Distributed Systems

Distributed Backup Service

Ângela Cardoso and Diogo Pereira

April 3, 2016

1 Chunk Backup Subprotocol

1.1 Objective

The basic protocol implemented in version 1.0 dictates that as long as space is available, each peer should backup all chunks. When a peer's space is depleted, then chunks with replication degrees above their threshold should be deleted to make room for new chunks. This leads to a lot of chunks being stored and deleted as well as a lot of multicast traffic from `REMOVED` messages. The objective of this enhancement is to develop an alternative chunk backup subprotocol, that guarantees that minimum replication degrees are met, without the disadvantages mentioned above. Also, the improved subprotocol should operate well with peers that do not have the enhancement implemented.

1.2 Solution

Our solution to this problem is a simple one. Upon receiving a `PUTCHUNK` message, instead of immediately storing the chunk and replying after a random delay between 0 and 400ms, the peer waits for a random interval between 0 and 800ms. During that interval, `STORED` replies that are received from other peers regarding that same chunk are counted. If at the end of the waiting period the minimum replication degree was met by other peers, the peer does not save the chunk. Otherwise, the chunk is saved and a `STORED` message is sent immediately. If after the chunk is saved another `PUTCHUNK` message arrives for the same chunk, the peer replies with a `STORED` message after a random wait between 0 and 200ms. We use a smaller wait period for replying to `PUTCHUNK` messages, so that other peers do not store the chunk unnecessarily. In a similar fashion, we allow for a possibly larger wait period before saving the chunk, in order to give plenty of opportunity for other peers to do it instead.

The implementation itself is mostly done inside the `PutchunkMessage` class, where we have the method `run_v_1_1` to handle `PUTCHUNK` messages according to this improved version of the protocol. To make use of this enhancement versions 1.1 or 2.0 should be used.

1.3 Limitations

Although this solution usually produces replication degrees equal to the minimum replication degree, some chunks still have higher replication degrees. This happens because peers may store the same chunk with very small time differences between them, so that when the replication degree is updated, it is already greater than necessary. However, in our tests, the minimum replication degree was only exceeded by one or two.

Even though this protocol works well with other peers that have only implemented the basic one, all peers that have the basic protocol will immediately try to store the chunk and then wait to communicate it, so other peers will take some time to notice and may unnecessarily store the chunk. This means that our solution will be much less effective if the number of peers without it increases.

2 Chunk Restore Subprotocol

2.1 Objective

In the original implemented protocol, all messages are sent using multicast channels. This means in particular that chunk recovery is done through multicast channels, leading to all peers receiving possibly large chunks when only one (the peer that sent the `GETCHUNK` message) needs to receive it. Of course, these multicast `CHUNK` messages are also used by other peers to determine that they no longer need to send the chunk themselves. The objective of this enhancement is to avoid sending chunks over multicast whilst maintaining interoperability with peers that use the basic protocol.

2.2 Solution

Since all peers are listening on a unicast channel for messages from the `TestApp`, we decided to use that same channel for this improvement, because it uses the peer `ID` as port, so nothing new has to be done in terms of messages. This way, if a peer using the enhanced version of this subprotocol receives a `GETCHUNK` message from an initiator that is also using the enhanced version, that peer will send two `CHUNK` messages. The first message will be sent to the private unicast channel of the initiator and it contains the actual chunk. The second message will be sent on the multicast channel, but its body will be empty. This way, other peers know that they should back off from sending `CHUNK` messages, without the hassle of receiving large unnecessary chunks themselves. On its side, the initiator peer is prepared to receive `CHUNK` messages on both the multicast and unicast channel. This way, if the real chunk is sent through multicast (by some peer with the basic implementation), it will be saved.

Part of the implementation is done inside the `Service` class, because that is where the peer listens to its unicast channel. In this class we have the `processRequest_v_1_2` method to channel all `CHUNK` messages received in the unicast channel through the usual `CHUNK` message protocol. The other part of the implementation is done inside the `GetchunkMessage` protocol, where the method `run_v_1_2` will guarantee that the appropriate `CHUNK` messages are sent on both channels. To make use of this enhancement versions 1.2 or 2.0 should be used.

