Fault-tolerant peer-to-peer image rendering system

Nathan S. Hartzler

Computer Science Undergraduate Program, Missouri State University. Springfield, MO 65897, United States

Abstract

An efficient and reliable image rendering system is a critical component for completing real-world computer-generated projects on time in the 3D and visual effects domain. Most image rendering systems, or render-farms, are designed as a centralized, distributed work system with (*N) worker* nodes receiving jobs from a single central *manager* server. The manager is solely responsible for receiving new job requests, sending those jobs to the worker nodes, and tracking the failure/success of the jobs. These systems inherently have a single point of failure as well as a single point of congestion. A fault-tolerant peer-to-peer image rendering system is proposed in this paper. The system decentralizes the parallel rendering system by making each node both a worker, able to complete jobs, and a manager, able to submit and track jobs. A method of synchronizing the application state between each peer is proposed and tested. The decentralized system is shown to provide promising fault-tolerance for render farms in the 3D work domains.

---

1. Introduction

The point of image rendering in the 3D and visual effects domains is to consider all transformations applied to an image via software and to calculate the color value of each pixel in the final image. This is a computationally complex process that becomes more complex when considering a sequence of images, as in a 3D film. The large time complexity for image rendering is one of the most notable problems. For example, in a 34-second scene for Disney/Pixar's Inside Out, the final render took 33 hours per frame to complete [1]. That means that the 34-second scene at 24 frames per second took 26,928 hours or just slightly over 3 years to render output images. At these speeds,

a single, one-core machine would require 635 years to render a 2-hour film. This time complexity is why distributed image rendering systems ("render farms") were invented.

A render farm typically comprises a diverse group of networked computers running specialized 3D and Visual Effects software such as Autodesk Maya [2], Adobe After Effects [3], or Blender [4]. Software installed on each computer (worker) listens on a specified port for network job commands from a central server. Users submit jobs to the central sever which parses the given sequence of frames into a list of smaller "chunks." These smaller chunks are distributed to all available workers. Workers begin rendering as soon as they receive a new chunk. For example, to render "Monster's University" Pixar used a render farm "composed of 2000 machines, and 24,000 cores" [1]. It took two years to render the 158,400 frames in the film so each of the workers may have processed 6 to 7 frame chunks each.

The key parts of a render farm are fast network connections between workers and the manager, powerful CPUs and GPUs in the workers, and close to a 99.99% system uptime. Traditionally a job manager is setup using a centralized topology. This becomes a single point of failure for the system. If the centralized job manager is unavailable, new jobs cannot be submitted and current jobs will not distribute frame chunks to the workers. For this reason, a more fault-tolerant peer-to-peer render farm is proposed. In the proposed system each worker also acts as a manager for the render jobs.

2. Related Work

A system related to the approach proposed in this paper is a system called P2Pixie [6]. P2Pixie is a hybrid-decentralized rendering system attempting to provide an open-source solution to communities "which often lack the resources to set up and administer a dedicated render farm." [5]. The application state management is not discussed in detail. The main difference from what is described in P2Pixie, and the system proposed here, is the structure of the peer network since "one of the peers must be identified as a Rendezvous peer, analogous to supernodes in other

partially decentralized P2P systems" [5]. This “Rendezvous” peer connection creates an undesirable point of failure since it is similar to a central manager. The fault-tolerant system proposed only depends on a set of centralized servers for initial worker startup and does not rely on partially centralized peers throughout the life of the system.

3. System Connections

The connections between workers comprise a fully connected mesh graph, see Figure 1. Each worker communicates with every other worker. A central server, or set of replicated servers, provides a list of known workers. These servers also act as non-rendering peers to provide a copy of the application state, described in *section 4B.*

