Distributed Systems (SDIS)
Class 2, Group 3
2020/2021

Serverless Distributed Backup Service
1st Practical Work

João Carlos Machado Rocha Pires (up201806079)
João Ricardo Ribeiro Cardoso (up201705149)

12-04-2021

Contents

1. Introduction

This report is intended to be a description for the enhancements made to the peers protocol and also for the concurrent execution of protocols.

2. Enhancements

Throughout the following sections, we describe the enhancements, the proposed solution, which includes both the idea behind the implementation and its reason, giving examples and making also an analysis of our solution in terms of problems and limitations.

2.1 BACKUP Enhancement

This scheme can deplete the backup space rather rapidly, and cause too much activity on the nodes once that space is full. Can you think of an alternative scheme that ensures the desired replication degree, avoids these problems, and, nevertheless, can interoperate with peers that execute the chunk backup protocol described above?

2.1.1 Proposed solution

In version 1.0, every Peer that receives a PUTCHUNK message stores the CHUNK and sends a STORED message after a random delay (between 0-400 ms). This would cause the backup space to be filled fast. Our solution was to have the Peer that receives the initial message store the chunk like initially, and after the random delay, check how many STORED messages for that chunk it has received, to determine the perceived replication degree of that chunk. If that perceived replication degree is higher or equal than the desired replication degree, it deletes the chunk again without sending the STORED message. If the replication degree has not been satisfied yet, it sends the STORED message like usually.

2.1.2 Limitations of the proposed solution

The only problem with our solution is that if two or more Peers send the STORED message in a really small interval, there is a possibility that the perceived replication degree ends up being higher than the desired replication degree. However, if a certain Peer becomes full, he still tries to remove chunks that have a higher than desired replication degree, so this problem would have little to no impact in system.

2.2 DELETE Enhancement

If a peer that backs up some chunks of the file is not running at the time the initiator peer sends a DELETE message for that file, the space used by these chunks will never be reclaimed. Can you think of a change to the protocol, possibly including additional messages, that would allow to reclaim storage space even in that event?

2.2.1 Proposed solution

To solve the problem above, our solution is as follows: every time a Peer comes back online, it sends, for each chunk it has stored with an unique file, a STORED message with the protocol version 2.0. A Peer doesn't need to send a message for each chunk, because if one chunk belonging to one file is to be deleted, then another chunk belonging to the same file will be deleted too anyways.

A Peer that receives that message will respond with a DELETE message if it doesn't have that file in the list of files that it backed up. Otherwise, it will not send any message.

Lastly, the first Peer (that was coming back online), just like it would normally do after receiving a DELETE message, deletes any chunks of the file with the same ID as the one in the message. Of course, a Peer only deletes a chunk if the senderID in the message matches the ID of the initiator Peer of that chunk.

2.2.2 Limitations of the proposed solution

Our solution has two limitations. The first is that it only works if the initiator Peer and the Peer that comes back online both operate with the version 2.0. The second is that it only works if the initiator Peer is online. Both these limitations are to be expected, and are more of a limitation of the system, rather than of our solution itself. Therefore, we consider the solution to be adequate for the problem.

3. Concurrency

For concurrency, we used an implementation based on threads.

3.1 Peer-TestApp Comunication using RMI

Since we used RMI for communication between Peer and TestApp, there is a thread being created when TestApp invokes a specified protocol on the PeerInterface. This thread is used to send, for example, PUTCHUNK messages when backing up a file, using the MulticastSender class.

3.2 Peer-Peer Comunication using Multicast Channels

A Peer creates one thread for each multicast channel using the MulticastReceiver class. These threads are used to receive messages on the corresponding multicast channel. When a message is received, we create a new thread to process the message using the MessageProcessor class. This implementation guarantees that multiple messages can be processed at the same time. We use ConcurrentHashMap instances to store most of our data because it is a class that supports full concurrency of retrievals and high expected concurrency for updates, making sure that processing messages at the same time doesn't cause any problems.

Peer.java

```
public class Peer implements PeerInterface {
    ...
    public static void main(String[] args) {
        ...
        new Thread(new MulticastReceiver(controlMultCast)).start();
        new Thread(new MulticastReceiver(backupMultCast)).start();
        new Thread(new MulticastReceiver(restoreMultCast)).start();
        ...
    }
}
```

MulticastReceiver.java

```
public class MulticastReceiver implements Runnable {
    ...
    public void run() {
        try {
            MulticastSocket socket = new MulticastSocket(port);
            InetAddress group = InetAddress.getByName(address);
            socket.joinGroup(group);
            while (true) {
                byte[] buffer = new byte[FileUtils.chunkSize +
                   MessageUtils.maxHeaderSize];
                DatagramPacket packet = new DatagramPacket(buffer,
                   buffer.length);
                socket.receive(packet);
                byte[] data =
                   Arrays.copyOf(packet.getData(),packet.getLength());

                new Thread(new MessageProcessor(data)).start();
            }
        } catch (IOException e) {...}
    }
}
```