Distributed Backup Service: Enhancements

All enhancements described in this document were implementing using a protocol version “2.3”. Also, all enhancements only become active if the “enhanced mode” flag is active when the peer is started (we call peers that are using the “enhanced mode” as “enhanced peers”).

Chunk backup subprotocol

As suggested in the project specification, the proposed chunk backup can deplete space quite rapidly and cause too much activity in nodes whose space is already full. To avoid those problems and still interoperate with “basic” peers, we added a slight modification to the functionality of the non-initiator peers of this protocol.

In the standard version of the protocol, peers who received a “PUTCHUNK” command would store the given chunk and then reply with “STORED” while keeping count of how many other peers also stored the chunk. “Enhanced” peers on the other hand, also wait a random delay of 0 to 400ms before responding with “STORED”, but if meanwhile the count of peers who store the chunk becomes greater or equal than that chunk’s desired replication degree, the peer discards the chunk and does not reply with “STORED”. The peer will still keep track of how many peers stored the chunk even if it does not store that chunk.

Using this technique, our system will still guarantee that a chunk’s desired replication degree is achieved, avoiding rapid space depletion and a lot of unnecessary activity in the peers.

File deletion subprotocol

A problem of the proposed chunk deletion subprotocol is that if a peer who has a chunk that belongs to another peer isn’t running when that peer deletes that chunk, it still keeps the chunk stored in disk and that space will never be cleared because the owner of the chunk will never start a “DELETE” subprotocol again.

To solve this issue, we added a couple of messages to the protocol version we created, and those messages are:

“EXISTS <version> <sender_id> <file_id><CRLF><CRLF>”

“WASDELETED <version> <sender_id> <file_id><CRLF><CRLF>”

Every peers keeps track of which files where deleted in non-volatile memory (metadata). This way, every time a file (either his or a peer’s file) is deleted, the peer stores that file’s id.

Since the problem happens when a peer is not active, we needed a solution that became active when the peer is initiated. Therefore, every time a peer is turned on, it gathers the identifiers of all files it has backed up for other peers, and for each of those identifiers it sends a “EXISTS” message through the control channel to query other peers of wether or not that file still exists in the system. Then, the peers wait for a couple of seconds to see what responses it gets. If a peer receives a “EXISTS” and it knows the file hasn’t been deleted, it ignores the message. If the peer knows that file hash has already been deleted, it waits a random delay of 0 to 400 ms to reply with a “WASDELETED” message through the control channel. If, meanwhile, the peer receives another “WASDELETED” message with the same “file_id”, it backs off from sending the message. Every time the initiator peer receives a “WASDELETED” message it deletes that file’s information and all chunks of that file that he has backed up.

This implementation solves the problem suggested at the project specification, avoiding situations where peers would keep chunks of files that no longer exist.

File restore subprotocol

The proposed file restore subprotocol can be problematic, because the message “CHUNK” is sent via a multicast channel (the restore channel) when only one peer will be interested in receiving it. Therefore, we made a slight change to the actions of “enhanced peers” to avoiding flooding the restore channel with unnecessarily big datagram packets.

To achieve this, we added a datagram socket to each “enhanced” peer, to serve as a private data channel. When a peer receives a “GETCHUNK” message, it checks wether or not the sending peer is using the “enhanced” version of the protocol (2.3). If it is not using this version, the peer executes the “standard” restore protocol. In the other case, the peer immediately sends the “CHUNK” message directly to the sender’s private data channel. Then, to avoid “flooding” the restore channel, the peer sends the same “CHUNK” message but with an empty body to that channel, so that non-enhanced peers don’t send the entire chunk through this channel.

This improvement can solve the problem of flooding a multicast channel without the need for it, while also ensuring compatibility with non-enhanced peers.

Reclaim subprotocol

The specification of the project suggests that if a backup subprotocol initiator fails before finishing, the chunks he is backing up might not achieve the desired replication degree.

To solve this issue, we made a slight change to the non-initiator backup peers.

When a backup subprotocol “enhanced” peer has waited for the initiator to finish and notices that the chunk has not yet achieved the desired replication degree, it waits for a random of 10 to 10.4 seconds to see if some other peer restarts the backup of this chunk. If no other peer restarts the protocol, the peer tries to backup the chunk again so that it might achieve the desired replication degree again, using the standard backup subprotocol (ensuring the replication degree even with “non-enhanced” peers).

This implementation solves issues for cases when the peers that initiate the backup subprotocol fail while doing this, assuring the replication degree when there are enough peers in the system to achieve the desired replication degree. However, this solution will cause a loop when there aren't enough peers in the system, but since we can assume the environment to be friendly, we also assume that the user does not request for a replication degree larger than the number of peers in the system.