2.3 Limitations

The only real limitations are due to the interoperability requirements. In fact, peers that do not have the improved version will still send all `CHUNK` messages through multicast. Also, initiator peers that use the basic version will force other peers to use multicast for `CHUNK` messages, even if all others have the improved version.

3 File Deletion Subprotocol

3.1 Objective

In the basic protocol, if a file is deleted while a peer is not running and that peer has backed up chunks of the file, then, since the peer will miss the `DELETE` messages, it will never be able to recover the space used by these useless chunks. The objective is to implement an addition to the protocol that allows for these chunks to be deleted when the peer comes back online. Also, even though it was not requested, we decided to include in this enhancement a solution to the problem of outdated replication degrees for chunks of a 'resurrected' peer.

3.2 Solution

Although the use of new messages was allowed for this implementation, we decided against it, in order to maintain a high degree of compatibility between different protocol versions.

When a peer comes back online after a period of absence, it starts by sending `GETCHUNK` messages for all its chunks. Each chunk for which a `CHUNK` message is then received is marked as not deleted, because clearly another peer has the same chunk. Then chunks that were not marked are deleted, because no other peer has the same chunk, therefore its file must have been deleted. The exception to this rule is for chunks whose replication degree is 1, because for those, we are not expecting to receive any `CHUNK` messages and that does not mean the file has been deleted. After the initial `GETCHUNK`, the peer sends `PUTCHUNK` messages for all the remaining chunks and uses the `STORED` replies to update its replication degree for each chunk.

Since this is done upon peer startup it is implemented in the class `Peer` itself, where we have the methods `checkChunks_v_1_3` and `initialGetchunk` to initiate the desired protocols. There is also a small change to the `ChunkMessage` class so that `CHUNK` messages received due to this initial `GETCHUNK` are properly handled. To make use of this enhancement versions 1.3 or 2.0 should be used.

3.3 Limitations

Due to our choice of not using new messages for this enhancement, the solution we devised has some real limitations. We have no way of determining if a chunk of replication degree 1 belongs to a file that has been deleted from the system. Also, if at the time of this initial `GETCHUNK` solution some other peers are offline, chunks that these offline peers have stored may be inadvertently mistaken for deleted chunks, since these offline peers will not reply.

In regards to our solution for the outdated replication degrees, it can only be used for the replication degrees of stored chunks, not for the files that the peer has sent for backup. For that, the peer would have to redo the whole backup of its files and we did not implement that. Also, as above, if other peers are offline while the replication degrees are being updated, the final count will very likely be below the real count, since their chunks will not be accounted for.

4 Space Reclaiming Subprotocol

4.1 Objective

In the original protocol, if the peer that initiates the `PUTCHUNK` protocol fails before finishing it, the replication degree may be below desired and nothing will be done about it. This happens both when a new file is being backed up, and when the replication degree of a chunk decreases bellow the threshold and a new `PUTCHUNK` is initiated for that chunk. The objective is to make sure that `PUTCHUNK` protocols are carried out all the way even if the initiator fails.

4.2 Solution

Our solution for this is to get all peers to count not only the `STORED` messages, but also the `PUTCHUNK` messages for each chunk. If, after a long enough period, the number of `PUTCHUNK` messages is less than 5 and the replication degree is not enough, a peer that stored the chunk and notices this will prepare itself to initiate the `PUTCHUNK` protocol. In order to guarantee that a single peer does this, we used a random delay between 0 and 400ms. If after this time no peer has reinitiated the `PUTCHUNK` protocol, the peer does it. We also ensure that the `PUTCHUNK` messages counter is reset after each protocol, so that any other `PUTCHUNK` protocol for that same chunk will also be reinitiated in case of need. Thus, our solution works for both uses of the `PUTCHUNK` protocol.

The implementation of this enhancement is done in the class `PutchunkMessage`, where we have the method `run_v_1_4` that is invoked for this version. This method then invokes the method `checkPutchunks` as a separate thread, to do the necessary waiting and checking. To make use of this enhancement versions 1.4 or 2.0 should be used.

4.3 Limitations

Our implementation only works if at least one peer has stored the chunk before the `PUTCHUNK` initiator fails, of course. Also, if a new file is being backed up and the peer dies halfway through it, the file may not be fully backed up, even if the chunks that make it through are backed up to the desired replication degree.