

SDIS – Sistemas Distribuidos – 2017/2018

Autor:

Tiago José Sousa Magalhães - 201607931

Contents

[***SUBPROTOCOLS*** 3](#_Toc510489119)

[***Dispatcher*** 3](#_Toc510489120)

[***Backup*** 4](#_Toc510489121)

[***Restore*** 6](#_Toc510489122)

[***Thread Registry*** 6](#_Toc510489123)

[***Enhancements*** 7](#_Toc510489124)

[***Backup*** 7](#_Toc510489125)

[***Restore*** 7](#_Toc510489126)

[***Delete*** 8](#_Toc510489127)

***SUBPROTOCOLS***

***Dispatcher***

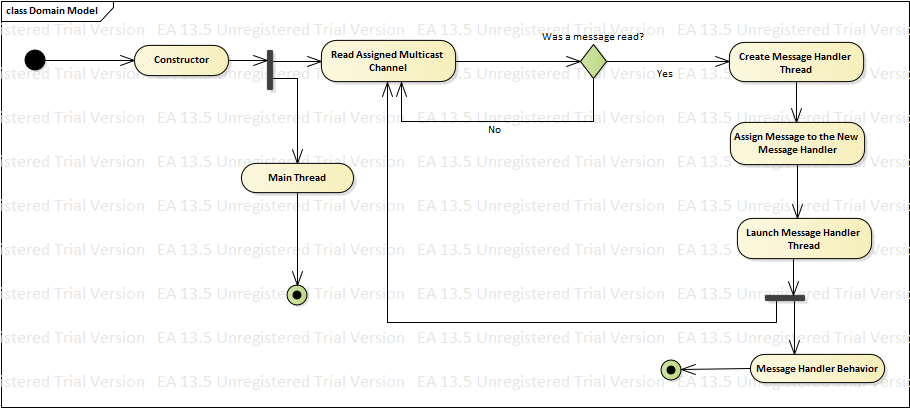
This implementation uses a hybrid of event driven and threaded architectures.

The dispatcher is divided into 2 key components the dispatcher reader and the dispatcher message handler.

The dispatcher reader the main source of concurrency in the event system, it is a threaded method that lives in an infinite loop trying to read its assigned multicast channel. When a message is read from the channel the dispatcher reader creates an instance of the message handler and gives it the read message to process, this message handler is then launched in an independent thread and continues reading. While creating a thread has inherent overhead and there might be a slightly better way to handle this we considered that this implementation fitter out design goals well enough, those goals being to minimize as much as possible the time the dispatcher reader is not reading the socket while maintaining architecture clean and easy to understand and iterate upon.

The dispatcher message handler’s behavior is not fundamental to the dispatcher’s concurrent architecture and as such we will not describe here how it behaves as it is a very simple message parsing and method invocation based on the parsed message. However, the message handler does play a key role in the concurrent behavior of the GETCHUNK and PUTCHUNK sub protocols, the role is notifying an active threads waiting for response messages, as the way this is done is specific to each of the sub protocols we will no go into the architecture in this section, we will reserve this for later explanation when describing the concurrent architectures of the 3 main protocols of the system.

Bellow we can see an activity UML diagram of the dispatcher.



UML Activity Diagram of the Dispatcher

***Backup***

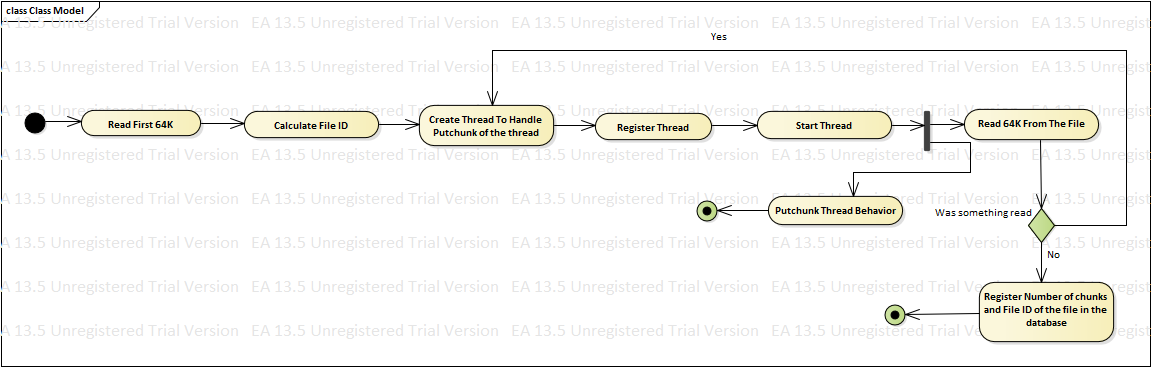
The backup protocol is nearly 100% concurrent, the only non-concurrent section being the reading and division of the file being backed up.

