2015 IEEE 16th International Symposium on High Assurance Systems Engineering

Modeling and Verifying Google File System

Bo Li*, Mengdi Wang*, Yongxin Zhao†, Geguang Pu†‡, Huibiao Zhu† and Fu Song†
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
*email: {tmacback,wangmengdi}@gmail.com
†email: {yxzhao,ggpu,hbzhu,fsong}@sei.ecnu.edu.cn

***Abstract*—Google File System (GFS) is a distributed file system developed by Google for massive data-intensive applications. Its high aggregate performance of delivering massive data to many clients but the inexpensiveness of commodity hardware facilitate GFS to successfully meet the massive storage needs and be widely used in industries. In this paper, we first 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. On that basis, both *relaxed consistency* and *eventually consistency* guaranteed by GFS may be revealed in our framework. Furthermore, the suggested CSP# model is encoded in Process Analysis Toolkit (PAT), thus several properties such as starvation-free and deadlock-free could be automatically checked and verified in the framework of formal methods.**

I. Introduction

In recent years, the cloud computing has become an issue of vital importance in engineering field [1], [3], [19]. In order to achieve the high performance for massive data-intensive applications in clouding computing, companies like Google, Amazon, Yahoo have proposed various framework to meet their massive storage needs [8], [6], [5], [13]. Google File System (GFS) 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. It is a typical distributed file system which shares many of the same goals as previous distributed file systems such as performance and scalability [8]. Besides, GFS is regarded as an impactful file system for the sake of its fault tolerance and high aggregate performance of delivering massive data to many clients.

Google File System is built from large clusters of many inexpensive commodity hardware and of highly insensitive to failures since its framework leverages the multiple replicas, which is to distribute the replica chunks across the cluster. However, due to the complexity of the network, delay of communication and concurrent user access, the maintenance of data consistency grows sophisticated and unpredictable. As a result, the demands of formalization of GFS, especially the read/write behaviors and the consistency model guaranteed by GFS, are of pretty important and highly urgent. Thus the formal modeling and verification of GFS become the focus of attention in both academic circles and engineering fields.

Many research efforts have been addressed to analyze and improve cloud computing architectures. Reddy *et al.* [14] provides a holistic way to evaluate the correctness of Hadoop system. Ono *et al.* [12] presents two methods based on Coq and JML to formalize the MapReduce framework and verify the running program of the selected Hadoop application. Yang *et al.* [21] models MapReduce framework using CSP method. All the efforts mainly focus on the high-level correctness of the whole system behaviors of the distributed architecture, 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# [7] with respect to GFS architecture. Roughly speaking, Google File system consists of multiple GFS clusters deployed for different application purposes. As the main parts of GFS cluster, both *master* and *chunkservers* are described as CSP# processes, which capture their corresponding behaviors. The *clients* from distinct machines which could heavily access massive data from a cluster and the interactions with the cluster are also formally modeled. Consequently, our CSP# model could precisely describes the behaviors of GFS. Furthermore, the whole suggested CSP# model could be encoded in PAT, which is a self-contained framework to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains.

On that basis, we could analyze several properties of the GFS such as divergence-free, non-termination and starvation-free properties, which are described as LTL formulas [11], [2] and automatically checked and verified using PAT, which indicate the high reliability of the system. More importantly, the *relaxed consistency* [9] model guarantees by GFS may be revealed in our framework. Our experiment shows that not *strong consistency* [10] but *relaxed consistency* is guaranteed by GFS since simultaneous read operations of the same position may return different results. Our model also demonstrates that GFS ensures *eventually consistency* [20], i.e., all accesses eventually return the same last updated value which provide that no new updates occurs.

The remainders of the paper are organized as follows. Section 2 briefly introduces the preliminaries of CSP#, PAT and also presents a overview of GFS. The CSP# model of Google File System is formalized in Section 3. Section 4 encodes the achieved model in PAT and verifies several important properties automatically. At last, Section 5 concludes the paper and presents the future work.

‡Corresponding author

978-1-4799-8111-3/15 $31.00 © 2015 IEEE
DOI 10.1109/HASE.2015.38

II. Preliminaries

In this section, we firstly introduce the preliminaries of CSP# and PAT respectively, by covering their most important and distinct features used in this paper. And then, we give a brief description of GFS.

A. The CSP# language

CSP# is a modeling and specification language designed for specifying concurrent system, which offers great modeling flexibility and efficient system verification by integrating high level CSP-like [4] operators with low level sequential programs constructs such as assignments and while loops.