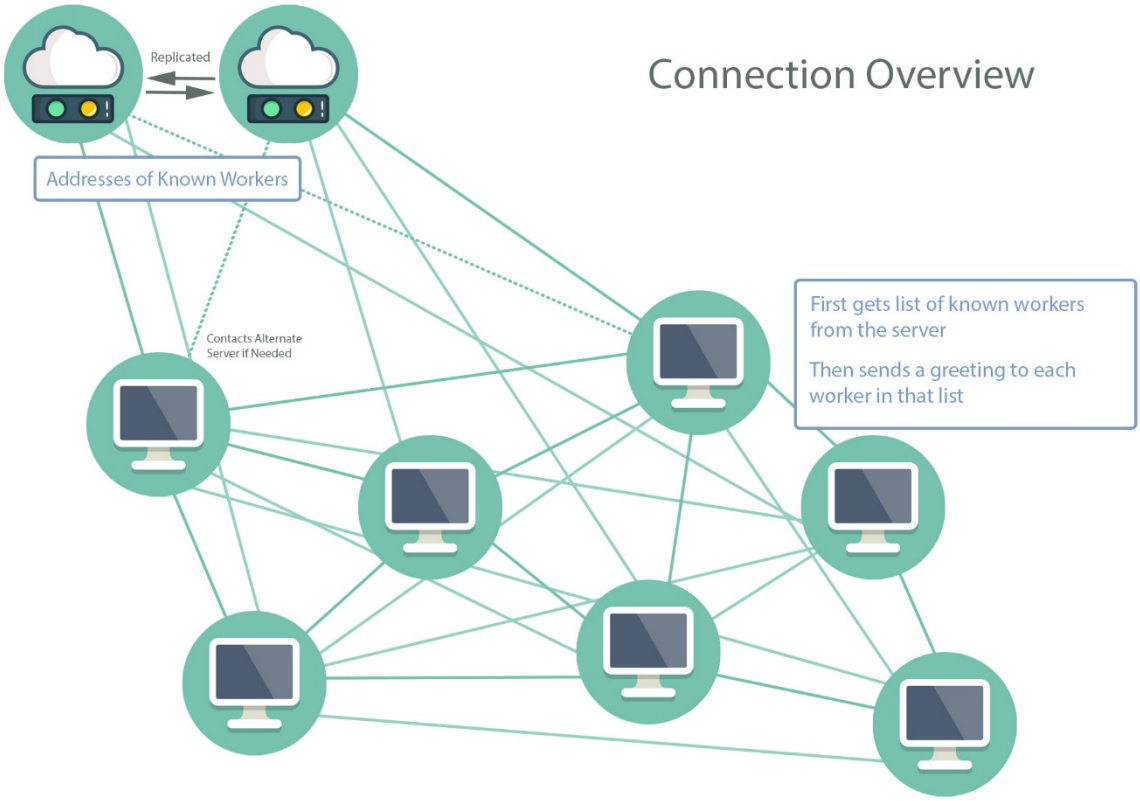

Figure 1

When a new worker joins the system, it first contacts one of the replicated central servers. The server adds this worker to its list of known workers and responds with the Known Worker List, (KWL). The KWL is a unique set containing non-repeating workers identified by the worker’s GUID. The worker stores the KWL in its local database.

It then creates a TCP connection with each of the workers on the KWL and sends a greeting message with the IP and other info about itself. When the other workers receive this greeting message, they add the greeting-worker to their own KWL in their local databases. In this way, the peer mesh network is maintained without the need for each worker to contact the central servers for a Known Worker List. When the worker is being shut down, the worker will attempt to send a disconnect message to each of the workers on its KWL. However, these disconnect messages may not always be generated in time before the worker shuts down. Therefore, an offline worker on the KWL is ignored when sending messages. When the offline worker comes back online, it rejoins the system as a new worker.

4. Peer-to-Peer Application State Problem

Once connections are established between all workers, the system is ready to accept render jobs. However, there is a problem unique to peer-to-peer networks that needs to be solved before the system can function. If a job is submitted to an individual worker, and that worker fails to communicate that job submission with its peers, then the worker's local database (the application state) becomes out of sync with the application state of its peers. This can be solved by using a central database that each worker connects to. This is the approach first used by Napster [6], a once popular p2p music sharing platform. Though this approach is simpler to grasp, it produces a critical single point of failure as well as a single point of congestion in the network. So then, if a local database at each worker is preferred, how is synchronization maintained between the application state of each worker?

To maintain sync of the application state among all workers, some form of communication between the workers needs to take place. It should also be noted that *relevant* changes to the local state should be communicated to every other worker. Non-relevant state changes would include things like when a user-interface view changes. A possible, although naïve, approach to the problem is to pass the entire state over the network every time a worker's

state changes. This approach is unrealistic on slower or larger networks and it becomes impractical if the application state is large. Another approach needs to be found.

4A. Research Contribution

There exists a local state management method wherein any upcoming change-of-state is formalized into a set of properties called an *Action* that is then applied to a set of pure functions that take the current state and the Action and output the changed state. This method creates a separation of the application state from the user or programmatic Action that will cause a change in state. The JavaScript library, ReduxJS [7] is one example of this state management method. ReduxJS was influenced by Facebook's Flux state management [8] and by the Elm programming language [9]. What these all share is separating application state from the actions that will change that state. This method is useful in solving the problem of synchronization among worker peers.

