Design and Optimization of Secure Byzantine Fault-Tolerant MapReduce on Large Cluster

Boopalan Mani [1], Sudha T [2]
Department of Computer Science and Engineering,
Muthayammal Engineering College, Anna University, Chennai, India

***Abstract*— Most Byzantine fault-tolerant state machine replication (BFT) algorithms have a primary replica that is in change of ordering the clients requests. Recently it was shown that this dependence allows a faulty primary to degrade the performance of the system to a small fraction of what the environment allows. In this paper we present Kerberos-based model with tokens for data blocks and processing nodes. We also especially interested in the performance of Byzantine fault-tolerant MapReduce framework that can run in two modes: (a) non-speculative (b) speculative. We designed the framework that they used around twice more resource instead of three times of alternative solutions. This novel mode of operation deals with those attacks at much lower cost.**

Keywords*— *MapReduce, Byzantine Fault Tolerance, Kerberos based model

I. INTRODUCTION

Many applications with high security and fault tolerance requirements can benefit from Byzantine fault-tolerant algorithms. These algorithms allow systems to continue to provide a correct service even when some of their components fail, either accidentally (e.g., by crashing) or due to malicious faults (arbitrarily). Several algorithms have already been presented in the literature: secure parallel algorithm [2], DRAM errors [9], sabotage tolerance mechanism [4], state machine approach [5].

Intrusion-tolerant systems are usually built using replications techniques. The idea is that there is a service that is replicated in a set of servers that execute requests from the clients. State machine replication (SMR) is one of these techniques, which allows making any deterministic distributed service fault or intrusion tolerant. In this form of replication all (non-faulty) servers have to execute all client's requests in the same order. Among these algorithms, Castro and Liskov's

PBFT [7] is often considered to be a baseline in terms of performance, probably because it was the first efficient algorithm in the area and many others derive from it.

Although it is crucial to tolerate crashes of tasks and data corruptions in disk, other faults that can affect the *correctness of results* of MapReduce are known to happen and will probably happen more often in the future [8]. A recent 2.5-year long study of DRAM errors in a large number of servers in Google datacenters,concluded that these errors are more prevalent than previously believed, with more than 8% DIMMs affected by errors yearly, even if protected by error correcting codes (ECC) [9]. A Microsoft study of 1 million consumer PCs showed that CPU and core chipset faults are also frequent. MapReduce is designed to work on large clusters and process large data, so errors will tend to occur.

Sarmenta proposed a similar approach in the context of volunteer computing to tolerate malicious volunteers that returned false results of tasks they were supposed to execute [4].However, he considered only bag-of-tasks applications, which are simpler than MapReduce jobs. A similar but more generic solution consists in using the *state machine replication* approach [5]. This approach is not directly applicable to the replication of MapReduce tasks, only to replicate the jobs, which is expensive. In the yearly 2010 Yahoo! Developers (*O'Malley et al.* 2010) attempted to improve the state of security [11]. Many security loopholes like poor default SASL (Simple Authentication Security Layer) quality of protection, incomplete authentication and lack of data security that flows between the nodes were identified (*Wei et al.*, 2009).

This paper introduces a replication based verification scheme which is decentralized scheme to run MapReduce securely. They also proposed a way to detect misbehavior of malicious users. Major concerns like SASL, and securing the channel are achieved through Kerberos based Authentication and through Remote Procedure Call via Secure Shell mechanism. In this study, the three technical challenges identified were resolved by transferring the tokens securely between mappers and reducers. Also presents a Byzantine fault-tolerant (BFT) MapReduce runtime system that tolerates arbitrary faults by executing each task more than once and comparing the outputs. The challenge was to do this efficiently, without the need of running 3f +1replicas to tolerate at most f faulty, which would be the case with state machine replication. The system uses several techniques to reduce the overhead. With f =1, it manages to run only two copies of each task when there are no faults plus one replica of a task per faulty replica, instead of a

replica of the whole job as in the result comparison scheme. In this paper we are especially interested in the performance of the BFT MapReduce system. Therefore, we designed it to work in two modes: non-speculative and speculative. What differentiates them is the moment when reduce tasks start to run. In non-speculative mode, f +1 replicas of all map tasks have to complete successfully for reduce tasks to be launched. In speculative execution, reduce tasks start after one replica of all map tasks finish. While the reduce tasks are running, it is necessary to validate the remaining map replicas' outputs. If at some point it is detected that the input used in the reduce tasks was not correct, the tasks will be restarted with the correct input.

