Distributed Backup Service

Final Report

Universidade do Porto

Faculdade de Engenharia

FEUP

Master in Informatics and Computing Engineering

Distributed Systems

Authors:
Duarte Nuno Pereira Duarte - 201109179
Ruben Fernando Pinto Cordeiro - 201109177

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

March 30, 2014

Contents

1 Architectural design

In order to ensure a proficient multi-threaded implementation of the sub protocols, several features of the *.NET Rx* framework were used.

Each one of the *Multicast* channels (MC, MDB and MDR) are modeled as both a MulticastListener and a MulticastBroadcaster class.

The MulticastBroadcaster class encapsulates a socket in order to send data to the network.

The MulticastListener class encapsulates a socket in order to receive data from the network. It also contains a reference to a *System.Reactive.Subject* object. This *Subject* is a provider of asynchronous push-based notifications: for each message received from the socket, all of the *Observers* subscribed to this channel are notified of it's arrival. See figure 1 for more details:

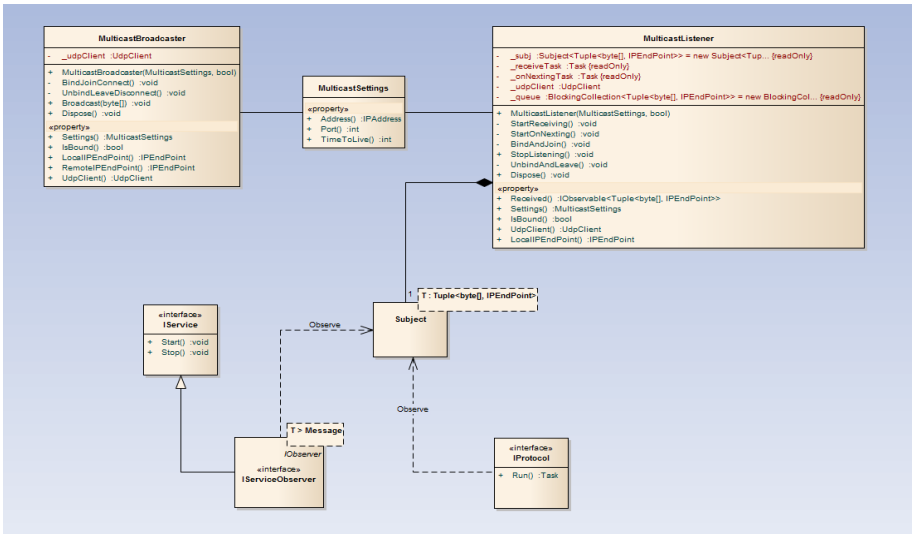

Figure 1: Channel UML diagram

Each one of the sub protocols described in the specification document is modeled as an **IProtocol** interface. This interface exposes a single method: *Run()*, that initiates an asynchronous System.Threading.Tasks.Task, containing the sequence of all the operations required for the protocol. Several instances of these protocols may be initiated by the peer in a concurrent fashion. See the UML diagram in figure 2:

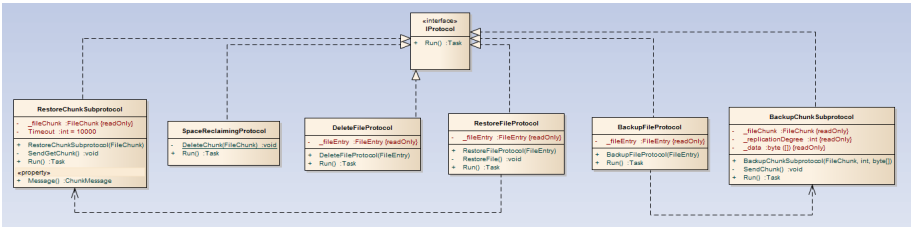

Figure 2: Protocols UML diagram

Additionally, each one of the **IProtocol** instances may subscribe to the multicast channels in order to receive message notifications.

