SDIS 2019/2020

Project 1: Distributed Backup System

Manuel Monge dos Santos Pereira Coutinho - up201704211
David Luís Dias da Silva- up201705373

This report contains implementation details of the first SDIS project “Distributed Backup System”. The first section explains our strategy and rationale behind the enhancements for the Backup, Restore and Delete protocols. In the second section, we describe the design of the application from a concurrency perspective.

1. Enhancements

1.1 Backup sub-protocol enhancement

For this enhancement, we used a strategy inspired by the design of the restore sub-protocol. In particular, with the mechanism used by peers regarding the CHUNK messages, in order to avoid multiple peers sending the same chunk via de MDR channel.

When a peer receives a *PUTCHUNK* message from an initiator (I), before sending the corresponding *STORED* message (which is sent after a delay between 0 and 400 ms according to the specification), it consults the chunk storage to check if the received chunk’s desired replication degree has been reached during the waiting time. If so, the peer may refrain from sending the *STORED* message and saving the received chunk.

By using this strategy, a peer only stores a chunk if it’s needed to achieve the desired replication degree, which provides an improvement over the original protocol because it reduces the amount of space that is used. If the system only contains peers using this enhancement, we can assure that at a given time, a chunk’s actual replication degree never surpasses the desired one.

Since this technique does not use any neither additional messages nor different communication protocols between peers, it’s usage does not prevent the peer from participating in a system with peers that don’t use the enhancement.

During development, we found that in some rare cases, the receiving peer might have determined that it should store the chunk. However, a STORE message from another peer might already have been “in transit”. This would cause the replication degree of that chunk to surpass the minimum replication degree. To work around this, we also gave the peer the ability to remove the “extra” chunk if it’s “peerID” among the N lowest who store the chunk, where N is the amount by which the replication degree is surpassed. This removal is done by sending a *REMOVE* message as described in the delete sub-protocol, which still preserves the interoperability.

There still is a slight chance that the Replication Degree is surpassed, i.e. in cases where *STORED* messages are received before the *PUTCHUNK* related to that chunk, being the only way to avoid it to start saving all the STORED messages, as opposed to only the ones for which we have already received a *PUTCHUNK* order. We decided not to do so, because from our interpretation of the specification followed that a peer should only be interested in messages about chunks that he is currently storing/backing up (excluding *PUTCHUNK* messages).

1.2 Restore sub-protocol enhancement

In order to reduce the amount of information that is transported in the MDR multicast channel through *CHUNK* messages due to the restore protocol, we devised a strategy that uses the TCP-protocol.

The peer that receives the *GETCHUNK* message and that is allowed to respond with the *CHUNK* message, after waiting for the specified delay, sends it as usual. However, instead of sending the chunk data in the message's body, it will send the needed information for a TCP connection to be established between itself and the initiator peer (socket's port number and the peer's IP address*). In this connection, the replicator peer acts as the server and creates the socket to which the initiator peer (client) must connect in order to receive the needed chunk. We also added timeouts to avoid situations where any one of the sides is stuck waiting for the connection to be established. By sending the data portion through a TCP channel we avoid sending the potentially large quantities of data through the multicast channel since only one peer (the initiator) has any interest in receiving it. By using TCP there is also benefits such as failure recovery and error handling.

Since a TCP connection requires active participation by two ends, for this protocol to take effect, both peers need to have the version that supports the enhancement. When a peer receives is preparing to respond to a *GETCHUNK* message he checks if it's version is greater or equal than his own. If so, he will respond with the enhanced message, if not he will use the regular protocol. Similarly, the initiator peer, upon receiving the CHUNK messages, will check if the sending peer's version supports the enhancement, which will determine how it will interpret the messages data segment, i.e. if the message is from the enhanced version he will use the contents of the data to open the non-initiators socket, else he will interpret the data segment as the chunk itself.

* The IP address of the socket can be obtained through the origin IP address of the UDP packet that contains the message

1.3 Delete sub-protocol enhancement

The goal of the delete enhancement is to remove the possibility of having peers storing chunks that have already been deleted in the system, due to being offline during the time in which the *DELETE* message was sent.

Upon sending a *DELETE* message, the initiator peer will store in a ledger (concurrent hash-map) the information about the deleted chunks and which peers currently replicate them. By implementing the enhancement, peers will now respond with a *DELCHUNK* message on the control multicast channel when they delete a chunk. The format of this message is similar to the *STORED*, with the only difference being the *MessageType* field. In response to this request, the initiator will remove that peer's ID from the entry in the corresponding chunk in the ledger. The information about the replicated file is only removed when all chunks have been deleted by all peers.

To solve the aforementioned problem of having a peer store a chunk indeterminately after it's already been deleted, if it was offline during the remove sub-protocol, we implemented an additional message (*GREETING*) on all peers that they send on the control multicast channel upon startup. The format of this initial message is the simplest of them all stripping it from its unnecessary fields (like the FileID present in every message). When a peer receives a greeting from another peer it shall check if it's deleted chunk ledger contains any reference to that ID. If so, it will send a *DELETE* message for that file, as defined by the standard protocol.

An issue that would continue to arise, would be the case in which the initiator peer is offline when the *GREETING* message is sent. This would cause the *DELETE* message not to be sent and the chunk to be stored indefinitely. To fix this, when starting up, the initiator peer will start the *DELETE* sub-protocol for each file in the deleted-chunk ledger.

This enhancement needs to be implemented by at least two peers in order to take effect. It also is interoperable with previous versions since, although non-enhanced peers won't send the *DELCHUNK* message, the reference for a deleted file is does not occupy significant space and *DELETE* messages are understood by all versions.

