October 2011

Tape-bridge Requirements, Analysis and Design

Revision : 3
Author : Steven Murray

CERN

Revision history

Revision	Date	Description
1	Tue, 19 Oct 2010	Initial version of document committed into SVN
2	Wed, 20 Jul 2011	Added file specific errors to the protocol between the tapebridged and tapegatewayd daemons
3	Wed,28 Oct 2011	Added the bulk-message protocol between the tapebridged and tapegatewayd daemons

Table of Contents

Introduction

The purpose of this document and the intended audience

This document describes the tape-bridge daemon (tapebridged) from problem statement, to analysis through to design. This document should be read by developers responsible for maintaining the tapebridged daemon and by developers working on the tape-gateway daemon (tapegatewayd) who need a detailed description of the TCP/IP protocol used between the tapebridged and tapegatewayd daemons.

The structure of this document

The next section of this document describes the requirements that are addressed by the tapebridged daemon. The section after describes the protocol used between the CASTOR remote tape copy daemon (rtcpd) and its client daemon (rtcpclientd). This document refers to this protocol as the rtcpd protocol. The legacy rtcpclientd daemon will be replaced by the tapegatewayd daemon. The tapegatewayd daemon speaks a simpler protocol than the rtcpd daemon. The main goal of the tapebridged daemon is to translate the rtcpd protocol so that the legacy rtcpd daemon can communicate with the new tapegatewayd daemon. The design section follows with the description of the two new protocols that the tapebridged daemon will introduce into CASTOR. The first protocol is already deprecated and is based on one message referring to a single file transfer. The second and hopefully final protocol is based on one message referring to a collection of many files to be transferred. This second "bulk-file" protocol was introduced to help make efficient "bulk" calls to the relational-database that stores the state of the tapegatewayd daemon. The design section then finishes by giving a high-level description of the design of the tapebridged daemon so that readers will be able to find their way around the source code of the daemon.

Requirements

A typical CASTOR installation can be divided into five types of server: central, stager, disk, tape and remote media changer. A central-server runs one or more CASTOR daemons that are responsible for managing the site-wide state of CASTOR. This state includes the file namespace, the tape-catalogue and the tape-drive scheduler. A stager-server runs one or more CASTOR daemons that are responsible for managing the state of a single stager. There is usually one stager per virtual organization at a site. A disk-server is physically connected to some disk-based storage and runs the CASTOR daemons responsible for accessing that data. A tape-server is physically connected to one or more tape-drives and runs the CASTOR daemons responsible controlling those drives. A remote-media-changer server is physically connected to a tape library and runs the CASTOR daemon responsible for controlling that library. Some hardware manufactures provide a per drive interface to the tape library in which the drive is located. In this case an instance of the CASTOR remote media-changer daemon is installed locally on each tape-server and there is no need for a remote-media-changer server.

CASTOR currently uses a pair of remote tape-copy daemons to coordinate the movement of data from disk to tape and from tape to disk. One member of the pair is called the remote tape-copy daemon, hereafter referred to as the `rtcpd` daemon. One instance of the `rtcpd` daemon runs on each of the CASTOR tape-servers. The other member of the pair is called the remote tape-copy client daemon, hereafter referred to as the `rtcpclientd` daemon. There is one instance of the `rtcpclientd` daemon per stager, meaning that one instance of the `rtcpclientd` daemon runs on one and only one of the possibly many stager-servers that make up a CASTOR stager.

There is currently a large discrepancy between the potential write performance of the tape hardware used by CASTOR and the actual measured performance. The problem lies with the fact that CASTOR uses a tape-format that requires 3 tape-marks per user-file written to tape. A normal tape-mark, hereafter referred to as a flushed tape-mark, marks the end of the current file on tape and flushes to tape all of the data currently buffered between the writing process and the physical tape. A flushed tape-mark can take around 3 seconds to complete due to the flushing of the data from the buffers. Considering the write speed of a tape-drive is currently 160 MB per second, the write performance of relatively small files is seriously reduced by their need to have 3 tape-marks each. The SCSI 2 specification allows for the use of immediate tape marks. These tape marks are written to tape but there is no flush of the buffers between the writing process and the physical tape. It has been demonstrated that tape write-performance is significantly improved by writing many small files using immediate tape-marks and then flushing the data buffers to make sure all data has been successfully written to tape.

Major modifications would need to be made to the rtcpclientd daemon in order to implement the idea of immediate tape-marks being enclosed between longer period flushes to tape. The protocol used between the rtcpclientd and rtcpd daemons would need to be modified so the rtcpcd daemon would be able to tell the rtcpclientd demon when it had flushed data to tape. The rtcpclientd daemon would need to be modified to understand this new protocol and it would need to be modified to only update the CASTOR namespace for a group of consecutive files once they had been flushed to tape.

The code maintenance costs of the rtcpclientd daemon have reached the point where it is impractical to make any major modifications. To make matters worse, it has also been requested that the rtcpclientd daemon be modified to keep all of its state in a database so that the daemon itself can be re-started without losing this state information. Given the difficultly to add new features and the high maintenance costs of the rtcpclientd daemon it has been decided to replace the daemon with the tape-gateway daemon, hereafter referred to as the tapegatewayd daemon.

Three major constraints were imposed on the development of the tapegatewayd daemon. Firstly the rtcpd daemon should not be significantly modified due to its critical role in the data path to tape. Secondly it must be easy to switch back and forth between using the rtcpclientd and tapegatewayd daemons. Thirdly a tape-server should be compatible with both stagers running the rtcpclientd daemon and stagers running the tapegatewayd daemon. The rational behind this latter constraint is the CASTOR tape-system is shared by all of stagers at a given site and the stagers are upgraded one by one to reduce the risk of a site-wide failure due to a bug introduced by the latest version of CASTOR.

The efficiency of the rtcpd protocol is heavily dependent on the number threads in the child processes of the rtcpclientd and rtcpd daemons. This dependency complicates the rtcpd protocol (especially its shutdown logic). The dependency also complicates the number of threads to be configured and used in the child processes of the rtcpclientd daemons. To run the rtcpd protocol efficiently it is necessary to configure the number of threads in the rtcpclientd daemon as a function of the number of threads that run in the child processes of the rtcpd daemon. The same rtcpclientd daemon communicates with rtcpd daemons that run with a different numbers of threads. The number of threads is different because it is a function of the amount of available main-memory and networking bandwidth. The rtcpd daemons running on newer and more powerful tape-servers will run with more threads than rtcpd daemons running on older lesser machines. Operators therefore must make a compromise between idle threads running in the child processes of the rtcpclientd daemon and fully supporting the most powerful tape-servers. The tapegatewayd protocol will be designed to have no dependency on the number of threads running in the child processes of the rtcpd daemon.