Besides explicitly initiating sub protocols, the backup service also provides a set of background services that reply to requests coming from the network. These background tasks are modeled as **IService** class instances. These services are subscribed to one or more multicast channels, thus receiving asynchronous push-based notifications. See figure 3 for more details:

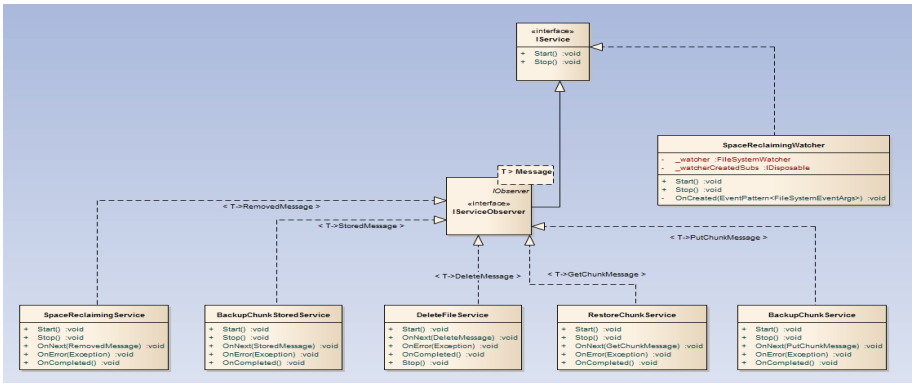

Figure 3: Services UML diagram

Finally, each message type sent/received through the network is modeled as a **Message** class. See figure 4:

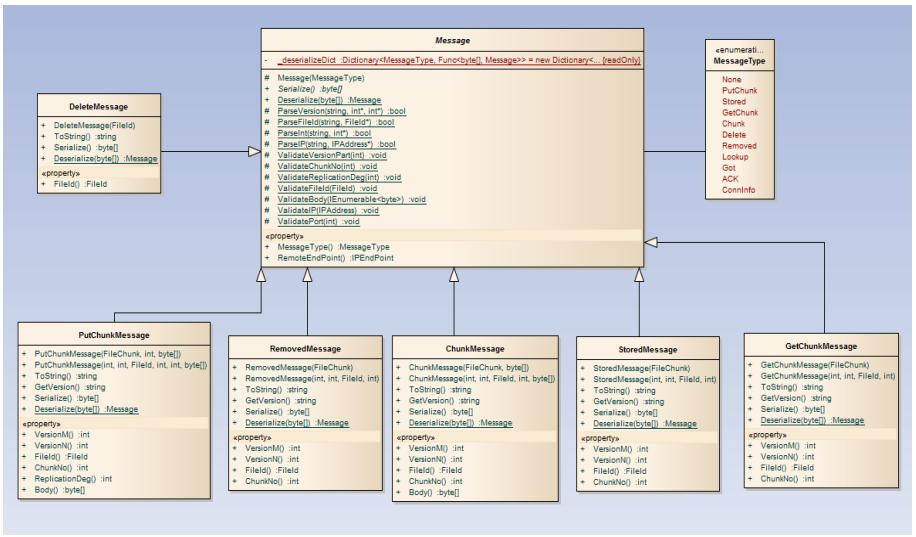

Figure 4: Messages UML diagram

2 Enhancements

All of the enhancements were implemented, the following sections describe the detailed implementation of each enhancement.

2.1 Data persistence

The information relative to the replication degree of each backed up chunk is stored in a **sqlite** database. This ensures constant data persistence. Besides the desired replication degree, the IP address of all the peers that backed up each chunk are also stored. These IP addresses ensure that duplicated **STORED** messages are not taken into account by each peer. This will be crucial to the enhancements described in section 2.2.

This database also stores the information about the files backed up by the peer.

2.2 Chunk backup subprotocol

Upon receiving the PUTCHUNK message, each peer checks it's local count of confirmation messages for the specified chunk after the random delay. If the count matches or exceeds the desired replication degree of the chunk, then the peer does not back up the chunk and doesn't reply with a confirmation message.

This optimization avoids the fast depletion of the backup space, since all the peers in the network stop backing up chunks with an actual replication degree higher than the desired.

2.3 Chunk restore protocol

In this enhancement, a peer sends the chunk via a TCP connection instead of sending it in a body of a CHUNK message via the MDR channel.

This new protocol is described in figure 5:

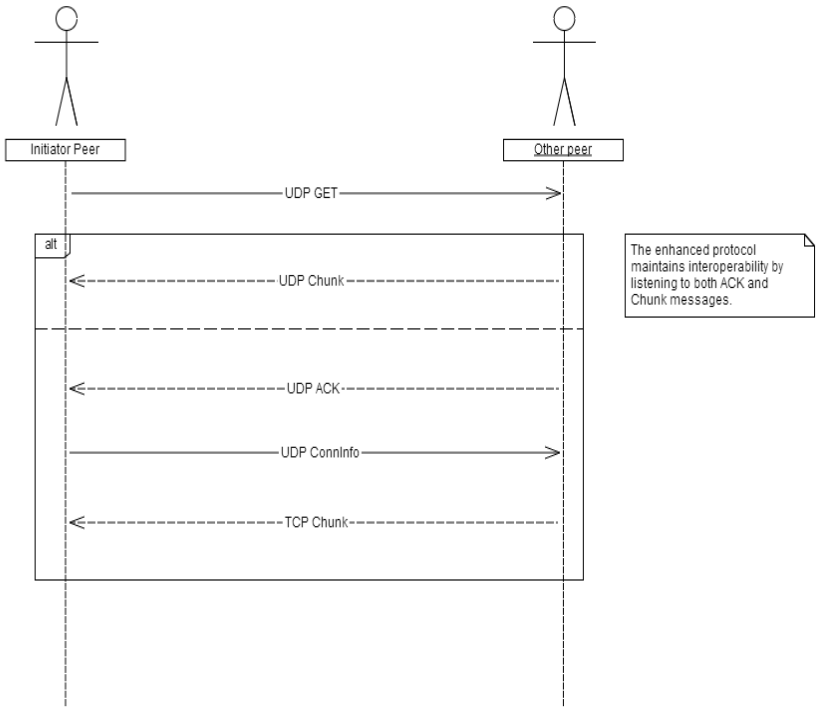

Figure 5: Enhanced restore chunk protocol sequence diagram

For this purpose, two new message types were added: **ACK** and **CONNINFO** messages. These messages have the following structure:

```
ACK <Version> <FileId> <ChunkNo> <CRLF> <CRLF>

CONNINFO <Version> <FileId> <ChunkNo> <InitiatorPeerPort>
<SelectedPeerIP> <CRLF> <CRLF>
```

InitiatorPeerPort : selected TCP port by the initiator peer.

SelectedPeerIP : IP address of the peer that has sent the first ACK message.

The initiator peer sends a GETCHUNK message via the MC channel. After that, the initiator peer waits for either a CHUNK or ACK message. If the CHUNK message is the first to arrive, the protocol finishes, otherwise, a CONNINFO message is sent via the MC channel. The initiator peer opens the TCP connection and waits for the other peer to open it as well. This ensures the protocol interoperability: we are still able to restore data from non conforming peers.

Upon receiving a GETCHUNK message, a peer with the requested chunk sends an ACK message after a random delay uniformly distributed between 0 and 400 ms. Only the peer with the IP address equal to the **SelectedPeerIP** field of the CONNINFO message actually opens the TCP connection and sends the requested chunk, every other peer discards it.

By setting up a TCP connection, we're sending chunks in a more reliable channel while reducing the load on the MC and MDR channels.

2.4 File deletion subprotocol

In order to reclaim space occupied by unused chunks, each peer periodically initiates a **Lookup** protocol.

This protocol consists in the following operations:

- Initiator peer:
    1. Create a list with all the distinct fileId's associated with every backed up chunk.
    2. Send a LOOKUP message for each distinct fileId via the MC channel and listen for GOT messages during a timeout interval (starts at 1 second).
    3. Repeat step 2 up to 5 times, doubling the timeout interval in each iteration.
    4. Delete all the backed up chunks associated to a fileId that wasn't acknowledged via a GOT message.
- Other peers:
    1. For every LOOKUP message received, check if the fileId corresponds to a backed up file. If it does, reply with a GOT message for the same fileId.

The LOOKUP and GOT messages have the following structure:

```
LOOKUP <Version> <FileId> <CRLF> <CRLF>
GOT <Version> <FileId> <CRLF> <CRLF>
```

This protocol ensures that only chunks belonging to active peers in the network are backed up.

2.5 Space reclaiming subprotocol

In order to tolerate chunk backup subprotocol failures and increase the replication degree of each chunk to the desired value, a new background service was implemented: **EnhancedSpaceReclaimingWatcher**. This service runs periodically and contemplates the following operations:

1. Initiate a chunk backup subprotocol for every chunk with an actual replication degree lower than the desired. Give priority to the chunks with the biggest replication degree offset.
2. Remove all the chunks with an actual replication degree higher than the desired.

Besides this new service, a new enhancement was introduced in the restore chunk protocol: upon receiving a GETCHUNK message, check the the database for the chunk entry. If there's an entry in the database but the actual chunk is missing in the backup directory, send a REMOVED message with the given fileId and chunkNo and remove the entry from the database, since the peer in question was supposed to have a local backup of the chunk. This enhancement tolerates faults from the local filesystem or external interference (chunk erased outside the program).