How to parallelize Rsync

Changjian Zhang, Deyu Qi*
School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
csa@scut.edu.cn

Abstract—With the advent of the data age, data backup has become the focus of academic research, especially the incremental backup technology based on Rsync. However, the current Rsync does not support parallel work because it needs to keep a long connection and needs multiple segments of communication to complete a single synchronization process. Therefore, this paper uses the Kafka framework to decouple the synchronization process of the Rsync algorithm, optimizes the synchronization process based on shadow data, and eventually endows the Rsync algorithm with the ability of parallel work. Experimental results show that the proposed method can effectively improve the concurrency of Rsync. This method will be beneficial to the performance optimization of the server in the backup system.

1. INTRODUCTION
With the development of 5G technology, the amount of data is growing explosively. Individuals and companies have to back up the data for the sake of data security. Due to their limited backup capacity, individuals and companies choose to back up their data to a third-party server[1,2]. With the continuous increase of stored data, the server in the backup system is facing the huge challenges brought by data synchronization. More and more data synchronization requests will be sent to the server, which will lead to the rapid growth of server disk I/O and CPU utilization. At the same time, the bandwidth consumption caused by synchronization can not be ignored. To avoid a large amount of bandwidth occupation due to synchronization, the server often uses incremental synchronization to complete the data synchronization request.

In the field of file incremental synchronization, the application of the Rsync algorithm is very common. Many incremental synchronization algorithms are optimized based on the Rsync algorithm[3–5]. To improve the processing capacity of a single synchronization request, Björn Nikolaj et al. proposed an incremental synchronization algorithm based on the variable-length chunking, which reduced the energy consumption of a single synchronization request[6]. Besides, Chao Yang et al. optimized the workflow of an incremental synchronization algorithm to reduce the server CPU load during download[7].

Meanwhile, the academic community has done more research on the data chunking algorithm in the Rsyn. Since the sliding chunking algorithm used in Rsync is based on the improvement of fixed-length chunking algorithm and solves the byte-shifting problem in by rolling check, some scholars begin to consider using content-defined chunking instead of sliding chunking to solve the byte-shifting problem. Among them, the earliest content-defined chunking algorithm used is Rabin fingerprint[8], which determines whether to set the cut point at the end of the window by the Rabin fingerprint in the fixed window. To solve the problem that it is hard to find a cut point with Rabin, R Raju et al. proposed the TTTD[9] algorithm, which adopts two preset values of the Rabin fingerprint, one is easy to achieve and...
the other is difficult to achieve. At the beginning of the algorithm, the difficult preset value is adopted. If the chunk length is too long, it is replaced by the easier one. Besides, Rabin fingerprints have the problem of length diversity. To solve this problem, n Bjhner et al. proposed the LMC algorithm\textsuperscript{6}, which no longer calculates Rabin fingerprint, but determines whether to set the cut point in the middle of the window by judging whether the maximum value in the window is in the middle, thus saving the time of calculating Rabin fingerprint. At the same time, because the window size can be set, the size of the chunk can be bound, and the distribution of chunk length is relatively stable. To speed up the verification of window data, Y. Zhang et al. proposed the AE algorithm\textsuperscript{10}, R. n. Widodo et al. proposed the RAM algorithm\textsuperscript{11}. By changing the verification method of window data, the chunking speed was accelerated. Also, the idea of parallel computing is applied to the data chunking algorithm\textsuperscript{12}\textsuperscript{13}\textsuperscript{14}.

However, there is little optimization of the backup server in academia. When synchronization requests increase, the concurrency provided by Rsync can no longer meet the needs of the backup server. Therefore, this paper improves Rsync in two aspects. Firstly, a special data structure, Shadow Data, is proposed to optimize the synchronization process. Then, a message queue based on the Kafka framework is proposed to decouple the synchronization process. Through these two improvements, Rsync can be processed synchronization requests in parallel.

2. MATERIALS AND METHODS
This section gives the reason why Rsync can't be processed in parallel and introduces the design idea of our method in detail.

2.1. Why is Rsync Difficult to Be Processed in Parallel
The Rsync service currently used in the industry is based on the algorithm of the same name proposed by Tridgell\textsuperscript{15}. The synchronization process is as follows: firstly, the server chunks the backup file and calculates the strong and weak check values of the chunks (MD5 and Adler-32), and sends the values to the client. Then, the client uses the method of rolling check to determine whether the chunks in the source file exist in the server, gets the different chunk information between the source file and the backup file, and sends the information back to the server. Finally, the server combines the backup file and differential chunk information to form a new backup file and replace the old one.

