Formalizing Google File System

Mengdi Wang*, Bo Li*, Yongxin Zhao† and Geguang Pu†‡
Shanghai Key Lab of Trustworthy Computing,
Software Engineering Institute,
East China Normal University, Shanghai, China
*email: {wangmengdi,tmacback}@gmail.com
†email: {yxzhao,ggpu}@sei.ecnu.edu.cn

***Abstract*—Google File System (GFS) is a distributed file system developed by Google for massive data-intensive applications which is widely used in industries nowadays. In this paper, we present a formal model of Google File System in terms of Communicating Sequential Processes (CSP#), which precisely describes the underlying read/write behaviors of GFS. Based on the achieved model some properties like deadlock-free, and consistency model of GFS can be analyzed and verified in the further work.**

I. Introduction

In recent years, the cloud computing has become an issue of vital importance in engineering field [1]. Google File System (GFS) [3] is a distributed file system designed and implemented by Google to cater for the rapid growing demands of massive data storage as well as concurrent client access. Many research efforts have been addressed to analyze and improve cloud computing architectures. But they do not investigate the underlying read/write mechanism along with the consistency model of GFS.

In this paper, we explore the underlying read/write behaviors of GFS and present a formal model based on CSP# [2] with respect to GFS architecture. Our CSP# model could precisely describe the behaviors of GFS. It not only formalizes the concurrent processes between the components but also clarifies the read/write operations.

This paper is organized as follows. Section 2 gives a brief introduction to GFS. The CSP# model of GFS are formalized in Section 3. At last, Section 4 concludes the paper.

II. GFS OVERVIEW

Google File System is a scalable distributed file system for large distributed data-intensive applications designed and implemented by Google Inc. in 2003. Benefits from the usage of multiple replications for every file chunk, GFS can provide strong availability and high reliability though the whole system is built from inexpensive commodity hardware. Besides, its high aggregate performance of delivering massive data to many clients meets the demand for massive storage.

Google File System typically consists of three main parts: *masters*, *clients* and *chunkservers*. The *Master* maintains all file system metadata. As the store units of GFS, chunks are managed by *chunkservers*. The *clients* interact with the *master* for metadata operations, and directly communicate with *chunkservers* to read and write content.

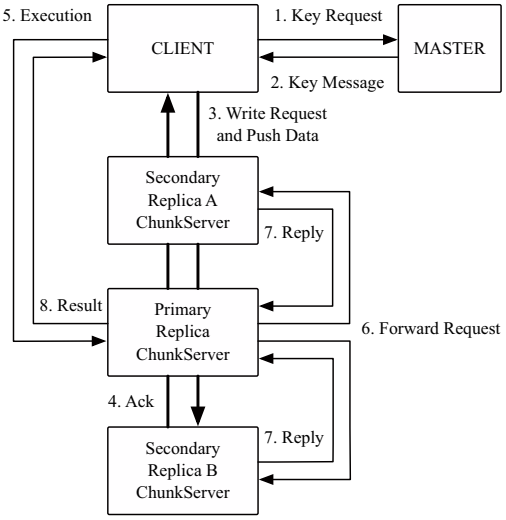

Fig. 1: Write Process

Fig. 1 depicts the overall flow of a write operation related to our formalized model. In the figure, there are five nodes. They are client, master, one *primary* chunkserver and two *secondary* chunkservers. The *MASTER* designate a *primary* server to control the write process while other *CHUCKSERVERS* contain the replica is called *secondary* replica server. The detailed write steps are listed as follows:

1) The client requests key from the master
2) The master returns the file location information
3) The client sends write requests and pushes the data to all the replicas.
4) The chunkservers reply acknowledge to the client.
5) The client sends a write execution request to the primary chunkserver.
6) The primary chunkserver forwards the write request.
7) All the secondary chunkservers reply to the primary.
8) The primay chunkserver replies the result to the client.

III. Modeling GFS in CSP#

In this section, we present a formal CSP# model of the Google File System.

‡Corresponding author

978-1-4799-6474-1/14 $31.00 © 2014 IEEE
DOI 10.1109/PRDC.2014.32

A. SYSTEM

There are three crucial processes running in parallel:

$$SYSTEM \triangleq MASTER \parallel CLIENTS \parallel CHUNKSERVERS$$

During the process, the *CLIENTS* issue the read/write requests, the *MASTER* manages the file location meta data and the *CHUNKSERVERS* execute the actual read and write operations. For brevity, the remainder of this paper will only focus on formalizing *CLIENTS* process.

B. CLIENTS

To clarify the *CLIENTS* process, some supplementary functions are firstly introduced as follows,

- $HasKey_i()$ checks whether the client contains the key message of the file.
- $StoreKey_i(KeyReq, KeyMsg)$ stores the requested key for further use.
- $GetN(hdl)$ returns the *id* of the nearest chunkserver which stores the replica.
- $GetRepl(hdl)$ returns all the chunkserver *id*s as an array according to the handle.