Analysis

The structure of this section

This section describes the rtcpd protocol. This is the protocol used between an rtcpd daemon running on a tape-server and an rtcpclientd daemon running on a stager-server. The next subsection describes the additional CASTOR daemons that participate in the rtcpd protocol. With these daemons described the subsection that follows goes on to explain how the rtcpclientd daemon obtains a tape for writing. The tape required when recalling files from tape is known in advance by the CASTOR name-server. A pair of child processes carries out the coordination of the file transfers for a single tape-mount. One of the pair is called the worker process and is forked and executed by the rtcpclientd daemon running on a stager-server. The other process of the pair is a child process forked by the rtcpd daemon running on a tape-server. The next subsection explains when this pair of processes is created in the rtcpd protocol. The final subsection explains how an rtcpclientd worker process and an rtcpd child process work together to coordinate the copying of files from disk to tape or from tape to disk.

The additional daemons of the rtcpd protocol

The following additional daemons need to be described briefly in order to help explain the rtcpd protocol:

- Volume and drive queue manager daemon (vdqmd)
- Volume manager daemon (vmgrd)
- Remote file access daemon (rfiod)
- Tape daemon (taped)

The vdqmd daemon is a site-wide drive scheduler that runs on the CASTOR central-servers. The vdqmd daemon supports redundancy by allowing more than one instance of itself to be ran on the CASTOR central-servers. The vdqmd daemon allocates free tape-drives to compatible tape-mount requests. A free tape-drive is matched with a compatible tape-mount request via the device-group-name (DGN). A DGN is a unique identifier string used to map tape-libraries to physically compatible tape-drives. A tape-drive is assigned the DGN of the tape-library in which it is located.

The vmgrd daemon is a site-wide tape-catalogue that runs on the CASTOR central-servers. The vmgrd daemon supports redundancy by allowing more than one instance of itself to be ran across the CASTOR central-servers. The information stored by the vmgrd daemon about each tape includes the estimated amount of free space, the tape-library where the tape is located, the logical tape-pool to which the tape belongs and the number of user files that have been written to the tape (please note that this count is not decremented when the CASTOR namespace entry for a file on the tape is deleted). The tape-library of a tape indirectly gives the DGN of that tape, which in turn is used by the vdqmd daemon to find a free and compatible drive when servicing a request to mount the tape. A tape-pool is an arbitrary grouping of tapes defined by the administrators of the CASTOR installation. A tape-pool can be composed of tapes from more than one tape-library. The vmgrd daemon does not store any information about the individual files stored on a tape. The CASTOR name-server daemon, which is not described any further by this document, stores this information.

Each disk-server runs an instance of the rfiod daemon. This daemon is responsible for providing remote access to the files located on the attached disk-based storage.

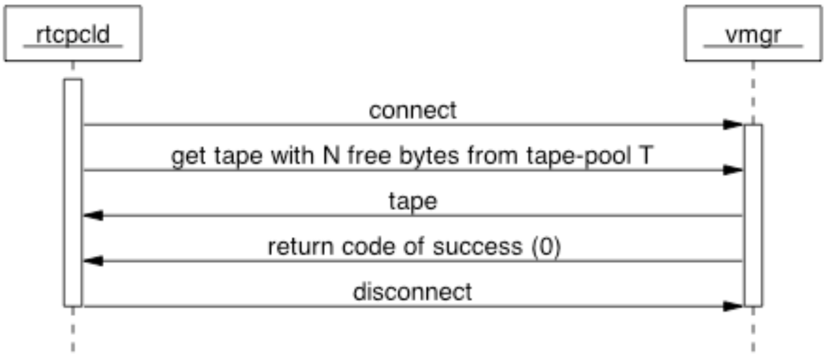
Each tape-server runs an instance of the taped daemon. This daemon is responsible for mounting and unloading tapes, tape-file positioning and for configuring tape-drives. When the taped daemon receives a client request, it forks and executes a helper process to service that request. The helper processes used by the taped daemon include mounttape, rlstape, posovl and confdrive.

Each tape-server runs an instance of the rtcpd daemon. This daemon is responsible for transferring data from disk to memory to tape and from tape to memory to disk. The daemon moves data to and from disk via connections made with the rfiod daemons running on the disk-servers. The rtcpd daemon moves data to and from tape via the appropriate Linux device driver (e.g. /dev/nst0). The rtcpd daemon delegates the tasks of mounting, positioning and unloading tapes to the taped daemon.

Getting a tape for writing

It is the responsibility of the `rtcpcld` daemon to ask for tapes to be mounted for either reading or writing. The tape to be mounted is known when a file needs to be recalled from tape. In the case of migrating files from disk to tape, an appropriate tape needs to be found.

