applied sciences

Article

An Experimental Implementation of a Resilient Graphic Rendering Cluster

Tibor Skala [1,*], Mirsad Todorovac [1], Miklós Kozlovszky [2] and Marko Maričević [1]

[1] Faculty of Graphic Arts, University of Zagreb, 10000 Zagreb, Croatia; mtodorov@grf.hr (M.T.); marko.maricevic@grf.hr (M.M.)
[2] John von Neumann Faculty of Informatics, Óbuda University, H-1034 Budapest, Hungary; kozlovszky.miklos@nik.uni-obuda.hu
* Correspondence: tibor.skala@grf.hr

Abstract: In this paper, we describe the challenge of developing a web front that will give an interactive and relatively immediate result without the overhead of complex grid scheduling, in the sense of the grid's lack of interactivity and need for certificates that users simply do not own. In particular, the local system of issuing grid certificates is somewhat limited to a narrower community compared to that which we wanted to reach in order to popularize the grid, and our desired level of service availability exceeded the use of the cluster for grid purposes. Therefore, we have developed an interactive, scalable web front and back-end animation rendering frame dispatcher to access our cluster's rendering power with low latency, low overhead and low performance penalty added to the cost of Persistence of Vision Ray rendering. The system is designed to survive temporary or catastrophic failures such as temporary power loss, load shedding, malfunction of rendering server cluster or client hardware, whether through an automatic or a manual restart, as long as the hardware that keeps the previous work and periodically dumped state of the automata is preserved.

Keywords: graphic rendering cluster; distributed rendering; multi-core Persistence of Vision Ray process; cloud rendering web front; grid complex scheduling system

Citation: Skala, T.; Todorovac, M.; Kozlovszky, M.; Maričević, M. An Experimental Implementation of a Resilient Graphic Rendering Cluster. *Appl. Sci.* **2021**, *11*, 12046. https://doi.org/10.3390/app112412046

Academic Editor: Alberto Gatto

Received: 8 November 2021
Accepted: 14 December 2021
Published: 17 December 2021

1. Introduction

Up to the point of writing this paper, we have published a lot of work on the cluster/grid rendering thematic. One of our published papers in the *Journal of Circuits, Systems and Computers* titled "Distributed Reliable Rendering Method for Parametric Modeling" [1] describes a proof-of-concept model of rendering using the parametric Persistence of Vision Ray model [2,3], client-server on-demand rendering architecture with SMTP (originally defined in RFC 821 and 822, updated in RFC 5321 [4]) and HTTP (originally defined in RFC 2616 [5], updated by RFC 2817, 5785, 6266 and 6585). This includes extensible protocol with the TCP connection keep-alive functionality, using open-source elements, configurable rendering architecture available at no cost besides the cost of hardware, with new solutions for simple configuration file mechanisms, and resilient software behavior based on best effort strategy of work distribution. Similar to SMTP and HTTP protocols, Internet protocol is not a requirement, and any reliable stream will suffice. Just like those protocols, and unlike FTP, GRC protocol transfers data such as an uploaded rendering model source package and resulting rendered images over the control connection stream.

The abovementioned paper presented an innovative way of creating a distributed reliable rendering method using a huge range of computers for parametric modeling. The main aim of the paper was to present a proof-of-concept solution that would use existing resources in order to make the modeling method more appropriate for applying complex computer services/jobs under distributed architecture. The implementation also included logistics and support for automatic computer clustering and for service/ job program execution. The primary goal was to use existing resources for useful applications in the 3D parametric modeling, image programming and simulation rendering. The logical extension to that work was our research in the field of a cloud-like service. The grid's complex scheduling introduced some stochastic behaviors in terms of unpredictable load, lags, delays and failures to deliver service. Yet, the greatest motivation to develop a cloud-like service was to justify investment and produce visible results for users who were not able to or not interested in obtaining a grid certificate in an institution that was not primarily about computer science, but accepted computer graphics [6]. Our choice of Persistence of Vision Ray as the rendering agent was motivated by the open source and free license of the program, which was also popular in the grid community and compatible with the Linux environment as well. The goal was to popularize new technology in a way that would be friendly to graphic design professionals and the student community. Our initial models included Tesla's electric motors, inspired by the anniversary of the Nikola Tesla's year of birth.

Early work involved the first attempt at a "Balkanized" grid using heterogeneous and asymmetrical implementation, demanding only a working GCC compiler and an OS with POSIX system calls. Later, a cluster was donated to explore advantages of grid architecture in the field of graphic rendering. The cluster had a total of 32 cores, which served as the render farm in the rack-mounted blade server architecture.

The second implementation was a major rewrite of the first render farm client. The goal was to implement the client efficiently enough so it would not be a bottleneck when the render farm would theoretically be expanded. The first implementation had a text terminal interface, after which a more user-friendly web front-end was designed.

Models were typically cyclic clock animations. The animation was divided into frames, each frame rendered independently. At the beginning of the process, all frames are in the "unrendered" state and in the UnrenderedFramePool object. The GRC-client then sends a model description to each server in the render farm, and then a keep-alive connection executes the handshake part of GRCP (Graphic Rendering Cluster Protocol) by sending the clock value for the particular frame and waiting for the Persistence of Vision Ray rendering job of the frame to finish. After that, the GRC-server process on the render farm server will produce a reply indicating that the particular result frame is done and declare its size, making it available to the client for download over the existing connection.

The procedure has been designed to produce an animation of a scene by calling a rendering farm node for each frame and then, having rendered all frames and transferred them to the web-front server's hard drive, calling ImageMagick's convert utility to produce an animated GIF.

The animated GIF is then displayed in a separate HTML frame.

Several layers of protection were added so the client could exhibit error recovery from transient and fatal errors, which were transmitted through a custom GRCP protocol reply status code common to protocols like SMTP, FTP or HTTP [5]. Finally, a web panel was added to provide a rendering progress meter to the user so they could know there was work in progress and not a server stall.

Several programmed actions were introduced during the course of development to make the cluster finish animation in a reliable and resilient way, in contrast to GRID animations from SEE-GRID project, which would spuriously fail for unknown GRID reasons and without an error code.

An effort was made to reduce frame dispatching overhead to the cost of rendering with Persistence of Vision-Ray SDL scenes. Where available, compile time and runtime optional Linux zero-copy kernel system calls were used, adding to cluster performance and eliminating some performance bottlenecks, thus increasing the theoretical limit of render farm servers that can be served via the GRC-client process which equally serves the text terminal and web front-end interface to the rendering farm. With a minimal scene box pov and a minimal rendering resolution, up to 3600 fps (frames per second) were reached on the 32-core cluster (on minimalistic benchmark Persistence of Vision Ray scenes), which demonstrated low overhead of managing animation frames and polling server nodes from the rendering farm. (However, this was only benchmarking. Real-life loads are something completely different.)

