Distributed Backup Service

FEUP - SDIS 2021

First Project - Group t6g12

João Diogo Vila Franca Gonçalves (up201806162)

Rafael Valente Cristino (up201806680)

Contents

1 Concurrency Design

Our concurrent implementation of the backup service's protocol is structured in the following way:

- One receiver thread per multicast channel. This thread starts worker threads that process the received messages, which allows for the processing of different messages received on the same channel at the same time.
- Multiple multicaster threads, one for each protocol instance, where each one is started by RMI for a given invocation of the `TestApp`. This allows for multiple protocol instances at a time.

1.1 Implementation

1.1.1 Receiver Multicast Threads and Worker Threads

The receiver threads are implemented in the form of the class `channels.ChannelListener`. This class is a thread, is instantiated once for each channel (MC, MDB and MDR) and has an infinite loop which upon the reception of a packet submits a task (worker thread) that will process it for execution.

Example of the packet reception loop in the class `channels.ChannelListener`:

```
(...)
MulticastSocket socket = channel.getSocket();
socket.joinGroup(channel.getHost());

while (true) {
    byte[] rbuf = new byte[65000];
    DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);

    socket.receive(packet);
    threadScheduler.execute(new Runnable() {
        @Override
        public void run() {
            byte[] data = packet.getData();

            Message msg = MessageParser.parse(data, packet.getLength());
            action.execute(msg, packet.getAddress());
        }
    });
}
(...)
```

1.1.2 Multicaster Threads

The multicaster threads are implemented as methods in the `Peer` class and only become separate threads when they are invocated by the `TestApp` (the threads are automatically created by RMI).

Example of a multicaster thread, in this case for the restore protocol, in the `Peer` class:

```
(...)
public Result restore(String fileName) throws RemoteException {
    if (!getPeerState().ownsFileWithName(fileName))
    {
        Logger.error("The file '" + fileName + "' doesn't exist in my history.");
        return new Result(false,
            "The file '" + fileName + "' doesn't exist in the peer's history.");
    }
    CompletableFuture<Result> f = new CompletableFuture<>();
    new Restore(f, configuration, getPeerState().getFileId(fileName)).execute();

    try
    {
        return f.get();
    }
    catch(Exception e)
    {
        Logger.error(e, true);
    }
    return null;
}
(...)
```

1.2 Noteworthy classes we used for concurrency

1.2.1 ScheduledThreadPoolExecutor

To support this concurrency design, we use the `java.util.concurrent.ScheduledThreadPoolExecutor` class to introduce the protocol's specified delays (in substitution of `Thread.sleep()`) and to execute the message processing tasks.

The use of this class eliminates the overhead of instantiating a new thread every time we need to run an asynchronous task, because the threads are instantiated once at the start of the program and reused throughout its execution. It is also better than using `Thread.sleep()` because with sleep the thread wouldn't be able to be reused while the timeout passed. By using the thread pool the thread is liberated and when the timeout finishes the task is executed in an unoccupied thread (if available, if not it waits in the queue).

Example of the use of thread scheduling to replace `Thread.sleep()` to send a `STORED` message:

```
(...)
configuration.getThreadScheduler().schedule(new Runnable() {
    @Override
    public void run() {
        try
        {
            Message msg = messageFactory.getStoredMessage(configuration.getPeerId(),
                                                          msg.getFileId(),
                                                          msg.getChunkNo());
            configuration.getMC().send(msg);
        }
        catch(Exception e)
        {
            Logger.error(e, true);
        }
    }
}, configuration.getRandomDelay(400), TimeUnit.MILLISECONDS);
(...)
```

1.2.2 AsynchronousFileChannel

To further eliminate blocking calls, we used the `java.nio.channels.AsynchronousFileChannel` when writing to the file system (to store chunks, files or the peer's non-volatile state) so that those calls wouldn't block the rest of the execution of the program.

This was useful because in this case, whenever we write, we don't need to wait for it to finish, we can just go on with the normal execution of the program. However, for reads, we would need the result of the read almost immediately after the reading call (if it was asynchronous we would have to wait in the same way), so we opted to keep using synchronous reads.

1.2.3 ConcurrentHashMap

In the places that needed the use of maps and also their concurrent access and modification we used the class `java.util.concurrent.ConcurrentHashMap<K,V>`, which supports full concurrency of retrievals and high expected concurrency for updates. Example of this is the class `state.PeerState`, which is shared among all threads and supports the Peer's non-volatile state.

1.2.4 ThreadLocalRandom

We opted to use the `java.util.concurrent.ThreadLocalRandom` class instead of `java.util.Random` because if we shared the same `Random` object between all the threads, they would all share the same seed. `ThreadLocalRandom` eliminates this issue and reduces contention among threads.