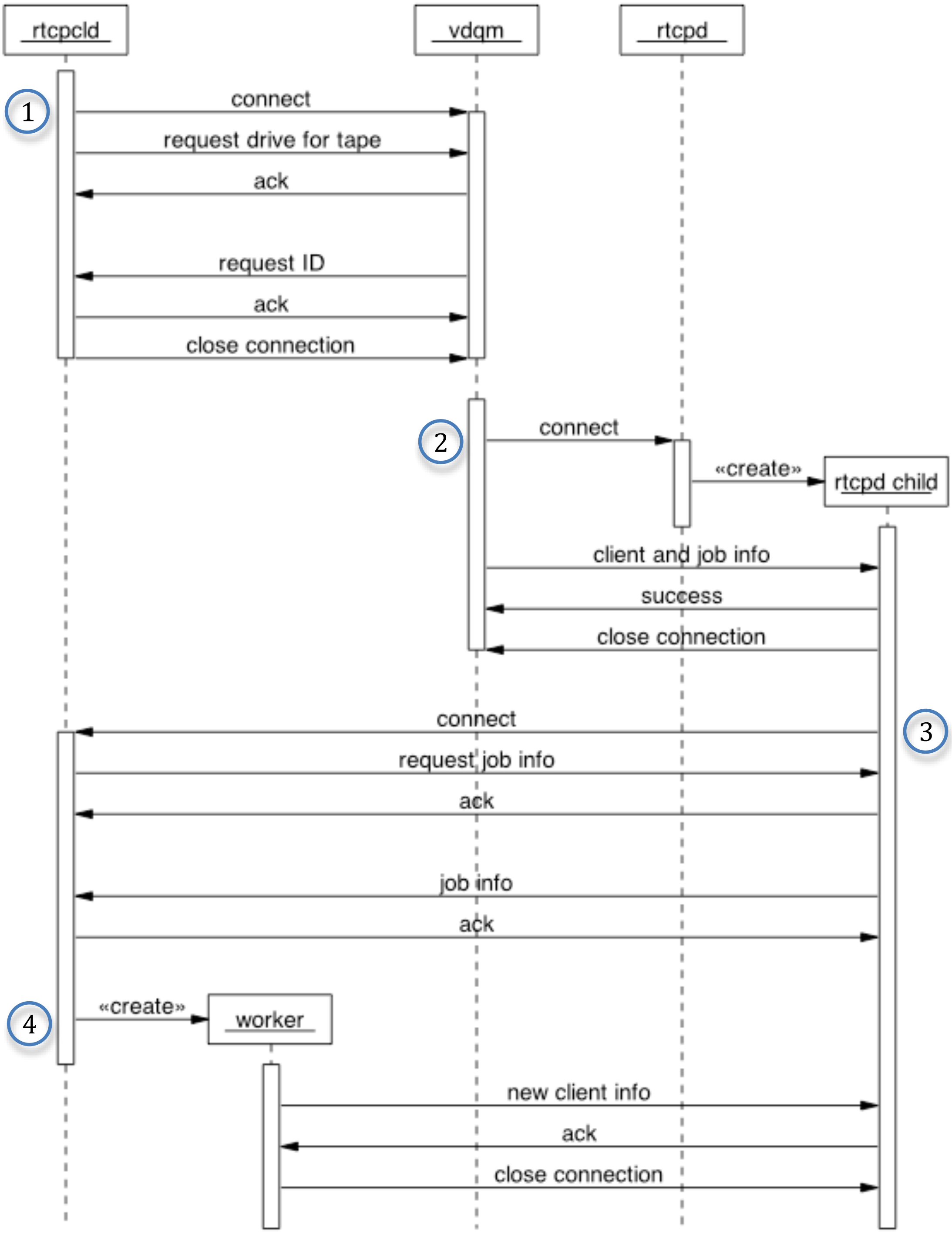
The following sequence diagram describes how the `rtcpclientd` daemon asks the `vmgrd` daemon for a tape for writing.


rtcpcld
vmgr
connect
get tape with N free bytes from tape-pool T
tape
return code of success (0)
disconnect

The `rtcpclientd` daemon connects to the `vmgrd` daemon via TCP/IP and sends a message requesting a tape for writing that satisfies two conditions. The tape should preferably have a certain amount of free space and it must belong to a specific tape-pool. In response to the request, the `vmgrd` daemon finds an appropriate tape and sets its status within the tape catalogue to `BUSY`. The volume identifier of the tape along with the position of the next file to be written is sent back to the `rtcpclientd` daemon.

Starting a pair of rtcpclientd worker and rtcpd child processes

The following sequence diagram shows the `rtcpclientd` daemon asking the `vdqmd` daemon for a free drive to mount a tape and it shows the creation of the `rtcpclientd` worker and `rtcpd` child processes.



The above sequence diagram has been divided into 4 steps to help describe the details.

At step 1 the `rtcpclientd` daemon has determined that a tape needs to be mounted for either migration or recall. As a result the `rtcpclientd` daemon sends a tape mount request to the `vdqmd` daemon. The request includes the volume identifier of the tape and the DGN associated with the tape via the library of tape. In response the `vdqmd` daemon queues the request and replies with a unique request identifier.

At step 2 the `vdqmd` daemon has asynchronously found a free and compatible drive that it can allocate to the pending tape mount request received earlier from the `rtcpclientd` daemon. The `vdqmd` daemon subsequently connects via TCP/IP to the `rtcpd` daemon running on the tape-server connected to the free tape-drive. The `rtcpd` daemon immediately forks a child process to handle the request and passes it the accepted TCP/IP connection. The `vdqmd` daemon sends to the child `rtcpd` process the TCP/IP connection details of the `rtcpclientd` daemon that requested the tape mount and information about the remote tape-copy job to be carried out. The job information includes the unique request identifier and the identity of the physical drive-unit to be used for the mount. The drive-unit needs to be identified explicitly because a single tape-server and its single instance of the `rtcpd` daemon may be associated with more than one physical drive. The volume identifier of the tape to be mounted is not passed from the `vdqmd` daemon to the `rtcpd` daemon.

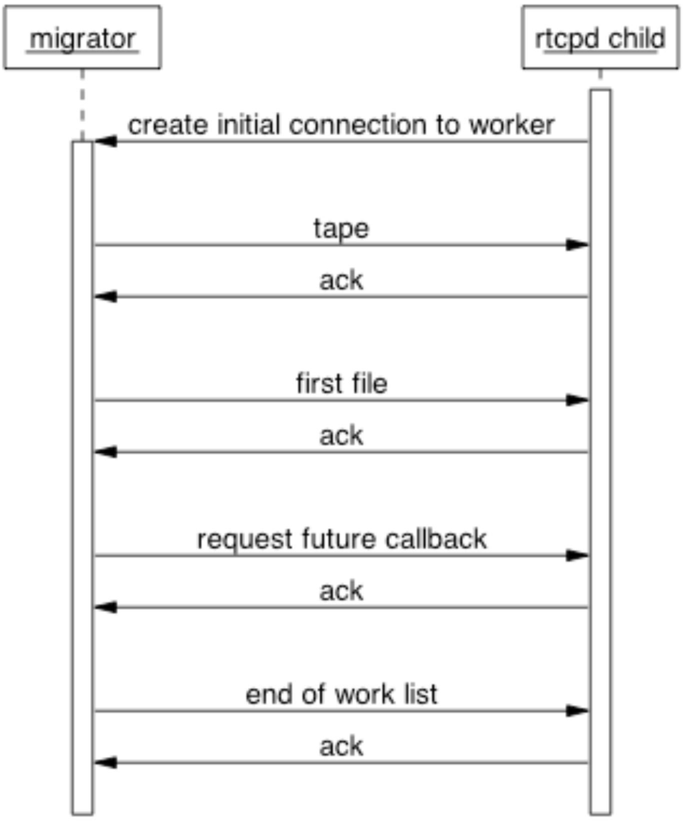
At step 3 the `rtcpd` child process connects to the `rtcpclientd` daemon, which immediately sends back a request for the remote tape-copy job information. The `rtcpd` child process sends the information to the `rtcpclientd` daemon, which uses the information to determine the volume identifier of the tape to be mounted and whether to mount the tape for read or write access.