The render farm servers are declared in a configuration file with server: port entry per line.

Pluggable interface to the Persistence of Vision Ray rendering engine allows for use of other render engines, in addition to ray tracing Persistence of Vision Ray, which introduces performance problems because it uses mathematical interpolation instead of vertex rendering. Additionally, Persistence of Vision Ray has the limitation of no hardware or GPU support; the entire rendering process is executed on CPU and in software.

Due to a small overhead to the cost of rendering, GRC-client/GRC-server and their GRCP communication protocol allowed for pluggable extension of the architecture which can be used to produce animations exported from open-source software like the Blender. This is enabled in the protocol and upload mechanism with corresponding MIME types (e.g., application/x-tar-povray).

2. Related Work

The first implementation included Debian Linux, MacOS X and Cygwin on MS Windows mixture or rendering agents in the farm. (Cygwin offers POSIX environment on top of Windows architecture.) This was recognized as architecture in between a "Balkanized" grid and a variant of a cloud environment [7]. The rendering farm was distributed across institutions of the University, across different hardware and OS architectures and machine performances, which made the selection of a FCFS (first come first served) simple scheduler a logical alternative to complex schedulers used by grid resource brokers.

Later, after acquisition of proper grid hardware from the project, the GRC service scavenged computer time not spent for serious grid uses (such as the SEE-GRID project) [8]. After improvements to the GRC-client and POVdriver web front-end client which rendered Tesla models on a web-front host in the DMZ area, the POVdriver was able to call the GRC-client, which was in turn polling nodes of a 4×8 core CPU cluster machine. Cygwin on MS Windows was phased out with the introduction of the BOINC client for volunteer computing, yet we considered re-introduction to deliver immediate performance compared to BOINC which involves long queueing on central hosts.

However, neither grid nor BOINC were able to fit closely to the web to deliver immediate performance. This is where GRC architecture fits in. After problems with bugs in the code, GRC suffered a major rewrite, having strengthened the code with an array of FSA (finite state automata), one per polled connection to the rendering server node, which maintained connection state per connection in an environment where unpredictable, stochastic events on machines and cores as a rule deliver different timing and states of connections will gradually stop being in unison after just a few frames. In clock animations, the view on scene changes resulted in variance of rendering time across the different frames of animation, as did the stochastic events in the operating system and hardware environment.

We had also closely monitored progress of Persistence of Vision Ray software development and success of its developers to deliver excellent multithread and multi-core implementation of the software, yet we preserved the per core reservation of render agents. The rationale for this was that there was spare time left unused if only one multi-core Persistence of Vision Ray process was run. If only one process was left, it could get allocated up to 800% of CPU time; in other words, all eight cores on a cluster node. We also experienced a recoverable error which we believe relates to Persistence of Vision Ray 3.7 release candidate RC3 version. (This error is a transient error resolved by restarting rendering, yet it pays off abundantly to deploy multiple rendering processes as an experience of 25% increase in rendering speed measured in fps, benchmarked on real scenes in our tests.)

Additionally, there is a continuous idea and motivation to allow for Blender's YafaRay pluggable render engine with XML-based scene description language, but at the time of writing it was not yet implemented. The rationale is in the fact that Blender is used by a much larger community for which we could develop a model upload and rendering service which would utilize our 32-core cluster and possible new equipment. The protocol with the working name GRP which is based on SMTP and HTTP with the addition of a RENDER command that tunnels sanitized Persistence of Vision Ray options to the Persistence of Vision Ray rendering agent on render farm servers is extensible enough to allow for Blender and YafaRay, and MIME-type-based distinction of formats will allow for a new format and new language. Alternatively, autorun.txt file will determine the source and include the language of the rendering job, but auto detection is also considered to minimize the knowledge and the trouble the MS Windows user must learn to use the service.

This auto-detection is feasible through analysis of the listing of the zip file or tar bundle with the job. In general, the rule of thumb is that another level of complication will discourage 50% of remaining users, if we are allowed to refer to a popular book by a renowned physicist Stephen Hawking.

In particular, the progress of the job and errors are difficult to present in real time over a web connection in the browser, so in anticipation of an AJAX-powered interface that would deal with greater intelligence with messages from the GRC-client, our text mode agent is at the time of writing more informative about what is going on. However, the web-front interface is more user-friendly. This confusing lack of information in the case of job failures imposes a demand on resilience of the GRC-client interface that polls the GRC-server on render farm nodes. We want the performance drop to be gradual and non-fatal. Errors ought to be treated gracefully and render jobs and animation frames restarted in case of Persistence of Vision Ray's stubborn or mysterious faults. Debug mode is left for final users so they can run the application in debug mode with their set of parameters, copy and paste the output and send it to our support.

Let us describe GRC-client: the text client's interface is presenting a select() call table with RWE (read, write and exception) state per connection and number of frames rendered on a particular node. The bottom line displays the number of frames rendered out of total number of frames in the animation, number of frames per second achieved in the job and ETA (estimated time left for the job in hours, minutes and seconds). The middle portion of the text terminal is reserved for the list of servers used, the state of FSA per connection with the name of the state colored in green, yellow or red (meaning OK, warning or transient problem or fatal error). If everything works, it will mostly display the GRC_SPNGWAIT (PNG wait) state as this is the state in which the client waits for rendering to finish, with occasional glimpses of GRC_SPNGRCVINPROG (PNG receiving in progress) state, which is the moment of writing the rendered frame or frame slice to disk.

In demonstrations, the GRC-client would sometimes fail embarrassingly, which the graphic rendering professionals understand since it happens also with much more expensive software, but we as developers were puzzled regarding the ways in which to make program more robust and resilient. The average user would simply lose interest if the interface does not seem to respond, so the problem was how to display any feedback until the final result arrived. A progress bar with a plus sign per frame seemed to work, but alas only in Internet Explorer and Firefox up to 3.6.x clients. Opera would describe the number of bytes loaded, but the rest of the tested browsers (Safari and Chrome and newer Firefox) behaved counterintuitively, waiting for the entire document to load before displaying a single byte of information, which in our case did not give an impression of faster loading.

Stereo animation production imposes additional demands on the perception of interaction, since seemingly irresponsive browser's interface will remain that way for a twofold amount of time (left and right eye image generation). However, about: config settings of Firefox did not work for us to produce progressive loading, so the only thing we could do was to recommend Firefox up to 3.6.× or IE browsers, which meant we would lose a significant number of users in anticipation of the AJAX-compliant interface.

