Distributed Backup Service

This report describes the enhancements for the Distributed Backup Service, implemented over the standard protocol described in the project's requirements.

Protocol Enhancements

Backup (version 1.1)

In addition to the base protocol implementation (1.0), we implemented an enhanced version of the protocol to allow for diminished activity on the nodes once the backup space is full, as well as restricting the replication of Chunks that already achieved their desired replication degree.

The standard subprotocol, after receiving a PUTCHUNK message, immediately saves the chunk and then waits a random interval (from 0 to 400 ms) to send the STORED message.

Conversely, after receiving a PUTCHUNK message, the enhanced version awaits a random interval (from 0 to 400 ms), listening for STORED messages of the corresponding chunk. In that interval, if the number of STORED messages fulfills the desired replication degree of the chunk, the thread exits; otherwise, the thread proceeds with saving the chunk, and immediately sends a STORED message.

In this way, this enhancement delays the depletion of the backup space, while ensuring the same replication constraints of the standard protocol are met.

Restore (version 1.2)

The problems of the standard restore protocol are based on two facts: using the UDP protocol to transmit data packets allows sporadic loss of chunks; and the fact that we are using a multicast channel to transmits the data chunks, where only one peer needs to receive them.

It was clear we needed to get use an alternative to the multicast channel in order to transfer the chunks, thus we employed a TCP connection, allowing for direct, lossless transfers.

First of all, the enhancement-compatible peer sends a ENH_GETCHUNK message to the multicast channel with an extra field (when comparing to the standard GETCHUNK message). This field transports the ip and port of the initiator peer, allowing the others peers to connect directly with the host. The IP is the ip of initiator peer and the port is (4444 + peerID), ensuring the ability to run several restores in the same network, simultaneously, as the ports are unique to each peer. This protocol only has the restriction of running at most 1 restoration of a peer initiator at a time (but may execute other protocols simultaneously).

- ENH_GETCHUNK <Version> <SenderId> <FileId> <ChunkNo> <IP:Port><CRLF><CRLF>

To handle each connection to the TCP server, we create a thread for each client to handle the chunks received and dispatch them correctly.

Our enhancement can interoperate with other peers because:

- If an enhanced peer receives a ENH_GETCHUNK:
    - The Peer sends a Chunk message to the multicast channel, without the body field, to avoid flooding the host;
    - The Peer sends the chunk via TCP directly to the host.
- If a “normal” peer receive a ENH_GETCHUNK
    - The Peer sends the message to the multicast channel

Once all the chunks are received and collected, these may be merged in order of their chunk numbers, consistently leading to the restored file (unlike the UDP based standard protocol).

Deleted (version 1.3)

Our enhancement ensures every node deletes chunks of deleted files, even if they were down when the DELETE message was sent, as long as any peer that received the DELETE message is up.

In order to achieve this, we implemented four simple responses to messages:

- Every time a DELETE message is received, the fileID is saved;
- Every time a STORED message is received, the name of the sender is saved as a mirror for the corresponding file;
- Every time a DELETED message is received, the name of the sender is removed from the list of mirrors of the corresponding file;
- Every time a PUTCHUNK message is received, if the file was previously deleted, that file is removed from the deletedFiles list (thus ensuring a file can be deleted and backed up multiple times);
- Every time an UP message is received (indicating a given peer is UP), if that peer is a mirror for a deleted file, a DELETE message is resent to the multicast channel.

The additional messages, UP and DELETED, are formatted as follows:

- UP <Version> <SenderID> <CRLF><CRLF>

- DELETED <Version> <SenderID> <FileID> <CRLF><CRLF>

Interoperability

All the implemented enhancements do not interfere with the application's interoperability with peers running the standard protocol.

Concurrency Implementation

We implemented a robust, resilient and efficient concurrent application. To this goal, we use divide the application's workload in small units, designed to be run in separate threads. Furthermore, nowhere in our application do we resort to "busy waiting", and every thread is accounted for, sometimes killing threads that aren't able to complete their purpose (for instance, a restore thread that couldn't receive all chunks).

The Peer class, which implements the RMI service methods (backup, restore, ...), features a MessageDispatcher, and a ScheduledThreadPoolExecutor. The MessageDispatcher is used to dispatch the handling of a message to the appropriate thread. And the executor service is used the schedule delayed tasks (such as sending messages after a random wait time), and to start initiators in seperate threads.

The MessageDispatcher runs on a seperate thread, and uses a BlockingQueue to communicate with the other threads. In this manner, every time a message is received, it's put in the BlockingQueue. And the thread continuously takes a Message from the queue and processes dispatches its handling to new Threads. This way, the Dispatcher thread blocks when no messages are received, yielding the CPU resources to the operating system.

This use of several threads, which use the same information regarding the local file system and the state of the network and its nodes, may lead to race conditions in an unprotected environment. In order to tackle this, we use Concurrent Collections, from the standard 'java.util.concurrent' package, every time a variable may be accessed and modified by several threads running simultaneously.

To further increase efficiency, protocols such as Backup, which issue resource-demanding tasks, with large writes/reads to disk, have their own ThreadPools, to issue Backup threads for each Chunk that needs to be backed up. As each chunk may need up to 15 seconds to complete its backup (the thread must wait for the corresponding STORED messages, and either stop or resend the

PUTCHUNK message up to 5 times), this decoupling of tasks from the main thread ensures a high concurrency and performance.

Tasks that may be issued by multiple threads, and do not correspond to a java Concurrent Collection, are protected with the synchronized keyword, as is the example of saving the Database to disk memory.

In tasks that require waiting a certain delay before being executed, we use a ScheduledExecutorService. This has several advantages: besides decoupling its execution from the main thread, it enables us to easily cancel their execution, using the “cancel()” method. The executor’s “schedule” method returns a Future instance, and this object may be used to cancel the task’s execution, with “futureObj.cancel()”, or to yield CPU resources until the task is completed, “with futureObj.get()”.

To attest our application fulfills the requirements for concurrency, we successfully tested the execution of several protocols simultaneously, such as the backup of a file (through the BackupInitiator) while the same peer stores (backs-up) chunks of other peers, the reclaim of disk space while the same peer executes the restore of a file, among other test examples. These tests were executed with peers in the same computer, as well as with peers in different computers.