The syntax of a subset of CSP# processes is shown below with short descriptions. More details about CSP# could be found in [7], [22], [15].

$P, Q ::= stop \mid skip$	– primitives
$\mid\ a \rightarrow P$	– event prefixing
$\mid\ ch!exp \rightarrow P$	– channel output
$\mid\ ch?m \rightarrow P$	– channel input
$\mid\ e\{prog\} \rightarrow P$	– operation prefixing
$\mid\ [b]P$	– state guard
$\mid\ P \Box Q$	– external choice
$\mid\ ifa(b)\{P\}else\{Q\}$	– conditional choice
$\mid\ P;\ Q$	– sequential composition
$\mid\ P \| Q$	– parallel
$\mid\ P \mid\mid\mid Q$	– interleaving

Where P and Q are processes, a is an event, e is a non-communicating event, ch is a channel, exp is an arithmetic expression, m is a bounded variable, $prog$ is a sequential program updating global shared variables, b is a Boolean expression. For every program P, we denote $\alpha(P)$ to stand for its alphabet, which denotes the events that the process can be performed.

The process *stop* never actually engages in any events. The process *skip* terminates and does nothing. The next three processes first engage event a, input $ch!exp$ and output $ch?m$ respectively, and then behave like P. In process $e\{prog\} \rightarrow P$, *prog* is executed atomically with the occurrence of event e. Process $[b]P$ waits until condition b becomes *true* and then behaves as P. In process $P \Box Q$, it behaves like a choice P and Q. Conditional choice $ifa(b)\{P\}else\{Q\}$ behaves as P if b is evaluated to be true, and behaves as Q otherwise. For process $P;\ Q$, Q starts only when P has finished. Process $P \| Q$ executes in parallel and synchronizes with common events while $P \mid\mid\mid Q$ runs all processes independently.

B. Process Analysis Toolkit

PAT is designed as an extensible and modularized framework for modeling and verifying a variety of systems such as concurrent systems, real-time systems [18], [16], [17]. It offers featured model editor, animated simulator and various verifiers. The editor provides a user friendly editing environment to develop system models. The simulator enables users to interactively and visually simulate system behaviors using facilities such as random simulation, user-guided step-by-step simulation, complete state graph generation, trace playback, counterexample visualization, etc. The verifiers implement various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. Furthermore, to achieve good performance, advanced optimization techniques are implemented in PAT, e.g. partial order reduction, symmetry reduction, process counter abstraction, parallel model checking.

PAT supports a number of different assertions to query about system behaviors or properties, denoted by keyword #assert.

- Atomic{P}: the keyword atomic ensures the atomic execution of process *P*.
- Deadlock: given *P* as a process, the assertion #assert *P* deadlockfree checks whether *P* is deadlock-free or not. Nonterminating and divergence-free properties verification can also be applied correspondingly.
- Reachability: the assertion #assert *P* reach *cond* asks whether *P* can reach a state at which some given condition *cond* is satisfied.
- Linear Temporal Logic (LTL): PAT supports the full set of LTL syntax, such as $\Box$ (*always*) and $\Diamond$ (*eventually*). In general, the assertion $P \models F$ checks whether P satisfies the LTL formula F.

C. Google File System

Google File System is a scalable distributed file system for large distributed data-intensive applications designed and implemented by Google Inc. in 2003. Benefiting 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. As a result, GFS became an influential paradigm for distributed file system nowadays.

Google File System typically consists of three main parts: *masters*, *clients* and *chunkservers*. Here one *master* and the *chunkservers* in its domain are considered as a *cluster*, either one *cluster* or multiple *Cluster* constitute a system. 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.

Compared with traditional distributed file system, the GFS offers a new method to write files, which is appending new data rather than overwriting existing data. Additionally, the file system API has deep coherent with the application design which improves the flexibility of the entire system. GFS does not guarantee strong consistency, it adopts a type of relaxed consistency called eventually consistency, which means if no new updates are made to the file system, all accesses will return the last updated values eventually.