The fact that the POVdriver is a separate program from the GRC-client limits what can be done in terms of AJAX programming. Teaching the GRC-client to "talk web" and run AJAX from the CGI environment is the next logical step, but a hard one, for the client would be required to both "talk to cluster" and "talk to web". Talking to the web and to render the farm at the same time imposes rigorous demands upon the development scheme and a new paradigm of a more intelligent object-oriented approach.

As a workaround, we developed an independent solution to the problem of the rendering task's visual feedback to the user [9]. A semaphore-driven extension allows us to periodically re-read dumped UnrenderedFramePool.obj and offer a visualization of the rendering process. Each frame in the pool is assigned a number and a color based on its state (unrendered, in progress, finished). Then a process reads this information upon signal from the semaphore it waits for and renders a suitable HTML representation of the rendering task in the form of a table of frames. In the beginning, all frames are in the "unrendered" state (grey), while in progress they will all go through the "in progress" state (yellow) until they all reach the "finished" state (green). This separate process dumps an HTML file which is downloaded and reloaded periodically in the browser's window, via an HTML <meta http-equiv="refresh" content="<time>"> reload tag, thus bypassing the visibility problem when it comes to lost functionality concerning the discontinuation of incremental loading and display of web browsers. At the end of the rendering process, the web interface displays the resulting animation.

However, under the hood our GRC-client hides several layers of behavior that make it deliver performance despite these shortcomings in the ruthless environment of users and computer clusters.

3. Experiments—Some Ways to Make Our Software Resilient

Our experiments, and especially presentations to computer graphic rendering industry professionals, have addressed several demands that would gladly be seen in a rendering agent and a rendering farm.

In respect to this, we have added layer upon layer of error recovery methods to our experimental implementation, on the level of:

- Recovery options in our GRP protocol
- Recovery of POSIX system calls
- Graceful restart after catastrophic failure
- Graceful exit to enable maintenance without loss of work
- Graceful continuation with servers that still work after catastrophic failure
- Restart of failed render farm nodes upon a POSIX signal
- Automatic restart of failed render farm nodes

As shown in Figure 1, this layered error recovery is believed to develop a resilient behavior in our software. We will analyze strong and weak sides of our solution.

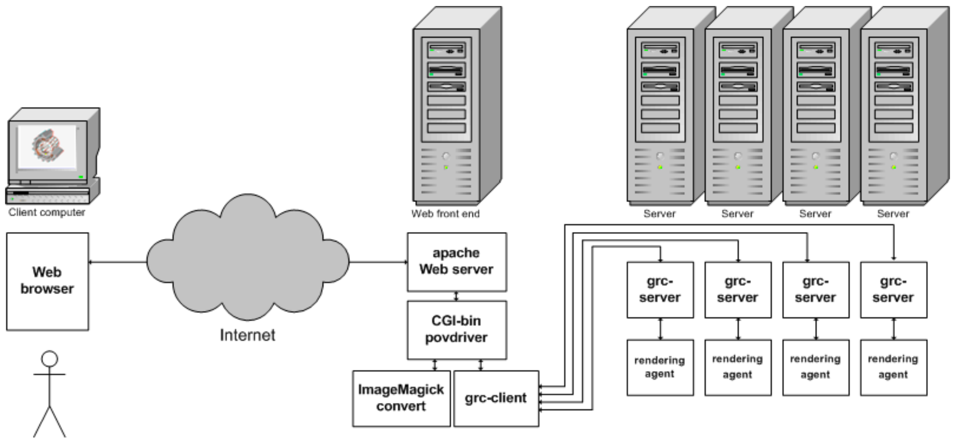

Figure 1. An experimental distributed cluster based on software resilience.

3.1. Recovery Options in Protocol

As described earlier and in [9], GRP graphic rendering protocol is based on proven SMTP and HTTP protocols with the extension of the RENDER command. Additionally, the GRP protocol has also inherited reply status codes, which indicate the severity of the reply from OK (2xx codes) and informational messages (1xx codes) to temporary failure (4xx codes) and catastrophic failure (5xx codes) of the render farm server host. Codes are used in the way that they are described in SMTP and HTTP protocol [4,5] which is no surprise to experienced open-source software programmers.

Notably, the reply status codes enable the GRC-client to distinguish server states without the complex parsing of the replies of a state of a render farm server where it is meaningful to restart the job from one in which it makes no sense to do so, or it might be beneficial to attempt afterwards, e.g., after the maintenance of /tmp partition on the render farm server.

In particular, Persistence of Vision Ray 3.7 RC3 seemed to have an unusual number of spurious failures that meant no hazard if rendering was simply restarted. Our conclusion was that this failure related to multicore use of processes and some kind of overload or starvation for resources, but at the time of writing we were not able to pinpoint the error in Persistence of Vision Ray 3.7 and produce a meaningful bug report since the error was not repeatable or reproducible. We were forced to learn through experimentation and trial and error that symmetric design of our cluster means that the error is transient without the increase in probability that the same node will fail again (Figure 2). This could be connected with RAM shortage (a cluster node had 2 GB of RAM).

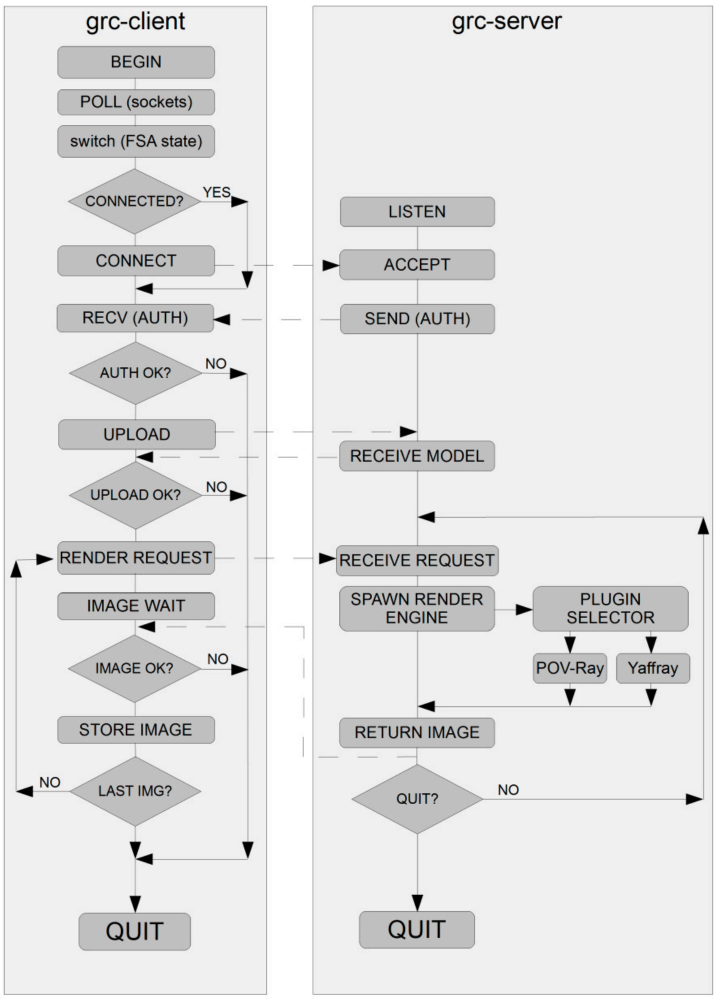

Figure 2. Algorithm flowchart for GRC-client and GRC-server.

