Distributed Replicated Files

CS244B Assignment 2
Mansoor Malik

Introduction

This specification defines the syntax, semantics, and timing rules for a distributed replicated file system (DRFS) protocol. The protocol allows clients to open, write, and close files that are managed by network file servers. The protocol also allows clients to commit and abort transactions. A two-phase commit protocol is used for committing transactions.

The DRFS protocol is layered on top of UDP. All messages are sent to a single multicast address and port. The multicast address and port that is used in our implementation is 224.1.1.1 and 44022 respectively. The protocol provides an addressing mechanism layered over the UDP protocol. A message consists of a common header followed by a payload. The header and payloads are defined later in this document.

Since messages are sent over an unreliable packet network, the DRFS protocol must deal with lost packets, delayed packets, and packets that arrive out-of-order. The protocol defines the semantics and timing rules for dealing with these types of events.

In addition, it is possible for a node to fail because of malfunctioning software or hardware or a failed network link. The protocol defines how this type of failure should be detected and what actions should be taken in response.

MessageHeader

	Magic Number	Version	Message Type	Source ID	Dest ID	Sequence Number
Byte	0	4	6	7	11	15
Offset	3	5		10	14	18

A common header precedes all messages sent by clients and file servers.

The first field is a magic number that must be inserted in all headers. If a packet is received with an incorrect magic number, the packet must be discarded. The magic number used in our protocol is 0xdeadbeef.

The version number is a reserved field that may be used in the future if more than one protocol version is supported. For now, all nodes must insert the value 0x0001. A node must discard any message that does not have this version number.

The Message Type field identifies the type of message. A list of valid message types is listed below:

0x00 = JoinRequest
0x01 = JoinResponse
0x02 = OpenFileRequest
0x03 = OpenFileResponse
0x04 = WriteBlock

0x05 = PrepareToCommitRequest
0x06 = PrepareToCommitResponse
0x07 = CommitRequest
0x08 = CommitResponse
0x09 = AbortRequest
0x0a = AbortResponse
0x0b = CloseFileRequest
0x0c = CloseFileResponse

A source ID is a random number chosen by a node when starting. The value of this field can range from 1 to (2^32)-1 and must be unique. The value of 0x00000000 is reserved as a broadcast address for all file servers and must not be used.

All messages are multicast to all nodes. Hence a node must examine each incoming message and discard any message where the destination ID does not match its own identifier or the broadcast address if the node is a file server. A node should also examine the source ID and discard any messages that match its own ID. This may happen since messages are multicast to all nodes, including the node that transmitted a message.

The sequence number is a monotonically increasing value that starts at one and increases by one every time a sender transmits a UDP packet. The maximum value of a sequence number is (2^32)-1. This value is large enough to ensure that a session ends before sequence numbers wrap around. The behavior is undefined in the unlikely event that sequence numbers wrap around.

JoinRequest

	Request Code
Byte Offset	0

The JoinRequest message allows clients to discover file servers. This request is sent as a multicast message to all file servers using the destination address of 0x00000000. The message has a request code field that can take on one of the following two values:
0x00 = RespondWithExistingId
0x01 = RespondWithNewId

The client sends 5 requests and waits for 1 second after each request to receive responses. The number of responses must match the number of servers that are passed in the InitReplFS function. If the number of responses does not match, the client must issue an error message and exit.

The client acts as a coordinator and detects duplicate identifiers. If a duplicate identifier is detected, the client must restart the process to discover file servers. It must send a JoinRequest message with a request code of RespondWithNewId followed by up to 4 additional requests with RespondwithExistingId. A file server that receives a request with RespondWithNewId must select a new random identifier and respond to the request.

JoinResponse

The JoinResponse message has no body. It consists only of a header.

All file servers must respond to a JoinRequest message by sending a JoinResponse message. The destination address in the header must be set to the identifier of the client that issued the JoinRequest. The source identifier must match the identifier of the file server that received the request.