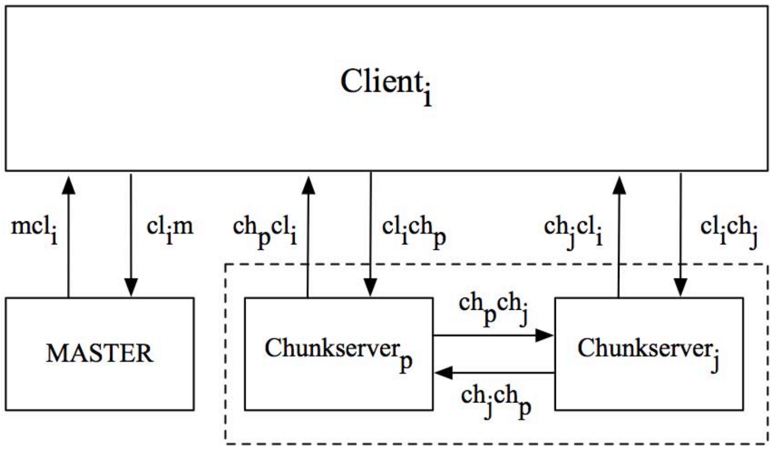


Fig. 1: Structure and channel of the system

III. Modeling GFS in CSP#

In this section, we present a formal CSP# model of the Google File System. Firstly, we list some notations and definitions which are used in the rest of the paper.

Notations

- I: the total number of clients.
- J: the total number of chunkservers.
- T: the replica number of each file block, presetted by the master.
- i: the identity of client.
- j: the identity of chunkserver.

Messages and Channels

We define several sorts of messages and the details are listed and explained as follows:

- $KeyReq \triangleq fn\#idx$
  KeyReq stands for the request of file location sent by channel cl_im from clients to the master. Here *fn* is the file name that applications need, and *idx* is the chunk index calculated and translated by the applications. To simplify the modeling, we treat read request and write request as the same one as they both exactly expect the handle and location of the data.
- $KeyMsg \triangleq \ \#_{t\epsilon[0,T-1]}(ChSvrId_t\#ChId_t\#ChOfs_t)$
  KeyMsg represents the reply to the request *KeyReq*. The notation $ChSvrId_t$ indicates the location of the t-th replica which stores the requested file block, $ChId_t$ means the chunk *id* on the corresponding chunkserver and $ChOfs_t$ stands for the offset within a certain chunk. These location information are stored and mapped with the requested *fname* and *index* by the master.
- $RReq \ \triangleq \ read\#ChHdl\#BRange$
  RReq stands for read request sent from the client to the chunkservers. Here *read* denotes the read operation. *ChHdl* and *BRange* stands for the chunkserver handle and byte range respectively which are the essential messages for the chunkkervers to locate the data.
- *Data* stands for the data sent by channel cl_ich_j from the client to chunkservers.
- $WReq \ \triangleq \ write\#Data$
  WReq stands for write request sent by channel cl_ich_j from the client to the chunkservers. Here *write* stands for the write operation. *Data* stands for the data send from the client to chunkservers.
- *WAck* stands for the ready writing state. It is sent from the chunkservers that store the replicas to the client after receiving the data indicating that the servers have received the write operator and cached the corresponding data successfully.
- *Exec* stands for an execution command. It is sent from the client to the primary chunkserver informing it to execute the write operation on each chunkservers that stores the replicas.
- *RRslt* stands for the result for a particular read request.
- *WRslt* stands for the completion message sent from the primary chunkserver to the client after all the executions.
- *Fwd* stands for the message sent from the primary chunkserver to the secondary replica chunkservers informing the start of storage process
- *Rpl* stands for the message sent from the secondary replica chunkservers to the primary chunkserver replying as a result of the storage process.

Next, we use the following channels to model the communications in the system:

- 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 $Client_i$.
- The channel ch_pch_j will represent the communication information sent from $ChunkServer_p$ to $ChunkServer_j$.

A. SYSTEM

As stated in the Google File System, a single master and multiple chunkservers compose a cluster. To brief the model, we use single cluster in the system. For the whole system, there are three crucial processes running in parallel, they are *CLIENTS*, *MASTER* and *CHUNKSERVERS*. We formalize the whole system as below.

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

During the process, some *CLIENT* issues the read/write request, the *MASTER* manages the file location meta data and the *CHUNKSERVERS* execute the actual read and write operation. There are some interactions between or within these processes. The *CLIENT* will query the file metadata from the *MASTER*. It can deliver read/write request to the *CHUNKSERVER* and get the replies. The *CHUNKSERVERS* themselves have some inner communications to implement the complete write operation.

B. CLIENTS

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

- $HasKey_i()$ checks whether the client contains the key message of the file, if not, it will request the master and get one.
- $StoreKey_i(KeyReq, KeyMsg)$ 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.

Besides, we propose a variable R stands for the *id* array of the chunkserver which stores the replica. $R[0]$ stands for the primary chunkserver and remains are the secondary replica chunkservers.