II. MAPREDUCE AND HADOOP

MapReduce is a mechanism used by Hadoop to distribute work across a cluster. There is a single master managing a number of slaves. The input file, which resides on a Hadoop distributed file-system (HDFS) throughout the cluster, is split into even-sized chunks replicated for fault-tolerance. HDFS is used to store the input the input splits and the final output of the job, but not the intermediate results (map outputs, reduce outputs) which are saved in local disc. Hadoop divides each MapReduce job into a set of tasks. Each chunk of input is first processed by a map task, which outputs a list of key-value pairs generated by a user-defined map function. Map outputs are split into buckets based on key. When all maps have finished, reduce tasks apply a reduce function to the list of map outputs with each key. Hadoop runs several maps and reduces concurrently on each slave – two of each by default to overlap computation and I/O. Each slave tells the master when it has empty task slots. The scheduler then assigns it tasks to accept new tasks of MapReduce process.

HDFS is implemented by using a single name node, the master node that manages the file name space operations (open, close, rename) and controls access to nodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on, and serve block operations(create, read, write, remove, replicate). Data nodes communicate to move blocks around, for load balancing and to keep the replication level on failures. MapReduce jobs are submitted to and managed by a centralized service called job tracker. This service creates one map task per input split and a predefined number of reduce tasks. Each node available to run Hadoop tasks run a software service called task tracker that launches the tasks. Using different nodes, the job tracker runs speculative tasks for those lagging behind and restarts the failed ones. The goal of speculative execution is to minimize a job's response time.

Response time is most important for short jobs where a user wants an answer quickly, such as queries on log data for debugging, monitoring and business intelligence. Short jobs are a major use case for MapReduce. Response time is also clearly important in a pay-by-the-hour environment like EC2. Speculative execution is less useful in long jobs, because only the last wave of tasks is affected, and it may be inappropriate for batch jobs if throughput is the only metric of interest, because speculative tasks imply wasted work. However, even in pure throughput systems, speculation may be beneficial to prevent the prolonged life of many concurrent jobs all suffering from straggler tasks. Such nearly complete jobs occupy resources on the master and disk space for map outputs on the slaves until they terminate. Nonetheless, in our work, we focus on improving response time for short jobs.

A. Tokens for security:

The new Hadoop version 0.21 provides improved security performance by creation of tokens at different layers of MapReduce process. Token is a Secret Key created by the Name node and Job tracker and is shared between the clients, Task tracker and Data nodes. Name nodes create Delegation Token and BlockAccess Token and Job tracker creates Job tokens to perform MapReduce job. Tokens like BlockAccess and Job token are designed with timestamp inorder to verify the tokens authenticity and its validity.

B. Delegation tokens:

Delegation token generated by Name node is created to enhance the security of Hadoop by allowing user authentication and pass credentials to all tasks of a job. It is generated to prevent flooding of
authentication requests at the start of a job and is shared as a secret key between the client and Name node/ Job tracker. For subsequent calls, Delegation token alone can be verified by Name node for each user instead of using Kerberos tickets.

C. BlockAccess token:

Inorder to securely access the contents of HDFS BlockAccess token is created. Actually Name node creates such tokens and is accessed by Data node. To access a file, clients communicate with Name node to find out which Data node the user intends to access inorder to fetch the file. Data nodes need to know from the Name node whether the client is authorized the client, information about the client block_id is passed using BlockAccess token from Name node to the owner of the file. Using the token, the owner/client can access the data block.