Table 1 shows the GRP protocol diagram of how it works on clock animation images for moments t = 0.00, 0.01, 0.02 . . . 1.00, or trace of one typical session.

Table 1. GRP protocol diagram (trace of one typical session).

```
S: 220 GRC server VERSION ready.
C: PUT model-tesla.tar.bzip2
C: Content-Length: 19192
C: Content-Type: application/x-tar
C: Content-Transfer-Encoding: 8bit
C: Content-Encoding: x-bzip2
C: <model data on the control connection>
S: 200 Upload OK.
C: RENDER BETA OPTS=+k0.00 +W640 +H480
S: 220 RESULT SIZE=64392
S: <image data on the control connection>
C: RENDER BETA OPTS=+k0.01 +W640 +H480
S: 220 RESULT SIZE=65342
S: <image data on the control connection>
C: RENDER BETA OPTS=+k0.02 +W640 +H480
S: 220 RESULT SIZE=65397
S: <image data on the control connection>
.
.
.
C: RENDER BETA OPTS=+k1.00 +W640 +H480
S: 220 RESULT SIZE=65536
S: <image data on the control connection>
C: GOODBYE
S: 221 Closing connection.
```

Notably, we were able to isolate this type of error and give it the code 411 REN-DERER ERROR to enable restitution of render farm server's node into functionality as shown in Table 2. Implementation does not guarantee that the same render server would get the same task, nor do we think it would matter (our examples consist mostly of clock animations, so the same Persistence of Vision Ray source is always assigned until restarting the GRC client) [10].

3.2. Recovery of POSIX System Calls

To an experienced programmer this chapter is self-assumed, but we have to emphasize the danger of an unhandled restart of system calls, which in asynchronous I/O and POSIX signal use can and do occur virtually everywhere in the code. This was handled by graceful treatment of EINTR, EAGAIN and ERESTART error codes from common system calls such as read(2) or write(2) and other states which do not represent fatal errors (but we also attempt to recover from fatal errors that do not imply the impossibility of rendering job on the cluster as a whole, such as abrupt failure of one or more connections, as long as there is still some capacity to survive the error or malfunction with degraded performance but eventually still finish).

Table 2. GRC protocol reply status codes.

220 GRC server ready.
200 Upload OK.
221 Closing connection.
404 No more jobs.
410 Rendering failed.
411 Renderer error %d.
412 Cannot open temporary directory.
413 Cannot use temporary directory.
420 Error in communication line.
421 Upload failed.
500 Internal server error $0 \times$ E0000001
500 Internal server error $0 \times$ E0000002
500 Internal server error $0 \times$ E0000003
500 Internal server error $0 \times$ E0000004
500 Internal server error $0 \times$ E0000007
501 Command not understood.
502 Command too long.
503 Extraneous input at the end of command.
504 Malformed command.
505 Malformed request.
506 No renderer executable installed (fatal).
507 No compress executable installed (fatal).
508 No image convert executable installed (fatal).
509 Error: PUT model first.
510 Unsupported content type.
511 Unsupported content encoding.
512 Unsupported content transfer encoding.
513 Filename too long.
514 Directory traversal not allowed.
515 Unable to find source in uploaded package.
516 Unknown error while unpacking the package.

Design based on POSIX select(2) call polling loop is believed to prevent copious restarts of non-blocking system calls read(2) and write(2) over an asynchronous non-blocking connection. Unlike waiting loops in assembly language programming, our design goal is the opposite: to save CPU cycles in a manner of relinquishing the processor or CPU core as soon as possible (ASAP). We believe we have done a fair job at this using 0.1% memory and 0.7% of system time CPU (2.0% under peak stress load, at the time of writing) as this scheduling overhead, as profiling has shown. Then the GRC-client goes back to wait on any socket events in the polling select(2) loop. It was not prudent to use the blocking read(2) or write(2) system calls as we serve many simulated parallel threads in one process.

A bug once used to cause 99% CPU load under certain circumstances, but we were able to isolate the root cause of the bug after a certain time and it was patched with a workaround. Object-oriented revamping of code resolved this error condition in an obvious way, natural to the problem. Often in popular software, runaway program states with 99% CPU use are not considered a condition serious enough to stop issuing a new version of software or to issue a patch, but we disagreed and pursued this condition, isolating this error in implementation in accord with the GNU software code of conduct.

This design was no surprise since we also asynchronously caught signals (discussed in the next chapters). It was essential to check all failures of system calls and errno numbers, together with possible conditions that require a system call restart. Their errno status serves as an input to the finite state automata, one per connection which corresponds to each rendering server.

Signal handlers just set flags that a signal was caught, so the main loop then deals with processing the condition as an FSA state. This reduces concerns over signal-safe functions.

3.3. Graceful Restart after Catastrophic Failure

With resilience in mind, we wanted to develop a contingency plan if hours of our render job were interrupted by a power outage in the cluster room, for example, or total Faculty or city block power outage (not uncommon in Croatia). This required a C++ object FramePool upon previous encapsulation of handling the unrendered frame pool from [11]. The object was made intelligent enough to dump and restore its state. Normally if the disk containing rendered frames and the state keeping file is not damaged, this suffices to avoid rendering frames or frame slices repeatedly. The state of disk files of unrendered frames is not sufficient to diagnose where the job stopped because the catastrophic failure could have occurred when only part of frame was saved in corresponding PNG, and this situation will require expensive diagnostics compared to the overhead of keeping and saving state of myFramePool every N seconds (in our case N = 10 s).

The problem is that copious saves of state after each frame creates a disk bottleneck, hence the periodic timed saving of state so at most 10 seconds of work would have been lost, adding to this the render time of unfinished frames, but there's no way to salvage those if the power outage occurrs before saving the state on the GRC-client host. However, this will mean losing at most <number_of_render_farm_servers> frames, and it could be a lot of time; but there is no remedy unless the designers of Persistence of Vision Ray decide to enable continuation from where it stopped in catastrophic cases by saving some state information as well (Figure 3).

