

SDIS 2015/2016 - 2nd Semester

Project 1 -- Distributed Backup Service

1. Introduction

In this project you will develop a distributed backup service for a local area network (LAN). The idea is to use the unused disk space of the computers in a LAN for backing up files in other computers in the same LAN. The service is provided by servers, one per computer, in an environment that is assumed cooperative (rather than hostile). Nevertheless, each server retains control over its own disks and, if needed, may reclaim the space it made available for backing up other computers' files.

1.1 Assumptions

The assumptions made regarding the environment and the system are essential when designing a distributed application. In this section, we try to make explicit all the assumptions that you can make, when devising your solution. We put them in a single section so that you can easily find them. Some assumptions stem from the environment in which the service is expected to operate on and are easy to understand. Other assumptions are not very realistic, but they are made to simplify your solution or its test.

We assume that the system is composed by a set of computers interconnected by a local area network. The service is provided by a set of servers, **possibly more than one per computer**, that participates in the backup service. A server manages local disk storage where it may store files, or parts thereof, and is identified by an integer, which is assumed to be unique and that never change.

We assume that the network may loose or duplicate messages, but that network failures are transient. I.e., if the sender keeps retransmitting a message, it will eventually reach its destination.

Likewise servers may fail by crashing, but they may also recover. Failure of a server may lead to the loss of files, not only the ones originally stored in that computer but also those it has backed up for other peers. We assume that the loss of data by a server is an event that is statistically independent of the loss of data by any other server, whether or not it is on the same computer. Furthermore, we assume that any files with metadata stored on a server are never lost on a server's crash.

Finally, we assume a local area network administered by a single entity in a friendly environment. All participants behave according to the protocol specified and do not attempt to take advantage of the service or to attack it. Furthermore, we assume that the participants do not modify, either intentionally or accidentally, the backed up data.

2. Service

The backup service is provided by a set of servers. Because no server is special, we call these servers "peers". (This kind of implementation is often called serverless service.) As stated earlier, **each peer is identified by an integer**, which is unique among the set of peers in the system.

2.1 Service Description

The purpose of the service is to backup files by replicating their content in multiple servers. We assume that each file has a "home" server, which has the original copy of the file. Although the file will be stored on some file system, which may be distributed, the backup service will generate an identifier for each file it backs up. This identifier is obtained by applying SHA256, a cryptographic hash function, to some *bit string*. Each implementation can choose the *bit string* used to generate a file identifier, as long as that choice generates file identifiers that are unique with very high probability, i.e. that bit string should be

unique. Furthermore, because the backup service is not aware of versions of a file, the bit string used to generate a file identifier should include data and or metadata that ensures that a modified file has a different fileId. As a suggestion you can combine the file metadata (file name, date modified, owner ...) and/or file data to generate that bit string.

The backup service splits each file in chunks and then backs up each chunk independently, rather than creating multiple files that are a copy of the file to backup. Each chunk is identified by the pair (fileId, chunkNo). The size of each chunk is 64KByte (where K stands for 1000). The size of the last chunk is always shorter than that size. **If the file size is a multiple of the chunk size, the last chunk has size 0.** A peer need not store all chunks of a file, or even any chunk of a file. The recovery of each chunk is also performed independently of the recovery of other chunks of a file.

In order to tolerate the unavailability of peers, the service backs up each chunk with a given degree of replication, i.e. on a given number of peers. The desired replication degree of a chunk depends on the file to which it belongs, and all chunks of a given file have the same desired replication degree. However, at any time instant, the actual replication degree of a chunk may be different from the one that is desired.

In addition to the basic functionality for backing up and recovering a file, the backup service must provide the functionality for reclaiming disk space on peers. **First**, as a requirement of the service, each peer retains total control on the use of its local disk space. If a server's administrator decides to reduce the amount of local disk space used by the backup service, it may have to free disk space used for backing up chunks. This will decrease the replication degree of the chunk, which may drop below the desired value. In that case, the service will try to create new copies of the chunk so as to keep the desired replication degree. **Second**, a file may be deleted. In this case, the backup service should delete all the chunks of that file. Actually, deletion of the chunks of a file, may happen not only when the file is deleted on its file system, but also when it is modified, because, for the backup system, it will be a different file. **Food for thought:** Is this a problem if we wish to keep the backups of multiple versions of a file?