D. Job token:

The token is created to securely run MapReduce jobs. It is generated by Job tracker and distributed to all MapReduce tasks (Task tracker nodes) in order to run jobs by providing a check as whoever comes with the token is authentic to run. When task communicates with Task tracker for results or computational purposes, Job token is used. Job tracker automatically renews tokens while jib is running and cancels tokens when jib is finished and hence it is not persistent.

E. Flow of tokens:

All communication between the nodes of distributed the nodes of distributed environment takes place via SecureShell (SSH). Despite transferring tokens via SecureShell, Hadoop services

are authentication protocol. Hadoop users who access services are authenticated through a Kerberos authentication protocol. Hadoop clients access its service via Hadoops Remote Procedure Call (RPC) library. Each users login name sent across the connection setup is authenticate through Kerberos based authentication to Name node in existing Hadoop 0.21 version. Kerberos has a key distribution center (KDC) which maintains Authentication Server and Ticket Granting Server. Ticket Granting Server is used to request a service ticket from ticket granting server. Client uses the service ticket to accept the delegation token from the Name node through RPC call. The three step Kerberos authentication for each call overloads the KDC on cluster for which a Delegation token is introduced. Figure 1 describes how the Kerberos service ticket and tokens flow during a MapReduce process. After inital user authentication through Kerberos the whole process involves running MapReduce jobs on a Hadoop cluster

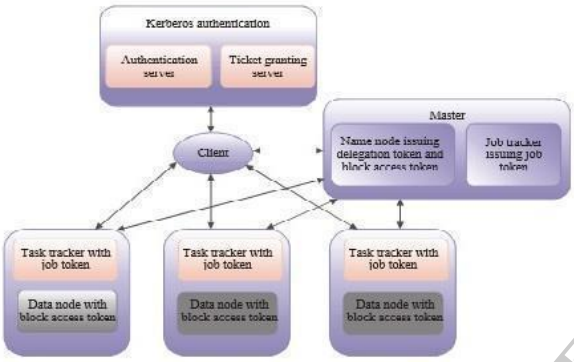

Fig 1: Map Reduce Architecture describing flow of tokens

and accessing HDFS blocks. To access HDFS using BlockAccess token, the access information of a BlockAccess token issued by Name node is used on Data node to verify its authenticity, The token enables its owner to access certain Data blocks. Here Data nodes do not enforce access control on data blocks. This allows an unauthorized client to read a data block when it knows the block_id. So, insecure channels in between the nodes allow malicious users/attacks to fetch other blocks. MapReduce jobs running on a cluster accepts the job tokens created, while running the task. The authenticity is checked with the users Delegation and Job token.

Handling attacks associated with token: Client Server interaction in a distributed environment like HDFS or MapReduce is prone to numerous attacks. So such an environment needs to be authenticated. Tokens are created and assumed to follow a secure channel. Besides communication security threats such as Denial of Service attacks, eavesdropping attacks, replay attacks, MapReduce faces issues while maintaining the integrity, confidentiality of data (Malley, 2010). Authenticated users obtaining a delegation tokens shares the token between the user and the Name node. The Delegation token need to be protected when passed over insecure channels.

An Attacker can affect the integrity of the MapReduce process in two ways.

1. When a client asks Name node for block_ids (location) of a file on HDFS, Name node checks that the client is authorized to access the file and send back block_ids along with a BlockAccess token for each block.
2. While running jobs on HDFS cluster an intruder who is capable of acquiring Job tokens can modify the results of a Map or Reduce task.

III. SYSTEM MODEL

Several risks related to HDFS data integrity, privacy and confidentiality of Hadoop users needs to be reduced by performing cryptographic encryption techniques on tokens when traversed from one node to another in Hadoop environment.