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:

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

The whole process of $Client_i$ can be described as three main parts, the key request operation, read operation and the write operation.

The key request operation is an auxiliary process. In the request 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 *KeyReq* message to the Master through the channel $cl_i m$. When the *KeyMsg* is received on channel mcl_i, it will be stored by function *StoreKey* for further use.

As defined in the GFS, the read/write sequence is illustrated in Figure 2 and Figure 3. The entire read/write flow can be interpreted as follows:

In Figure 2, the three nodes represent the *CLIENT*, *MASTER* and *CHUNKSERVERS*. When an application is proposed to read data from the GFS, the following sequence of action occurs:

1) The client requires the chunk data of certain file from the master.
2) The master replies the chunk data information, and the client will cache these information for further use.
3) The client sends a read request to the closest chunkserver.
4) The client reads the data from the chunkserver.

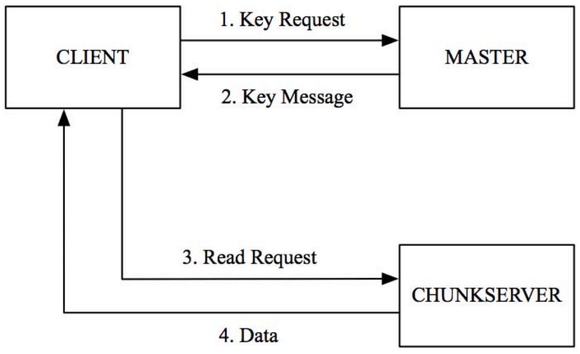


Fig. 2: Read Process

Figure 3 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 *CHUNKSERVERS* 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. The chunkservers will store the data in cache.
4) The chunkservers reply acknowledge to the client.
5) After receiving the acknowledge of all the replica, the client sends a write execution request to the *primary* chunkserver.

6) The *primary* chunkserver forwards the write request. All the replicas will execute the operation in the same serial order assigned by the *primary* chunkserver.
7) When the operation completed, all the *secondary* replica chunkservers reply to the *primary* which indicate the completion.
8) The *primay* chunkserver replies to the client to report the errors during the executions.

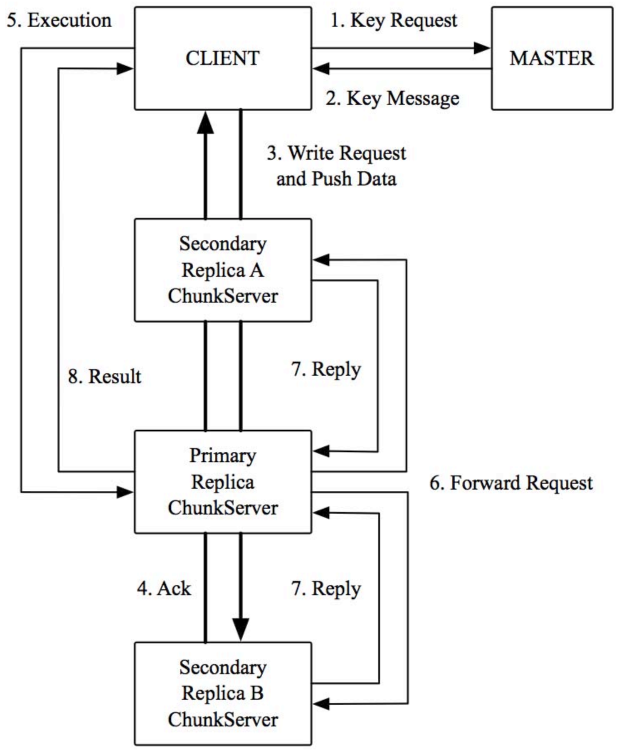

5. Execution
CLIENT
1. Key Request
MASTER
2. Key Message
3. Write Request and Push Data
Secondary Replica A ChunkServer
8. Result
7. Reply
Primary Replica ChunkServer
6. Forward Request
4. Ack
7. Reply
Secondary Replica B ChunkServer

Fig. 3: Write Process

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

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

For the read operation, firstly, the $Client_i$ will first 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.