As described, the backup service knows only about chunks of the backed up files, which are identified by the fileId. It knows nothing about the file systems where they are kept. Of course to be of practical use, the mapping between the fileId kept by the backup system and the name of that file (and possibly its file system) needs to survive a failure of the original file system. This problem can be solved in different ways, but you are not required to do it for this project. For this project, and to keep it feasible for all of you, we will assume that this mapping is never lost.

2.2 Service Interface

This year, the service interface will be specified, to facilitate the demoing of your project. It will be based on a command line interface (CLI) application, which will then have to communicate with the local servers. Essentially, it will provide the functionality required to test the protocol. We will publish further details in about 2 weeks.

3. Protocol

In this section we describe the protocol that is executed by the peers.

The protocol used by the backup service comprises several smaller subprotocols, which are used for specific tasks, and that can be run concurrently:

1. chunk backup
2. chunk restore
3. file deletion
4. space reclaiming

Many of these subprotocols are initiated by a peer. To distinguish it from the other peers, in the description of these subprotocols, we call that peer the **initiator-peer**. The other peers are called peers only. The role of initiator-peer can be played by any peer, depending on the particular instance of the subprotocol.

All subprotocols use a **multicast channel, the control channel (MC)**, that is used for control messages. All peers must subscribe the MC channel. Some subprotocols use also one of two multicast data channels, MDB and MDR, which are used to backup and restore file chunk data, respectively.

Note The IP multicast addresses of these channels should be configurable via 6 command line arguments of the server program, in the following order MC, MDB, MDR, with the IP multicast address of each channel before the respective port number. These arguments must follow immediately the first command line argument, which is the **server id**.

3.1 Message Format and Field Encoding

In this subsection we define a generic format for all messages. Below, in the subprotocol description, we specify the format of each message by specifying the fields that must be present. When they are used in a message, they must be encoded as described herein.

The generic message is composed by two parts: a header and the body. The header contains essentially control information, whereas the body is used for the data and is used in only some messages.

Header

The header consists of a sequence of ASCII lines, sequences of ASCII codes **terminated with the sequence '0xD' '0xA'**, which we denote <CRLF> because these are the ASCII codes of the CR and LF chars respectively. Each header line is a sequence of fields, sequences of ASCII codes separated by spaces, the ASCII char ' '. **Note that:**

1. there may be more than one space between fields;
2. there may be zero or more spaces after the last field in a line;
3. the header always terminates with an empty header line. I.e. the <CRLF> of the last header line is followed **immediately by another <CRLF>, without any character in between.**

In the version described herein, the header has only the following non-empty single line:

```
<MessageType> <Version> <SenderId> <FileId> <ChunkNo> <ReplicationDeg> <CRLF>
```

Some of these fields may not be used by some messages, but all fields that appear in a message must appear in the relative order specified above.

Next we describe the meaning of each field and its format.

<MessageType>

This is the type of the message. Each subprotocol specifies its own message types. This field determines the format of the message and what actions its receivers should perform. This is encoded as a variable length sequence of ASCII characters.

<Version>

This is the version of the protocol. It is a three ASCII char sequence with the format <n> ' ' <m>, where <n> and <m> are the ASCII codes of digits. For example, version 1.0, the one specified in this document, should be encoded as the char sequence '1' ' ' '0'.

<SenderId>

This is the id of the server that has sent the message. This field is useful in many subprotocols. This is encoded as a variable length sequence of ASCII digits.

<FileId>

This is the file identifier for the backup service. As stated above, it is supposed to be obtained by using the SHA256 cryptographic hash function. As its name indicates its length is 256 bit, i.e. 32 bytes, and should be encoded as a 64 ASCII character sequence. The encoding is as follows: each byte of the hash value is encoded by the two ASCII characters corresponding to the hexadecimal representation of that byte. E.g., a byte with value 0xB2 should be represented by the two char sequence 'B' '2' (or

'b' '2', it does not matter). The entire hash is represented in big-endian order, i.e. from the MSB (byte 31) to the LSB (byte 0).

<ChunkNo>

This field together with the FileId specifies a chunk in the file. The chunk numbers are integers and should be assigned sequentially starting at 0. It is encoded as a sequence of ASCII characters corresponding to the decimal representation of that number, with the most significant digit first. The length of this field is variable, but should not be larger than 6 chars. Therefore, each file can have at most one million chunks. Given that each chunk is 64 KByte, this limits the size of the files to backup to 64 GByte.

