M123: Distributed Systems 2017

Final Project

A secure and anonymous storage service

Sakellari Elisavet
M1386

July 2017

A secure and anonymous storage service (Freenet-like).

Introduction

I have implemented a simulation of a secure and anonymous storage service, similar to Freenet. Specifically, I have implemented the algorithms as described in Freenet [1] for routing, new node joining, and file encryption, with a few differences.The project is implemented in Java, and the nodes communicate with each other through RPC (Java RMI). The tests have been conducted on my personal laptop, and all the nodes reside on the same computer, so all the lookups for the remote objects are done on the localhost.

Algorithms

The project follows the algorithms as described in the Freenet [1] paper for node joining, insert and request routing, and file encryption and decryption. There are however some differences which I describe in detail below.

Node joining

A new node enters the system by contacting one of the nodes that already participate in the system. In my simulation, the new node is supposed to already know somehow the address of the initial node it will contact and not discover it through other means (for example DNS). When the new node first contacts the node of the system the process of the 'new node announcement' begins, for a number of hops that the new node has decide. The number of hops corresponds to the number of nodes that will be contacted during the procedure of the new node announcement , and that will be informed about the new-coming node (add it in their routing table). These nodes also play a role in 'constructing' the unique key that will be the identifier of the new node. The procedure followed for the key construction is exactly like the one described in [1]. Through this procedure the new node acquires its unique key, by which it will be chosen among the other nodes in the routing procedures. The new node adds the only node it knows about in its routing table, and in its first insert or request it will contact this node, until more nodes are added in the routing table. Every node in the system has also a unique pair of a **private** and a **public key**.

File encryption

A node that wants to add a new file in the system must first **fragment** it in smaller pieces, encrypt those pieces, and then create an ***indirect file*** which will have the information about these pieces. This information consists of the **Content Hash Keys (CHKs)** of the smaller files and their decryption keys. Each decryption key is different for every 'fragment' file and is randomly produced during the insertion procedure. The indirect file is also encrypted. The inserting node also **signs** the data of all the files with its private key.

For other nodes to retrieve the file, the inserting node must provide them with its public key and a secret string ID with which it has encrypted the indirect file. Once another node has retrieved and decrypted the indirect file, it must then search for all the smaller pieces of the original file by using their CHKs. Once it has acquired them and collected them together, it must decrypt them by using the decryption keys that are paired with the Content Hash Keys in the indirect file, and verify the data by using the public key of the inserting node.

The following diagram shows the architecture of the file fragmentation.

