Backup SubProtocol Enhancement (1st enhancement)

Implemented Solution

The implementation of this enhancement consist in modifying the behavior of the previous ChunkBackupReceiver method *run()* which was responsible for processing a *putchunk* and subsequent *STORED* messages.

In the new implementation, upon receiving a *putchunk*, if the peer doesn't own the file, it will not try store the chunk immediately. Instead it will wait a fixed delay and a random delay. After these delays, it verifies if there have already been received enough *STORED* messages to satisfy the replication degree. If enough messages have been received, the peer refrains from storing the chunk. If instead the peer already owns the chunk it will only wait the random delay to send the *STORED* message confirmation.

We present this behavior below in a more schematic form.

1. If a peer already has a copy of the chunk received
    a. Wait random delay
    b. Reply with *stored*
2. If a peer does not own a copy of the chunk
    a. Wait fixed delay
    b. Wait random delay
    c. Count the number of *stored* messages related to the chunk received so far
    d. If the number of *stored* messages is lesser than the replication degree
        i. Store chunk

When a reply is sent the peer will wait for the rest of the maximum possible delay (which the random delay didn't use), plus a little more just for precaution, and once this wait finishes the number of stores will be counted again. This way, even when the peer verifies that replication degree was achieved, will take into account possible answers that could be sent later in that given interval.

Backup SubProtocol Enhancement Efficiency

The protocol described can work with peers that use the default protocol however the efficiency it provides is not meaningful unless more peers use the same enhancement. The more peers use this enhanced version (and less use the default) the more efficient the process becomes.

Delete SubProtocol Enhancement (3rd enhancement)

Implemented Solution

The implemented solution consists in sending *GETCHUNK* of certain chunks after recovering from a crash/failure. The *GETCHUNK* is only sent in two situations and if no answer to this GETCHUNK is received we send a second one just for precaution. If no answer is received again we assume the file associated with the *GETCHUNK* sent was deleted.

The first case where a *GETCHUNK* is sent is when there is at least one chunk of a file missing in our peer chunk collection. If our peer owns the chunks number 1, 2 and 4 for example, then either someone else has the chunk number 3 or the file can't be recovered, so it is safe to assume that it was deleted. The only situation where this procedure might fail is when some peer that owns the chunk we ask for is offline. To implement this, the peer checks its data to find a missing chunk for each file, there may be missing more than one but we only select the first found, and then tries to confirm the existence of the file's chunk in the peer group.

The second case where a *GETCHUNK* is sent is when there is some chunk of a file with desired replication degree above one. Similarly to the first case we only select a chunk per file if any satisfy the mentioned condition. In this case is a bit riskier to assume the occurrence of a file deletion when a confirmation is not received, our peer assumes the rest of the peers were kept online and tried to keep the desired replication degree of the file but even if this was the case there is the possibility, when most peers' storage is already full, of the file being currently owned only by our peer. Selecting and confirming additional chunks of a file wouldn't solve the problem completely and would make the process of identification of deleted file longer but would decrease the occurrence of these mistakes so for testing and simplicity purposes we decided to use only one chunk per file.

Delete SubProtocol Enhancement Efficiency

The strong point of this implementation is that it works with peers independently of their protocol and the weak is that is that is not the safer available. A safer alternative could be achieved, for example, by synching the peers' clocks and keeping track of the deletes would be possible to query other peers about deleted files in case of a crash but this alternative solution would be more complex, would require an additional protocol and would need that at least two peers in the network used the enhanced protocol.

Backup Enhancement (4th Enhancement)

Motivations

Identifying and dealing with a backup protocol failure directly can be "tricky" and somewhat the "wrong question to ask". Before exposing our solution we would like the motivations behind it.

Regarding the "trickiness", we could try to identify the failure by the replication degree achieved but there is the chance of the number of peers not being high enough to achieve it or of the group already having all their storage space used. Alternatively we could use special messages but that would not have any efficiency unless more than one peer used the enhanced protocol. Preventing and/or trying to correct possible occurrences of the problem seem to be better approaches than dealing with it directly. Our solution uses the "correct possible occurrences" approach.

Regarding pertinence of the project's script there are other situations that also leave the effective replication degree below the desired replication degree such as the ones referred in the previous paragraph. These situations are no less important than the failure of a backup, after some time the state of the group might change and these situations could be corrected. Our solution prioritizes precarious chunks independently of the reasons behind their current replication degree.

At least our solution also turns one weak point in our implementation into a possible strong one. Because we do not take into account stores received outside of "putchunt iteration" (receiving a putchunk resets the replication degree count), we do not keep track of the the specific peers that own the chunk or process STORED messages outside of a putchunk iteration we can easily estimate a replication degree that is lesser than the actual one. Our solution helps correcting this estimate and allows our peer to cope with possible dead peers, which could be a problem if we kept track of the peers that own certain chunks.

Implemented Solution

Our solution is consists in periodically, using a random time interval (contained within a minimum and maximum interval), start the backup subprotocol for one owned chunk. The selection of this chunk is somewhat random. We prioritize precarious chunks – that have higher *desired degree of replication – degree of replication* value – by giving them a higher chance to be selected but any owned chunk can still be selected. This way our peer can avoid always selecting the same chunk, which could happen, for example, if a certain chunk couldn't achieve its desired replication degree due to the number of available peers.

The chance of selection of a chunk is given by its index in a list (in reality the used structure in code is not a list but for explanations purposes let's assume it is) which contain all chunks sorted by their precarity in ascending order (when the precarity is the same the older files have lower indexes). The following formula shows the probability of a chunk being select:

$$\frac{2 * index}{|number\ of\ chunks| * (1 + |number\ of\ chunks|)}$$