Distributed Backup Service for the Internet

Distributed Systems 2019/2020

João Pedro Ribeiro
Francisco Almeida
Ricardo Pinto

Integrated Master in Informatics and Computing Engineering

May 2020

Table of Contents

Overview

This report describes the implementation of a distributed backup service for the internet. The developed application uses a centralized design, in which a central peer, which we will refer to as the server, acts as an orchestrator for the communication between other peers. The communication between nodes in the system is done using the TCP protocol. The backup service supports 4 operations: file backup, file restore, file deletion and space reclaim. In order to increase concurrency, the application makes use of multithreading allowing for the concurrent execution of several operations.

Protocols

This section describes the protocols used by the service during its operation. Besides RMI for the communication between the test client and peer and TCP and SSL used for the communication between nodes, each operation supported by the system uses a specific application protocol that works on top of TCP. These protocols are BACKUP, RESTORE, DELETE and RECLAIM. Besides these, there is another protocol called CONNECT which refers to the connection established between peers and the server.

RMI

The RMI protocol is used in the communication between the test client and the peers. The peer exposes remote methods that the client can call which correspond to the operations supported by the system. Below is the *main.sdis.peer.Peer* interface, which defines these methods.

```
public interface Peer extends Remote {
   void backupFile(String filePath, int replicationDegree) throws IOException;

   void restoreFile(String filePath) throws RemoteException, IOException;

```

```
    void deleteFile(String filePath) throws IOException;

    void reclaimSpace(long maxDiskSpace) throws RemoteException, IOException;

    void retrieveState() throws RemoteException;

    String getAccessPoint() throws RemoteException;
}
```

This interface is then implemented by the *main.sdis.peer.PeerImpl* class.

Application Protocols

As mentioned previously, each operation uses an application protocol. Each one of these protocols uses a set of messages which are exchanged between peers and the server and between peers as well. The general format for every message is as follows:
<message_type> <sender_address/port> [args...]

CONNECT

The CONNECT protocol is used by peers to tell the server that they are active. When a peer is launched, it sends a CONNECT message to the server.. Upon receiving this message, the server registers the peer in a list of connections and replies with a CONNECTED message. Every 10 seconds, the server goes over its list of connections and sends a PING message to every connected peer. If a peer fails to respond with a PONG message, it is removed from the list. This list is essential for the execution of other application protocols.

BACKUP

The BACKUP protocol is used by the initiator to backup a file in the system. When initiated, the peer first sends a GETBACKUPPEERS message to the server. This message takes the following arguments:

- The File ID of the file
- The desired replication degree

Upon receiving this message, the server responds with a BACKUPPEERS message containing the list of addresses of peers that are available to backup the file. This list is returned by the *getBackupPeersMethod()* inside the *main.sdis.server.Server* class and is a randomized list with size equal to the desired replication degree of peers that have not yet backed up the specified file, excluding the initiator peer as well.

```
public synchronized List<InetSocketAddress> getBackupPeers(GetBackupPeersMessage
message) {
        List<InetSocketAddress> peersOfFile =
storage.getPeersOfBackedUpFile(message.getFileId());

        List<Connection> randomConnections = new ArrayList<>(connections);
            randomConnections.removeIf(conn ->
conn.getClientAddress().equals(message.getSenderAddress())
                    || (peersOfFile != null &&
peersOfFile.contains(conn.getClientAddress())));
            randomConnections = Utils.randomSubList(randomConnections,
message.getReplicationDegree());

        return getConnectionAddresses(randomConnections);
    }
```

After receiving the list of suitable peers, the initiator peer sends a PUTFILE message to each of them. Upon receiving a PUTFILE message, a peer stores the file. The arguments of the PUTFILE message are as follows:

- The File ID of the file
- The desired replication degree
- The file's data (bytes)

After storing the file, the peer replies to the initiator peer with a STORED message. Upon receiving this response, the initiator peer sends a CONFIRMSTORED message to the server. This message takes the following arguments:

- The File ID of the stored file
- The InetSocketAddress of the peer who stored the file
- The desired replication degree of the file

Upon receiving the CONFIRMSTORED message, the server stores the confirmation and replies with an OK message. If the system fails to replicate the file with the desired replication degree, the initiator peer restarts the BACKUP protocol after 2 seconds, up to a maximum of 5 times. The initiation of the BACKUP protocol is handled by the *main.sdis.peer.protocol.Backup* class.