The problem of catastrophic render server failure, such as disk malfunction or power unit malfunction on part of the cluster, will not damage the rendering process as a whole. This had been tested thoroughly, and the system call will be interrupted with the loss of connection error. After that, the unrendered frame is simply returned to the unrendered frames pool and eventually reassigned to a working render farm node. The consequence is that the rendering process is not stopped if the rendering farm is geographically distributed across areas that will not suffer power outage at the same time (the location of the web front-end host and povdriver is still power-critical).

Yet, this made our cluster fail to render forever on a set of hosts after temporary outage of power as well. Stopping and restarting the GRC-client did the trick, but potentially long jobs were needlessly lost and Persistence of Vision Ray processes on the GRC-server hosts continue to render until done despite the process being futile and despite its attempt to be efficient.

This was, in part, resolved with the next level of resilience.

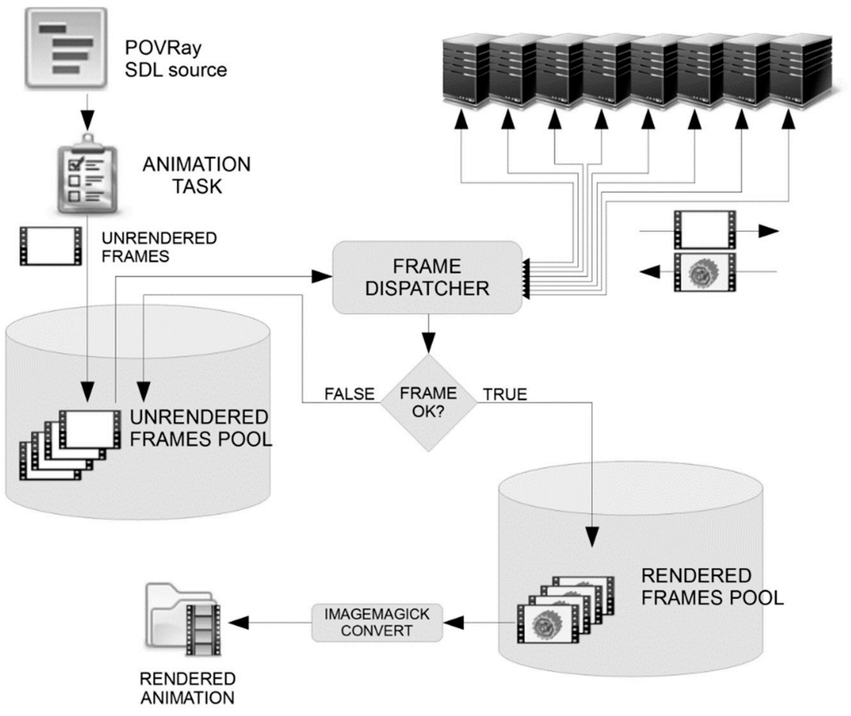

Figure 3. Communication protocol.

3.4. Graceful Exit to Enable Maintenance without Loss of Work

This graceful exit was implemented in text version only, on pressing CTRL + C as shown in Figure 4. At that point, the GRC-client receives the SIGINT signal, and the signal handler sets a flag upon which all simulated threads begin to finish rather than appropriating new unrendered frames from the unrendered frames pool myFramePool and initiating new rendering jobs on the cluster over the established connections.

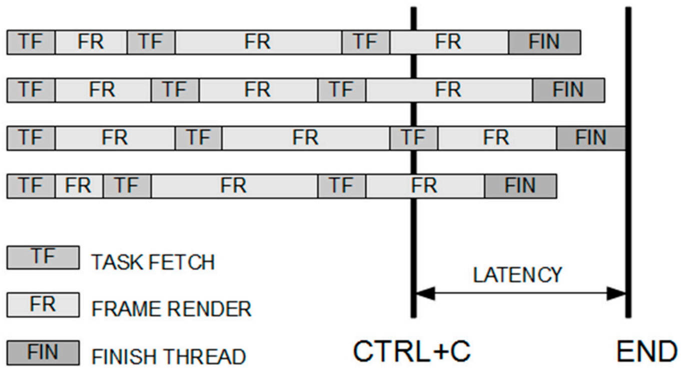

Figure 4. Timeframe and stopping latency for distributed cluster.

The second CTRL + C interrupts ungracefully with an emergency stop, in which case current rendering jobs are lost and restart is available only up to the state in last timed dump of myFramePool. However, this is not recommended except for an emergency because of the lingering rendering jobs remaining on the server. A method for asynchronous delivery of the SIGINT signal to the server side is required via an out-of-band message, but this is not yet implemented.

Restart with option –C is provided to continue where the saved job state stopped in both cases. Transferring the emergency stop signal to the rendering farm hosts is not yet implemented, so they will finish currently rendering frames despite the break and have them discarded after completion. This incurs a performance penalty on the cluster, but the emergency stop is meant to be used only sparingly in special cases of emergency. (The urgent STOP signal to the cluster requires an asynchronous extension to the GRC protocol and is planned for the next phase of development.)

Right now, the web-front implementation of graceful exit/restart is thought of as possible to implement with a job ticket and some AJAX programming in the interface [12].

3.5. Graceful Continuation with Servers That Still Work after a Catastrophic Failure

As mentioned earlier in this article, the GRC-client is able to finish the job despite catastrophic failure in all but one render farm server host. The performance penalty is in reduced throughput. This is not a new feature; it was present already in the previous version [1] as a consequence of encapsulation dealing with the unrendered frame pool through a very limited number of functions. Now those functions are methods, and only the means to save/restore the state of frame pool to disk were added.

This feature is useful in case of fatal failure of the host that cannot be fixed in time, such as hardware malfunction, and would matter for the render job to finish. Sometimes, the performance penalty of a slowed down cluster and reduced throughput is acceptable when not a single frame of rendering job is lost [6,13].

Catastrophic failure, including physical destruction of a render server farm member by fire, will cause connection failure that will be handled in a manner of returning the frame to the unrendered frame pool, just as any other error.

We believe the reason for this action being so successful in making software behavior resilient is by following the inherent logic of the problem. Using assert() assertions wherever it made sense helped catch a significant number of bugs that would have otherwise been uncaught and were evasive from detection in a program that, as a rule, never runs the same way twice (due to event-driven programming).

It is also possible to replace the render_config_file.cfg file and continue the rendering job on an entirely different cluster that has the GRC server installed (before or afterwards), in the case of the rendering farm suffering a catastrophic unrecoverable error.

3.6. Restart of Failed Render Farm Nodes upon a Posix Signal

In particular, in one demonstration a computer graphic industry professional inquired about the possibility to restart jobs on a subset of the rendering farm after a kernel upgrade, for example, or in the MS Windows world, an automatic or WSUS update that required a reboot. This was a real concern even for professionals who were using professional rendering software such as that used in making blockbuster movies [1].