At step 4 the `rtcpclientd` daemon forks and executes a worker process in order to coordinate the reading or writing of the tape. When reading from tape the worker process used is the `recaller` process and when writing to tape the worker process used is the `migrator` process. The worker process creates its own TCP/IP listening port, the details of which it then sends to the `rtcpd` child process via the TCP/IP connection made between the `rtcpd` child process and the parent `rtcpclientd` process. The worker process then closes the TCP/IP connection it inherited from its parent and waits for the `rtcpd` child process to reconnect using the newly created listening port of the worker process.

Coordinating a remote-copy

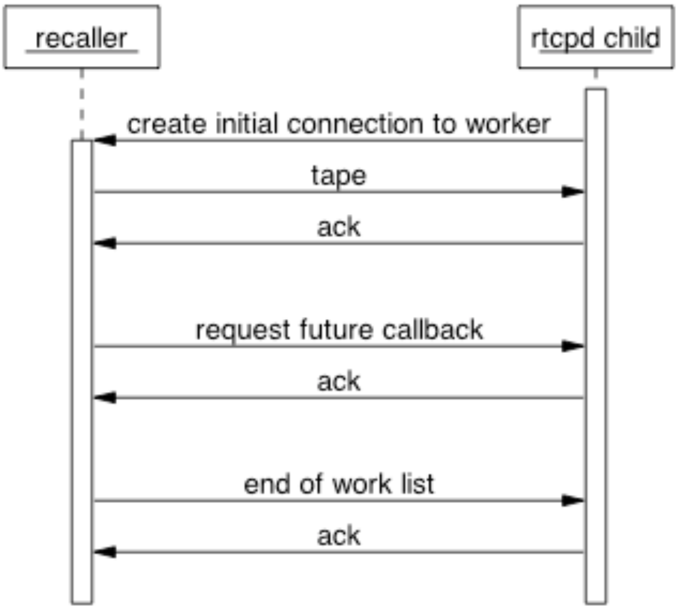
The `migrator` worker process exchanges more initial messages with the `rtcpd` child process than the `recaller` worker process does. When migrating files from disk to tape there is the opportunity to start copying data from disk into the memory of the tape-server whilst the tape is being mounted.

The following sequence diagram shows the initial messages exchanged between a `migrator` process and an `rtcpd` child process.

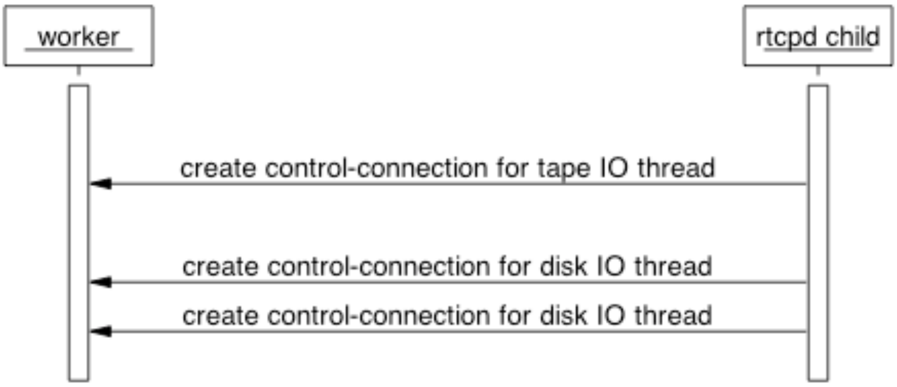


The `rtcpd` child process connects to the `migrator` process that was waiting for the connection after sending the details of its own newly created TCP/IP listening port. The newly created connection with the `migrator` process will hereafter be referred to as the initial connection to the worker. This name is given because more connections will be made latter between the `rtcpd` child process and the worker process and they will be treated differently during the shutdown logic of the `rtcpd` protocol. Once the initial connection has been made, the `migrator` process immediately sends a work list to the `rtcpd` child process containing the tape to be mounted, the first file to be migrated and a request for the `rtcpd` child process to request more work in the future.

The following sequence diagram shows the initial messages passed between a `recaller` worker process and an `rtcpd` child process. The messages are the same as those exchanged between a `migrator` process and an `rtcpd` child process except the first file to be transferred is not sent because the tape has to be mounted before any data can be transferred to the memory of the tape-server.



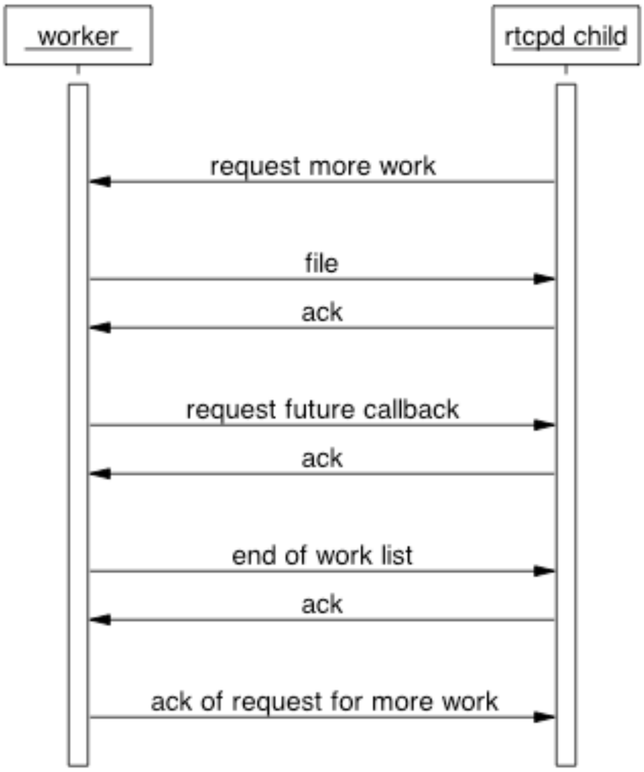
Once the `rtcpd` child process knows which tape to mount it internally creates one tape IO thread for reading or writing files to tape and one or more disk IO threads for remotely reading or writing disk files using TCP/IP connections to the `rfiod` daemons running on the disk-servers. A separate control-connection between the `rtcpd` child process and the worker process is made for each of these IO threads as shown by the next sequence diagram. Please note that the number of disk IO threads is configurable and the following sequence diagram assumes the `rtcpd` child process is configured to have two.


worker
rtcpd child
create control-connection for tape IO thread
create control-connection for disk IO thread
create control-connection for disk IO thread

The main loop of the `rtcpd` protocol can begin now that the IO threads of the `rtcpd` child process are created and they have access to a set of TCP/IP control-connections with the worker process. For the sake of simplicity this document will temporarily treat all of the connections between the `rtcpd` child process and the worker process as if they were one, and in fact the protocol does not rely on a specific connection sending and receiving specific messages until the shutdown logic starts.