OpenFileRequest

	Filename
Byte	0
Offset	127

A client sends an OpenFileRequest message to open a file. The message is multicast to all file servers using the broadcast address of 0x00000000. The message has one field that contains the filename to be opened. The sender must ensure that the filename is null terminated.

To allow for packet loss, this message may be transmitted up to 5 times. The client must wait 1 second for all responses to be received by replicated file servers before retransmitting. The client must ensure that all file servers have responded with a valid file descriptor.

OpenFileResponse

	File Descriptor
Byte	0
Offset	3

The OpenFileResponse message has a single field that is the file descriptor. The file descriptor is 1 for the first file that is opened. It is incremented by 1 for every subsequent file open request.

WriteBlock

	Block Identifier	File Descriptor	Byte Offset	Block Size	Buffer
Byte	0	4	8	12	16
Offset	3	7	11	15	527

The WriteBlock message is multicast from a client to all servers. The first field in this message consists of a block identifier. This is a monotonically increasing value that starts at 1 and increments by 1 for every write. A server uses this value to determine if it is missing blocks. The file descriptor must be a valid file descriptor for an open file. The byte offset specifies the offset from the beginning of a file and the block size specifies the size of the buffer. The buffer is a sequence of bytes that must be written to the file. The block size cannot be larger than 512 bytes and the file size cannot exceed 1 MB.

PrepareToCommitRequest

	Transaction Identifier	File Descriptor	First Block Identifier	Last Block Identifier
Byte	0	4	8	12
Offset	3	7	11	15

This message consists of four fields. The first field is a transaction identifier. It is a monotonically increasing value that starts at 1 and increases by 1 every time a client attempts to commit file changes. The second field is a file descriptor. The last two fields are the first and last block identifiers that were transmitted by a client using the WriteBlock message. A server uses these identifiers to determine if any write blocks are missing.

All servers must respond to this request. The packet containing the request from the client, or the response from the server, may be lost. Therefore, the client may transmit a request up to 5 times. The client will set a timeout period of 1 second for all servers to respond.

File servers may respond to this request by asking for the retransmission of missing write blocks. Hence, a client must retransmit the missing write blocks and then retry again. If a client retries after sending a missing write block, it may once again transmit a request up to 5 times to allow for lost requests or responses.

PrepareToCommitResponse

	Transaction Identifier	Response Code	Number of Missing Blocks	Missing Block Identifiers
Byte	0	4	5	9
Offset	3		8	521

All file servers respond with a PrepareToCommitResponse message after getting a PrepareToCommitRequest message. This message consists of four fields.

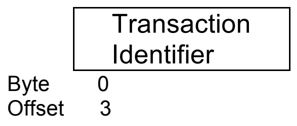
The first field is the transaction identifier that was transmitted in the request.

The response code from a file server may be one of the following values:
0x00 = VoteYes
0x01 = VoteNoMissingWriteBlock
0x02 = VoteNoInvalidFileDescriptor

If a file server is missing a block, the server will respond with a VoteNoMissingWriteBlock response code. The third field in this message includes the number of missing blocks. The values can range from 0 to 128, the maximum number of blocks that can be transmitted before a commit. The last field will include the missing block IDs. Each block ID is 4 bytes.

The file server should ensure that the file descriptor is valid. If it is not then it should set the response code to VoteNoInvalidFileDescriptor.

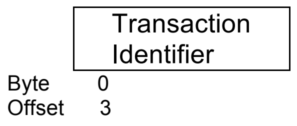
CommitRequest



If all file servers respond with a yes vote, a client can send a CommitRequest message. The client must ensure that the number of servers that voted yes is equal to the number of servers specified in the InitReplFs call.

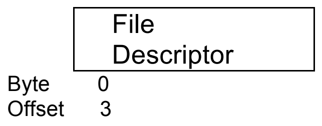
This message contains a single field that is the transaction identifier that was sent earlier to all file servers. The client expects all servers that voted yes to respond to this request. The packet containing the request from the client, or the response from the server, may be lost. Therefore, the client may transmit a request up to 5 times. The client will set a timeout period of 1 second for all servers to respond before retrying.