<ReplicationDeg>

This field contains the desired replication degree of the chunk. This is a digit, thus allowing a replication degree of up to 9. It takes one byte, which is the ASCII code of that digit.

Body

When present, the body contains the data of a file chunk. The length of the body is variable. As stated above, if it is smaller than the maximum chunk size, 64KByte, it is the last chunk in a file. The protocol does not interpret the contents of the Body. For the protocol its value is just a byte sequence.

3.2 Chunk backup subprotocol

To backup a chunk, the initiator-peer sends to the **MDB multicast data channel** a message whose body is the contents of that chunk. This message includes also the sender and the chunk ids and the desired replication degree:

```
PUTCHUNK <Version> <SenderId> <FileId> <ChunkNo> <ReplicationDeg> <CRLF><CRLF><Body>
```

A peer that stores the chunk upon receiving the PUTCHUNK message, should reply by sending **on the multicast control channel (MC)** a confirmation message with the following format:

```
STORED <Version> <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>
```

after a random delay uniformly distributed between 0 and 400 ms. **Food for thought:** Why use a random delay?

IMP: A peer must never store the chunks of its own files.

This message is used to ensure that the chunk is backed up with the desired replication degree as follows. The initiator-peer collects the confirmation messages during a time interval of *one second*. If the number of confirmation messages it received up to the end of that interval is lower than the desired replication degree, it retransmits the backup message **on the MDB channel**, and doubles the time interval for receiving confirmation messages. This procedure is repeated up to a maximum number of five times, *i.e. the initiator will send at most 5 PUTCHUNK messages per chunk*.

Hint: Because UDP is not reliable, a peer that has stored a chunk must reply with a STORED message to every PUTCHUNK message it receives. Therefore, the initiator-peer needs to keep track of which peers have responded.

A peer should also count the number of confirmation messages for each of the chunks it has stored and keep that count in non-volatile memory. This information is used if the peer runs out of disk space: in that event, the peer will try to free some space by evicting chunks whose actual replication degree is higher than the desired replication degree.

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?

3.3 Chunk restore protocol

This protocol uses the same multicast control channel (MC) as the backup protocol, but uses a different multicast channel for data, the multicast data recovery channel (MDR).

To recover a chunk, the initiator-peer shall send a message with the following format to the MC:

```
GETCHUNK <Version> <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>
```

Upon receiving this message, a peer that has a copy of the specified chunk shall send it in the body of a CHUNK message via the MDR channel:

```
CHUNK <Version> <SenderId> <FileId> <ChunkNo> <CRLF><CRLF><Body>
```

To avoid flooding the host with CHUNK messages, each peer shall wait for a random time uniformly distributed between 0 and 400 ms, before sending the CHUNK message. If it receives a CHUNK message before that time expires, it will not send the CHUNK message.

Enhancement: If chunks are large, this protocol may not be desirable: only one peer needs to receive the chunk, but we are using a multicast channel for that. Can you think of a change to the protocol that would eliminate this problem, and yet interoperate with non-initiator peers that implement the protocol described in this section?

3.4 File deletion subprotocol

When a file is deleted from its home file system, its chunks should also be deleted from the backup service. In order to support this, the protocol provides the following message, that should be sent on the MC:

```
DELETE <Version> <SenderId> <FileId> <CRLF><CRLF>
```

Upon receiving this message, a peer should remove from its backing store all chunks belonging to the specified file.

This message does not elicit any response message. An implementation, may send this message as many times as it is deemed necessary to ensure that all space used by chunks of the deleted file are deleted in spite of the loss of some messages.

Enhancement: If a peer that backs up some chunks of the file is not running at the time the initiator peer sends a DELETE message for that file, the space used by these chunks will never be reclaimed. Can you think of a change to the protocol, possibly including additional messages, that would allow to reclaim storage space even in that event?

3.5 Space reclaiming subprotocol

The algorithm for managing the disk space reserved for the backup service is not specified. Each implementation can use its own. However, when a peer deletes a copy of a chunk it has backed up, it shall send to the MC channel the following message:

REMOVED <Version> <SenderId> <FileId> <ChunkNo> <CRLF><CRLF>

Upon receiving this message, a peer that has a local copy of the chunk shall update its local count of this chunk. If this count drops below the desired replication degree of that chunk, it shall initiate the chunk backup subprotocol after a random delay uniformly distributed between 0 and 400 ms. If during this delay, a peer receives a PUTCHUNK message for the same file chunk, it should back off and restrain from starting yet another backup subprotocol for that file chunk.