The main loop of the rtcpd protocol can be divided into two message sequences. The first, hereafter referred to as the "what's next?" sequence, is the sequence of messages that exchanges which file should be transferred next and when the session of all the files to be transferred is finished. The second sequence of messages, hereafter referred to as the "transfer feedback" sequence, is the sequence of messages used to give feedback on individual files being transferred to tape or to disk.

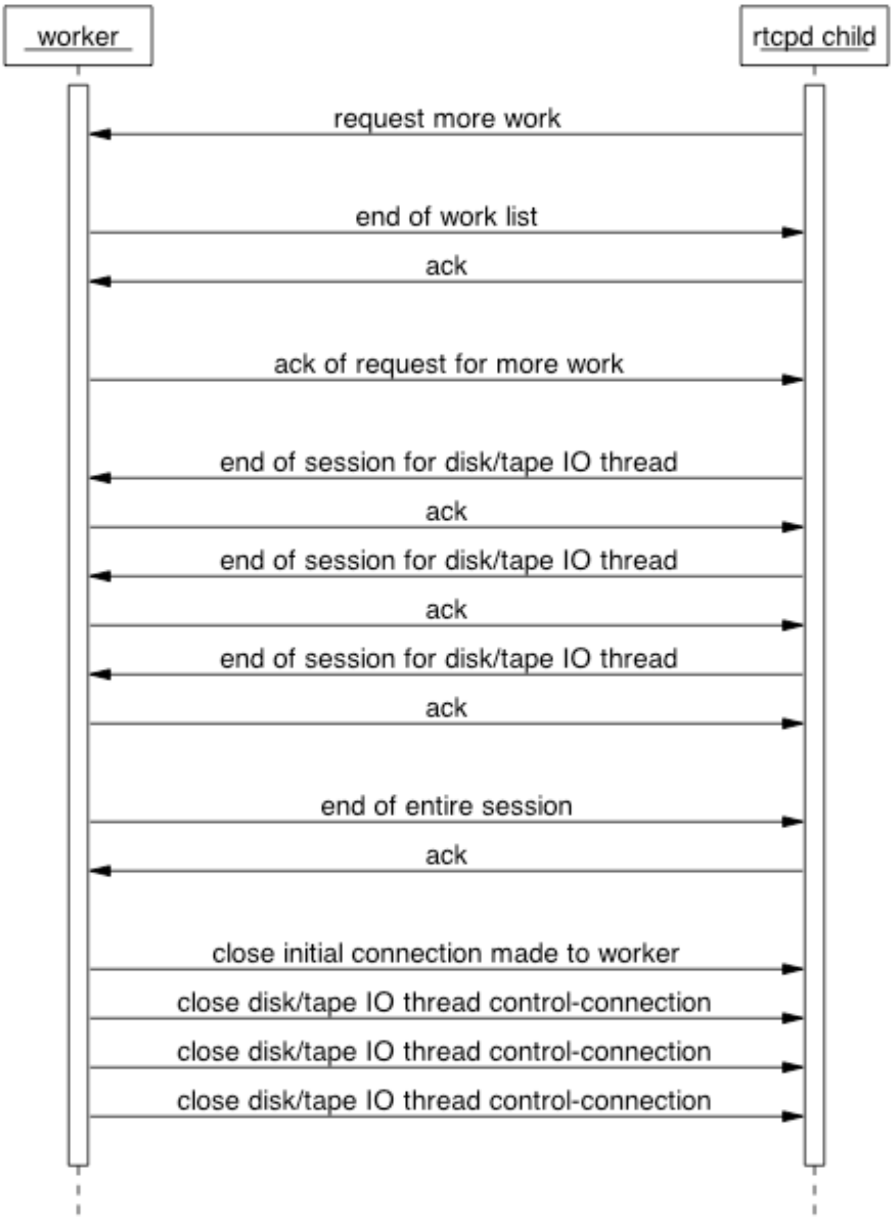
The following sequence diagram describes the "what's next?" message sequence.



Each iteration of the main loop starts with the rtcpd child process requesting more work from the worker process. In response the worker process replies with a work list of one file to be transferred and a request for the rtcpd child process to ask for more work in the future.

A session of transferring one or more files can end in two ways. The worker process could run out of files to be transferred or an error could occur on either the worker process side or the `rtcpd` child process side. It should be noted that reaching the physical end of a tape when migrating files from disk to tape is considered an error even though it is expected.

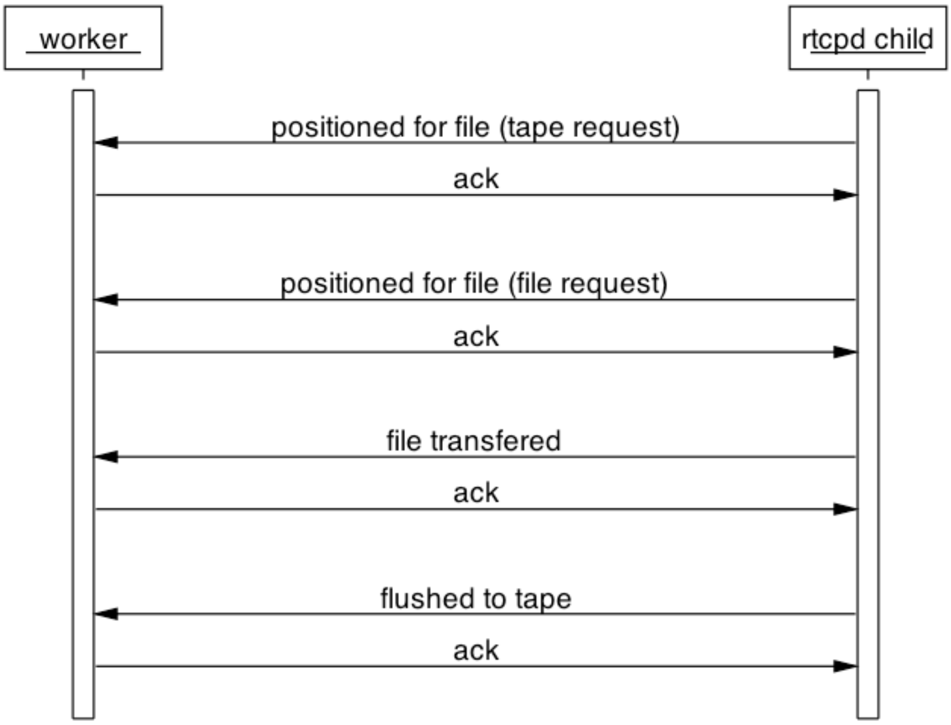
The following sequence diagram shows the end of the "what's next?" sequence when the worker process has ran out of work to do (files to transfer).