If each worker is using the described state-management method, then a novel solution to the state sync problem is to communicate each relevant state-changing Action to the other workers over the network. Here in this approach will be called Distributing State-Changing Actions (DSCA). When using this approach each worker maintains its local application state and communicates only its relevant changes of state to the other workers through small-payload Action messages. The Action messages have a consensus of *Types* and a formal structure which includes a *Payload*.

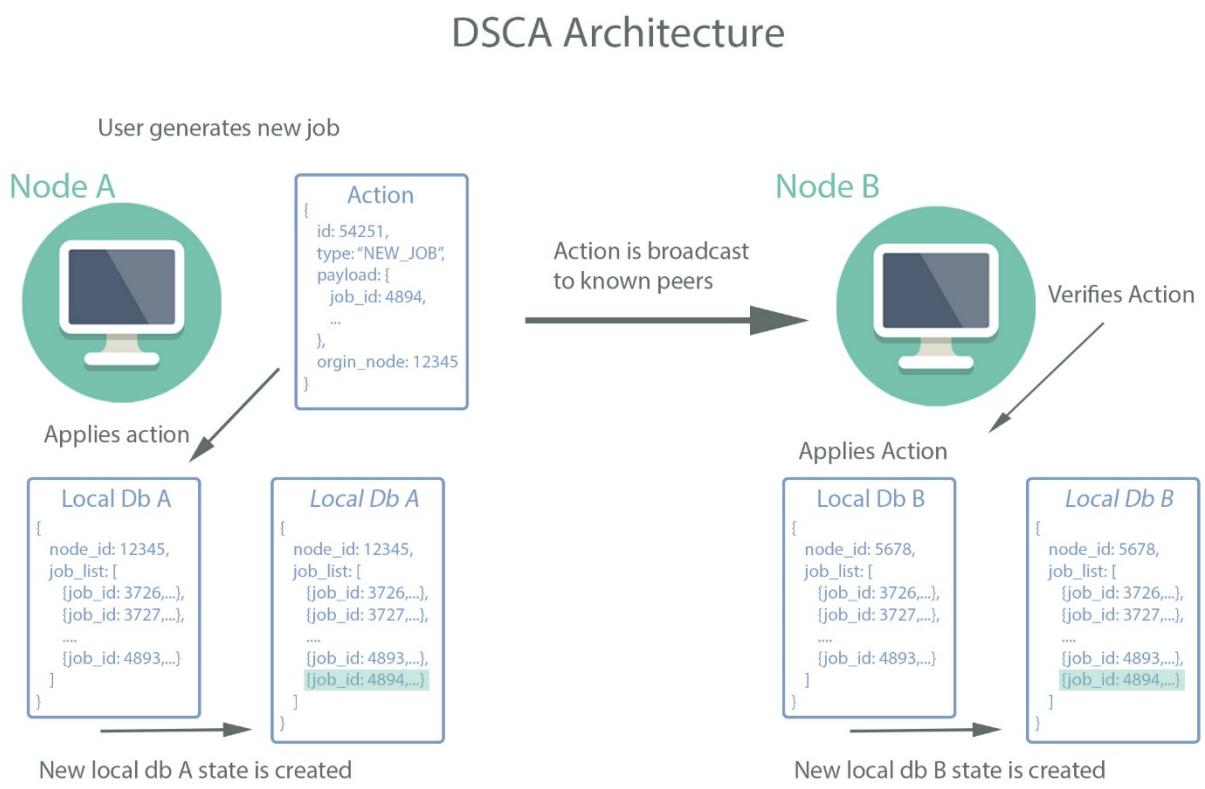

Figure 2

As the example shows in Figure 2, when a user submits a new render job the Action is generated with the type of "NEW_JOB" and the Payload containing the properties of that new job that need to be inserted into the local database. A worker passes this Action to it's own state-manager but also sends a message to each of the workers in the KWL. When the other workers receive a new Action, they pass the Action to their own state managers which output the changed state.

4B. Central Servers as Peers in the Worker Network

In the system proposed, the central servers maintain a copy of the KWL to send to a worker during its initial boot in the system. Using the DSCA approach, the central servers can also maintain a copy of the synced application state. Then, when a new worker is requesting the KWL, it can also receive a copy of the current system's state. In this way, a new worker can join the network and immediately complete active render jobs in the system.

