Implementing an Automated Data Replication System for the Genesis II Platform

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Assad Aijazi

Spring, 2020.

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Andrew Grimshaw, Professor of Computer Science, Department of Computer Science

1. Introduction

1.1 Background/Purpose

Data Replication, or the duplication of objects (directories or files) across different Genesis containers, is desirable for two major reasons: availability and performance. Replication increases availability by ensuring if a client needs to access an object from a container that is unavailable, it can be rerouted to a different container that has a replica of that object. Replication can also improve performance by dispersing load for a specific object across different containers. Additionally, replicas can be used to place objects at containers closer to the clients that access them, which can increase client performance as well. Due to these listed benefits, the goal of this project is to design and implement a data replication system in an existing grid computing platform, the Genesis II platform.

1.2 The Genesis II Platform

The Genesis II platform is an open source, standards based grid platform designed to support both high-throughput computing and secure data sharing. Through a component called the Global Federated File System (GFFS), clients can access programs and files running on other computing sites, such as other college campuses, as if they were located on the local filesystem. In addition to having a grid client, participating sites can also host containers that can be linked to the GFFS to be accessed by other clients. Containers contain a variety of services for various resources, such as ByteIO (files), and RNS (directories), which allow grid clients to access these resources.

As of now, the grid client supports the creation of replicas of objects on a particular container via the *replicate* command. However, since replication must be done manually, it is difficult to fully leverage its usefulness since many decisions involving replication (i.e. when to

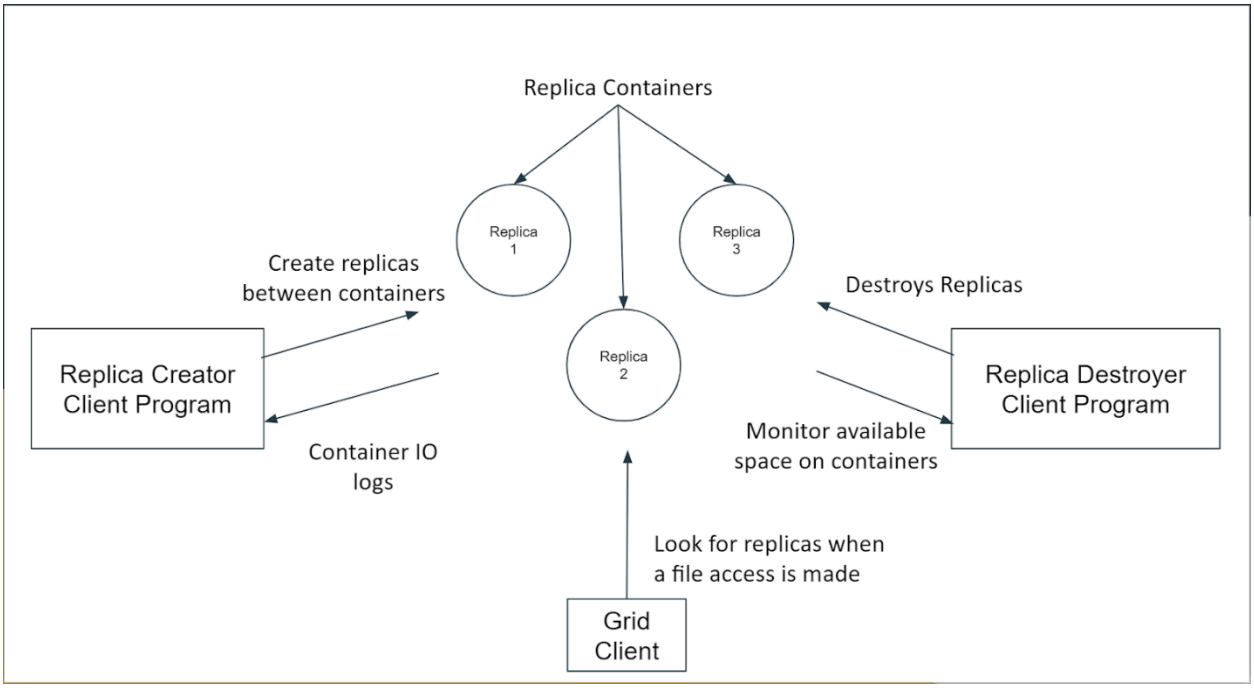
replicate, what object to replicate, where to replicate to etc...) have to be made in real time. Thus, the goal of this project is to create a system to automate the decision-making involved in the replication process to allow the benefits of data replication to be fully realized.