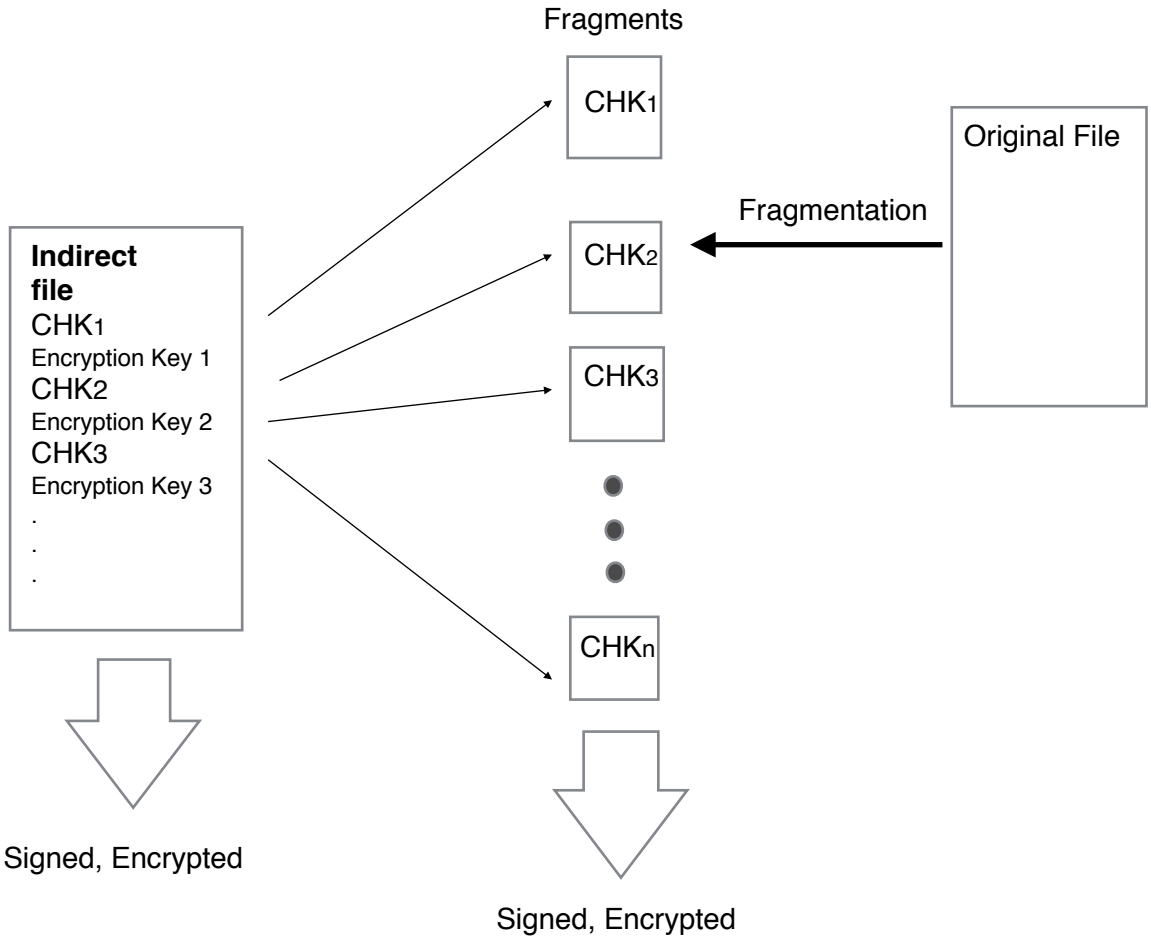


Diagram 1. The architecture of the file fragmentation technique.

Basic routing algorithm

A node that is looking for a specific file key must choose a node among the entries in its routing table to forward the request. This is done by calculating a ***lexicographic distance*** between the key it is looking for and the key that each node in the routing table has. The node chooses the node whose key is ***lexicographically closer*** to the key it is looking for, and forwards the request to it. If this node is not available, which means it may have failed or departed from the network*, it chooses the next node whose key is lexicographically closer to the key it is looking for.

* In my implementation, a node failure or departure, is simply achieved by removing the entry of this object from the rmiregistry.

File insertion

To insert a file, the basic routing procedure described above is used in order to forward the file key.

When the insertion procedure begins, a **random string ID** for this file is created, and which is used to **encrypt** the indirect file. Then the **unique file key** for this file is produced by XOR-ing the hash of the string ID with the hash of the public key of the inserting node. The same procedure is followed when a node wants to retrieve the file, so it must first calculate the file key of the indirect file. As mentioned earlier, a node that wants to insert a file in the system it must first fragment it and then follow the insertion procedure for each one of the fragment-files with their Content Hash Keys as the file keys, and for the indirect file with its file key.

My implementation deals with collisions. In case the file key already exists in some node, the node that holds it forwards the file back to the upstream node which also stores it and adds the

node that has the file key in its routing table. The same happens with all the nodes in the route that forwarded the file key. When a node inserts a file, it sets a maximum number of hops for which it wants the procedure to continue. This maximum number of *hops* also plays the role of the *file replication,* since the procedure will not stop until zero hops are remaining. In case no collision is found, the file with its file key is stored on the nodes that have been successfully contacted during the procedure. This way the newly-inserted file gets replicated.

The inserting node must **publish** its public key and decryption string ID of the file to the nodes it wants to allow access to.

File request

As mentioned in the previous algorithm, a node calculates the file key of the file it is looking for from the public key and the decryption string ID it has acquired from the owner of the file.

The basic routing algorithm is used once more, but in this case a 'collision' is treated as a success because it means that we have found the file we are looking for. The node that has the file forwards the data to its upstream requestor, which in turn stores them in its datastore. The same happens with all the nodes that have been contacted along the procedure. The nodes also add an entry in their routing table with the information about the node on which we found the data, if this entry does not already exist.