The backup protocol has 2 main components to it and a defining requirement for the dispatcher.

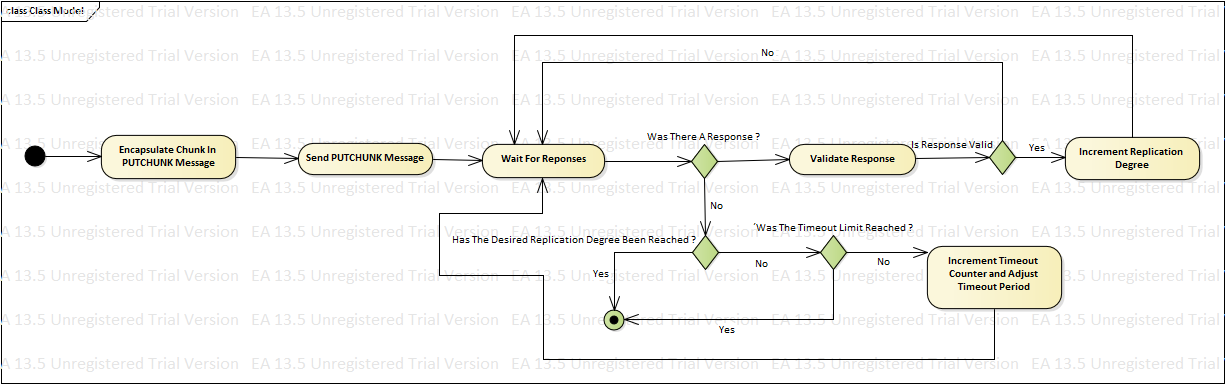
The first main component is the file reader and splitter, from the name we understand that if reads the file and splits however this is not done in 2 different stages but at the same time, basically we read the file in chunks of 64 thousand characters. After reading each chunk we then encapsulate it in an object that will handle the protocol of sending it over the network and receiving answers. When we are done with this we register in a file database the number of chunks and file ID of the file. There are a lot of details that have been left out as they do not contribute to the concurrency of the protocol.

The second main component is responsible for handling sending a chunk over the network and receiving its responses we call this the PUTCHUNK stub thread. To achieve this, it implements the runnable interface, when a thread is launched for it the body is encapsulated in a PUTCHUNK message and sent over the data backup channel, it then proceeds to wait for responses. Waiting for response exerts a need on the dispatcher, a need to signal the threads whenever a message directed at it is received and then pass along the message to the thread. For this to work when a thread that handles a chunk is created we register it as a thread that is waiting for responses to a PUTCHUNK message directed at a file ID and specific chunk, we also give the thread a queue where inbound messages are kept. Since the thread implements the runnable interface we have access to the wait() and notify() methods these allows us to wait until a message for the thread is received. As such the cycle of events is rather simple, the thread starts by sending the PUTCHUNK message and then proceeds to wait for responses, if the queue is empty we will wait a duration of time equal to the current timeout period. If we are able to read the message we validate it and increment the replication degree counter and reset the timeout period back to 1 second and repeat the process until we timeout either at the max timeout time of 16 second or we timeout and the desired replication degree was reached. From the dispatcher´s side when it parses a stored message it will access the registry of PUTCHUNK threads and search for one with the file ID and chunk number of the message, if there is one it adds it to the inbound queue of that thread and notifies it.