worker
rtcpd child
request more work
end of work list
ack
ack of request for more work
end of session for disk/tape IO thread
ack
end of session for disk/tape IO thread
ack
end of session for disk/tape IO thread
ack
end of entire session
ack
close initial connection made to worker
close disk/tape IO thread control-connection
close disk/tape IO thread control-connection
close disk/tape IO thread control-connection

The exit process starts with the `rtcpd` child process asking for more work and the worker process replying with an empty work list because there are no more files to be transferred and the `rtcpd` child process should not ask for more work in the future. The tape and disk IO threads of the `rtcpd` child process then start to individually send end of session messages when they have finished all the work they have been each assigned to do. When all the tape and disk IO threads have sent their end of session messages, the worker process sends an end of entire session message to the rtcpd child process over the initial connection to the worker process. The worker process then closes all of its connections with the rtcpd child process.

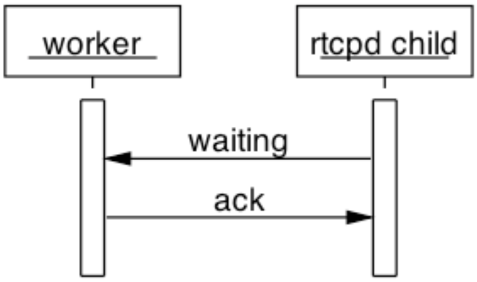
If an error occurred then an abort sequence is started. This sequence has been seen to fail and for the purpose of designing the tapebridged daemon a simpler solution has been devised. This document does not describe the broken abort message sequence between a worker process and an rtcpd child process.

The following sequence diagram describes the "transfer feedback" message sequence.


worker
rtcpd child
positioned for file (tape request)
ack
positioned for file (file request)
ack
file transfered
ack
flushed to tape
ack

For each file transferred the rtcpd child process sends four messages. The first two signal the tape has been positioned for the file transfer, the third states that the file has been written to the tape drive but not yet flushed to tape and the fourth states that the data buffers have been flushed to tape and the file is now safely stored on tape. The two messages signaling that the tape has been positioned are semantically equivalent but are sent using two different messages structures, the tape request structure and the file request structure. This document does not go into any more detail about these structures, as will be discussed later, if these two messages do not report any errors then they are acknowledged by the tapebridged daemon and nothing is relayed to the tapegatewayd daemon. In the case of them reporting an error they are acknowledge by the tapebridged daemon and the errors are then propagated to the tapegatewayd daemon.

The following sequence diagram shows a fifth type of message that the `rtcpd` child process can optionally send to the worker process. The message is called a waiting message and it gives the worker process a human readable text message explaining what the `rtcpd` process is currently waiting for. As will be explained later, if the waiting message does not represent an error then the `tapebridged` daemon will simply acknowledge it. If however it reports an error then the `tapebridged` daemon will acknowledge it and then send an error to the `tapegatewayd` daemon.