There are several reasons why Rsync cannot be processed in parallel.

First, the synchronization process needs to keep a long connection. In synchronization, the server and the client need to chunk the file and calculate the check values, which requires a certain period. During this period, the server and the client must be connected, because once disconnected, the whole synchronization process will be interrupted.

Secondly, the synchronization process needs multi-segment communication. Rsync algorithm needs at least four communication processes, including the client sending synchronization request, the server sending backup file summary information, the client sending differential block information, and the server sending synchronization results. Therefore, the parallel processing of Rsync must save and maintain the socket information and the connection information during the parallel process, which requires very good configuration of the server nodes when the concurrency is large.

Finally, the Rsync service in Linux responds to multiple synchronization requests by threads forking, and the parallel processing requires a very strict plan of system resources, which is difficult to achieve and easy to make mistakes.

2.2. Shadow Data
In the process of Rsync, the server needs to send the chunk information of the backup file to the client. However, the backup file has existed in the client once, since it is from the client and the server will not edit it.

Based on the above analysis, this paper proposes a special data structure, shadow data, which is used to store the chunk check information of the source file in the process of one synchronization, that is, the
chunk check information of the backup file in the next synchronization. The information is stored in the client, so it will cause extra disk cost to the client, which will be discussed in the experiments.

![Diagram of shadow data structure](image1.png)

**FIGURE I.** The specific structure of shadow data.

The specific structure of shadow data is shown in Figure I.

As can be seen in the figure, shadow data stores the chunk check information of the source file. In the next synchronization, this information can be used to replace the backup file chunk check information sent by the server. Therefore, one communication process can be saved, and at the same time, the amount of calculation generated by the server's one-time chunking and calculation of chunk check information can be saved.

![Diagram of communication flow with and without shadow data](image2.png)

**FIGURE II.** The communication flow of file incremental synchronization with and without shadow data.

In Figure II, this paper gives the communication flow of file incremental synchronization with and without shadow data.

As can be seen from the figure, based on the shadow data, the client can directly calculate the different chunk information between the source file and the backup file locally, without waiting for the message from the server. Therefore, the first three communication processes in Rsync can be reduced to one: the client sends the synchronization request and simultaneously sends the different chunk information between the source file and the backup file. In the fourth communication process, the server informs the client of the synchronization results. This communication step can be replaced by maintaining a unified virtual attempt of backup files in the server and providing a service to query the synchronization results. Therefore, the improved Rsync only needs one communication to complete a single incremental file synchronization.
2.3. Message Middleware Based on Kafka Framework
Since Rsync can complete the incremental synchronization of files through one communication, this paper considers decoupling the synchronization request so that the server is only responsible for receiving the request, and the processing of the synchronization request is handed over to the distributed cluster such as Hadoop, Spark, Flink, etc.

To achieve the above purpose, this paper proposes a distributed message middleware based on the Kafka framework. In Figure III, this paper shows the incremental file synchronization process after adding the Kafka framework..

As can be seen in Figure III, after receiving the synchronization request, the message receiving server simply transforms the synchronization request and sends it to the Kafka cluster. The queue consumer server reads the synchronization request from the Kafka cluster, converts the synchronization request into the task of the distributed computing framework, and then submits it to the distributed computing cluster. The distributed computing cluster processes the synchronization request and updates the synchronization result to the state management server, which exposes the service, and the client can access the service to query the synchronization results.

With the help of the Kafka framework, this paper decouples the receiving and processing of synchronous requests. After receiving the synchronization request message, the message receiving server can directly disconnect, which avoids the problem of maintaining a large number of long connections. At the same time, as a task in the distributed computing cluster, the synchronous request can be well parallel handled, which greatly increases the processing performance of the backup server.

3. EXPERIMENT AND ANALYSIS

3.1. Experimental Data
The experimental data in this paper are from the public data of New York taxi driving data in 2013, which can be downloaded from the website https://chriswhong.com/open-data/foil_nyc_taxi/. Since the
purpose of the experiment is to compare the proposed method with Rsync with a large number of synchronization requests, we split the dataset and get 10 datasets with a size of 500M. The experimental data used in the experiment in this section are shown in Table I.

