MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
SUMY STATE UNIVERSITY
UKRAINIAN FEDERATION OF INFORMATICS

PROCEEDINGS
OF THE V INTERNATIONAL SCIENTIFIC CONFERENCE

ADVANCED INFORMATION SYSTEMS AND TECHNOLOGIES

AIST-2017
(Sumy, May 17–19, 2017)

**SUMY
SUMY STATE UNIVERSITY
2017**

The Vth International Conference «Advanced Information Systems and Technologies, AIST 2017»
17-19 May 2017, Sumy, Ukraine

Creating Highly Available Distributed File System for Maui Family Job Schedulers

Andrii Onishchuk
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

This article describes a way to implement a distributed file system for MAUI job scheduler, which solves the problems of low scalability and unreliability of data storage, as well as a problem of problem of data inaccessibility due to failures in software or hardware. The architecture which is suitable for MAUI GRID systems is suggested.

***Keywords* – distributed file system, MAUI task scheduler, file system scalability, high reliability of file systems, name node, datanode.**

I. INTRODUCTION

In recent years a rapid increase in popularity of distributed systems has been seen. The reason this is in their greater reliability, scalability and power. That's why a need for simple and convenient software that simplifies the user experience for such a systems has drastically increased.

Currently to run MAUI task user of GRID system must manually copy the files to each node in the system, which increases the likelihood of errors. Also this approach has a number of other disadvantages compared to using the concept of a distributed file system for organizing the files workflow. The advantages of using this concept are scalable and reliable data storage, data accessibility and low cost of equipment for storing files.

II. ARCHITECTURE OF DISTRIBUTED FILE SYSTEM FOR MAUI

Distributed File System (DFS) is a file system where the file pieces (blocks) are stored on a bunch of computers connected with high-bandwidth network [1]. The system which is described in this article is a subtype of DFS, which is used for Maui Scheduler.

The system has two main types of nodes: namenode and datanode. And two types of ancillary nodes: journalnode and standby namenode. Figure 1 shows the architecture, placement and interaction of these tytpes nodes. Namenode is a master node in DFS. There is only one active namenode in the DFS. It stores metadata of files, as well as information about where data are stored in the cluster file. Metadata examples are file names, their types, permissions, data about blocks location within the network. Namenode does not store any file blocks. This is done in order to reduce the load of it. In most file operations first, and sometimes the only call is made to namenode. The exception is a write operation to a file, which requires coordination between all types of file system nodes.

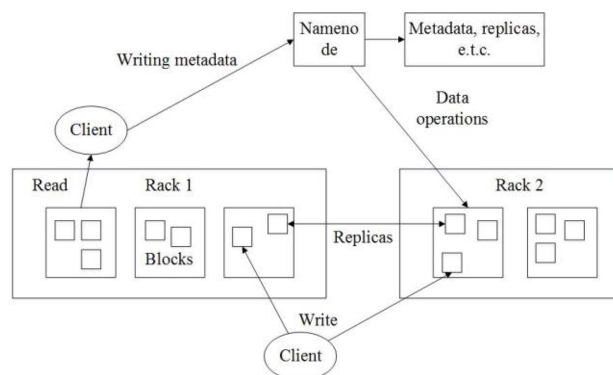

Figure 1. Architecture of RFS

MAUI jobs communicate with namenamenode every time they want to find a file, or add / copy / move it.

Datanode – is a slave node (master/slave achritecutre), the main purpose of which is to preserve data blocks. To take advantage of the RFS one should have more than a single datanode in a GRID system. Each node is aware of the blocks on it. The blocks are accessible namenode tells the user an exact location of them of datanode. In addition, datanode can replicate file blocks to improve system reliability. Also due to replication, this type of nodes usually does not require administrators to install RAID drives.