Design

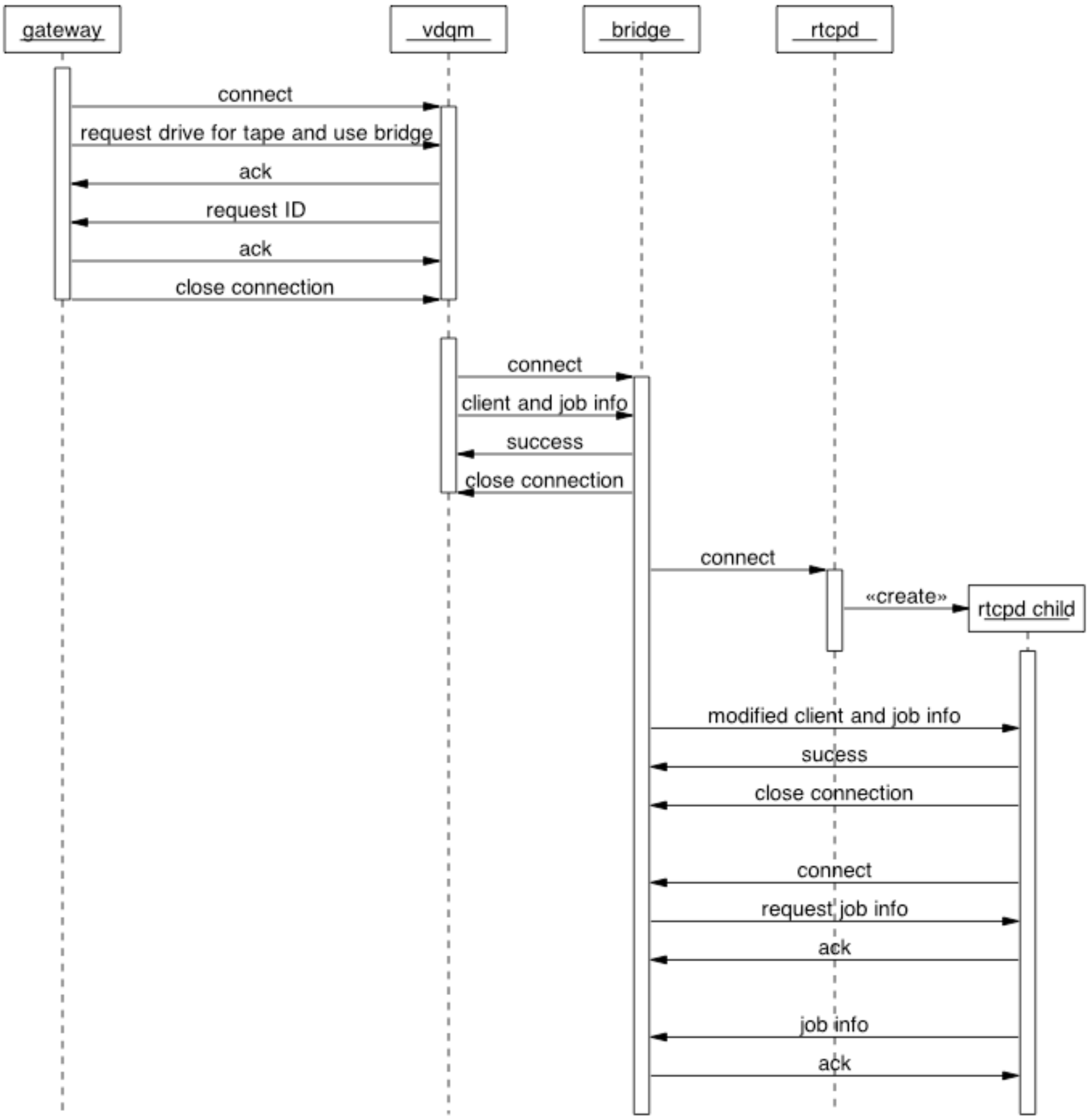
The structure of this section

The next subsection describes the message sequences that make up the new remote tape-copy protocol to be used between the `tapegatewayd`, `tapebridged`, `vdqmd` and `rtcpd` daemons. This protocol will hereafter be referred to as the `tapgatewayd` protocol. The next subsection lists all of the messages that can be sent from the `tapebridged` daemon to the `tapegatewayd` daemon together with the possible response messages. The next subsection gives the class diagram of the C++ classes that represent the messages sent between the `tapebridged` and `tapegatewayd` daemons. The last subsection gives an overview of the internal design of `tapebridged` daemon. The goal of the section is to give the reader enough information to be able to navigate comfortably through the source code, which has doxygen comments in the C++ header files.

The tapegatewayd protocol

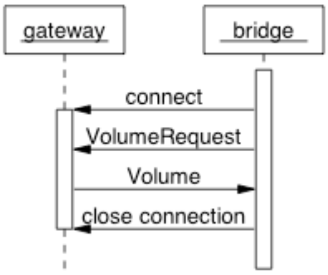
In order for a tape-server to support legacy stagers running the `rtcpclientd` daemon and newer stagers running the `tapegatewayd` daemon, a tape-server should provide direct access to the `rtcpd` daemon as well as access to the `tapebridged` daemon. Providing support for both the `rtcpd` and `tapegatewayd` protocols poses a problem for the `vdqmd` daemon when it wishes to start a remote tape-copy job on a tape-server with a free tape-drive. The `vdqmd` daemon needs to know whether it must contact the `rtcpd` daemon directly or whether it should contact the `tapegatewayd` daemon. The solution taken is to introduce a new type of client request that asks the `vdqmd` daemon for a free drive and to use the `tapebridged` daemon.

The tapebridged daemon acts as both the vdqmd and rtcpclientd daemons to the rtcpd daemon and its child process. The following sequence diagram shows how the tapegatewayd daemon requests and acquires a free drive to mount a tape. Please note that the name tapegatewayd has been shortened to gateway and the name tapebridged has been shortened to bridge in order to keep the sequence diagrams less cluttered.



Once the tapebridged daemon has finished sending the job information to the the rtcpd child process and the child process has reconnected to the tapebridged daemon ready for work, the tapegatewayd daemon then goes on to request the tapegatewayd daemon for the volume to be mounted and whether it should be mounted for read or write.

The following sequence diagram sequence shows the tapebridged daemon requesting the volume and access mode from the tapegatewayd daemon.

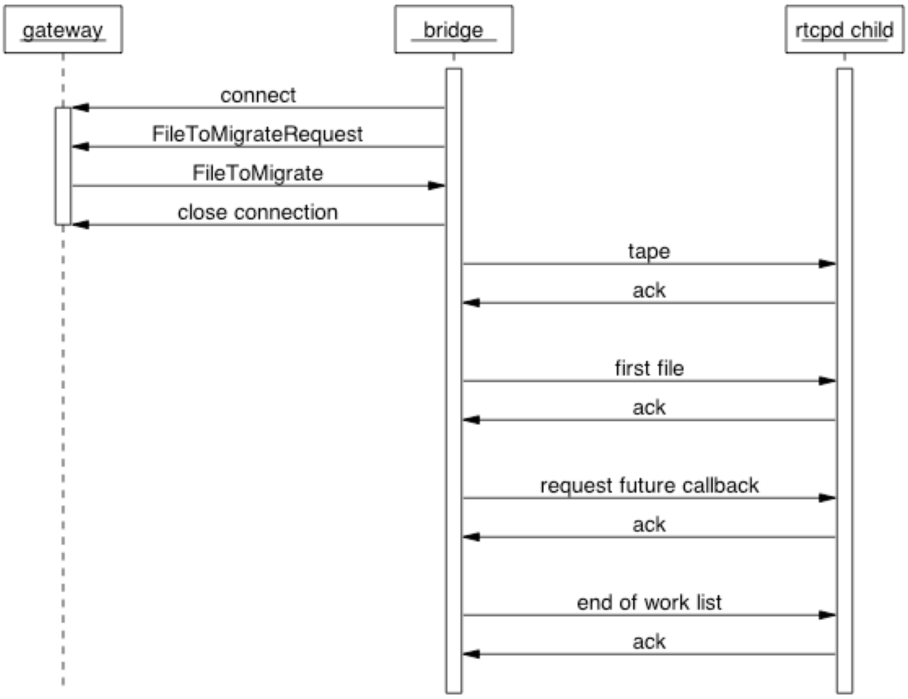


When designing the tapegatewayd protocol it was decided to create a new connection for each request made to tapegatewayd daemon. This decision was taken to keep open the possibility of distributing requests across many instances of the tapebridged daemon.