C. MASTER

A cluster can be classified into two main parallel components: maintaining the file location and storing the file data cross multiple chunkservers. In our model, the master has only one duty, which is to maintain the directory system and file location. When the client sends a file key request, the master replies with the specific file information. Since we put the focus on the read/write process of the system, there exists some minor differences between our formal model and GFS. In the GFS architecture, the master offers some other services like heartbeat communication, logger and so on. These procedures are designed to maintain the status in the whole system. To simplify the model as well as clarify the process, we do not consider these actions essential for our purpose because read/write operations can still function correctly without them. The behavior of *MASTER* is formalized as follows:

$$Master \triangleq |||_{i\epsilon[0,I-1]}\ (cl_im?KeyReq \rightarrow mcl_i!KeyMsg)$$

The whole process of *MASTER* could be described as follows. The master receives the request keys *KeyReq* from the client through channel cl_im. After some inner processing, it replies to the client the *KeyMsg* on channel mcl_i.

D. CHUNKSERVERS

The process *CHUNKSERVERS* is composed by many chunkservers, each of which can be set a unique *id* marked as j and corresponds to a chunkserver progress. Under such condition, *CHUNKSERVERS* could be formalized as follows.

$$CHUNKSERVERS \triangleq ||_{j\epsilon[0,J-1]} ChunkServer_j$$

The *CHUNKSERVERS* receives the read/write request from the client, executes actual operation and replies to the client the results. A single chunkserver process $Chunksever_j$ can be formalized as below:

$$
\begin{aligned}
&ChunkServer_j \triangleq\\
&\qquad (cl_ich_j?RReq \rightarrow ch_jcl_i!Data \rightarrow ChunkServer_j)\\
&\qquad \Box(cl_ich_j?WReq \rightarrow ch_jcl_i!WAck\\
&\qquad \rightarrow (PriWrite_j \Box RplWrite_j) \rightarrow ChunkServer_j)\\
&PriWrite_j \triangleq\\
&\qquad cl_ich_j?Exec \rightarrow \{R = GetRepl(hdl)\}\\
&\qquad \rightarrow |||_{t\epsilon[1,T-1]}\ (ch_jch_{R[t]}!Fwd \rightarrow ch_{R[t]}ch_j?Rpl);\\
&\qquad \rightarrow ch_jcl_i!WRslt \rightarrow ChunkServer_j;
\end{aligned}
$$

$$\begin{aligned}&RplWrite_j \triangleq\\ &\qquad (ch_j ch_{R[0]}?Fwd \rightarrow ch_{R[0]}ch_j!Rpl)\\ &\qquad \rightarrow ChunkServer_j;\end{aligned}$$

Some functions used in the *ChunkServer*$_j$ process are defined as follows,

- *GetRepl*(hdl) returns all the chunkserver ids as an array according to the handle.

The whole process of *ChunkServer*$_j$ can be described as follows. Firstly, *ChunkServer*$_j$ receives a request from *Client*$_i$ on channel cl_ich_j. Secondly, if this request is a read operation, the chunkserver will return the *Data* on channel ch_jcl_i directly and stop the process. Otherwise some steps remain. Since data is replicated on multiple chunkservers, its consistency needs to be assured. To fulfill this requirement, several interaction between all the replica chunkservers are required. According to the GFS design, we separately define two Chunksever write process: *PriWrite*$_j$ for the primary process and *RplWrite*$_j$ for the secondary process.

PriWrite$_j$ is the storing process for a *primary* chunkserver, the detailed is illustrated as below:

Firstly, the process receives the *Exec* message on the channel cl_ich_j. Secondly, it obtains the array R of all the replica chunkservers by the means of function *GetRepl*. Thirdly, the process send a *Fwd* to each *secondary* replica chunkservers on channel $ch_jch_{R[t]}$ and receives *Rpl* on channel $ch_{R[t]ch_j}$ simultaneously. Finally, the *primary* chunkserver sends the *WRslt* to the client through channel ch_jcl_i.

RplWrite$_j$ is the process for *secondary* chunkserver, the precise steps are depicted as follows:

When the *secondary* replica chunkserver receives the *Fwd* instruction on channel $ch_jch_{R[0]}$. It will store the data through inner process and return a *Rpl* information on channel $ch_{R[0]ch_j}$.

IV. Checking Model with PAT

In this section we first encode the CSP# model formalized in previous section through PAT platform which supports composing, simulating and reasoning of concurrent systems, in which the read and write behaviors are simulated in terms of the PAT simulator. On that basis, we then show that our suggested CSP# model follows these properties such as *deadlock-free*, *divergence-free*, *nonterminating* and would not lead to *starvation*. At last, by observing and investigating the data consistency, we demonstrate that both relaxed consistency and eventually consistency are guaranteed by Google File System.