How does the namenode choose which datanodes to store replicas on? There's a tradeoff between reliability and write bandwidth and read bandwidth here. For example, placing all replicas on a single node incurs the lowest write bandwidth penalty since the replication pipeline runs on a single node, but this offers no real redundancy (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Maui GRID systems to date have run in), there are a variety of placement strategies. Indeed, MAUI changed its placement strategy in release 0.17.0 to one that helps keep a fairly even distribution of blocks across the cluster. And from 0.21.0, block placement policies are pluggable.

MAUI's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first (off-rack), chosen at random. The third

replica is placed on the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes on the cluster, although the system tries to avoid placing too many replicas on the same rack.

Once the replica locations have been chosen, a pipeline is built, taking network topology into account. For a replication factor of 3, the pipeline might look like Overall, this strategy gives a good balance among reliability (blocks are stored on two racks), write bandwidth (writes only have to traverse a single network switch), read performance (there's a choice of two racks to read from), and block distribution acrossthe cluster (clients only write a single block on the local rack).

For fast file blocks access, namenode caches often used blocks. Therefore, increasing the number of nodes on the RAM data although it may result in a slightly faster performance, but not critical.

Another part of the file system is an RFS client. RFS client is a software library that allows you to work with the file system using simple unix-like commands. Allowed commands are: ls, rm, mkdir, touch, some other RFS specific commands are: copyFromLocal, copyToLocal. These commands allow you to copy files or folders from the local file system to RFS and vice versa. Another RFS client is MAUI RFS client integrated with MAUI API interface. It is used to do file operations through built MAUI interface [2].

Let's take a look at the example of the interaction of the nodes in case of basic file operations. The most complex operation in terms of nodes interaction is the RFS file write operation (WRITE). It consists of four steps described below:

1. Call to namenode for a list of datanodes which should receive file blocks.
2. Uploading of file blocks to the given nodes.
3. Replication of the received blocks by datanodes.
4. Sending the information about the file blocks location to the namenode.

Operation of changing file attributes (CHANGEATTR) such as permissions, name, location is done with one call to namenode. The same applies to the file removal operation (RMFILE), creating a folder (MKDIR) or empty file (TOUCH). File copy operation is implemented through a series of calls from namenode to datanodes.

Let's define RFS file block concept in greater details. RFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 64 MB, although many HDFS installations use 128 MB blocks. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

Let's describe the permission model in our DFS. It has a permissions model for files and directories that is much like the POSIX model. There are three types of permission: the read permission (r), the write permission (w), and the execute permission (x). The read permission is required to read files or list the contents of a directory. The write permission is required to write a file or, for a directory, to create or delete files or directories in it. The execute permission is ignored for a file because you can't execute a file on DFS (unlike POSIX), and for a directory this permission is required to access its children [3].

Each file and directory has an owner, a group, and a mode. The mode is made up of the permissions for the user who is the owner, the permissions for the users who are members of the group, and the permissions for users who are neither the owners nor members of the group.By default, DFS runs with security disabled, which means that a client's identity is not authenticated. Because clients are remote, it is possible for a client to become an arbitrary user simply by creating an account of that name on the remote system. This is not possible if security is turned on. Either way, it is worthwhile having permissions enabled (as they are by default; see the dfs.permissions.enabled property) to avoid accidental modification or deletion of substantial parts of the filesystem, either by users or by automated tools or programs.

III. Organization for highly reliable distributed file system.

The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high-availability of the filesystem. The namenode is still a single point of failure (SPOF). If it did fail, all clients—including Maui jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Maui system would effectively be out of service until a new namenode could be brought online.On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.We remedy this situation by adding support for RFS.

In this implementation there is a pair of namenodes in an active-standby configuration. A few architectural changes are needed to allow this to happen:

- The namenodes must use highly-available shared storage to share the edit log. When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new

entries as they are written by the active namenode.
- Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, using a mechanism that is transparent to users.
- The secondary namenode's role is subsumed by the standby, which takes periodic checkpoints of the active namenode's namespace.