Next, the channels used in the model are illustrated below:

- The channel cl_ich_j will represent the read or write data requests sent from $Client_i$ to $Chunkserver_j$.
- The channel ch_jcl_i will represent the replies sent from $ChunkServer_j$ to $Client_i$.
- The channel cl_im will represent the key requests sent from $Client_i$ to Master.
- The channel mcl_i will represent the key information sent from Master to $Client_i$.

Clients could simultaneously access to the GFS, each of which may generate a unique application *id* marked as i and corresponds to a client progress. Under that condition, *CLIENTS* can be formalized as follows.

$$CLIENTS \triangleq \parallel_{i\epsilon[0,I-1]} Client_i$$

The $Client_i$ process represents the application to fetch chunk information from the master and do read/write operations on the chunkservers. It can be formally defined as follows:

$$\begin{aligned} Client_i \triangleq\ & ifa(!HasKey_i(fn, idx))\{GetKey(fn, idx)\} \\ & \rightarrow \{hdl = GetHdl(fn, idx)\} \\ & \rightarrow (Read_i(hdl, R) \Box Write_i(hdl, BRange, Data)) \\ & \rightarrow Client_i \\ GetKey_i(fn, idx) \triangleq & \\ & cl_im!KeyReq \rightarrow mcl_i?KeyMsg \\ & \rightarrow StoreKey(KeyReq, KeyMsg) \rightarrow Skip \end{aligned}$$

The whole process of $Client_i$ can be described as three main parts, the key request operation, read operation and the write operation. Here, we propose a variable R stands for the *id* array of the replica chunkservers. $R[0]$ stands for the primary chunkserver and remains are the secondary chunkservers.

The key request operation is an auxiliary process. In the operation, a client will call the *HasKey* function with the file name *fn* and chunk index *idx* to check whether the corresponding *KeyMsg* is stored or not. If it does not hold the *KeyMsg*, it will send a *KeyReq* message to the Master through the channel cl_im. When the *KeyMsg* is received on channel mcl_i, it will be stored by function *StoreKey* for further use.

To reflect the characteristic of the read/write sequence, the formalized read and write is given as follows:

$$\begin{aligned} Read_i(hdl, BRange) \triangleq\ & \{j = GetN(hdl)\} \\ & \rightarrow (cl_ich_j!RReq \rightarrow ch_jcl_i?RRslt \rightarrow Skip) \\ Write_i(hdl, BRange, Data) \triangleq\ & \{R = GetRepl(hdl)\} \\ & \rightarrow \parallel|_{t\epsilon[0,T-1]} (cl_ich_{R[t]}!WReq \\ & \rightarrow ch_{R[t]}cl_i?WAck) \rightarrow cl_ich_{R[0]}!Exec \\ & \rightarrow ch_{R[0]}cl_i?WRslt \rightarrow Client_i \end{aligned}$$

For the read operation, firstly, the $Client_i$ will get the id j of the closest chunkserver which stores the replica through function *GetN*. Secondly, it sends the $ChunkServer_j$ a *RReq* through channel cl_ich_j. Thirdly, the $ChunkServer_j$ will return the *Data* to the $Client_i$ through channel cl_ich_j. Finally, the $Client_i$ operates some inner process to handle data, while this part is not taken into our consideration.

Similar to the read operation, in the process of write operation, firstly, the $Client_i$ will send the *WReq* messages to every $ChunkServer_{R[t]}$ through channels $cl_ich_{R[t]}$ which contain *write* instruction and data *Data* to write. Secondly, $Client_i$ will receive all the *WAck* messages through channels $ch_{R[t]}cl_i$. Lastly the $Client_i$ will issue the *Exec* operation to the $ChunkServer_{R[0]}$ through channel $cl_ich_{R[0]}$ and wait to receive *WRslt* through channel $ch_{R[0]}cl_i$ after the *primary* server has finished all the actions.

IV. CONCLUSIONS

In this work, we have provided a CSP# model of Google File System which precisely describes the underlying read/write behaviors of GFS. In future work, we will encode the formal model into PAT [4] to check and verify the properties like deadlock-free and consistency model in our formal framework to enhance the reliability of the system.

Acknowledgements Geguang Pu and Yongxin Zhao are partially supported by NSFC Project No.61021004 and No. 61321064. Mengdi Wang is partially funded by Shanghai Knowledge Service Platform No. ZF1213. Bo Li is partially supported by SHEITC Project 130407.

REFERENCES

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] Jin Song Dong and Jun Sun. Towards expressive specification and efficient model checking. In *TASE*, page 9, 2009.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, 2003.

[4] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.