Food for thought: The loss of REMOVED messages may lead to an overestimation of the number of copies of a file chunk, and consequently its actual replication degree may be lower than the desired replication degree. One way to try to prevent this would be to add a response message. Can you think of other alternatives? What are the pros and cons?

Enhancement: If the peer that initiates the chunk backup subprotocol fails before finishing it, the replication degree of the file chunk may be lower than that desired. Can you think of a change to the protocol, compatible with the chunk backup subprotocol, that could tolerate this fault in an efficient way? Try to come up with a solution that works in both cases of execution of the chunk backup subprotocol, i.e. both when a chunk is being backed up for the first time and when a copy of the chunk is deleted.

3.6 Protocol Enhancements

If you choose to enhance any of the subprotocols described above, or to create new subprotocols to add some features, you must be careful to ensure interoperability with the subprotocols defined in this document.

If possible, you should avoid changing or adding any message. If you find that that is unavoidable, you should adhere to the following rules:

1. The header of each message shall be a sequence of lines, such that it does not break the general format rules used in the header definition:
 1. The last header line is always an empty line, i.e. the <CRLF> ASCII character sequence
 2. Each header line terminates with the <CRLF> ASCII character sequence
 3. Fields in a header line are separated by the space ASCII char
2. If you have to change messages defined herein, do not change the respective header line, instead add new header lines.
3. Any message either new or modified must use a version different from '1' . '0', the version of the messages defined in this specification.

4. Final Remarks

4.1 What and how to submit?

You should submit all the source code files via the SVN repository of the Redmine project that you must create for SDIS. Your project **id** shall be named **sdis1516-t<n>g<p><q>**, where <n> is the number of your section (turma) and <p><q> are two digits with the number of your group, e.g. sdis1516-t3g06. In addition to the source code files, you should submit a plain ASCII file named README with instructions for compiling and running your application.

Furthermore, if you have implemented any enhancement to the protocol specified in this document, you should submit also a report, a PDF file named protocol.pdf also via the SVN repository of your Redmine project. For each enhancement you have implemented, the protocol should include its specification and explain its rational. This should not be longer than one page per enhancement.

4.2 Grading criteria

We will test your implementation also with that of other groups, and possibly our own, to ensure that the protocols are implemented in an interoperable way. Therefore, we suggest that as you develop your project you test it with that of other groups.

A proficient concurrent implementation of the subprotocols without enhancements described in this document is worth a project grade of 80%. To achieve concurrency, you can use either multiple threads or Java NIO.

By implementing each of the 4 suggested enhancements, you will get an additional 5%. Please note that your enhanced subprotocols must still interoperate with non-enhanced subprotocols, therefore you must be very careful in the design of these enhancements.

You will have the opportunity to evolve this service in the second project. (The idea is to "move" the service from a LAN to the Internet.) Actually, if you wish to get a very high grade in this course (say 18 or above), it is likely easier if you choose this path than if you choose the implementation of a web service. This is because this path is likely to be more difficult and the project difficulty determines the maximum grade you can get.

Annex A: Interoperability

Your implementation must interoperate with that of other students: we will test this. That is your service must be able to execute the four tasks mentioned in the previous point using only the messages in [Section 3](#). You should consider both the behavior as an initiator peer and otherwise. (Of course, if there are not enough remote peers implementing your enhancement, the corresponding enhanced protocol may fail.)

To avoid interference between the execution of non-interoperable implementations, a service must drop messages that it does not understand. This is a general rule of every protocol.

If you implement any enhancement, it is probably easier to ensure interoperability if you provide two operating modes: one with the enhancements and the other without. The latter shall use only the messages defined in this document.

You should carry out interoperability tests as soon as possible. The chunk backup subprotocol is a good starting point. If your implementation is able to execute this subprotocol with implementations from other colleagues, then it should not be hard to get the implementation of the other subprotocols to interoperate.

This can be done rather early in your development and without much effort. You can provide a `.class` file that prints the messages to the standard output. This way, other groups can use that file, to generate messages to test their own code, in particular the code that parses the messages.