The initial requesting node once again sets a maximum number of hops for which we will be contacting nodes. If there are no remaining nodes and we still have not found the file key we are looking for, then a message that the request failed is returned to the initial node. In case of success, the initial requestor also becomes a holder of the pieces of the file it requested (and of the indirect file), since it keeps them in its datastore, and following requests from other nodes for these files can be serviced from it.

Performance - Security/Anonymity

As also mentioned in the Freenet paper [1], the performance of the system gets better as time goes by, since gradually the nodes find out more information about other nodes in the system. and cache replicas of the fragments of the files. Also, the technique that uses the lexicographic distance between the file keys and the nodes keys, helps by in some way 'clustering' files with specific keys in specific nodes. However it should be mentioned that the file keys are derived in such a way that they give no information about the content of the files. Even in the case of the Content Hash Keys, no information can be derived from the key. Also, the nodes that keep encrypted parts of files in their datastores have no way of extracting or decrypting the content of those files, since their decryption keys are kept separately in encrypted files (the indirection files) on other nodes of the system. One can only decrypt those files by acquiring the initial decryption key (decryption string ID) of the indirect file, which the owner of the file gives only to the nodes it trusts.

Evaluation

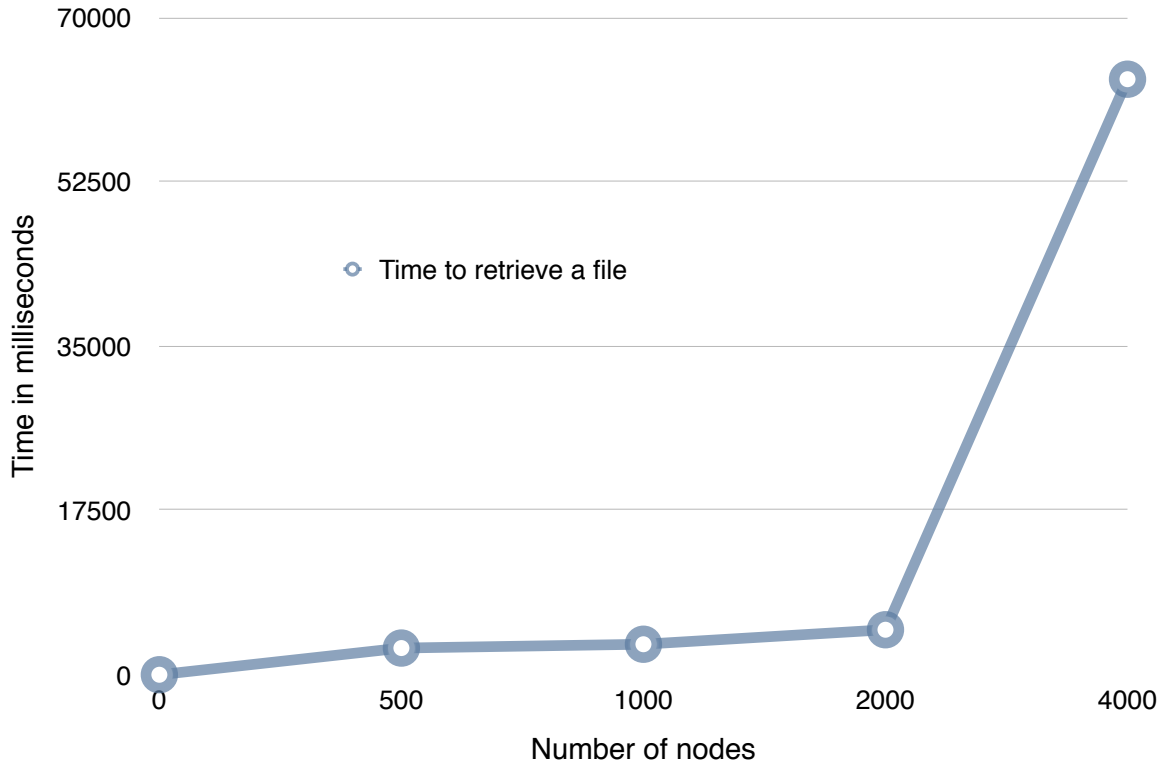
I have conducted some experiments on my personal laptop in which I have measured and compared times and path lengths of get requests, on clusters of maximum of 4000 nodes. I have used files of size 115 KB and fragments of size 2 KB (of course also larger files can be stored in the system).