A. Using Asymmetric encryption mechanism:

The proposed technique is to replace the symmetric HMAC-SHA 1 system supported in the version Hadoop 0.21 by a public key system. In the case of BlockAccess token, instead of storing the same symmetric key over several Data nodes, a public-private key pair is generated between the Name node and every Data node. Similarly, job token are encrypted using a public key shared by the Job tracker and the task. When a job is submitted by a Hadoop client, the Job tracker first generates key pairs with each of the Task tracker. When a job is split to the various Task trackers, the job token ensures as secure communication. In this case, the token is encrypted by the key which is available only at the encrypting end i.e., Job tracker and sent to the corresponding Task tracker node. At this end, the task tracker decrypts the token using its private key and checks the authenticity. Inorder to ensure that token is arrived from the appropriate Job tracker; its digital signature is appended with the tokens. Similarly, during data access from HDFS, public keys for encryption are stored in the Name node and each of the Data nodes has the corresponding private keys. The blocks at the respective Data nodes can be retrieved by verifying the corresponding encrypted tokens.

B. Using symmetric encryption mechanism:

The symmetric hashing is replaced by a symmetric encryption mechanism. In the case of BlockAccess token, instead of storing the same symmetric key over several Data nodes a private key is generated between the Name node and every Data node. Similarly job tokens are encrypted using a private key and is shared by the Job tracker and each of the tasks. When a job is submitted by the user, the Job tracker generates private keys and shares it with each of the Task trackers. When a job is split to the various Task trackers, the job token ensures the secure communication. The Job tracker encrypts the Job token by the private key. It is decrypted using the same private key and checked accordingly at the Task tracker node. Inorder to access data from HDFS, BlockAccess tokens are encrypted using private keys that are maintained by Name node and each of the Data nodes which receives the encrypted form of block_id from the user, checks the token by decrypting it with

the same private key sent by the Name node.

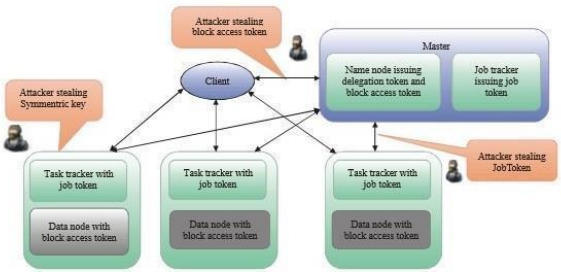

Fig 2: Depicting an attacker stealing Block access Token, Symmetric key and Job token when traversed between Name node and Data

C. *Refinement in the existing HMAC-SHA 1 mechanism of Hadoop's MapReduce:*

The model shares the same secret key across nodes to compute the hash value of tokens like BlockAccess and Job token. The algorithm designed generates a unique secret key for each and every node instead of using the same key. Such concept introduced, reduces the risk of stealing or destroying the contents of all blocks and is exposed only to the block of HDFS. An even malicious user who steals a secret key is unable to access the contents from other blocks and is exposed only to the block for which the secret key is assigned. The implementation of such mechanism involves key_init method to set the default parameters for the key. But generate keys method in this case uses the actual crypto method to generates as many secret keys as the mapper nodes. After that the processes are connected by secure channels, so no messages are lost, duplicated or corrupted. In practice this is provided by TCP connections. We assume the existence of a hash function to produce message digests. This function is collision-resistant, i.e., it is infeasible to find two inputs that produce the same output (e.g., SHA- 1 or SHA-3, recently chosen by the NIST). Our algorithm is configured with a parameter f. In distributed fault-tolerant algorithms f is usually the maximum number of faulty replicas but in our case the meaning of f is different: f is the maximum number of faulty replicas that can return the same output given the same input. Consider a function F, map or reduce, and that the algorithm executes several replicas of the function with the same input I, so all correct replicas return the same output O. Consider also the worst case in which there are f faulty replicas that execute the function F and $F_1(I)$ $= F_2(I) = ... = F_f(I) = \neq O$. The rationale is that f is the maximum number of replicas that can be faulty and still allow the system to find out that the correct result is .

If the system selects the correct output by picking the output returned by f +1task replicas, it will never select O because it is returned by at most f replicas. Similarly to the usual parameter f, our f has a probabilistic meaning (hard to quantify precisely): it means that the probability of more than f faulty replicas of the same task returning the same output is negligible.