2. Design

2.1 Overall Design

The project can be divided into three major components, each of which corresponds to the main categories of decisions involved in the data replication process: creating replicas, using replicas, and destroying replicas. Creating replicas will be managed by a separate client program, which will use the container IO logs to analyze reads, writes, creation, and deletion of files by clients and use that information to create replicas on the participating containers. The client code will have to be updated to make sure that whenever an access is made to an object, it will be responsive to the created replicas and utilize them, rather than access the original object every time. Destroying replicas will be managed by another separate client program, which will have access to the container IO logs, will analyze the available space available on the containers, and delete replicas according to some algorithm (LRU). A diagram of the design is shown below in figure 1.

Figure 1: Data Replication System Design



2.2 Creating Replicas

The replica creation process will be managed by a separate client program running on a location separate from the containers. The program will be fed log files from the different replica containers, and analyze the information contained to make decisions involving replica creation. There are 3 major decisions that will be made: when to create a replica, what to replicate, and where to replicate. With regards to when to replicate and what to replicate, the main piece of information that will be used to make these decisions is the amount of times a file/directory has been accessed. The client program will be responsible for tracking the amount of times a singular file/directory has been accessed (including creates, reads, and writes), using the information contained within the log files. When the amount of accesses exceeds some predetermined threshold, the program will make the decision to replicate the resource.

A situation that should be avoided from using this method is accesses that occurred in the distant past skewing present replication decisions. For example, assume that the threshold of accesses for a resource to be replicated is 10. If 9 accesses occurred in the distant past, it does not make sense for those to weigh so heavily as to trigger a replication of a resource that was accessed once in the present. In order to remedy this process, the resource access counts will be aged by halving the counts whenever a defined period of time has occurred. The period of time to use for this will have to be experimentally determined.

In regards to where the program will replicate the resource to, several options were considered in the design process. One choice was to arbitrarily pick one of the replicas at random to replicate to. The main advantage of this is ease of implementation, as it requires no additional information to make the decision. However, this approach is limited in that it has potential performance ramifications. If a container is chosen at a suboptimal location for the clients that utilize the given resource in terms of latency or bandwidth, it may result in slower access times for the clients, making it take longer to access the resource. Another potential solution would be to build a list that assigns ranges of ip addresses to specific containers. The program could then use this list to determine what container to replicate to based on the ip address of the client. This would require to track not a resource's access history but also the ip address of the client that made the access. While this solution would provide the most performance benefit, there is significant overhead in both developing such a list, and also keeping track of client's ip addresses. There is also the problem of resolving the decision of which container to use in the case of multiple clients accessing the resource. Another potential solution would be to experimentally determine the best container to use for each client, either through some measure

of latency like ping or by geographical distance. The drawbacks of this are the tediousness of developing such experiments, and it would not scale well in the case of numerous clients.

While using resource access counts is a viable primary metric to use for data replication, a limitation of this approach is the focus on temporal locality over spatial locality. With the current approach, resources will be replicated based on how quickly they are accessed in terms of time. However, this completely ignores spatial considerations; in many use cases, if a resource is accessed frequently, it is likely that resources located near it on the directory tree will be accessed as well. Thus, replicating not only the resource itself, but resources around it, would yield more optimal performance benefits. However, for the purposes of this project resource access history was considered a sufficient metric.

2.3 Using Replicas

Once replicas of objects can be created, changes to the client will have to be made so that clients can utilize these replicas. Thus, whenever an access to an object needs to be made, first, you must query the resolver service to determine if the object has any replicas. A cache will be utilized to ensure that you do not have to query for replicas every time. Then, a decision of what replica to use will have to be made. In order to assist with this decision, the client preferences will be updated to contain metadata that can help determine which hostname corresponds to a local replica. Then, if there is a replica located at that particular hostname, the client will utilize that replica. Otherwise, the client will pick a replica from the list at random, which will ensure that load for a particular object will be dispersed across all of the different replicas of that object.

2.4 Destroying Replicas

In order to ensure that old replicas do not take up available resources, a separate client program will be responsible for destroying replicas. This program will monitor the available

space of all of the replica containers. When the amount of space taken up by replicas exceeds a certain threshold (say 80%), the program will make the decision to delete replicas to get the memory usage back under the threshold. To accomplish this, the program will have access to the container access logs (and hence the size of every created replica) as well as a list of replicas from each container that are available for deletion.

