Local Area Network Distributed Backup Service

Enhancements specification

Universidade do Porto

Faculdade de Engenharia

FEUP

Distributed Systems

3^{rd} **year**

Master in Informatics and Computing Engineering

Class 4 - Group 7

José Peixoto	200603103	ei12134@fe.up.pt
Pedro Moura	201306843	up201306843@fe.up.pt
Paulo Costa	201206045	ei12086@fe.up.pt
Marina Camilo	201307722	up201307722@fe.up.pt

May 29, 2016

Contents

1 Sub-protocols enhancements

In order to simplify the development of the new security and scalability features, a code restructuring was made to make use of the sub-protocols enhancements described below as the default system for the peer communication. In short, retro-compatibility is now broken.

1.1 Backup sub-protocol enhancement

The default backup sub-protocol relies on sending a `PUTCHUNK` message to the `MDB` multicast data backup channel and replying with a `STORED` message for each chunk backed up without checking if any other peer had already stored it, resulting in excessive replication.

In order not to rapidly deplete the backup space available in all peers, the backup enhancement sub-protocol waits between 0 to 400 milliseconds allowing for the arrival of other peers `STORED` messages regarding that same chunk and compares the current and desired replication degrees to decide if it also needs to store it.

Peer-initiator

```
while counter < 5
    send PUTCHUNK
    if received STORED count >= replicationDegree
        return
```

Peer

```
if received PUTCHUNK
    random delay 0-400 ms
    if currentReplicationDegree < desiredReplicationDegree
        store chunk
```

This solution ensures a reduced over-replication of the chunks while still allowing for restoration.

1.2 Restore sub-protocol enhancement

The default restore sub-protocol relies on sending a `GETCHUNK` to the `MC` multicast control channel and waiting for a few milliseconds for new incoming messages containing the desired `CHUNK` to restore. According to the specification, "each peer shall wait for a random time uniformly distributed between 0 and 400 ms, before sending the `CHUNK` message", preventing overloading the `MDR` multicast data recovery channel.

In the enhanced version of this sub-protocol, the peer-initiator opens a server socket and binds it to a random `TCP` port and sends a `GETCHUNK` request with that port in the header to the multicast control channel. If the remote peer has the chunk, an attempt to establish a direct one-to-one reliable `TCP` connection is made, replacing the need for the `MDR` multicast data recovery channel for the `CHUNK` packet transit. Communicating in this manner also prevents other uninterested peers from receiving irrelevant `CHUNK` messages.

Peer-initiator

```
create socketChannel server
send GETCHUNK request to the MDC
if received the CHUNK
    store the CHUNK data
```

Peer

```
if received GETCHUNK
    CHUNK = retrieveLocalChunk(fileID, chunkNo);
    if socketChannel connection is established
        send CHUNK
```

1.3 Delete sub-protocol enhancement

In the default delete sub-protocol, a peer that received a `DELETE` request message shall remove from its backing store all the chunks belonging to the specified file.

The enhanced version adds a new confirmation message that is sent from the peers that successfully deleted their copies of the chunks: the `DELETEACK`.

```
DELETEACK <Version> <SenderId> <FileId> <CRLF><CRLF>
```

It uses the same format as the predefined delete message and allows the peer-initiator to count and compare the received confirmations of the `DELETE` request and expected file replication degree.

Peer-initiator

```
create thread
replicationDegree = chunkReplicationDegree
waitingTime = 15 seconds
while waitingTime < 16 minutes && replicationDegree > 0
    send DELETE
    replicationDegree -= DELETEACKS
    waitingTime *= 2
```

Peer

```
if received DELETE
    delete chunks
    send DELETEACK
```

1.4 Reclaim sub-protocol enhancement

The default reclaim sub-protocol attempts to delete some local chunks in order to free the requested disk space and communicates through the multicast control channel that it has removed the selected chunks, sending the `REMOVED` message. The other network peers, upon receiving the `REMOVED` message for a given chunk, compare the new replication degree of the chunk to its desired degree and in case the former is lower, the backup sub-protocol for that chunk might be initiated.

When the backup sub-protocol fails before its completion (both when a chunk is being backed up for the first time and when a copy of the chunk is deleted), the replication degree of the file might be lower than expected, effectively lowering its replication degree for some chunks. In the developed enhanced version, when a peer receives a `PUTCHUNK` request through the multicast data backup channel containing a chunk it has already backed up, a waiting period for the maximum backup sub-protocol time is made, followed by the re-initialization of the backup sub-protocol, this time by another peer, if the desired replication degree wasn't reached.

Peer

```
if received PUTCHUNK(c) && hasChunk(c)
    wait BACKUP_MAX_CONFIRMATION_WAIT;
    clearInbox();
    wait MAX_DELAY_MILLISECONDS;
if no PUTCHUNK(c) was received &&
        currentReplicationDegree < expectedReplicationDegree
    start backup(c)
```

2 Security

Contrary to what was lectured, the security layer was in fact added as an after-thought. The developed security features described below attempt to provide integrity, authenticity and confidentiality when transmitting the chunks through the network.

2.1 Integrity and authentication

The developed integrity chunk check scheme also provides peer authentication, because the `SHA-256` hashing function used to compute the chunk digest is also combined with a shared symmetric key that is known by all the authorized peers of the `LAN`. The hash-based message authentication code system prevents the undesired modification of the chunk content by third parties, also allowing the authentication of each message's sender.