Get-request, time metric

In this test the time retrieval for a file right after it has been inserted in the network is counted. I have experimented on clusters of 500, 1000, 2000, and 4000 nodes.

Number of nodes / Operation	join	put	get
500	5	5	10
1000	5	5	10
2000	5	5	10
4000	5	5	15

Table 1. Maximum number of hops for this test on different sizes of clusters.



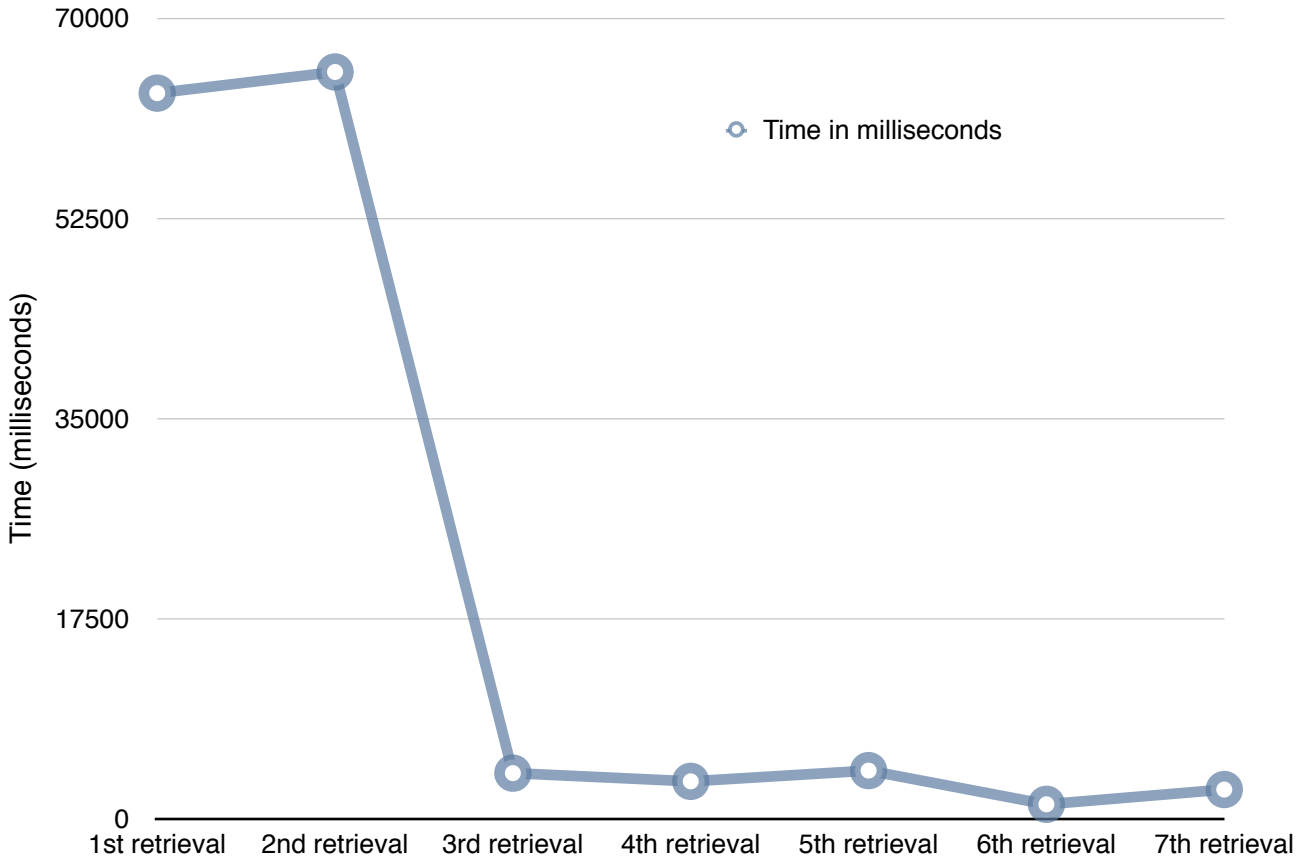
Graph 1. Time for the retrieval of a file right after it is inserted in the network.