3.2. Experimental Setup

| TABLE I. The experimental data. |
|-------------------------------|
| Dataset | Size(MB) | Number(piece) |
| Dataset1 | Above 500 | 10 |

The experimental platform configuration of this paper is given in this section. The test environment of this paper is based on three virtual machines. The hardware allocation and network configuration of the three virtual machines are shown in Table II. Deploy HDFS, Zookeeper, Kafka, and Spark cluster in three virtual machines. HDFS is used to store backup files of backup server, Kafka cluster is used as message middleware to store synchronization request message, Zookeeper cluster is responsible for controller election of Kafka cluster and high availability of Spark cluster, and Spark cluster is responsible for parallelization processing of synchronization requests. The role assignment in the virtual machine is shown in Table III.

| TABLE II. The hardware allocation and network configuration of the three virtual machines. |
|-----------------------------------------------|
| Name | OS | Memory (GB) | Disk (GB) | IP (192.168.0) |
| Centos1 | Centos7.2 | 4 | 200 | 101 |
| Centos2 | Centos7.2 | 2 | 100 | 102 |
| Centos3 | Centos7.2 | 2 | 100 | 103 |

| TABLE III. The role assignment in the virtual machine. |
|-----------------------------------------------|
| Name | HDFS | Kafka | Spark | Zookeeper |
| Centos1 | NN broker | Worker | zk |
| Centos2 | DN broker | M,W | zk |
| Centos3 | DN broker | M,W | zk |

3.3. Results and Analysis

This paper compares the proposed method with Rsync from three aspects.

3.3.1. Client Disk Occupancy

Because the shadow data needs to be stored in the client, there will be additional storage costs on the client-side. Therefore, this experiment compares the disk cost of the proposed algorithm and Rsync in the client-side.

At the same time, we make random changes to the experimental data to simulate the data changes and then use the Rsync algorithm and our method to synchronize these files. Table IV shows the client disk occupancy collected in the experiment.

As can be seen from Table IV, Rsync does not generate additional disk consumption. Our method generates about 4% additional disk space of the source file. This data is the shadow data. Therefore, the method in this paper has certain disk requirements for the client, which is not suitable for specific devices. When using our method, a reasonable decision needs to be made according to the hardware conditions of the client.
3.3.2. Concurrency

We deploy the Rsync service and the message receiving server in our method on Centos1. The stress test of the Rsync service is to establish long connections continuously with the help of the PSSH tool, while the stress test of the message receiving server is to send synchronization requests continuously with short connections by JMeter tool. The reason for this difference is that Rsync service keeps the connection state during the synchronization process, while the message receiving server can disconnect after receiving the synchronization request.

When testing, if the Rsync service is hosted on xinetd, the Rsync client starts to report errors, as shown in Figure IV, when the concurrency of PSSH is set to 30. The explanation for this error is that Rsync service refused to be connected. If the Rsync service is started directly in the form of a deamon, as shown in Figure V, the error in Figure IV starts to be reported when the concurrency of PSSH is set to 550.

![FIGURE IV. Error information in the Rsync server.](image)

![FIGURE VI. Error code in JMeter.](image)

In the test of our method, the TCP request we set is in the default way. Using the JMeter tool, we can make sure that when sending the connection request, a string is sent to simulate the synchronous request in the real situation, and then verify the received data from the server. Then, the concurrency is set to 1000, 3000, 5000, 8000, 10000, 12000, 15000, 20000, 25000 and 30000. It is found that when the concurrency reaches 20000, the CPU utilization of Centos1 reaches 100% for a long time, but the

| Dataset | Disk Space (MB) |
|---------|----------------|
|         | Our Method     | Rsync |
| 1       | 19.2           | 0     |
| 2       | 19.5           | 0     |
| 3       | 17.1           | 0     |
| 4       | 20.9           | 0     |
| 5       | 18.6           | 0     |
| 6       | 25.7           | 0     |
| 7       | 16.5           | 0     |
| 8       | 22.6           | 0     |
| 9       | 21.3           | 0     |
| 10      | 24.0           | 0     |

**TABLE IV. The client disk occupancy.**
correctness of the request can still be guaranteed because the return message from the server can be correctly received. However, when the concurrency reaches 30000, error messages begin to be received. We check the error messages and find that the error code is 500, as shown in Figure VI, which means that the server fails to provide services normally. The maximum number of concurrency that the Rsync service and our method can bear in the extreme case is shown in Figure VII.

3.3.3. Synchronization Time
To prove the feasibility of our method, we implement the whole process, in which the Spark is selected as the distributed computing framework. The experimental environment of this paper is implemented in the virtual machine environment. To simulate the network delay, this paper uses the TC tool in Linux to increase the network port delay of the Centos1. In the actual test, considering the low concurrency of Rsync, the synchronization time of a single file is tested. In this paper, 10 synchronization times are obtained based on ten datasets, and then the average value of these times is calculated. The experimental results are shown in Figure VIII.