At first it seemed infeasible, but by using a combination of signals and flags we made it conceivable after the implementation of a graceful stop on CTRL + C. POSIX signal SIGUSR1 was used to signal to the GRC-client that it was time to attempt to restore connections to render servers on all hosts that were lost in the past and are present in render_config_file.cfg at the start time. This option does not (at the time of writing) allow adding new render farms interactively after a job has started. (Perhaps a job for another signal [5], but the job can be gracefully stopped, render_config_file.cfg file adjusted, and the rendering continued, so this is possible with the existing code and little effort on user's side.)

This is somewhat cumbersome, but it amounts to finding the PID of the GRC-client and sending it the SIGUSR1 signal, as if:

```
% ps –ef | grep grc-client
mtodorov 25817 10807 0 11:26 pts/3 00:00:00 ./grc-client -C -c -n120 -W400 -H300
-O+A0.3 -J0.0 +FN abyss.tar.gz
% kill –SIGUSR1 25817
```

Alternatively, top utility may be used to find the PID of the GRC-client to send it the signal.

However, the alternative (automatic restart described in the next chapter) is introducing certain system overhead as well, such as pending connection attempts to a server that is temporarily down, so both methods are kept.

3.7. Automatic Restart of Failed Render Farm Nodes

This behavior was intended to restart connections to clusters in the case of temporary downtime, temporary grid power outage, load-shedding on server-side power grid, or a catastrophic failure like hardware failure that was repaired. Current implementation is to try and attempt restart every N seconds. Since GRC-client is "blind" to the state of render farms unless it tries to "ping" potentially "empty space", an unreachable network address, this behavior could lead to ping overload of the network with a large number of render servers in the farm if not used with caution. This, however, saves the need to service the GRC-client process and sends it a signal with a terminal command, and then later restart. This is also a web-friendly solution, for web users as a rule will not be given access to a terminal able to send *kill* command to GRC-client process (allowing for the fundamental difference of enabling web-front end jobs and the possibility to automatically restart after temporary failures.) Therefore, the optimal retry time will depend on the kind of job that is running, e.g., from five seconds in an interactive web job task to a good minute in a rendering that is scheduled overnight. It is configurable via a command line option [14], and the client deploys an "exponential back off" strategy, used in the Ethernet protocol as well, to avoid network overload with connection attempts while the server is down.

Adding new render servers interactively during the program run is another desired feature that is not yet implemented, but it is viable in our code. Some progress had been done in this direction with the development of the ConnectionPool class. In this aspect, adding and retirement of servers in the active render farm on-the-fly is a desirable feature, but it would require complete OO revamping of the connection establishment, hiding the complexity of the connection-making process. A vast part of this had already been done, i.e., by moving the connection initiation code of asynchronous connections towards the inside of the polling select() loop and FSA-simulated threads rather than running outside of the main loop, which previously disallowed reconnection in the case of losing one. This revamping of code is yet to be fully justified, apart from enabling signal SIGUSR1 and timed restarts of servers. Restarting the connection became as easy as resetting the connection to GRC_SINITCONN state on corresponding FSA-simulated thread. The rest is handled by an FSA-driven engine which handles connection and GRC protocol states in a "smart" way.

However, in order to fully harvest the benefits of object-oriented dealings with a connection pool, a logical and reasonable coding model is required, one that follows the logic of the problem. Objectification of FramePool and ConnectionPool had already made certain things possible where a non-object oriented, non-encapsulated approach would needlessly complicate the code, increase overall complexity and create places susceptible to bugs or make implementation fundamentally infeasible. Making certain procedures class methods of a "smart" object allows for dealing with asynchronous events and cleanup in case of unexpected termination of the client or server program. Making each object "know" how to release allocated resources enables a new functionality when compared to having the same code in the "plain" functions in main program.

3.8. Measured Performance Bottleneck

The test suite box.pov from Persistence of Vision Ray demo scene database in minimal 80×60 pixel resolution is rendered at a well-pleasing 1390 fps (frames per second), the speed at which the GRC-client consumes 100% of processor time, or an entire CPU core, which as a proof of concept that works by almost two orders of magnitude better than requirements for real-time rendering. As of writing, real-time animation is not a goal in the GRC project; this speed is only used to benchmark and measure bottlenecks in performance. At a normal resolution of 1024×768, box.pov is rendered at typically 35 to 40 fps. In this example, the GRC-client is no longer a bottleneck, but it occupies about 15% to 20% of CPU core load. By experience we can tell that in this case the bottleneck is Persistence of Vision Ray itself with its rendering speed limit. We were unable to speed up Persistence of Vision Ray, having only provided handshaking by two orders of magnitude faster than it is required, adding little or insignificant overhead to bare Persistence of Vision Ray rendering CPU consumption. This lower overhead has increased scalability as a consequence [15], as the client could serve more rendering servers without choking its CPU core on the job (the multi-threaded implementation of the client is on the to-do list as well).

We were happy to learn that the GRC-client and GRC-server presented no performance bottleneck, or present one only in extreme conditions, such as 1390 fps rendering of a very small and simple benchmark scene. Real-life animations have much larger complexity than box.pov, meaning that the CPU load the GRC-client will impose on the web-front host will be on average, in our experience, from 0.7% to 2.0%. Later improvements have raised benchmark performance to up to more than 3500 fps for a box.pov benchmark simple scene example, which is sufficient to demonstrate client-server handshaking of the web-front server and the rendering farm plus the overheads of the protocol and of the Persistence of Vision Ray process startup. As in the cluster, as a rule some Persistence of Vision Ray process will always run and have preloaded binary executable, and the overhead is mostly reduced to that of fork() and exec() POSIX system calls with the few copy-on-write page faults. Using faster mmap() system call instead of open() and read() maps the model directly into memory, and resulting frames are also mapped before sending onto the network socket. Where available, Linux-specific system call sendfile() is used [16]. Because the copying is done within kernel, this is more efficient than read() and write(), which would require copying of image data to and from the user space. This call is also safer than mmap(), which can sometimes fail with bus error (signal SIGBUS) after which there are very few options in signal recovery but to close the server process (but the whole rendering process will not be interrupted as the animation frame will simply be returned to the unrendered frame pool). Additional zero-copy Linux system calls are also being considered, like splice() and vmsplice(), to avoid further user space copying overhead in the GRC-client and GRC-server.

(The use of these Linux-specific high performance system calls requires conditional compilation of GRC-server and isolation of Linux-specific code in #ifdef . . . #endif clauses.)