Path length and time of a get request after a few get operations

In this test I calculated the path length and time for the retrieval of a file, while the system is working. I have created a graph that combines the order of the file retrievals with the time/path length. The maximum number of hops are shown in the following table. File size = 115KB fragmented in pieces 2KB each.

Number of nodes/ Operation	join	put	get
4000	5	5	20

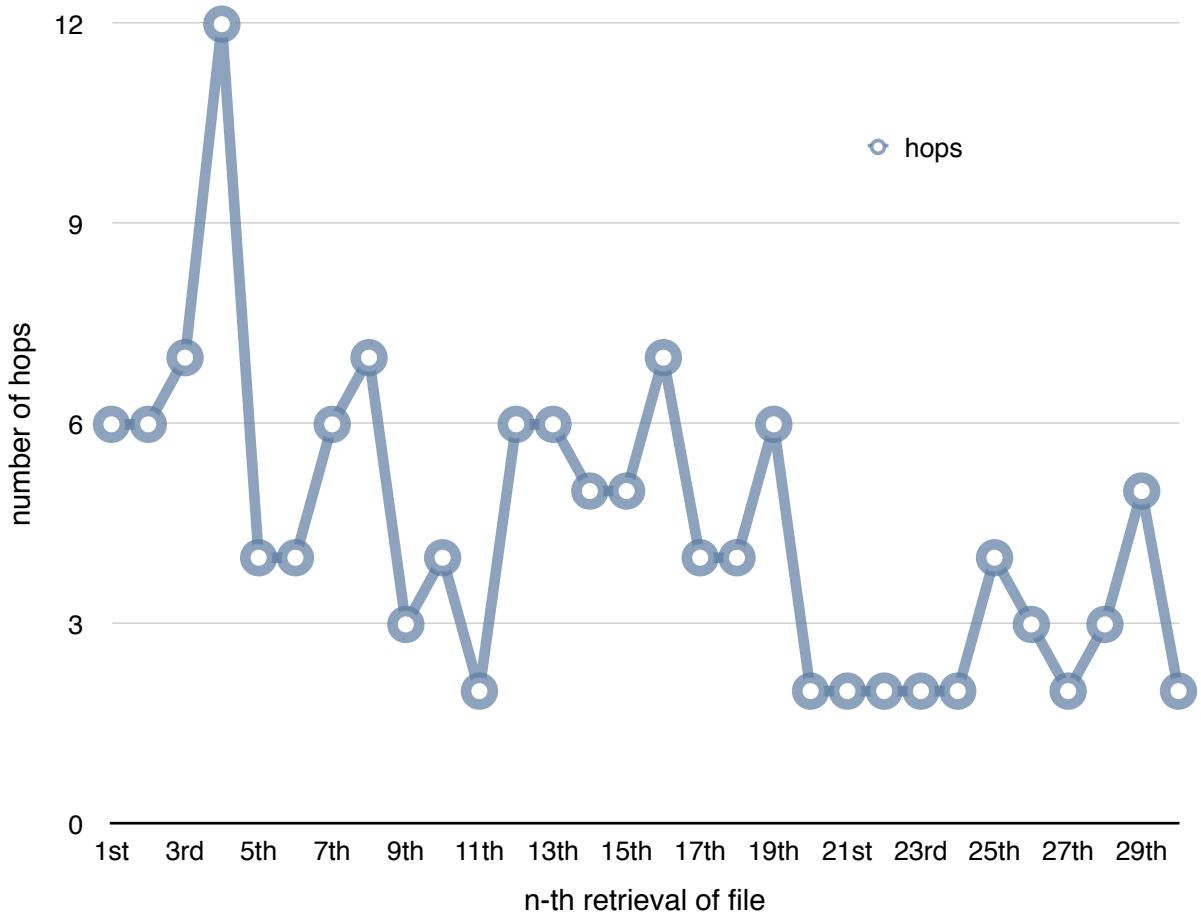
Table 2. Maximum number of hops for join, put, and get operation,



Graph 2. Time for the retrieval of a file by serial retrievals from different nodes.

As we can see from the Graph 2, the time needed for the retrieval of a file gets better by time, since more and more nodes get replicas of the pieces of the file, as well as of the initial indirect file.

In the following graph is presented the path length for retrievals of a file from different nodes. The same as in the test for time, the path length gets somewhat reduced while the system is working and the request for this file becomes faster serviced.