2.2 Chunk confidentiality

The chunk confidentiality is ensured using a public key or asymmetric encryption system. In the starting process, each peer, reads or computes a new `RSA 4096 bit` length key-pair. A peer uses its public key to encrypt the file and then splits the cyphered data output in chunks in order to send them across the network. On restoration, after all chunks data are retrieved and merged, the decryption process is possible only to the peer which detains the corresponding private key, effectively reversing the previous process and decrypting the cypher text into plain-text of the original file.

3 Fault tolerance

Each time the database data structures are significantly changed during runtime, a local backup (save) is done by simply rewriting these data structures to disc. In addition, this important meta-data can also be remotely backed up through the test application graphical user interface, by manually clicking on the `"Backup metadata"` button. This action simulates an update by deleting and backing up again the files in case this peer's meta-data was already uploaded to the network. The meta-data files are also encrypted using the network shared AES key, so that even if the peer loses its own private RSA keys it can still recover this information.

In case some database meta-data is lost it can be retrieved by clicking the `"Restore metadata"` button. This action restores the database files and shuts down and reboots the peer network activity, so it can apply the new downloaded restored files.

4 TestApp graphical interface

The former command-line companion test application was improved to provide a new graphical interface that improves the interaction with the peers.

4.1 Key functionalities

- Cross-platform using the Java programing language

- Native operating system look and feel
- Easy to connect/disconnect to different peers
- Browse the system folders and select multiple files at once
- Multiple file backup/restoration/deletion
- Operation progress information
- On demand meta-data backup and retrieval

4.2 Screenshots

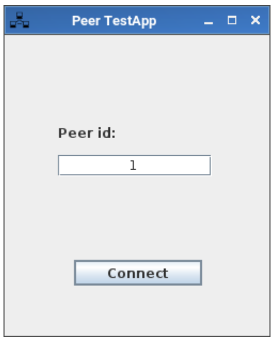
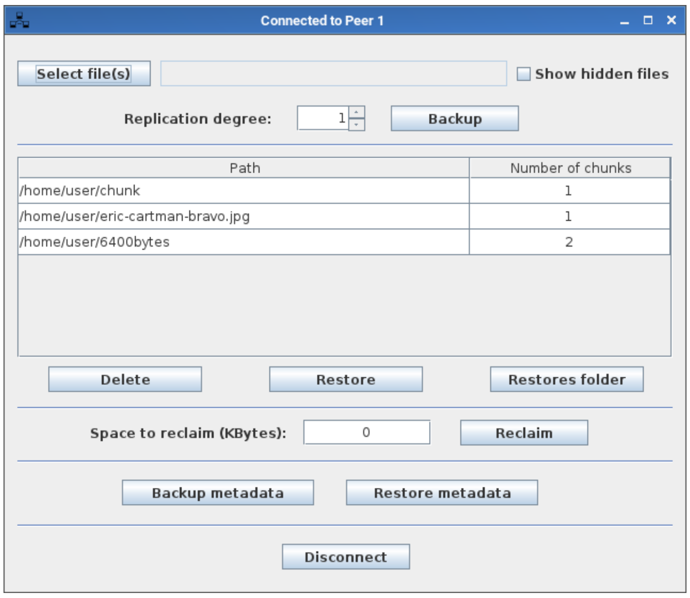
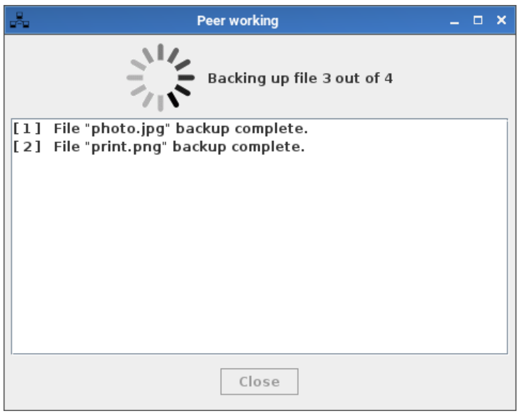


Figure 1: TestApp connection window



Path	Number of chunks
/home/user/chunk	1
/home/user/eric-cartman-bravo.jpg	1
/home/user/6400bytes	2

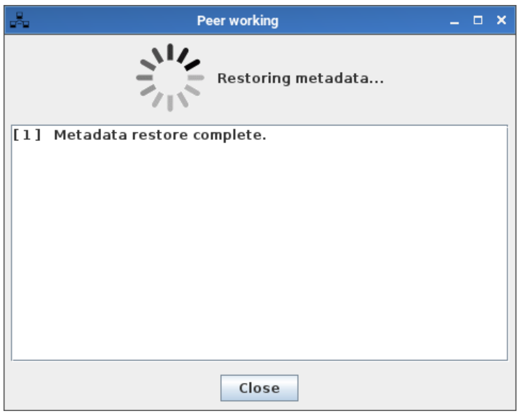
Figure 2: ActionTester main window



Backing up file 3 out of 4

```
[1]  File "photo.jpg" backup complete.
[2]  File "print.png" backup complete.
```

Figure 3: PeerRequest dialog with progress reports



Restoring metadata...

```
[1]  Metadata restore complete.
```

Figure 4: Meta-data restoration UI

5 Project development effort

5.1 Project development time estimations

Inside classes 6 hours.

Outside classes 40 hours.

Total 46 hours.

5.2 Individual contribution rates

José Peixoto 80%

Pedro Moura 18%

Paulo Costa 1%

Marina Camilo 1%