5. Pros and Cons of DSCA

Two of the obvious disadvantages to the DSCA approach are 1) The data synchronization may not happen in real-time and 2) The workers can become "chatty" with Action messages and congest the network. The first disadvantage can be illustrated by considering two workers over a slow network connection. Let's say that the user on Worker A submits a job that requires *all available workers*. This would generate Action-A and be communicated across the network. At the same time, the user on Worker B is submitting a job that requires *all available workers.* Worker B is on a slow network connection and has not yet received Action-A. From Worker B's perspective, the job that generated Action-B should be first in the system's queue. It is only once Action-A is received from the network that Worker B knows that all the other workers are unavailable. The DSCA approach provides "best-effort" state synchronization between workers and is a tradeoff between 100% state synchronization guarantees available in a centralized database and overall system fault-tolerance provided by decentralization.

The second disadvantage can be avoided if the system is carefully implemented to only communicate relevant Action messages that are never sent to the same worker twice. In the implementation used for the Experimental Analysis, *section 6*, there is effort put towards minimizing the number of Action messages sent across the network. As an example, the numerous logs created during the 3D render process are kept on the local database and not automatically communicated. They are only sent between workers if a specific "GET_LOG" action is sent. This helps reduce chattiness over the network.

One major advantage of the DSCA approach is the flexibility it provides for each local database. Although the Action messages need to have the same consensus of Type and structure, there are unlimited variations available to the local database state in that not every worker need implement the same set of state-change functions for a given Action. This allows individual nodes to receive the same messages being sent in the system but behave differently from their peers if needed. They receive the same Action messages as the workers but the shape of their state only contains the few branches that are needed to pass along to any new workers in the system.

6. Experimental Analysis

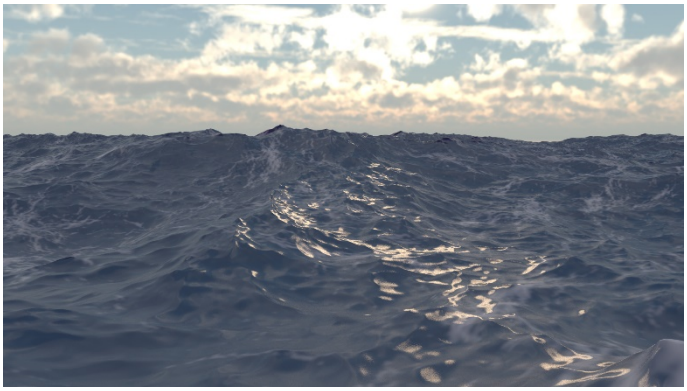

Figure 3 Frame from the Ocean Simulation Scene [15]

The proposed render system was tested on three fronts: DSCA Synchronization, Render Job Completion, and Fault Tolerance. The system was implemented using JavaScript on the NodeJS Runtime [10] using the ElectronJS Desktop Framework [11]. The resulting open-source implementation was named *peerfarm* and is available on GitHub [12] https://github.com/PocketOfWeird/peerfarm. Tests were run using 3 to 6 networked workers and 1 central server. The tests used an Ocean Simulation scene, Figure 3, created with Autodesk Maya [2] and the Arnold Render Lighting system [13].

6A. DSCA Synchronization

Hypothesis: State synchronization is maintained when using DSCA.

Procedure: During a 60-frame render, the application state was exported to a separate text file upon each state change. This text export procedure ran on the state changes from 6 peers during the render job. After completion of the render job, a *diff* algorithm was run on each of the text files, compared by creation date.

	W1	W2	W3	W4	W5	W6
W1						
W2						
W3						
W4						
W5						
W6						

Figure 4

Results: The diff results in Figure 4 show no differences between the shared state of the exported text files. W1, …, W6 are the workers 1-6 and the cell is green if the diff found no difference.

6B. Render Job Completion

Hypothesis: The proposed distributed system decreases render completion times compared to a centralized system.

Procedure: The peerfarm implementation was given a series of jobs using 5 frame chunks but varying in number of total frames and number of available workers. There was a 15-frame job running on 3 workers and then a 30-frame job running on 6 workers. These jobs were run identically, using the same worker computers, on a centralized render system, RenderPal by Shoran Software [14].

Figure 5

Results: Peerfarm ran longer than RenderPal in all time trials as shown in Figure 5. This may not be due to the design of the system but rather the JavaScript implementation. Runtime was not a major decision factor when implementing the proposed system in the NodeJS Runtime [10] and ElectronJS Desktop framework [11]. Presumably these coding conveniences caused runtime bloat due to the large memory space used by ElectronJS and the single-threaded nature of NodeJS.