CommitResponse



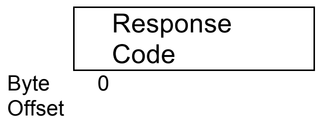
All file servers must send a CommitResponse message after receiving a CommitRequest message if they voted yes to the PrepareToCommitRequest that was sent earlier.

AbortRequest



A client sends an AbortRequest message to discard all changes since the last commit. The file descriptor is the only field in this message. The packet containing the request from the client, or the response from the server, may be lost. Therefore, the client may transmit a request up to 5 times. The client will set a timeout period of 1 second for all servers to respond before retrying.

AbortResponse



This message has one field. All replicated file servers must respond with an AbortResponse message after receiving an AbortRequest message. The response code will be success as long as a valid file descriptor was used in the request. The valid values for the response code are:

0x00 = AbortSuccess
0x01 = AbortFailure

CloseFileRequest

	File Descriptor
Byte Offset	0 3

A client sends a CloseFileRequest message to close a file. All unsaved changes must be committed before closing a file. The client is responsible for issuing the PrepareToCommitRequest and CommitRequest messages before issuing a CloseFileRequest message if any updates were pending.

The file descriptor is the only field in this message. All replicated file servers must respond to this request with a CloseFileResponse message. The packet containing the request from the client, or the response from the server, may be lost. Therefore, the client may transmit a request up to 5 times. The client will set a timeout period of 1 second for all servers to respond before retrying.

CloseFileResponse

	Response Code
Byte Offset	0

A replicated file server must respond with a CloseFileResponse message after receiving a CloseFileRequest message. The response code will be success as long as a valid file descriptor was used in the request. The valid values for the response code are:

0x00 = CloseSuccess
0x01 = CloseFailure

Evaluation

This section describes the merits and disadvantages of using our approaches to replication versus conventional reliable transport.

In our protocol, all communication from client to file servers is multicast using UDP. Therefore a message is only sent once between one client and many servers. This makes the system inherently more scalable as more clients and file servers are added to a system. If we had used TCP connections between client and file servers than a message would need to be transmitted over each individual connection. For example, using a TCP connection, writing a 1 MB file to 5 file servers would result in 5 MB of data being transferred over the network.

The downside of using UDP is that it is not reliable. Hence, an application has to deal with lost packets, packets that are delivered out of ordered, or packets that are delayed. This is a disadvantage because the application needs to handle this complexity instead of relying on a transport protocol such as TCP. For example, when a server does not respond to a request from a client, we do not know whether (a) the request was lost or delayed, (b) the response was lost or delayed, (c) a network link failed, or (d) a file server went down because of a software or hardware failure. The client deals with these issues by retransmitting a request up to a maximum of 5 times and waiting at least 1 second for a response before retrying. If no response is received after 5 attempts then a client concludes that either a file server went down or a network link failed.

TCP itself has limited usefulness if a network link fails or a file server went down because of a software or hardware failure. Even if we used TCP we would need to set a timeout before reading from a socket. Otherwise, the read could block forever and never return if a file server was no longer reachable.

An advantage of our approach is that we can discover file servers dynamically by sending discovery messages over a multicast channel. This would not have been possible using a transport protocol such as TCP, which is connection based. We would have needed another mechanism such as static configuration files or a name server.

Another advantage of our approach is that all of our messages are datagrams, which means that each datagram corresponds to a message. A reliable protocol such as TCP uses streams, which makes the implementation more complex as mechanisms are needed to detect boundaries between messages.

Another advantage of our approach is observability since all messages are multicast between clients and file servers. For example, in a system with many clients and servers, all nodes can observe all actions. Hence, other nodes would observe actions taken by a malicious user. On the other hands, if confidential data needed to be saved between a client and file server, we would need to encrypt it which takes away the benefits of observability. This is discussed in more detail in the next section.

A disadvantage of our approach is that even though we use a two-phase commit protocol, there is no mechanism to recover from file server failures. This is discussed in more detail in the next section.