This increased efficiency of 3500 fps has no effect in real life rendering examples but removes the bottleneck from the GRC-client’s and GRC-server’s frame handling routines completely, as the system limit on open file handles will be reached first unless there has been a change done by the administrator on the resource NOFILE [2] (the system limit on the number of open files). As current implementation uses one connection per CPU core [17], the number of total cores in the cluster is limited with this resource limit parameter. (On our Debian jessie system, this number is currently limiting the cluster size to a theoretical limit of 65536 cores, but recent releases such as Bullseye have hard system limits increased and the maximum number of files is 1048576).

Additional optimization is that the standard system(3) function is not called, but a replacement mysystem() for the original function involves running a shell process to start up the desired command, with the consequential overhead and performance penalties.

Table 3 shows the benchmark results for a test scene box.pov and for a real-life model of a Tesla 3 phase electric motor on a 12-node cluster [18]. Both the asynchronous (induction) and the synchronous motors are modelled. Minimal image size is supposed to test the entire system’s overhead in submission of scenes, rendering and communication protocol. The minimalistic resolutions serve only to measure the bottlenecks.

Table 3. Benchmarking results for a test scene and for a real-life model of a Tesla electric motor.

Resolution	fps (box.pov)	fps (motor.tar.gz)
4×3	769.50	103.70
8×6	887.80	80.00
16×12	850.10	45.50
20×15	810.30	36.40
40×30	322.00	10.60
80×60	265.10	3.30
120×90	226.20	1.71
200×150	158.60	0.76
400×300	71.20	0.23

4. Future Development

The reader might be curious about whether this experiment has a continuation. In particular, several issues were tested only to the state of "proof of the concept" reliability, where there is natural space for improvements. Several concepts were implemented via hacks involving AIO (asynchronous input-output), vector of FSA (finite state automata) and socket polling [19]. Such sections of the program are today advised to be implemented by using multi-threading support in C++ programming language, enabling the distributed load over system's CPU cores in a multi-core system. There is also space for improvement in making the code pure C++ and for abstraction of concepts which were implemented with hacks.

Additional obvious improvement would be implementation of pluggable rendering agents support [20]. This is practically done already with pluggable compression/decompression and none/zip/tarball packaging of the model source, including images. Implementation of pluggable Yafaray support would not be expensive at this point, allowing Blender rendering at the same time.

We see room for experimental use of our software by advanced students or rendering professionals who are interested in open-source modeling in Persistence of Vision Ray and Blender and who possess advanced skills.

Use of the proposed multi-threaded extension to GRC protocol would increase the possible number of CPU cores by a factor of four or eight, or by the number of cores per host, since a connection would be used per host, not per core. With the parameters of our cluster and Debian jessie operating system, this would impose a theoretical limit of 524,288 cores, which is more than we expected or find possible.

An out-of-band asynchronous message on the protocol control connection could be implemented to allow for immediate stop of rendering that would terminate lingering Persistence of Vision Ray processes.

While all effort and coding discipline was made to make the code safe (and neither client nor server will run in a privileged mode) and avoid buffer overruns or memory leaks, the GRCP protocol itself cannot be considered safe: endpoints are not authenticated, and there is a need to address this issue in eventual future development, especially if the rendering process might be distributed across organizations and domains rather than running on a single cluster on local area network. Possible authentication might be executed with a username, password and a "secret", known only to client and server. To avoid a security compromise, a safe hashing function like SHA-256 or stronger will be used, or a layer of TLS sockets.

In the first scenario, secret and additional unique information would be XOR-ed and then a hash would be generated from the resulting value. Both ends should agree on the hashing result for the authentication to be approved, much like RADIUS protocol (defined in RFC 2865 [21], updated by RFC 2868, RFC 3575, RFC 5080, RFC 6929, RFC 8044) (but that one uses weak MD5 for hashing as implemented in most environments).

The models for rendering are transferred in cleartext, and a layer of TLS encryption would be required if this is not tolerable (in the case of rendering with concern of copyright of model source that might be stolen by a man-in-the-middle attack). An extension like AUTH and STARTTLS command would be required. This is, however, not yet implemented at the time of writing.

Additional consideration is needed for the possibility of introducing a connection ID like in HTTP/3 over QUIC protocol defined in draft standard [22]. Initial experimentation with implementation of HTTP/3 in the popular nginx server [23] tested on our site had shown that a connection tracking mechanism is at least theoretically possible to implement in GRC protocol, which would enable rendering from roaming clients. However, this inevitably incurs the need to keep the state of the rendering jobs on the GRC rendering server.

Introducing a connection ID might also enable the implementation of reconnecting to old rendering jobs from different browsers, different client hosts, an entirely different organization or home computers, allowing for mobile work force use and working from home. This extension would allow for elimination of the default 5 min browser timeout limit on interactive rendering (text GRC client does not have such a limit).

5. Conclusions

This paper's intent is the dissemination of the SEE-GRID project for which our hardware was donated. In our work based upon the SEE-GRID project, we were scavenging cycles of the process time of a donated cluster which was not used in grid operation. We were presented a task that required web-front interface to a cloud-like rendering service on a cluster, since we wanted to introduce new technology to users whom we did not want to require to issue access accounts to the cluster or GRID infrastructure itself. Architecture based on interfacing POVdriver web-front service with already proven and tested with CLI GRC-client software which managed rendering on the cluster and was designed to allow such interfacing of cluster rendering power with the Persistence of Vision Ray tracing graphic software from a common web browser.

We have made a stack of layered and independent provisions to make our software robust and resilient, including the use of asynchronous I/O and polling of connections, graceful restart of connections and services in GRP protocol and in an automatic restart in case of broken connections, upon a POSIX signal from Linux operating system environment, or because of a timeout, server failure, hardware malfunction, power outage, or load-shedding, making it a proof-of-concept poor man's rendering cluster infrastructure. Graceful stop and restart have been provided upon a keyboard interrupt signal, upon which the GRC-client completes current frame rendering and then stops. The state is preserved in an UnrenderedFramesPool object periodically dumped on the disk subsystem. Makefile shell script is provided to allow the graceful restart of nodes in render farms without disrupting the rendering as a whole by the implicit restart of connections on timeout.

The system presents little or negligible overhead in real-life usage with complex scenes, such as 0.7% to 2.0% in test loads. The GRC-client driver does not choke on the job until a rendering speed of 1390 fps in benchmarking and bottleneck analysis, which is far beyond what is required for normal operation. Later removals of the bottlenecks in implementation have increased maximum benchmark throughput to up to 3500 fps on our 32-core cluster, which more than doubled scalability of our solution, further reducing animation frame management overhead and allowing the glue software itself to theoretically serve in a real-time rendering environment, provided that Persistence of Vision Ray tracing agent would be substituted with a rendering program that uses hardware optimizations and cluster's GPU units. Rendering options can be tunneled through an existing OPTS parameter of the RENDER command, and the existing infrastructure of uploading with MIME types and pluggable rendering software allows for such extension. Benchmarks show that the GRC overhead would be almost negligible in a scenario of rendering 30 fps or even 120 fps.

