it is debatable whether archival data management should be made transparent to application designers. If application developers need to be proactively aware of the data archival services, we believe the model to integrate remote web visualization with data archives is a fruitful direction, as piloted by ParaViewWeb and MIDAS [15].

Scalability and Performance. Two potential factors affect Tapestry's performance. First, the number of containers is directly related to how many concurrent requests can be answered. Second, Tapestry uses OSPRay as the renderer. OSPRay is optimized for many-core architectures by way of threading and vectorization. Given the finite number of cores available on each computing node, increasing the number of requests to be handled simultaneously (i.e. increasing containers) and increasing the rendering performance (i.e. increase the cores used by OSPRay) become competing objectives. For the datasets used in this paper, 20 containers / node (48 cores per node) seems to work well. To volume render larger datasets, however, we may be required to reduce the number of containers.

Assuming medium sized datasets (such as the supernova dataset we used for testing), a web page containing 5 hyperimage elements, and that each active viewer's browser emits 5 rendering requests/second (one from each of the 5 hyperimage elements), our 3-node cluster can support roughly 100 concurrent viewers on that web page. For today's scientific computing, this system is small; scalability on a bigger system requires further evaluation.

In larger scenarios with many users, the single endpoint that clients talk to could become a bottleneck. However, in reality, given that Docker uses an Ingress load balancer, any node can be the endpoint. Moreover, multiple Docker swarm managers can be set up for additional fault tolerance.

6 Conclusion and Future Work

In this work, we have studied how to encapsulate volume rendering into an easily-accessible service that can be non-invasively embedded into existing popular websites about science topics. The challenge we focused on was the scalability of audience. The methodology that guided our architecture design is to separate application space from system space, allowing for multiple concurrent users.

The application space presence of embedded volume renders is about two lines of code per hyperimage. All other management issues, such as user interaction, rendering requests, response retrieval, etc. are abstracted away. The system space rendering service is stateless and refactored into virtualized containers. These containers can then leverage the same microservice management infrastructure on today's cloud hosting facilities with elastic and on-demand scaling.

In the future, we plan to add support for unstructured grid data. We also plan to add full six degrees-of-freedom camera movement, which requires sending new camera look-at attributes to the server. We would also like to investigate the potential of caching renders. On the client-side, the local buffer can be easily reused as a cache, however more investigation needs to be done for the server end. Additionally, we plan to expand the functionality of Tapestry to deliver a larger variety of scientific visualization applications on the web. Collaborative editors and interactive project management tools are two examples of potential applications. Another area of potential is web-based scientific storytelling for educational uses.

Acknowledgments

The authors would like to thank the anonymous reviewers of this and previous versions of the manuscript for their valuable comments and suggestions. The authors are supported in part by NSF Award CNS-1629890, Intel Parallel Computing Center (IPCC) at the Joint Institute of Computational Science of University of Tennessee, and the Engineering Research Center Program of the National Science Foundation and the Department of Energy under NSF Award Number EEC-1041877. The supernova data set is made available by Dr. John Blondin at the North Carolina State University through DOE SciDAC Institute for Ultrascale Visualization.

References

[1] Software Container Platform - Docker: https://www.docker.com/.

[2] *NVIDIA IndeX*, 2016.

[3] Amazon Web Services. Scaling based on metrics, 2017 (accessed March 20, 2017). `http://docs.aws.amazon.com/autoscaling/latest/userguide/policy_creating.html`.

[4] U. Ayachit. The Paraview guide: a parallel visualization application. 2015.

[5] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Proc. of IEEE Visualization*, pp. 135–142. IEEE, 2005.

[6] J. M. Blondin and A. Mezzacappa. Pulsar spins from an instability in the accretion shock of supernovae. *Nature*, 445(7123):58–60, 2007.

[7] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[8] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil. VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372. Oct 2012.

[9] D. Donzis, P. Yeung, and D. Pekurovsky. Turbulence simulations on $O(10^4)$ processors. In *TeraGrid 2008 Conference*, 2008.

[10] A. Evans, M. Romeo, A. Bahrehmand, J. Agenjo, and J. Balt. 3d graphics on the web: A survey. *Computers and Graphics*, 41:43 – 61, June, 2014.

[11] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pp. 55–63. IEEE, 2001.

[12] J. Gao, J. Huang, C. R. Johnson, and S. Atchley. Distributed data management for large volume visualization. In *Proc. of IEEE Visualization*, pp. 183–189. IEEE, 2005.

[13] Google Cloud Platform. Scaling based on cpu or load balancing serving capacity, 2017 (accessed March 20, 2017). `https://cloud.google.com/compute/docs/autoscaler/scaling-cpu-load-balancing`.