To keep the change of Rsync as the only variable, we implement the Rsync algorithm and then add our method to it. The experimental data are obtained on our implementation.

It can be seen from Figure VIII that the synchronization time of our method is slightly longer than the Rsync. This is because our method has experienced Kafka’s disk drop and Spark’s shuffle process when processing a synchronization request, which results in some additional processing time. However, this paper only tests the synchronization of a single file at one time. In the case of high concurrency, the advantages of the distributed computing framework in our method will be reflected.
4. CONCLUSIONS
This paper analyzes the workflow of Rsync and finds out the reason why it is not suitable for parallel processing. To give Rsync the ability to work in parallel, this paper proposes a synchronization process optimization method based on shadow data. At the same time, it uses Kafka to decoupling the receiving and processing of one synchronization request, thus effectively parallelizing Rsync. The experimental results show that the proposed method can effectively improve the concurrency of Rsync, and can keep the same level with Rsync in the synchronization time of a single file. This method will effectively increase the synchronous request response-ability of server in the backup system.

ACKNOWLEDGMENT
Supported by the National Natural Science Foundation of China (Grant No. 61070015), Guangzhou Industrial Technology Major Project (Grant No. 201802020035) and Frontier and Key Technology Innovation Projects in Guangdong Province (Grant No. 2014B010110004).

REFERENCES
[1] Rahumed A, Chen H C H, Tang Y et al.. A Secure Cloud Backup System with Assured Deletion and Version Control[A]. 2011 International Conference on Parallel Processing Workshops, ICPPW 2011, Taipei, Taiwan, Sept. 13-16, 2011[C]. 2011: 160–167.
[2] Thomas J E, Galligher G C. Improving Backup System Evaluations in Information Security Risk Assessments to Combat Ransomware[J]. Computer & Information Science, 2018, 11(1): 14–25.
[3] Xiaokang Zhu. Research and implementation of key technologies of cloud disk based on EITP [D]. South China University of Technology, 2017.
[4] Tongzhou. Research on efficient data synchronization model based on priority hybrid architecture[D]. Nanjing University of Posts and Telecommunications, 2015.
[5] Guibao Chen. Research and implementation of data center synchronization software with high performance and high availability [D]. Xi'an University of Electronic Science and Technology, 2017.
[6] Bjørner N, Blass A, Gurevich Y. Content-dependent chunking for differential compression, the local maximum approach[J]. Journal of Computer & System Sciences, 2006, 76(3–4): 154–203.
[7] Drago I, Mellia M, Munafò M et al.. Inside Dropbox: Understanding Personal Cloud Storage Services[A]. Proceedings of the 2012 Internet Measurement Conference[C]. New York, NY, USA: Association for Computing Machinery, 2012: 481–494.
[8] MO R. Fingerprinting by Random Polynomials[EB/OL]. 1981. https://www.researchgate.net/publication/242608934_Fingerprinting_by_Random_Polynomials/related.

[9] Raju R, Moh M, Moh T. Compression of Wearable Body Sensor Network Data Using Improved Two-Threshold-Two-Divisor Data Chunking Algorithms[A]. 2018 International Conference on High Performance Computing & Simulation (HPCS)[C]. 2018: 949–956.

[10] Zhang Y, Feng D, Jiang H et al.. A Fast Asymmetric Extremum Content Defined Chunking Algorithm for Data Deduplication in Backup Storage Systems[J]. IEEE Transactions on Computers, 2017, 66(2): 199–211.

[11] Widodo R N S, Lim H, Atiquzzaman M. A new content-defined chunking algorithm for data deduplication in cloud storage[J]. Future Generation Computer Systems, 2017, 71(jun.): 145–156.

[12] Ni F, Lin X, Jiang S. SS-CDC: A Two-Stage Parallel Content-Defined Chunking for Deduplicating Backup Storage[A]. Proceedings of the 12th ACM International Conference on Systems and Storage[C]. New York, NY, USA: Association for Computing Machinery, 2019: 86–96.

[13] Won Y, Lim K, Min J. MUCH: Multithreaded Content-Based File Chunking[J]. IEEE Transactions on Computers, 2015, 64(5): 1375–1388.

[14] Zhi Tang, Youjip Won. Multithread Content Based File Chunking System in CPU-GPGPU Heterogeneous Architecture[A]. B. Carpentieri. 2011 First International Conference on Data Compression, Communications and Processing[C]. Los Alamitos, CA, USA: IEEE Computer Society, 2011: 58–64.

[15] A. Tridgell. Efficient Algorithms for Sorting and Synchronization[EB/OL]. 1999. https://www.samba.org/~tridge/phd_thesis.pdf.