A major design question is choosing which algorithm to use for deleting replicas. While it is impossible to create an optimal algorithm due to the lack of access information, a sufficient heuristic can be created based on the available information to choose which of the available replicas to delete. Potential variables to use as part of the heuristic include replica size, access time, and number of existing replicas of the resource. To simplify this algorithm for the purposes of this project, a basic least recently used (LRU) algorithm will be used, where the replica that was accessed the least recently will be deleted. In the case that the memory usage still exceeds the threshold, this algorithm will be called repeatedly until the threshold is met. An important constraint on this algorithm is making sure that the resource has other replicas on other containers. In other words, a deletion of the replica will not result in the permanent deletion of the resource. To ensure proper availability, it may be required that a certain number of replicas (maybe 2 or 3) exist for the resource before it is considered a candidate to be deleted.

3. Implementation

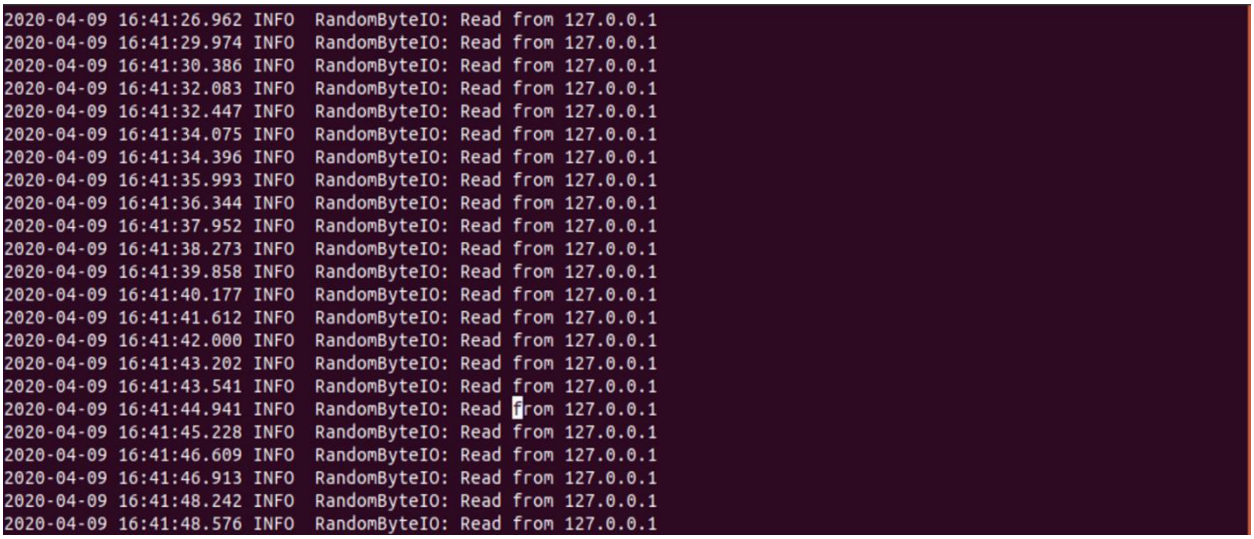
While the entire implementation was not able to be completed in the available time frame, a basic proof of concept was created to demonstrate the viability of the design. In general, the basic functionality of the creation of replicas was completed, as well as the basic functionality of the using replicas based on the available replicas.

3.1 Creating Replicas

Several updates had to be made to the base Genesis II code in order to support creating replicas. First, the io log file had to be updated to also output the grid path of the file that was accessed in the logged action. This path was necessary to obtain the Endpoint Reference (EPR) of the file that is needed by the replicate command to actually perform the replication. The grid path, however, was not readily available, however, as the io log file is updated at the container level, and the grid path was only available on the client side. To get around this, a resource property, grid path, was added to the ByteIO and RNS types, and was set upon creation of the resource. Once this resource property was set, the container could query for the value from the EPR, and output its value to the log file. Figure 2 shows an io log file entry before and after the changes.

Figure 2: Container IO Log File Before and After Changes

Before:



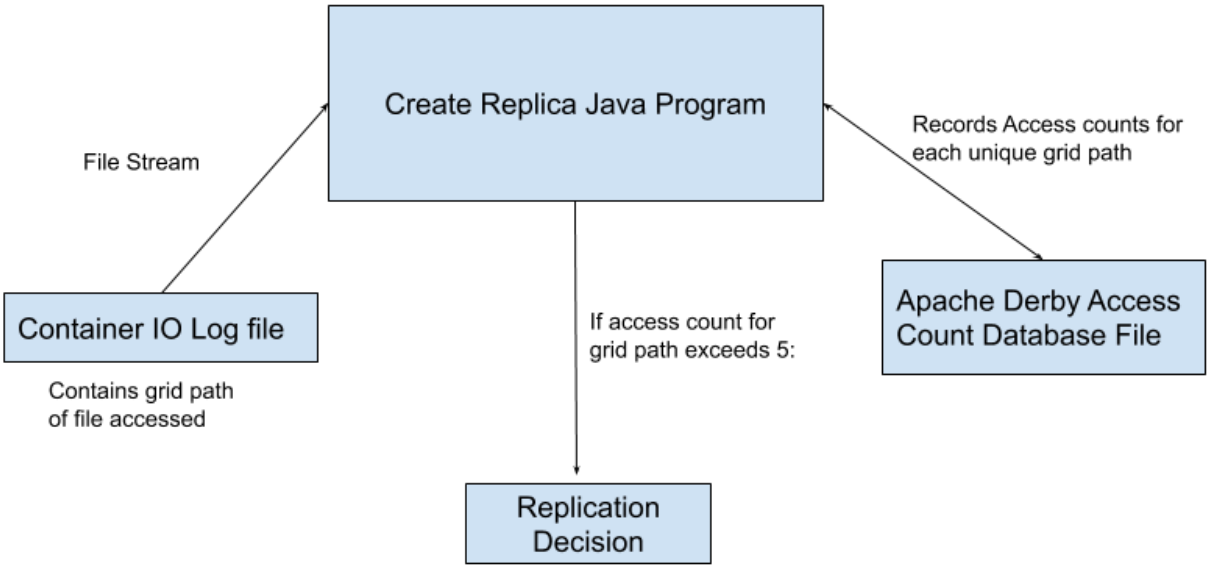
After:

```
2020-04-09 16:41:34.396 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:35.993 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:36.344 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:37.952 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:38.273 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:39.858 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:40.177 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:41.612 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:42.000 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:43.202 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:43.541 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:44.941 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:45.228 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:46.609 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:46.913 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:48.242 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:48.576 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:50.036 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:50.377 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:51.798 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:52.172 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:53.737 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
2020-04-09 16:41:54.035 INFO  RandomByteIO: Read gridPath '/home/xsede.org/userX/sandbox/test2.txt' from 127.0.0.1
```

As noted, the new log file entry now contains a gridPath field, which shows which file on the grid was accessed. This new field is used by the client program to make replication decisions.

Once, the log file changes were made, the client program for creating replicas was set up. The program was implemented as a Java program. The program would create a file stream to the log file, parse the entries for grid paths that were accessed. The program would then keep track of the access counts of the grid paths using a simple Apache Derby embedded database. If the counts exceeded a certain threshold, (arbitrarily picked to 5 for testing purposes), then the program would output the decision to replicate. In this implementation the container to replicate to was picked at random. Figure 3 shows a basic flow diagram of the program implementation structure.

Figure 3: Create Replica Program Structure



3.2 Using Replicas

In order to look for replicas, the code for resolving a grid path was updated. Rather than just pick the first Endpoint Reference, the existing list replicas command was used to check if any existing replicas of the file or directory existed. If it was the case, then one of the other replicas was picked at random.

4. Further Work

A great deal of work on this project still needs to be done to bring the proof of concept implementation to production-level ready. In addition to completion of implementation, a testing system is still needed to ensure that the proposed system is actually beneficial in terms of performance.

4.1 Completion of Implementation

All three aspects of the project still have implementation details specified in the design that will still need to be completed. In terms of creating replicas, the main component that needs to be completed is the algorithm for selecting which container to which to replicate a resource. The current implementation is to pick a container at random, which will on average not provide any performance boosts to clients. The design in previous sections proposes several alternatives, such as developing a table that maps specific ip address ranges to containers, or performing some method of experimentally determining the best container through a method like ping. Another missing component from the implementation is the aging of access counts to ensure that accesses in the distant past are not weighed as heavily in present replication decisions.Additionally, with regards to the using replicas, the caching of resolving requests still needs to be implemented to ensure the resolver service is not overloaded. Finally, the system for destroying replicas has been completely unimplemented; in a real system, this component would be integral to making sure replicas can still be created without running out of disk space too quickly.

4.2 Testing

Once implementation is complete, a testing procedure will need to set up in order to measure if the replication system improves performance. A way to perform this test is to set up an experiment measuring the access time from different clients to a file, comparing the results with the data replication system in place with a baseline without the data replication system in place. If the data replication system is successful, one would expect that eventually access times with data replication will improve overall, as clients are accessing a replica of the file at a site that is faster to access than before. Another thing to measure is the load of client accesses to a container. With proper data replication, the overall load should be more spread out across the

different replica containers since every client will have several containers to choose from. If neither of these two metrics show any improvement, then that would be a sign to reassess the design for any shortcoming and analyze the implementation for any bottlenecks.