[14] F. Guo, H. Li, W. Daughton, and Y. H. Liu. Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Physical Review Letters*, 113(15):1–5, 2014. doi: 10.1103/PhysRevLett.113.155005

[15] J. Jomier, S. Jourdain, U. Ayachit, and C. Marion. Remote visualization of large datasets with midas and paraviewweb. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pp. 147–150. ACM, New York, NY, USA, 2011. doi: 10.1145/2010425.2010450

[16] S. Jourdain, U. Ayachit, and B. Geveci. Paraviewweb, a web framework for 3d visualization and data processing. In *IADIS international conference on web virtual reality and three-dimensional worlds*, vol. 7, p. 1, 2010.

[17] P. Kanuparthy, W. Matthews, and C. Dovrolis. DNS-based ingress load balancing: An experimental evaluation. *CoRR*, abs/1205.0820, 2012.

[18] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross. Toward a general i/o layer for parallel-visualization applications. *IEEE Computer Graphics and Applications*, 31(6):6–10, 2011.

[19] Kitware. Vtk.js, 2017 (accessed June 10, 2017). `https://github.com/Kitware/vtk-js`.

[20] X. Li and H.-W. Shen. Time-critical multi-resolution volume rendering using 3d texture mapping hardware. In *Proc. IEEE/ACM Symp. on Volume Visualization and Graphics*, pp. 29–36. IEEE, 2002.

[21] Mathieu Stefani. Pistache http server, 2017 (accessed June 16, 2017). `http://pistache.io`.

[22] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proc. of IEEE Symp. on Volume Visualization*, pp. 81–90. ACM, 2000.

[23] C. Pahl. Containerization and the PaaS cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[24] V. Pascucci, G. Scorzelli, B. Summa, P.-T. Bremer, A. Gyulassy, C. Christensen, S. Philip, and S. Kumar. The visus visualization framework. *EW Bethel, HC (LBNL), and CH (UofU), editors, High Performance Visualization: Enabling Extreme-Scale Scientific Insight, Chapman and Hall/CRC Computational Science*, 2012.

[25] J. Sanyal, S. Zhang, J. Dyer, A. Mercer, P. Amburn, and R. Moorhead. Noodles: A tool for visualization of numerical weather model ensemble uncertainty. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1421–1430, 2010.

[26] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-oriented Approach to 3D Graphics*. Kitware, 2006.

[27] R. Sisneros, C. Jones, J. Huang, J. Gao, B.-H. Park, and N. Samatova. A multi-level cache model for run-time optimization of remote visualization. *IEEE transactions on visualization and computer graphics*, 13(5), 2007.

[28] J. Stubbs, W. Moreira, and R. Dooley. Distributed systems of microservices using docker and serfnode. In *Science Gateways (IWSG), 2015 7th International Workshop on*, pp. 34–39. IEEE, 2015.

[29] G. Tamm and P. Slusallek. Plugin free remote visualization in the browser. In *SPIE/IS&T Electronic Imaging*, pp. 939705–939705. International Society for Optics and Photonics, 2015.

[30] G. Tamm and P. Slusallek. Web-enabled server-based and distributed real-time ray-tracing. 2016.

[31] W3C. Embedding custom non-visible data with the data attributes, 2017 (accessed March 21, 2017). `https://www.w3.org/TR/2011/WD-html5-20110525/elements.html#embedding-custom-non-visible-data-with-the-data-attributes`.

[32] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil. Ospray-a cpu ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, 2017.

[33] R. Wilhelmson, M. Straka, R. Sisneros, L. Orf, B. Jewett, and G. Bryan. Understanding tornadoes and their parent supercells through ultra-high resolution simulation/analysis.

[34] Q. Wu, J. Gao, M. Zhu, N. S. Rao, J. Huang, and S. Iyengar. Self-adaptive configuration of visualization pipeline over wide-area networks. *IEEE Transactions on Computers*, 57(1):55–68, 2008.

[35] C. S. Yoo, R. Sankaran, and J. H. Chen. Direct numerical simulation of turbulent lifted hydrogen jet flame in heated coflow. 2007.

[36] H. Yu, K.-L. Ma, and J. Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proc. of the ACM/IEEE Supercomputing Conference (SC'04)*, pp. 49–49. IEEE, 2004.

[37] C. Zach, S. Mantler, and K. Karner. Time-critical rendering of discrete and continuous levels of detail. In *Proc. of the ACM Symp. on Virtual Reality Software and Technology*, pp. 1–8. ACM, 2002.