A. Model Implementation

According to the suggested formal CSP# model, we respectively encode four processes, which are *SYSTEM*, *CLIENTS*, *MASTER*, *CHUNKSERVERS*. In fact, the implementations of processes are straightforward. All the channels and messages also follow the aforementioned definitions.

However, the initialization is not trivial since the deploy of the initial storage state which is involved with many chunkservers is much complicated. For this reason, we introduce multi-dimension array $mem[J][chunkNum][chunkSize]$ to describe the storage model, the values of which represent the stored data. Thus all read operations may return the expected data and write operations could modify the designed data as well. By this method, we could discuss the data consistency by observing and comparing the mutation of data. Besides, all the read/write operations are randomly provided in the simulation process by *CLIENT*. At last, similar to the initialization of the storage model, we also set the default meta-data for the distributed system GFS which are stored into another array $filePos[FileNum][FileLength][T]$, which are initialized by random data.

Note that the whole system are parameterized by several configurable parameters, such as J, T which stand for ChunkServer number and replica number respectively. Without loss of generality, here we set I, J, T to 2, 3, 3 and *chunkNum* to 6 as default. Obviously, the actual values may vary with the practical case.

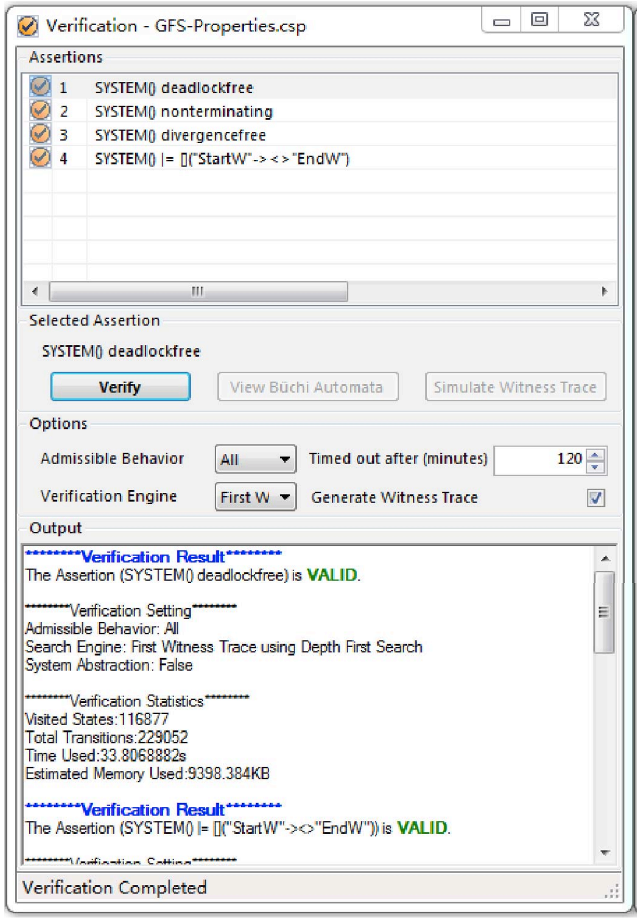
B. Properties Checking

In this subsection we investigate several properties of GFS based on the suggested model. The interesting properties *deadlock-free*, *nonterminating* and *divergence-free* are automatically checked in PAT. Meanwhile, the *starvation-free* property described as LTL formula is also verified and every client may eventually execute its operation as long as its operation is available. The left in Figure 4 shows the result of the verification in PAT.

- **Deadlock-free Property** the property guarantees that the model would not progress into a deadlock situation, which means there is no state such that the model has no further movement except for the terminated state.
- **Nonterminating Property** the property guarantees that the model will not run into a terminating situation.
- **Divergence-free Property** a divergence of a process is defined as any trace of the process after which the process behaves chaotically. This *divergence-free* property assures out model is well-defined without ambiguousness.
- **Starvation-free Property**: $SF \triangleq StartW \Rightarrow \Diamond EndW$
  the LTL formula above means the *CLIENT* will eventually complete write operation after it sent out requests using Global and Future operations, where formulas *StartW* and *EndW* describe the start point and the end point respectively.