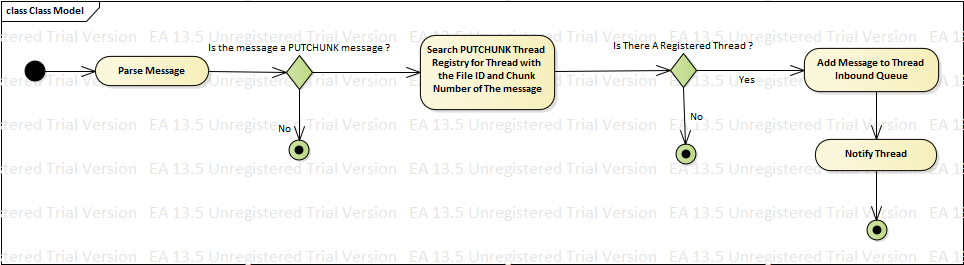
Bellow we can see highly simplified UML activity diagrams of the 3 behaviors described above.



UML Activity Diagram for File Reader and Splitter



UML Activity Diagram for the PUTCHUNK Stub Thread



UML Activity Diagram for the Dispatcher Message Handler

***Restore***

The concurrency implementation of the restore protocol is handled identically to the backup protocol, varying only in details specific to the restore protocol and the registry that is accessed and as such we have decided not to explain it here as it would essentially be a mirror of the previous section.

***Thread Registry***

The thread registry is fundamental to the concurrent behavior of the restore and backup protocols, since it as an area that can create a bottleneck in the system we decided to give a very brief explanation on how it was designed.

To implement this registry we an ConcurrentHashMap which is an implementation of the Hash Map that lock the data at the bucket level instead of locking it at the map level, this provides a huge performance benefit as the data is only locked if two thread try to access the exact same data at the same time, allowing for a much more concurrent access to the registry data.

From the tests realized in [this article](https://dzone.com/articles/java-7-hashmap-vs) we are able to see that a ConcurrentHashMap offer roughly 2x the performance of other Hash Map implementation in a heavily parallelized environment while still maintaining data integrity.

# ***Enhancements***

While we did not implement any enhancement due to lack of time in this section we outline out theories for the implementation of the suggested enhancements and had we had time these would have been the ways we would have implemented them.

## ***Backup***

The way we intend to solve the issue of replication degrees much larger than required is to extend the protocol to allow for deleting an individual chunk instead of the whole file and also extend it to allows us to target a specific peer, something like say:

DELETECHUNK <Version> <SenderID> <FileID> <ChunkNo> <TargerPeerID> <CRLF> <CRLF>

Upon receiving this message if the peer is the intended target it should delete said chunk. As for the initiator peer as soon as he reaches the required replication degree whenever it receives a STORED message after that it will send a DELETECHUNK message to the network directed at the Peers who´s storage of the chunk is unneeded.

An option for reducing the network stress is to switch over to a connection-oriented architecture by instead of sending the chunk body sending an IP and port to whom the peers connects to download the chunk body, thus reducing network load and giving individual initiator peers fine grained control over how much replication degree is present.

## ***Restore***

Enhancing the restore protocol to only send the chunk body to the peer who requested is very simple and the same as the second possibility presented for enhancing the backup subprotocol. Essentially a peer that will send a GETCHUNK request first sets up a TCP server that is listening for connections, the GETCHUNK message is then extended to include the IP and Port of this TCP server, the first peer to connect to this server will send the chunk body and all further connections will be denied, this heavily reduces network strain and makes sure only the intended recipient receives the chunk.

The GETCHUNK extension would look like:

GETCHUNK <Version> <SenderID> <FileID> <ChunkNo> <ServerAddress> <ServerPort> <CRLF> <CRLF>

## ***Delete***

To enhance delete to handle when the file was deleted but a given chunk was not online we need to use the concept of Death Certificates.

Death Certificates are essentially logs of all the files that have been deleted from the network.

So, in our theoretical implementation whenever a DELETE message is sent a Death Certificate must be issued all the peers need to add the file to their death certificate list. So, whenever a peer comes online it queries the network for death certificates and the peer send their list of death certificates, if a certificate that the peer does not have is received it deletes the file and issues a certificate for himself.

This is however never foolproof as it only works for as long as there are other peers that have the death certificate for the files, however as well all know magic is not real and therefore we will always need someone in the network with knowledge of the death certificate in order to inform peers who are not aware as the information cannot come from thin air.