6C. Fault Tolerance

Hypothesis: Workers continue to accept and render frame chunks if the server crashes.

Procedure: A 60-frame render job was submitted in 5-frame chunks to the peerfarm. After the submission was received by the 6 designated workers the central server, which initiates new workers, was halted. For RenderPal, the same 60-frame job was submitted. After the submission was received by the same 6 workers, the RenderPal Server was halted.

Results: The peerfarm workers continued to communicate with one another and continued rendering to the second set of 5-frame chunks. The centralized RenderPal workers, seen in Figure 6, were unable to retrieve the second set of 5-frame chunks from the server.

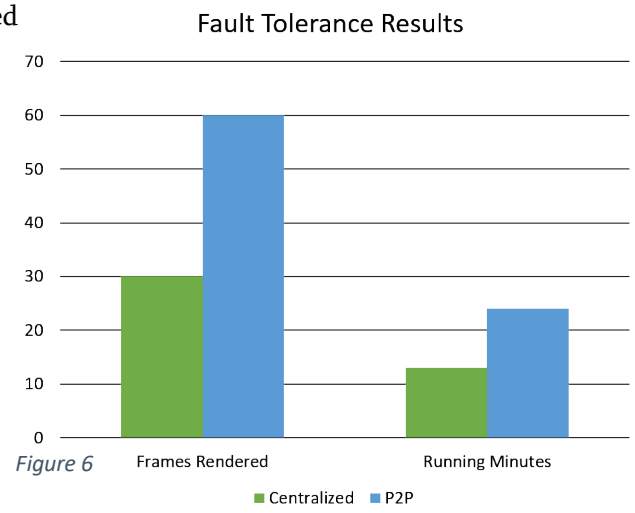

Figure 6

7. Conclusion and Future Work

The proposed peer-to-peer image rendering system is more fault-tolerant than an analogous centralized system. The running time of the proposed system needs more research with a more runtime-conscience implementation. The novel Distributing State-Changing Actions approach to distributed application state synchronization does maintain best-effort state synchronization across peers in the system. This approach is likely generalizable to other p2p applications. Future work could be done to integrate the work of p2p file sharing systems and eliminate the need for a central file store for the Render Scene files and Image Output Directories. This may lead to even greater decentralization and fault-tolerance in the system.

References

[1] "Rendering", *Sciencebehindpixar.org*, 2018. [Online]. Available: http://sciencebehindpixar.org/pipeline/rendering.

[2] "Maya | Computer Animation & Modeling Software | Autodesk", *Autodesk.com*, 2018. [Online]. Available: https://www.autodesk.com/products/maya/overview.

[3] "Adobe After Effects | Visual effects and motion graphics software", *Adobe.com*, 2018. [Online]. Available: https://www.adobe.com/products/aftereffects.html.

[4] "About — blender.org", *blender.org*, 2018. [Online]. Available: https://www.blender.org/about/.

[5] "P2Pixie: A P2P Network Rendering Tool", *semanticscholar.org*. [Online]. Available: https://pdfs.semanticscholar.org/c82c/e43daae694c497608244badf1fa61eb6dab6.pdf.

[6] Q. Vu, M. Lupu and B. Ooi, *Peer-to-Peer Computing*. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2010, pp. 15–18.

[7] D. Abramov, "Redux · A Predictable State Container for JS Apps", *Redux.js.org*, 2015. [Online]. Available: https://redux.js.org/.

[8] "Flux | Application Architecture for Building User Interfaces", *Facebook.github.io*, 2015. [Online]. Available: http://facebook.github.io/flux/.

[9] E. Czaplicki, "Introduction · An Introduction to Elm", *elm-lang.org*. [Online]. Available: https://guide.elm-lang.org/.

[10] "Node.js", *Node.js*. [Online]. Available: https://nodejs.org/en/.

[11] "Electron | Build cross platform desktop apps with JavaScript, HTML, and CSS.", *Electronjs.org*. [Online]. Available: https://electronjs.org/.

[12] N. Hartzler, "PocketOfWeird/peerfarm", *GitHub.com*, 2018. [Online]. Available: https://github.com/pocketofweird/peerfarm.

[13] "Arnold Renderer | Autodesk | Arnold", *Arnoldrenderer.com*. [Online]. Available: https://www.arnoldrenderer.com/arnold/.

[14] "RenderPal V2 | Render Farm Manager", *Shoran Software* [Online]. Available: https://www.renderpal.com/.

[15] A. Goodwin, *Ocean Simulation*. Springfield: Missouri State University Department of Art & Design, 2018.