One problem with this implementation would be if the *DELCHUNK* message is lost and it never reaches the initiator peer, because the reference to this replicator would never be erased from the pending delete chunk ledger. One of the ways to avoid this, would be to make this *DELCHUNK* part into a sub-protocol like the backup described in the specification. This way the non-initiator peer would send X amount of times the *DELCHUNK* waiting for a confirmation sent by the Initiator peer and only then it would remove the stored chunk stored.

We opted for our solution not only because this new subprotocol would require a new message (the confirmation), but also would increase the message traffic significantly: for each instance of the *DELETE* sub-protocol we would have, at least, N more messages sent (N being the number of non-initiator peers), while in our implementation, if some *DELCHUNK* is lost, only 3 messages would be sent in case of the initiator or corresponding peer stops running and comes back (bear in mind that we are comparing the best case of the "solution" with worst case of our implementation).

2. Concurrency

During the development of this project, we designed the system to support multiple concurrent protocol instances. To achieve this and maintain a sustainable performance while using threads we used *thread pool*s. Thread pools allow us to sustain the heavy use of threads that we needed by creating a predetermined amount of threads beforehand (*TaskManager* class constructor), which eliminate the computational cost that derives from creating and destroying threads. However, we must note that the performance of thread pools may be limited by the CPU of the machine that runs the peer, as such, a study could be made on the optimal sizes of the *thread pools* in order to fit any particular system. The sizes of the thread pools we used can be controlled through the *MC_THREAD_POOL_SIZE* and *TASK_MANAGER_THREAD_POOL_SIZE* variables on the Protocol class.

From a concurrency perspective, we split our system into three sections: the protocol/tasks section (*subprotocol*, *taks* and *client* packages), the channels section (*channel* package), and the storage section (*storage* package).

The class “TaskManager” contains a scheduled thread pool. Together with this class, we created the concept of Tasks (package *task*) which have unique ID’s and that can be added to the task manager to run immediately (*Task* class) or after a fixed/random delay (classes *RandomDelayedTask* and *FixedDelayedTask*) using the thread pool (functions *addTask* and *addDelayedTask*). The manager stores the tasks that have been submitted for execution in a concurrent hash-map, which allows the system to keep track of tasks that are or will be executed and to cancel them if necessary (functions *protocolInProgress* and *cancelTask)*. This functionality is useful to avoid the same peer engaging in protocols for files which are currently being handled by other protocols. The task manager is used by the Peer class to start the execution of a subprotocol in a thread after a request from the TestApp (e.g. Peer class, method *backup*). This allows the program to execute many subprotocols instances at the same time. We’ve tested this functionality by executing several file backups from the same peer at the same time, while also executing backup,restore and delete protocols from other peers. In this sense, we might consider the Peer class as a dispatcher for requests from the *TestApp*.

The multicast channel classes (*Channel* class and subclasses) each contain a thread pool, which is used to execute threads that handle received messages (*handler* function on *Channel* class). This allows for the processing of several messages received on a channel at the same time. The handlers call functions that process the message given its type (*manage* method, classes *ControlChannel*, *BackupChannel* and *RecoverChannel*), taking the necessary actions. These individual handlers interface with the storage section in order to access information about the chunks/files or elicit the alteration of the storage data and the task manager in order to schedule new messages/protocols or cancel queued ones (i.e. *handlePutChunk*, *handleStored*).

The storage section (storage package) contains classes and data structures that handle the IO operations with the file system and that store the information about the replicated files/chunks (BackupSystem class). This section needed to be severely optimized for concurrent use since it is accessed by all other sections of the system, i.e. at a given time

many protocol threads and/or message handlers may need to check or update the replication status of some file/chunk. To do this we used the Concurrent Hash Map data structure as a basis for storing the needed information due to its inherent capabilities of dealing with concurrent accesses on the hash-map's buckets. However, we still needed to use synchronized blocks and methods in order to avoid race conditions in segments with access/modification sequential operations. An example of this occurs in operations that access the linked lists that store the ID's of the chunk replicators (class *ChunkInfo*), which may lead to "race conditions" in situations where a peers reads information about the list, such as if it contains the ID of a given peer, and updates said list according to the result (method *addReplication,* class *ChunkInfo*), or in situations where the list's need to be iterated (method *removeReplication ,* class *ChunkInfo*).

This section also runs a thread periodically (using a scheduled thread pool) to store the contents of its hash-maps that contain information about the chunk replication status, files that are backed up and/or in process of being deleted, into non-volatile memory avoiding the loss of information in case of Peer failure. The time interval between the storage of this data-structures is regulated by the DISK_BACKUP_INTERVAL_SECONDS in the *Protocol* class. For doing this we also needed to ensure that the ledgers were not being handled while the thread as backing them up. For this we used synchronized calls.

Initially the system used *Thread.sleep* in threads that were running protocol's whose specification implied waiting (i.e. backup sub protocol). This is problematic because it would be occupying one thread of the pool unnecessarily, since the thread would be idle. Instead of waiting, these threads now work by scheduling another task to be run after the waiting period as passed, this way the current thread is freed and can be used by other tasks during this time, i.e. instead of waiting the specified delay before sending retrying the backup sub-protocol for a given chunk, the thread will schedule a sub-protocol to be run after that delay (*BackupSubProtocol* class).

We also explored the possibility of substituting our file I/O operations with an implementation using asynchronous file channels (using *java.nio*), however due to the time constraints and the return that we would obtain doing so, we decided not to use it in our implementation.