IV. SECURE BFT MAPREDUCE ALGORITHM

The algorithm designed to generates a unique secret key for each and every node instead of using the same key. By using key_init method for setting up the default parameters and generate_keys function for generating public and private keys. The private keys generated were based on different values of commonly used public key values such as 17,257 and 6553. The public and private key pair created in the Name node of cluster show in Table 1. The Job token are encrypted using public key shared by Job tracker would start 2f+1 replica of each map task in different nodes and task trackers. Job tracker start also 2f+1 replica of each reduce task. Each reduce task fetches the output from all replicas, picks the most voted results, processes them and stores the output in HDFS. At this end, the task tracker decrypts the token using its private key and check for authenticity.

Nodes of Cluster scheduled to run Map/Reduce tasks	Private Key		Public key
Node 31	4	59E+029	65537
Node 8	6	14E+017	17
Node 23	2	71041713	65537
Node 32	2	50889438643593	17
Node 17	5	70E+017	65537
Node 26	1	54E+020	65537

Table 1: Asymmetric Keys created in Name node and Distributed to nodes of Cluster

The first simplistic solution is very expensive because it replicates everything 2f +1times: task execution, map task inputs reading, communication of map task outputs, and storage of reduce task outputs. Starting from this solution, we propose a set of techniques to avoid these costs:

A. *Deferred execution:*

Crash faults, which happen more often, are detected using Hadoop standard heartbeats, while arbitrary faults are dealt using replication and voting. Given the expected low probability of arbitrary

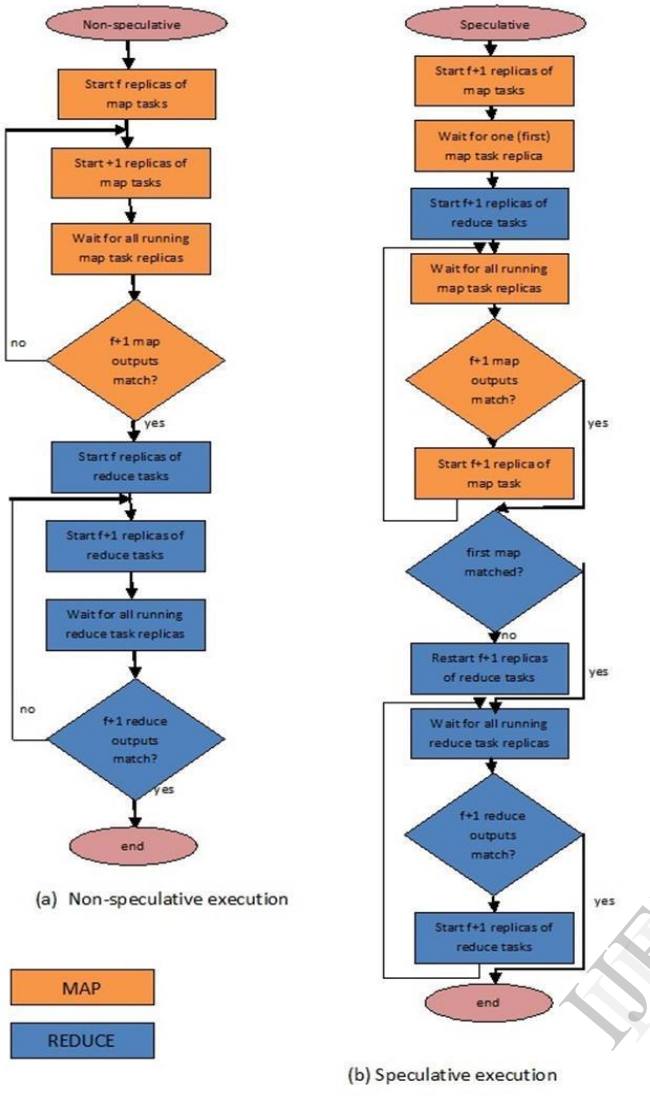

Fig 3: Flowchart of (a) non-speculative and (b) speculative execution

faults, there is no point in always executing 2f+1 replicas to obtain the same result almost every time. Therefore, our job trackers start only f +1 replicas of map and reduce tasks. After map tasks finish, the reduce tasks check if all f +1replicas of every map tasks produced the same output. If some outputs do not match, more replicas are started until there are f +1 matching replies. At the end of execution, the reduce output is also checked to see if it is necessary to launch more reduce replicas. This algorithm is represented as a flowchart in Figure 3(a).