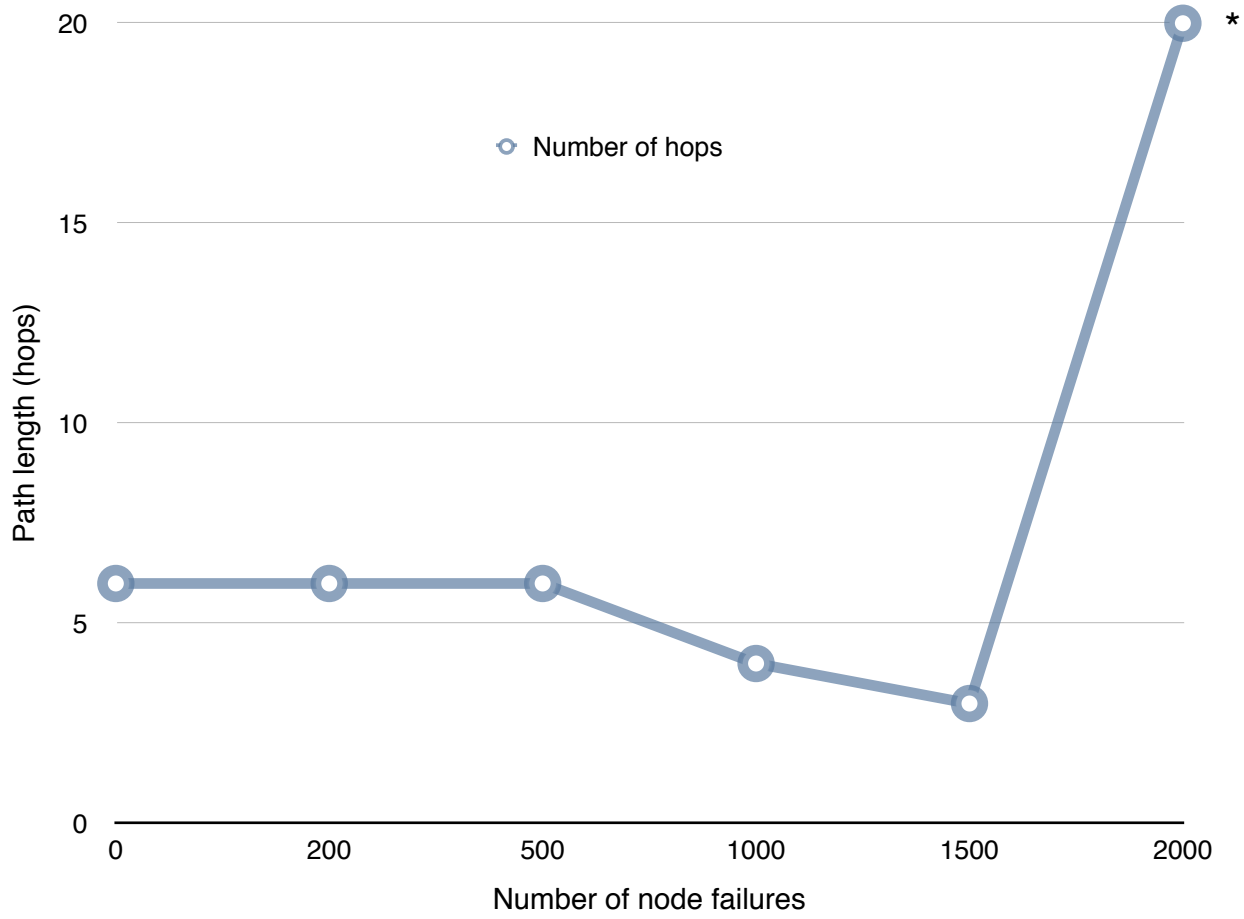
Graph 3. Path length for the retrieval of a file by serial retrievals from different nodes.

Node failures

In this experiment I have measured the time and path length needed for a get request after some nodes have failed or departed from the network. This test was conducted on a cluster of 3000 nodes. The maximum number of hops are shown in the following table.