RESTORE

The RESTORE protocol is used by the initiator peer to restore a previously backed up file. When initiated, the peer first sends a GETRESTOREPEERS message to the server. This message takes as argument the File ID of the file. Upon receiving this message, the server responds with a RESTOREPEERS message containing the list of addresses of peers that have previously stored the file. This list is returned by the *getRestorePeersMethod()* inside the *main.sdis.server.Server* class and is a randomized list of the peers that stored the file. After receiving the list of suitable peers, the initiator peer iterates over the list sending a GETFILE message to each peer until one of them sends the file. This message takes as a single argument the File ID of the file. Upon receiving a GETFILE message, a peer fetches the file from its storage and sends it to the initiator peer inside a FILE message, which takes as arguments the File ID of the file and its data (bytes). The initiation of the RESTORE protocol is handled by the *main.sdis.peer.protocol.Restore* class.

DELETE

The DELETE protocol is used by the initiator peer to request the deletion of all copies of a previously backed up file. When initiated, the peer first sends a GETDELETEPEERS message to the server. This message takes as argument the File ID of the file. Upon receiving this message, the server responds with a DELETEPEERS message containing the list of addresses of peers that have previously stored the file. This list is returned by the *getDeletePeersMethod()* inside the *main.sdis.server.Server* class. After receiving the list of suitable peers, the initiator peer sends a DELETE message to each one of them. This message takes as argument the File ID of the file. Upon receiving a DELETE message, a peer deletes the file from its local storage and replies with an OK message. After receiving the OK message, the initiator peer sends a CONFIRMDELETE message to the server. This message takes as a single argument the FileID of the file and the InetSocketAddress of the peer who deleted the file. Upon receiving this message, the server registers the removal of the file from the system. If a peer fails to respond, the initiator peer restarts the DELETE protocol after 5 seconds, up to a maximum of 5 times. The initiation of the DELETE protocol is handled by the *main.sdis.peer.protocol.Delete* class.

RECLAIM

The RECLAIM protocol is used by the initiator peer to reclaim some of or the entirety of its local storage space. When initiated, the peer counts the used space and compares with the maximum disk space given. If the used space is over the maximum space, the peer deletes a file sending a GETRECLAIMPEERS message to the server. Upon receiving the message with the File ID of the deleted file, the server searches which peers can backup the file again, discarding the peers that already did a backup of the file, the peer that asked for the backup of that file initially and the peer that is reclaiming the space, sending the list of possible peers in a RECLAIMPEERS message. The initiator peer then proceeds to send a PUTFILE message to one of the peers on the list sent by the server, thus making the replication degree the same as before the deletion of the file. After storing the file, the peer sends a STORED message to the initiator peer. When this message is received, the initiator peer sends a CONFIRMSTORED message to the server. The server stores the confirmation and replies with an OK message.

After deleting and making sure that the replication degree of the file is the same, the initiator peer certifies that the used space is below the maximum space, if not, the initiator peer does the reclaim cycle until the used space is lower or equal to the given maximum disk space.

Concurrency Design

This section describes the design and implementation decisions that contribute towards the system's concurrency.

Threads

In order to support the concurrent execution of protocols and handling multiple messages simultaneously, the system runs tasks in parallel throughout several threads. For example, the initiation of each protocol is done by the initiator peer on a separate thread, thus allowing the peer to initiate multiple protocols without having to wait for the previous execution to finish. Moreover, both the peers and the server are listening for incoming messages on a separate thread and upon receiving a message, the handling is passed to a specific handler that knows how to take care of that message. For example, here is a part of the code of the main.sdis.peer.MessageReceiver class, which is a thread running in loop on the peer waiting to receive messages.

```
switch (message.getMessageType()) {
                   case PING:
                         executorService.execute(new PingHandler(peer, (PingMessage)
message, out));
                       break;
                   case PUTFILE:
                                   executorService.execute(new PutFileHandler(peer,
(PutFileMessage) message, out));
                       break;
                   case GETFILE:
                                   executorService.execute(new GetFileHandler(peer,
(GetFileMessage) message, out));
                       break;
                   case DELETE:
                                executorService.execute(new DeleteFileHandler(peer,
(DeleteMessage) message, out));
                       break;
                   default:
                       break;
```

```
            }
```

After accepting a connection on its Socket, the peer passes the handling of the message to a specific handler (which runs on a separate thread) and continues executing other tasks while the message is being processed.

Thread Pools

Every thread execution in the system is done using a *ScheduledExecutorService*, which internally makes use of a scheduled thread pool. The system uses this service through the *main.sdis.common.CustomExecutorService* singleton.

Thread safe data structures and methods

Since this solution makes no use of *Thread.sleep()* for synchronization, special care is required to avoid concurrent modification of objects. To achieve this, all methods that can be accessed concurrently by multiple threads are marked with the *synchronized* keyword. This creates a lock on the object to which the method belongs thus not allowing it to be executed by multiple threads simultaneously and avoiding the concurrent modification of its attributes. Moreover, all data structures that can be modified concurrently use a thread safe implementation. For example, the *backedUpFiles* attribute of the *main.sdis.server.Server* class is a *ConcurrentHashMap*.