The strength of our solution is not in our hardware implementation and our cluster, but in scalability and extensibility of our software based on low overhead and low performance penalty for network outages and graceful recovery from all predicted “unpredictable” failures. It is designed to be more robust than our environment demands, based on insights from actual graphic industry professionals renowned in the field who attended our demonstrations. We established our design with a single thought in mind—finish the job at all costs if only one render node survives and allow for a graceful restart of rendering in case it does not, possibly on new equipment, as long as the hardware that preserved previous work survives. Due to object-oriented design, it is possible to restart rendering with a different cluster and continue rendering only unrendered, unfinished frames as long as the new rendering server's rendering engine has the same semantics of the Persistence of Vision-Ray SDL language on rendering nodes in the farm. This is done by replacing or editing the render_config_file.cfg in the rendering directory, while leaving dumped information about the FramePool object and restarting. This process would allow for continuation of the rendering job from where it had stopped on new hardware in case the original hardware suffered malfunction or we lost permission to use it for rendering, but the job already done was too significant or lengthy in duration to be lost or repeated.

This experimental architecture was meant to popularize cluster, grid and cloud use for rendering at the Faculty of Graphic Arts, University of Zagreb, as most of the rendering is still done on a single client-side personal computer. This, we believe, justifies this experiment in cluster rendering and pluggable, especially if it will, through the existing pluggable render driver system, include Yafaray and popular Blender support.

The solution is completely made out of open-source components, with our code also being open source and GPL licensed.

Author Contributions: Conceptualization, T.S. and M.T.; methodology, T.S. and M.T.; software, T.S. and M.T.; validation, T.S., M.T., M.K. and M.M.; formal analysis, M.T.; investigation, M.M.; resources, T.S.; data curation, M.T.; writing—original draft preparation, T.S. and M.T.; writing—review and editing, T.S.; visualization, M.M.; supervision, M.K.; project administration, M.M.; funding acquisition, T.S. and M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: The researchers would like to express gratitude for the financial aid received.

Conflicts of Interest: The authors declare that there is no conflict of interest.

References

1. Skala, T.; Todorovac, M.; Skala, K. Distributed reliable rendering method for parametric modeling. *J. Circuits Syst. Comput.* **2013**, *22*, 1250090. [CrossRef]
2. Afgan, E.; Bangalore, P.; Skala, T. Scheduling and planning job execution of loosely coupled applications. *J. Supercomput.* **2012**, *59*, 1431–1454. [CrossRef]
3. POV-Team. POV-Ray Persistence of Vision Ray-Tracing Software. 2021. Available online: http://www.povray.org/ (accessed on 2 December 2021).
4. Klensin, J. Simple Mail Transfer Protocol. October 2008. Available online: https://datatracker.ietf.org/doc/html/rfc5321 (accessed on 2 December 2021).
5. Fielding, R. Hypertext Transfer Protocol—HTTP/1.1. Available online: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html (accessed on 10 October 2021).
6. Skala, T.; Skala, K.; Afgan, E. Impact of 3d graphic structure complexity to the rendering time. *J. Circuits Syst. Comput.* **2013**, *22*, 10–16. [CrossRef]
7. Skala, T.; Todorovac, M. Comparison of stereo displaying techniques in POV-Ray 3D generated scenes. In Proceedings of the iiWAS2008, Linz, Austria, 24–26 November 2008.

8. Pavkovic, N.; Skala, T.; Vidić, V. Automatic Enlarge and Deployment of Computer Cluster Using Dual-Boot Approach. *Automatika* **2013**, *54*, 242–251. [CrossRef]
9. Skala, T.; Cviljušac, V.; Mustač, K. Development of algorithm for continuous generation of a computer game in terms of usability and optimization of developed code in computer science. *Acta Graph. J. Print. Sci. Graph. Commun.* **2017**, *28*, 55–58. [CrossRef]
10. Kerrisk, M. Linux Programmer's Manual—Sendfile(2). 2021. Available online: https://man7.org/linux/man-pages/man2/sendfile.2.html (accessed on 22 November 2021).
11. Kerrisk, M. Linux Programmer's Manual—Mmap(2). 2021. Available online: https://man7.org/linux/man-pages/man2/mmap.2.html (accessed on 11 December 2021).
12. Chong, A.; Sourin, A.; Levinski, K. Grid-based computer animation rendering. In Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, Kuala Lumpur, Malaysia, 29 November–2 December 2006.
13. Sheharyar, A.; Bouhali, O. A Framework for Creating a Distributed Rendering Environment on the Compute Clusters. *Int. J. Adv. Comput. Sci. Appl.* **2013**, *4*, 117–123. [CrossRef]
14. Burns, B. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*; O'Reilly Media: Newton, MA, USA, 2018; p. 20181.
15. Campbell, L.; Majors, C. *Database Reliability Engineering: Designing and Operating Resilient Database Systems*; O'Reilly Media: Newton, MA, USA, 2017.
16. Akenine-Möller, T.; Haines, E.; Hoffman, N. *Real-Time Rendering*, 4th ed.; Taylor and Frances Group, LLC: Philadelphia, PA, USA, 2018.
17. Jeffers, J.; Reinders, J. *High Performance Parallelism Pearls*; Elsevier Inc: Amsterdam, The Netherlands, 2015; Volume 2.
18. Gooding, S.; Arns, L.; Smith, P.; Tillotson, J. Implementation of a Distributed Rendering Environment for the TeraGrid. In Proceedings of the 2006 IEEE Challenges of Large Applications in Distributed Environments, Paris, France, 19 June 2006; IEEE: Manhattan, NY, USA, 2006; pp. 13–21.
19. Tanenbaum, A.S.; Steen, M.v. *Distributed Systems: Principles and Paradigms*, 2nd ed.; CreateSpace Independent Publishing Platform: Scotts Valley, CA, USA, 2016.
20. Kleppmann, M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, 1st ed.; O'Reilly: Newton, MA, USA, 2017.
21. Rigney, C.; Willens, S.; Simpson, W.; Rubens, A. Remote Authentication Dial in User Service (RADIUS). June 2000. Available online: https://datatracker.ietf.org/doc/html/rfc2865 (accessed on 2 December 2021).
22. Bishop, E.M. Hypertext Transfer Protocol Version 3 (HTTP/3). 2 February 2021. Available online: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34 (accessed on 2 December 2021).
23. Sysoev, I. Our Roadmap for QUIC and HTTP/3 Support in NGINX. 12 July 2021. Available online: https://www.nginx.com/blog/our-roadmap-quic-http-3-support-nginx/ (accessed on 2 December 2021).