2 Protocol Enhancements

These enhancements are executed when the protocol version "1.1" is supplied to the peer.

2.1 Backup

The vanilla backup protocol makes it possible for all peers to store the same chunk while not being aware that they are saving it unnecessarily. For example, in a pool of 20 peers, one chunk that is backed up with a desired replication degree of 4 may very well end up having a real replication degree of 19 (excluding the initiator peer). This happens because when the peers store the chunk they do not evaluate whether it has already been stored enough times. This may deplete the overall system backup space much more quickly than it should if the chunk only existed in its desired replication degree.

To avoid this issue, while ensuring the desired replication degree and being able to inter-operate with peers that execute the vanilla chunk backup protocol, we came up with a solution that is based on how the peer processes an incoming `PUTCHUNK` message. When such a message arrives the peer starts storing the `STORED` messages he has received and, just like in the vanilla version, the peer randomly waits up to 400ms. When that time has passed, he checks the number of `STORED` messages that he received. If that number is equal or greater than the desired replication degree he aborts and doesn't store the chunk. Otherwise, he sends a `STORED` message and stores the chunk.

This solution also has problems. It depends on the distribution of `STORED` messages over those 400ms of delay. If we have 10 peers or more, that would mean that many stored messages would be ignored by peers that happen to save the chunk at more or less the same time. One thing that improved this was, instead of waiting a random amount up to 400ms, waiting (400 + rand(2000))ms. During the first 400ms, the peers which upon reception of the `PUTCHUNK` message already had the chunk stored send their `STORED` messages (after rand(400)ms). The peers that are storing the chunk for the first time will always wait between 400 and 2400ms. After that, they execute the procedure described in the previous paragraph. With this modification, the first delay of the backup initiator peer was also changed to 3000ms to accommodate for the extra time that the peers will take to store the chunk. This slows down the backup process a bit, but ensures a better memory expenditure.

This is implemented in the `channels.handlers.strategies.EnhancedBackupStrategy` class.

2.2 Restore

In the vanilla version of the restore protocol, peers reply to `GETCHUNK` messages with a multicast `CHUNK` message that includes the chunk data in the message body. That means that the body of the chunk will be sent to all peers, even those who didn't ask for it. That wastes a lot of bandwidth unnecessarily.

To improve on this, we devised a solution that uses TCP to establish a connection between the peer who wants to get the chunk and the peer who is going to send it.

When a peer receives a GETCHUNK message, he waits a random amount of time up to 400ms. If during that time he didn't receive a CHUNK message (if he received it would mean that another peer already replied to the request and that he can abort), he starts a TCP server using a ServerSocket initialized with the 0 value - which chooses an arbitrary port - and with a configured timeout of 5 seconds (in case there was a conflict and other peer also replied to the GETCHUNK message, which means that no peer will connect to his TCP server because the initiator peer already obtained the chunk). The setup of the TCP server is implemented in the channels.handlers.strategies.EnhancedRestoreStrategy class.

After setting up the TCP server, the peer sends the CHUNK message, but instead of the body being the chunk data, it contains the port number in string format. Once he receives a connection to the server, he sends the chunk body and then closes the socket.

Meanwhile, when the initiator peer receives the CHUNK message, it gets the other peer's address from the UDP DatagramPacket of the CHUNK message, and the port number from the message body and connects to the TCP server, from where it reads the chunk data. The handling of the CHUNK message is done in the channels.handlers.RestoreChannelHandler class.

This solution also works with peers that implement the vanilla protocol. If a peer receives a GETCHUNK message with a version of 1.0, it replies with the chunk's data in the body of the CHUNK message, even if he is running on the 1.1 version of the protocol. It only sets up the TCP connection if the GETCHUNK message comes with a version of 1.1.

2.3 Delete

In the vanilla version of the delete protocol, if a peer that stored chunks of a given file is down and that file's initiator peer deletes the file, that peer will have chunks that aren't supposed to exist occupying its storage.

To fix this we devised a solution that includes a new message, the FILECHEK message:

```
<Version> FILECHECK <SenderId> <FileId> <CRLF><CRLF>
```

When a peer initiates, it sends to the multicast control channel one FILECHECK message for each of the files that he has stored (independently of the number of chunks that he has of each one). This is implemented in the actions.CheckDeleted class.

Upon reception of that message, a peer this has marked that file as deleted waits a random amount of time up to 400ms, and if at the end of that time he didn't receive a DELETE message for that file, replies with its own DELETE message (which causes the peer that sent the FILECHECK message to delete the file). This is implemented in the channels.handlers.ControlChannelHandler class.