Distributed Backup Service

Project 1 report

Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática e Computação

T2G11 Group

Catarina Almeida - up201606334@fe.up.pt

João Almeida - up201504874@fe.up.pt

April 14th, 2019

Index

1. Introduction

For the first project of *Sistemas Distribuídos*, we were proposed to develop a **Distributed Backup Service for a local area network (LAN)** based in the specification made available for this purpose.

As our implementation supports the concurrent execution of protocols, this report aims to explain our rationale behind it.

Furthermore, it also describes the enhancements our team put effort to improve this project.

2. Concurrency design

In our implementation, four channels of communication were created: the control channel (**MC.java**), the backup channel (**MDB.java**), the data recovery channel (**MDR.java**) and client's communication channel using RMI, starting from a **RMInterface.java** that extends Java's **Remote** class.

The peer communication channels (MC, MDB and MDR) extend an abstract class **Channel** that extends Java's **Thread**. We use one thread per multicast channel for receiving the messages sent via the respective channel. We can only execute one protocol at a time.

Each thread handles the messages received via the respective channel serially in an endless loop and in each iteration it receives a message on the respective multicast channel and processes the received message. As we have implemented the communication between the TestApp and the Peer using RMI, our multicaster thread is the thread created by the RMI runtime to handle the backup operation invoked by the TestApp.

It were used **ConcurrentHashMap** structures in order to save the information of our database. This approach is the most appropriate in multi-thread environments, allowing atomic operations, preventing the information from being corrupted. In addition, it confers scalability and a better performance to our application.

2.1 Relevant classes for concurrency design

2.1.1 Message.java

This class is the main core of our peer-to-peer communication. It receives a DatagramPacket and its size and extracts the header and body parts of each message. As requested, it accommodates the values of

<MessageType> <protocolVersion> <senderID> <fileID> <chunkNo> <replicationDegree> <CRLF>

This approach allows a better handling and manipulation of the messages, as well as extensions to other versions of the protocol.

2.1.2 Peer.java

This Java class is the main worker of the protocol, adapting to each job according to the request by multicast thread. Its **main** method executes a thread per channel and sets up the RMI, either creating it or joining it if already running.

Our approach to maintain each peer's state in a non-volatile memory is using Java's **object serialization**. This mechanism allows an object to be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in it. Our peer can retrieve each state from the filesystem using the method `serialize_storage.`

2.1.2 PeersInfo.java

This class associates with each peer the filesystem ecosystem: its **storage**, **receivedChunks**, **spaceUsedOnDisk** and files with backup. It also stores and updates its **maxStorage**.

3. Enhancements

3.1 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?

To ensure that each chunk has the desired replication degree we created the following condition:

When a peer saves a chunk in the filesystem all peers update the currReplicationDegree of that chunk. Before saving the chunk in the filesystem all peers should wait between 0 and 400 ms, after that period they should check if the currReplicationDegree is smaller than the ReplicationDegree, if it is smaller, the peer can save the chunk in filesystem.