C. Relaxed Consistency

In this subsection we focus on the storage model adopted by GFS by observing the data consistency. Generally, we investigate several situations including *replicas conflict* to judge what model consistency guaranteed by our achieved formal model. In detail, both the positions and the contents of read operations are recorded into additional arrays which could be used to check the identity of two replicas. Besides, we check whether the expected states are reachable in terms of the keyword *reaches*. Note that PAT's model checker usually performs a



```
1   SYSTEM() deadlockfree
2   SYSTEM() nonterminating
3   SYSTEM() divergencefree
4   SYSTEM() |= []("StartW"-><>"EndW")
```

SYSTEM() deadlockfree

Admissible Behavior: All
Timed out after (minutes): 120
Verification Engine: First W
Generate Witness Trace: ✓

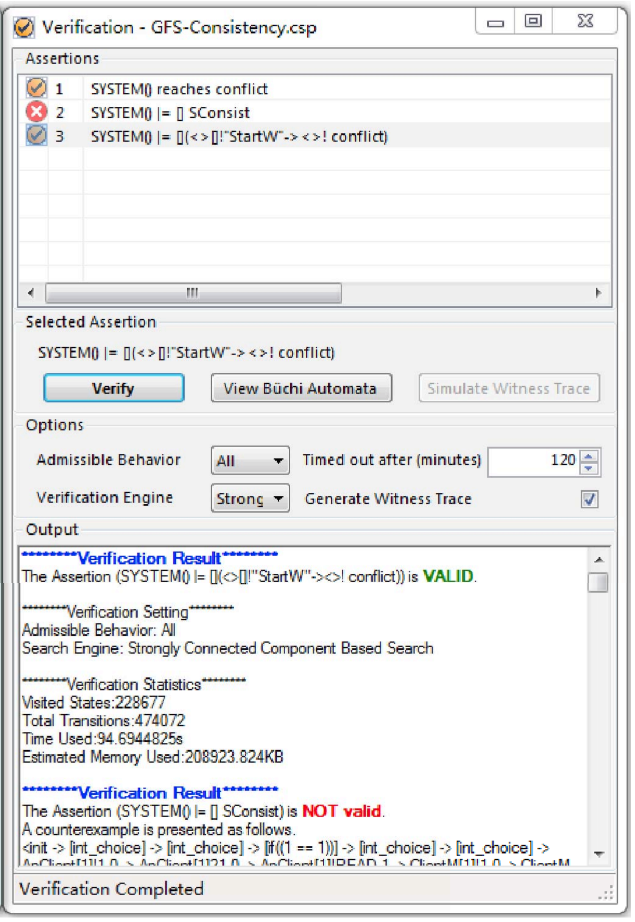
```
*********Verification Result*********
The Assertion (SYSTEM() deadlockfree) is VALID.

*********Verification Setting*********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

*********Verification Statistics*********
Visited States:116877
Total Transitions:229052
Time Used:33.8068882s
Estimated Memory Used:9398.384KB

*********Verification Result*********
The Assertion (SYSTEM() |= []("StartW"-><>"EndW")) is VALID.
```

Verification Completed



```
1   SYSTEM() reaches conflict
2   SYSTEM() |= [] SConsist
3   SYSTEM() |= [](<>[]!"StartW"-><>! conflict)
```

SYSTEM() |= [](<>[]!"StartW"-><>! conflict)

Admissible Behavior: All
Timed out after (minutes): 120
Verification Engine: Strong
Generate Witness Trace: ✓

```
*********Verification Result*********
The Assertion (SYSTEM() |= [](<>[]!"StartW"-><>! conflict)) is VALID.

*********Verification Setting*********
Admissible Behavior: All
Search Engine: Strongly Connected Component Based Search

*********Verification Statistics*********
Visited States:228677
Total Transitions:474072
Time Used:94.6944825s
Estimated Memory Used:208923.824KB

*********Verification Result*********
The Assertion (SYSTEM() |= [] SConsist) is NOT valid.
A counterexample is presented as follows.
<init -> [int_choice] -> [int_choice] -> [if((1 == 1))] -> [int_choice] -> [int_choice] ->
```

Verification Completed

Fig. 4: Verification Results on PAT

depth-first-search algorithm to repeatedly explore unvisited states until a state at which the condition is true is found or all states have been visited. By this method, we demonstrate that not strong consistency but relaxed consistency is guaranteed by GFS, in particularly, the eventually consistency is also ensured in our model.