If the active namenode fails, the standby can take over very quickly (in a few tens of seconds) because it has the latest state available in memory: both the latest edit log entries and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), because the system needs to be conservative in deciding that the active namenode has failed. One of the main advantages of a distributed file system is the possibility of highly reliable file storage and access. The problems that might disrupt high reliability access to data or damage the data are [4]:
- hardware or software failure of namenode
- hardware or software failure of datanodes
- unavailability of data due to network problems

Hardware or software failure of datanode is solved using file blocks replication. Replication in this sense is excessive copying of data blocks between datanodes. The number of nodes which store data blocks is called replication factor. By setting up a high replication factor and replication to different network segments we can achieve reliable access to the files in the event failure of multiple nodes at once, or even of the entire network segments. In order to for file system to work when namenode failure occurs, the introduction of two types of auxiliary nodes is required. They are journalnode and stanby namenode.Two separate machines are configured as namenodes. At any given time, exactly one namenode is active and the other is in standby. Active namenode is responsible for all client operations in the cluster, while other one is in standby mode and is not used, however it still retains enough information about the state of the file system to ensure a rapid transition to it, if necessary. For standby namenode to be synchronized with the active node, both namenodes are connected with a group of individual nodes, so-called journalnodes. When any namespace change is performed by active node, it registers record modification to the journal nodes. Standby namenode is capable of reading the log journalnode and constantly monitors changes in it. As a standby namenode sees the changes, it applies them to its own namespace. In case of failure of the active namenode, reserve one, after reading all the logs from the journalnodes declares itself active. This ensures that the namespace is fully synchronized before the transition to another node happens. In order to provide a fast failover, it is also necessary that the standby node have up-to-date information regarding the location of blocks in the cluster. In order to achieve this, the datanodes are configured with the location of both namenodes, and send block location information and heartbeats to both.It is vital for the correct operation of an GRID system that only one of the namenodes is active at any point in time. Otherwise, the namespace state would quickly diverge between the two, risking data loss or other incorrect results. In order to ensure this property and prevent the so-called "split-brain scenario," the journalnodes will only ever allow a single namenode to be a writer at a time. During a failover, the namenode which is to become active will simply take over the role of writing to the journalnodes, which will effectively prevent the other namenode from continuing in the active state, allowing the new active to safely proceed with failover.Inaccessibility due to network problems can be easily eliminated by organizing redundant network topology.The underlying concept behind network redundancy is to provide alternate paths for data to travel along in case a cable is broken or a connector accidentally un-plugged. However, Ethernet as standard cannot have rings or loops in the network as this will cause broadcast storms and can ultimately cause the network to stop working. An Ethernet network cannot have two paths from point A to point B without a mechanism in place to support this type of topology [5]. To achieve redundancy, the network infrastructure (switches) must support redundancy protocols designed to negate the usual problems of putting loops into an Ethernet network, maintaining a default data path and switching to an alternate one when a fault occurs.

CONCLUSIONS

This paper described the architecture of a distributed file system for MAUI, which achieves greater scalability and high reliability and availability of data storage.

REFERENCES:

[1] Ilya Ganelin. Spark: Big Data Cluster Computing in Production, Wiley, March 2016. p.21-32.
[2] Site Maui [Electronic resource] // Access mode: http://www.adaptivecomputing.com/products/open-source/maui/
[3] Neal Kobel. Distributed File Systems: Distributed Computing Architecture, CreateSpace Independent Publishing Platform, December 2016. p.41-67
[4] EC-Council. Computer Forensics: Investigating File and Operating Systems, Wireless Networks, and Storage (CHFI), 2nd Edition (Computer Hacking Forensic Investigator) 2nd Edition, Course Technology, April 2016. p.18-26
5. A. Tanenbaum, Computer Networks 5th By Andrew S. Tanenbaum (International Economy Edition), Prentice Hall, January 2010. p.648-697