B. Digest outputs:

f+1 map outputs and f+1 reduce outputs must be matched to be considered correct. These outputs tend to be large, so it is useful to fetch only one output from some task replica and compare its digest with those of the remaining replicas. With this solution, we avoid transferring the same data several times causing additional network traffic, and we just transfer data from one replica and the digests from the rest.

C. Tight storage replication:

We write the output of all reduce tasks to HDFS with a replication factor of 1, instead of 3 (the default value). We are already replicating the tasks and their outputs will be written on different locations, so we do not need to replicate these outputs even more. A job starts reading replicated data from HDFS, but from this point forward, the data that is saved in the HDFS by each (replicated) task is no longer replicated.

D. Speculative execution:

Waiting for f +1matching map results before starting a reduce task can worsen the time for the job completion. A way to deal with the problem is for the job tracker to start executing the reduce tasks immediately after receiving the first copy of every map output (see Figure 3(b)). Whenever f +1replicas of a map task finish, if the results do not match, another replica is executed. If f +1replicas of a map finish with matching results but these results do not match the result of the first copy of the task, then the reduces are stopped and launched again with the correct inputs. For a job to complete, f +1matching map and reduce results must be found for all tasks, and the reduces must have been executed with matching map outputs.

V. CONCLUSION AND DISCUSSION

The paper presents RSA encryption algorithm, a novel Byzantine fault-tolerant algorithm that tolerates performance attacks by changing the primary replica whenever a batch of pending requests is accepted for execution. This way of tolerating these attacks is much simpler and more efficient than other solutions in the literature. This novel mode of operation also does some load balancing among the servers, allowing an improvement of Practical BFT's throughput in the fault-free case.

The model is designed with providing three levels of security first at user level, second at the MapReduce process level and third at the HDFS level. To reduce the complexity involved in maintaining the tokens, a simple hashing technique methodology is explained that offers a reliable way to run the MapReduce process. Such mechanism prevent Distributed Denial of Service attack, Replay attacks on nodes involved in MapReduce process and stealing of keys from the nodes of cluster. As well as our algorithm masks these faults by executing each tasks more than once, comparing the outputs of these execution, and disregarding non-matching outputs. This simple but powerful idea allows our secure BFT MapReduce to tolerate any number of faulty task executions at the cost of one re-execution per faulty task. The framework is designed that they used around twice more resources instead of three times of alternative solutions. And also supports considerable cost for critical applications.

As future work, we plan to study the possibility of running MapReduce in several datacenters in order to tolerate faults that severely impact a subset of them. Furthermore, we aim to

study to what extend similar schemes can be applied to generalizations of MapReduce like Dryad or Pig Latin.

REFERENCES

[1] Pedro Costa, Marcelo Pasin, Alysson Bessani, Miguel Correia, " n the Performance of Byzantine Fault-Tolerant MapReduce," in *IEEE Transactions on Dependable and Secure Computing*, vol.22, no.4.,Mar.2013

[2] M.K.Suresh Gautham and Sowmy Narayanan, "Improving Security of Parallel Algorithm Using Key Encryption Techniques," in *Proceedings of the 12th asian network for scientific information Publications*, vol.1, no.12, 2013.

[3] T. White, Hadoop: The Definitive Guide. 'Reilly, 2009.

[4] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, vol. 18, pp. 561–572, Mar. 2002.

[5] F. B. Schneider, "Implementing fault-tolerant service using the state machine aproach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[6] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, Nov. 2002.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in Proceedings of the 19th ACM
Symposium on Operating Systems Principles, 2003, pp. 29–43.

[8] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, 2007.

[9] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, 2009, pp. 193–204.

[10] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programing Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[11] 'Malley, ., K.Zhang, S. Radia, R.Marti and C.Harell, 2010. *Hadoop security design*,
[12] http://carfield.com.hk/document/distributed/hadoop-security-design.pdf.

[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Syste*