- **Replicas Conflict**: different replicas of the same file block may be different due to the nondeterminism of multiple updates to different replicas. For simplification, we just randomly select two replicas of the same file block to illustrate this properties, which are described by the following assertions:
  $\#\text{define}\ \mathit{conflict}\ \mathit{content}1! = \mathit{content}2;$
  $\#\text{assert}\ \mathit{SYSTEM}\ \text{reaches}\ \mathit{conflict};$
  where the variables $\mathit{content}1$ and $\mathit{content}2$ define different replicas of the same file.
- **Strong Consistency**: guarantees that shared state behaviors like centralized system will not occur any exceptions caused by concurrent execution. In terms of distributed file system, this property indicates simultaneously read requests should obtain identical results. We check whether the results from two read operations are equal by observing the additional arrays, an emergence of inequality reveals the violation of strong consistency during the read process. The formulas are implemented by the assertions as below:
  $\#\text{define}\ \mathit{SConsist}\ (\mathit{SamePos} \rightarrow \mathit{SameCon});$
  $\#\text{assert}\ \mathit{SYSTEM}\ \models \Box \mathit{SConsist};$
  where the variable $\mathit{SamePos}$ and $\mathit{SameCon}$ indicates whether relative read operation regards to the same position and obtain the same contents.
- **Eventually Consistency**: All accesses eventually return the same last updated value provided that no new updates occurs, which is described as a LTL formula and this property can be naturally expressed as follows:
  $\#\text{assert}\ \mathit{SYSTEM} \models \Box(\Diamond\Box!\mathit{StartW} \Rightarrow \Diamond!\mathit{conflict});$
  where "Start" describes the start point of write operations.

The verification results of the three properties are shown in the right in Figure 4. Note that the strong consistency is not guaranteed by our model since two read operations at the same time may obtain different contents, i.e., the relaxed consistency model is adopted by GFS because eventually consistency is ensured by the experiment result.

V. Conclusion and Future work

In this work, we have presented a CSP# model of Google File System which precisely describes the underlying read/write behaviors of GFS. The suggested model is divided into three parts, i.e., *MASTER*, *CLIENTS* and *CHUNKSERVERS* which actually capture the actual behaviors respectively. And then we encode the formal model into PAT, which facilitates to check and verify the properties such as non-determinism, starvation-free and deadlock-free in our formal framework. Beside, we also demonstrate that both relaxed consistency and eventually consistency are guaranteed by GFS.

In future work, we will improve our formal model by introducing component failures and time behaviors. More interesting and novel properties will also be investigated. Ultimately, we can provide a comprehensive understanding of GFS and give a formal approach to verifying GFS to enhance the reliability of the system.

Acknowledgment

Geguang Pu and Yongxin Zhao are partially supported by NSFC Project No. 61021004 and No. 61402176. Mengdi Wang is partially funded by Shanghai Knowledge Service Platform No. ZF1213. Huibiao Zhu is partially supported by SHEITC Project 130407 and Bo Li is partially supported by National High Technology Research and Development Program of China (2012AA011205).

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] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[3] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[4] C.A.R.Hoare. *Communication Sequential Processes*. Prentice Hall International, 1985.

[5] Brian F Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James J Kistler, PPS Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, et al. Building a cloud for yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

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

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

[9] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[10] M.D. Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31(8):28–34, Aug 1998.

[11] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[12] Kosuke Ono, Yoichi Hirai, Yoshinori Tanabe, Natsuko Noda, and Masami Hagiya. Using coq in specification and program extraction of hadoop mapreduce applications. In *Software Engineering and Formal Methods*, pages 350–365. Springer, 2011.

[13] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Cloud Computing*, pages 115–131. Springer, 2010.

[14] G Satya Reddy, Yuzhang Feng, Yang Liu, Jin Song Dong, Sun Jun, and Rajaraman Kanagasabai. Towards formal modeling and verification of cloud architectures: A case study on hadoop. In *Services (SERVICES), 203 IEEE Ninth World Congress on*, pages 306–311. IEEE, 2013.

[15] Ling Shi, Yongxin Zhao, Yang Liu, Jun Sun, Jin Song Dong, and Shengchao Qin. A utp semantics for communicating processes with shared variables. In *ICFEM*, pages 215–230, 2013.

[16] Jun Sun, Yang Liu, and Jin Song Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 307–322. Springer, 2009.

[17] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Trans. Softw. Eng. Methodol.*, 22(1):3:1–3:29, March 2013.

[18] 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.

[19] B T OGRAPH and Y RICHARD MORGENS. Cloud computing. *Communications of the ACM*, 51(7), 2008.

[20] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[21] Fan Yang, Wen Su, Huibiao Zhu, and Qin Li. Formalizing mapreduce with csp. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 358–367. IEEE, 2010.

[22] Yongxin Zhao, Jin Song Dong, Yang Liu, and Jun Sun. Towards a combination of cafeobj and pat. In *Specification, Algebra, and Software*, pages 151–170, 2014.