Future Directions

This section discusses the extensions, refinements, and modifications to our protocol and implementation that would be required for real deployments. The answers to this discussion include considerations of high availability, performance, security, and the semantics of file systems.

The current implementation makes use of a two-phase commit protocol. However, the functionality of a transaction coordinator or manager is implemented in the client itself. This is undesirable because in a real deployment, a distributed file system must support multiple clients. Therefore, one of the enhancements would be to have a separate node act as a transaction coordinator. The transaction coordinator itself could be replicated for high availability.

In the current two-phase commit implementation, file servers do not log transactions to disk before responding with their votes. This scheme would not work in a real deployment for the following reason. Assume that a transaction coordinator sends a PrepareToCommitRequest to all file servers and that all file servers respond with a yes vote. The transaction coordinator then sends a CommitRequest to all servers. All file servers that are still functioning will commit the file changes to disk. However, if one of the file servers failed after sending their yes vote, they need to be able to recover to a correct state. A transaction log would allow them to determine which changes were pending when they sent their yes vote. Once the file server is back on line, it could check with the transaction coordinator to determine if they should commit or abort the changes in their log file.

At present, the loss of a single file server means that a client can no longer commit file changes. For example, if a temporary network failure occurred between file servers then the contents of files may no longer be correct if writes continued to occur on some file servers but not others. In real deployments however, the expectation is to have mechanisms in place to withstand the failure of a single node.

It is outside the scope of this project but several solutions are possible if we want to modify the system to have higher availability. One possibility is to separate the file system into metadata and extent servers. Some distributed file systems such as the Google File System use this scheme. The metadata server would make use of expensive high-availability machines whereas the extent servers would make use of commodity servers. The intent of this suggestion is to have a file system that is more robust in the face of failures.

The existing implementation only supports a single client and a single open file. But the existing protocol can be used to allow a user to open multiple files. The protocol can be extended to support more than one client as an identifier uniquely identifies each client. The difficulty of supporting multiple clients would occur if two clients tried to open the same file. In that case, we could prevent a client from opening a file that was opened by another user.

The existing protocol does not support common file system operations for reading, copying, and deleting files. The existing protocol also does now allow clients to create and delete directories or to list the files that are in a directory. These enhancements would be needed for real deployments.

It would be inefficient to read a file from multiple file servers. So the protocol would have to be modified to read from a single file server. We would need to enhance the protocol to support ReadBlockRequest and ReadBlockResponse messages. The messages would have fields for a file descriptor, an offset into the file, and the number of bytes to read. The number of bytes could be capped to a reasonable level such as 512 bytes.

Similarly, we could enhance the protocol by adding DeleteFileRequest and DeleteFileResponse messages. The client would need to pass the name of the file to be deleted in the request. The response would confirm whether the delete operation had succeeded or failed. To ensure correctness, a file would need to be deleted either from all servers or none at all. Therefore, a two-phase commit scheme would need to be used.

The protocol can also be enhanced to list all files in a mounted directory. A client can transmit a FileCountsRequest message to a single file server. The file server can respond with a FileCountsResponse message that specifies the number of files in a directory. If there are N files and each file is indexed from 0 to N-1, then a client can retrieve all file names by sending FileNameRequest messages with the index number of a file name. The file server can respond with the file name. If file names are limited to a maximum length, then multiple file names can be retrieved with each request and response.

A major shortcoming of the existing protocol is that there is no user access control and authentication. The lack of user access control means that any user can open any file. Clearly this is not acceptable for a file system that would be used in a production environment. In addition, mechanisms for authenticating users are also required. Both features would be required in a real deployment.

In addition, all communication between clients and file servers happens in the clear. Therefore, an attacker could monitor network traffic and read the contents of confidential file. This may not be acceptable in many real life deployments. Ensuring confidentiality requires that the file contents and possibly the file metadata be encrypted. However, encrypting the communication between clients and file servers would defeat the benefits of observability that were described earlier.