Number of nodes/ Operation	join	put	get
3000	5	5	20

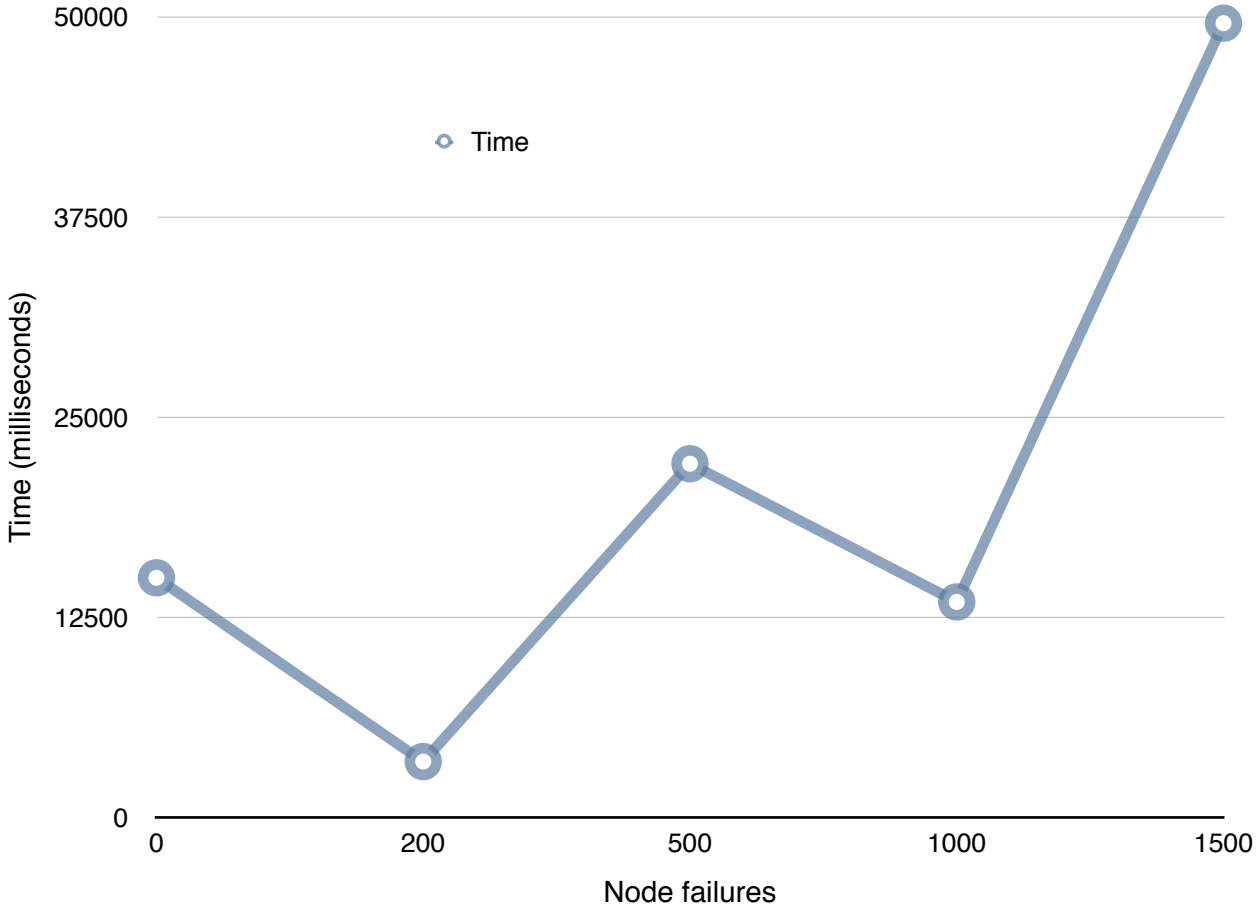
Table 3. Maximum number of hops for join, put, and get operation on a 3000-node cluster.



Graph 4. Path length for the retrieval of a file during node failures.

* When 2000 out of 3000 nodes of the system have departed or failed, most file requests were unable to be serviced.

In the next graph is presented the time needed for the retrieval of a file during node failures or departures. From the graph we can see that the time for a file retrieval gets only a little worse, while at the failure of half the nodes (1500 out of 3000) the retrieval time gets significantly increased.



Graph 5. Time for the retrieval of a file during node failures.

References

[1] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong, **"Freenet: A Distributed Anonymous Information Storage and Retrieval System"**, International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability. New York, NY: Springer-Verlag, 2001