SDIS

Project 1

Distributed Backup Service

Clara Alves Martins up201806528
Daniel Filipe Amaro Monteiro up201806185

Concurrency Design

Our project consists of a concurrent implementation of a distributed backup system. The peer class is **`Server`**, while the testing class is **`TestApp`**.

In our **`Server`** class, we implement a concurrency pattern using three main threads:

- The main thread is the one that offers the backup system based on the **`ClientEndpoint`** class.
- The second and third threads listen to the Multicast Control Channel and the Multicast Data Backup Channel. The **`ListenerThread`** class is the base for these two threads.

We chose not to use a thread to infinitely listen to the Multicast Data Restore Channel since the messages transmitted in this channel are always a response to the `GETCHUNK` message, and therefore would not benefit from multithreading. The **`Server`** class is responsible for spawning and managing the peer's threads (see lines 73 - 78 of `Server.java`).

When running our program, the **`TestApp`** will execute a **`ClientCommand`**. There are five client commands:

- **`BackupCommand`** - initiating the backup protocol and sending the `PUTCHUNK` messages for all the chunks in the file.
- **`DeleteCommand`** - initiating the delete protocol for the specified file and sending the `DELETE` message.
- **`ReclaimCommand`** - initiating the reclaim protocol in the desired peer.
- **`RestoreCommand`** - initiating the restore protocol, sending the `GETCHUNK` message for each file chunk, and recreating the file.
- **`StateCommand`** - initiating the state protocol and returning the peer's current state.

The **`TestApp`** connects to the RMI registry and looks up the access point given on the command line. Then, it will call the correct **`ServerCommands`** method implemented on the **`ClientEndpoint`** class.

The **`ClientEndpoint`** class will be in charge of starting all the implemented protocols on the peer side.

For each available message, we provide an `answer()` method. This method allows us to do the message processing and send responses if necessary. Since some of these answers require extensive processing, they will be wrapped in their own thread to increase concurrency and avoid losing messages from the multicast channels (due to buffers filling up).

We have the following message classes, each with its own unique behavior:

- **`BackupSenderMessage`** - sent by the backup initiator peer, initiates the backup protocol in all the peers that receive it.
- **`BackupReceiverMessage`** - sent by the peers that have stored a file chunk, indicates to the other peers that this peer stored it.
- **`RestoreSenderMessage`** - sent by the restore initiator, initiates the restore protocol in all the peers that receive this message and have the chunk stored in their storage.
- **`RestoreReceiverMessage`** - sent by the faster peer that has the file chunk stored, transmits the chunk back to the peer that initiated the restore protocol.
- **`DeleteSenderMessage`** - sent by the delete initiator peer to make all the other active peers delete the specified file.
- **`ReclaimReceiver`** - sent by the peer that, after suffering a storage space reclamation and losing storage space, needs to delete this chunk from its storage.

The Backup Protocol uses two message types: **`BackupSenderMessage`** and **`BackupReceiverMessage`**. The Restore Protocol also uses two message types: **`RestoreSenderMessage`** and **`RestoreReceiverMessage`**. The Delete Protocol uses only one message type: **`DeleteSenderMessage`**. And the Reclaim Protocol also uses only one message type: **`ReclaimReceiver`**.

The State Protocol doesn't need peer-to-peer interaction, meaning that there are no messages exchanged between them.

To implement the State Protocol, we persistently store the state information (once every 20 seconds) in a JSON file unique to each peer. The **`PeerState`** class is responsible for maintaining this file, and it also manages the replication degrees of the backed-up chunks.

Backup Enhancement

Proposed: 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?

Implementation

The backup enhancement is a way to ensure the desired replication degree while trying to avoid taking up too much space on the peers and reducing the number of transmitted messages once the peers can't store any more chunks. To accomplish this, we divided this enhancement into two distinct parts:

- Avoid taking up unnecessary space on the peers
- Avoid retransmitting if there is no more space on the peers

To avoid taking up unnecessary space on the peers, we established that when a peer notices that the other peers already met the replication degree, it won't need to store the chunk itself. This way, we can reduce the number of replications beyond the desired replication degree and reduce the space taking on the peers.

To avoid retransmission when there is no more space on the peers, we established that after the initiator peer has received the same number of answers from the same peers, it stops trying to back up the file. When the answers repeat themselves, it is highly likely that either the rest of the peers is full or the number of peers is insufficient to meet the desired replication degree. Either way, the maximum possible replication degree has been acquired, and there is no point in continuing to send messages.

Delete Enhancement

Proposed: 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 reclaiming storage space even in that event?

Implementation

The delete enhancement can be thought of as a way to reclaim storage space when a file gets deleted while one of the peers that stored this file is not running. With that in mind, we created 3 new messages:

- `<Version> START <SenderId>`
- `<Version> DELETED <SenderId> <FileId>`
- `<Version> REMOVE <SenderId> <FileId> <TargetId>`

With these messages, we were able to identify which peers correctly deleted the specified file by having each peer send the DELETED message after receiving the DELETE message and removing the file from their storage. After receiving all the DELETED messages, we can identify which peers didn't delete the file and store that information in our peer state.

When a peer starts its execution, it shall send a START message. When the other peers receive that message, they will go through their states and verify if that peer needs to delete a file. If that occurs, they will send a REMOVE message containing both the file id and the target peer id. The target peer id is the id of the peer that needs to delete the specified file, the id of the peer that has just started.

When a peer receives the REMOVE message, it simply removes the file from its state and storage, reclaiming all the space occupied by those chunks. After that, it sends a REMOVED message to notify the other peers that it has deleted the file from its storage.