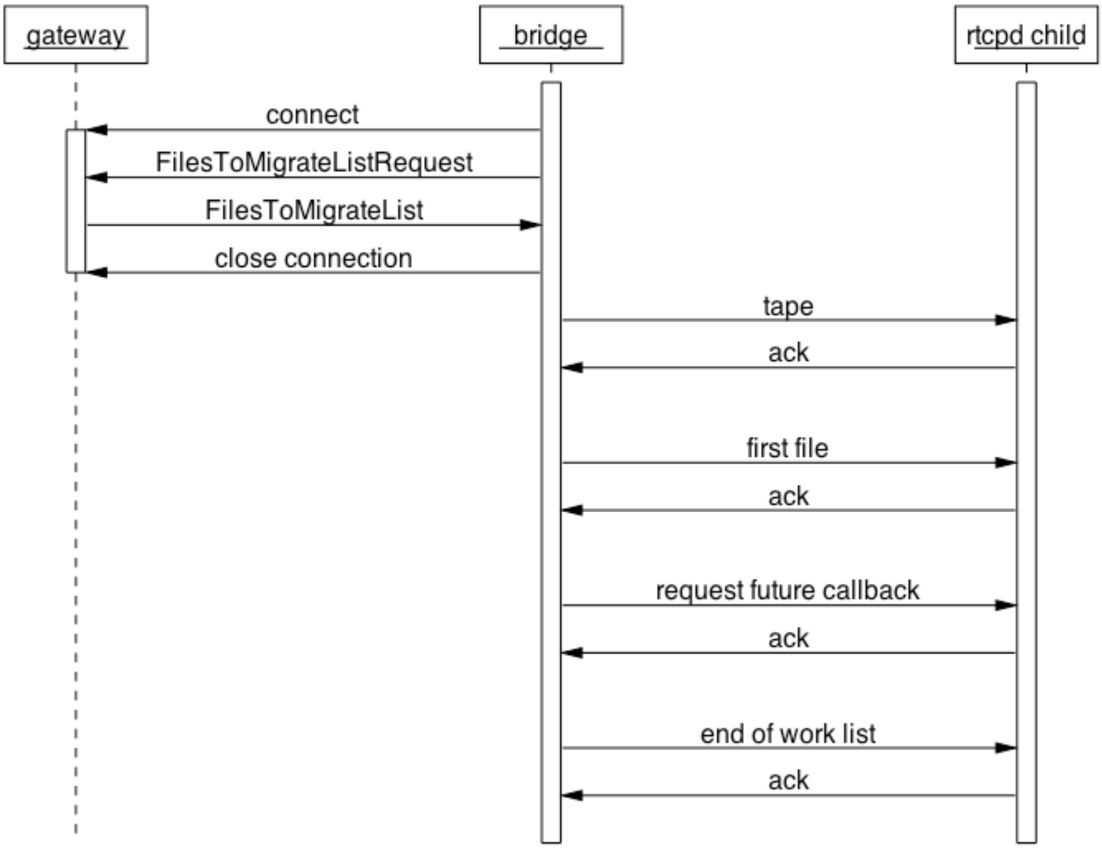
There are two versions of the tapegatewayd protocol. The first, hereafter referred to as the single-file protocol, is designed around the idea that a single request or feedback message refers to a single file. The second, hereafter referred to as the bulk-file protocol, is designed around the idea of a single request or feedback message referring to a group of files. The bulk-file protocol deprecates the single-file protocol and was introduced to help improve the efficiency of the interface to the relational-database responsible for keeping the state of the tapegatewayd daemon persistent.

A given version of the tapebridged daemon either supports only the single-file protocol or only the bulk-file protocol. There is no version of the tapebridged daemon that supports both protocols. Unlike the tapebridged daemon, the current version of the tapegatewayd daemon supports both the single and bulk file protocols. This dual protocol support facilitates the early deployment tests being carried out at CERN where we a mixture of legacy and up-to-date versions of the tapebridged daemons are deployed across the CERN tape-servers.

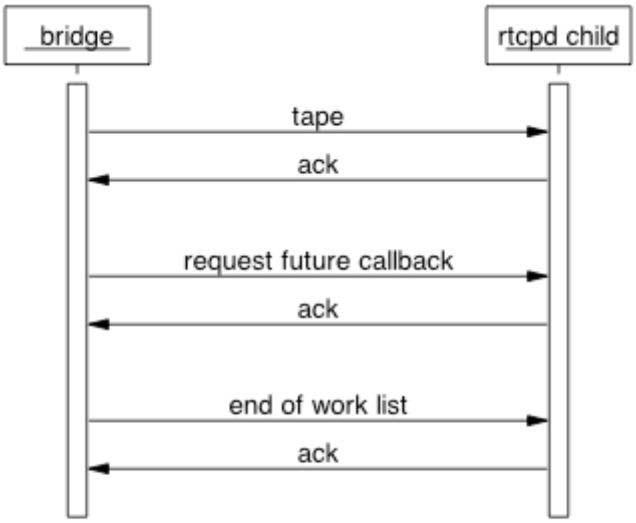
The following sequence diagram shows how a legacy version of the tapebridged daemon uses the deprecated single-file protocol to ask the tapegatewayd daemon for the first file to be migrated before sending the first work list to the rtcpd child process. The first work list consists of the tape-volume to be mounted, the first file to be migrated and a request for the rtcpd child process to ask for more work in the future.



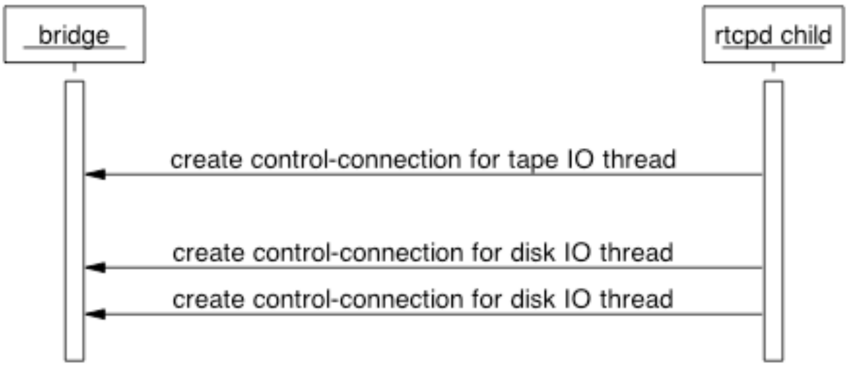
The following sequence diagram shows how the current version of the `tapebridged` daemon achieves the same goal using the bulk-file protocol. It should be noted that the current version of the `tapebridged` daemon uses a bulk-file request (a `FileToMigrateListRequest`) message to request one and only one file. A future version of the `tapegatewayd` daemon will ask for more than one file.



When recalling a file from tape to disk, the `tapebridged` daemon sends a work list containing only the tape-volume to be mounted and the request for a future callback, as shown by the following sequence diagram.



As already described in the analysis section, once the rtcpd child process knows which tape to mount it internally creates one tape IO thread for reading or writing files to tape and one or more disk IO threads for remotely reading or writing disk files using TCP/IP connections to the rfiod daemons running on the disk-servers. A separate control-connection between the rtcpd child process and the tapebridged daemon is made for each of these IO threads as shown by the next sequence diagram. Please note that the number of disk IO threads is configurable and the following sequence diagram assumes the rtcpd child process has two.

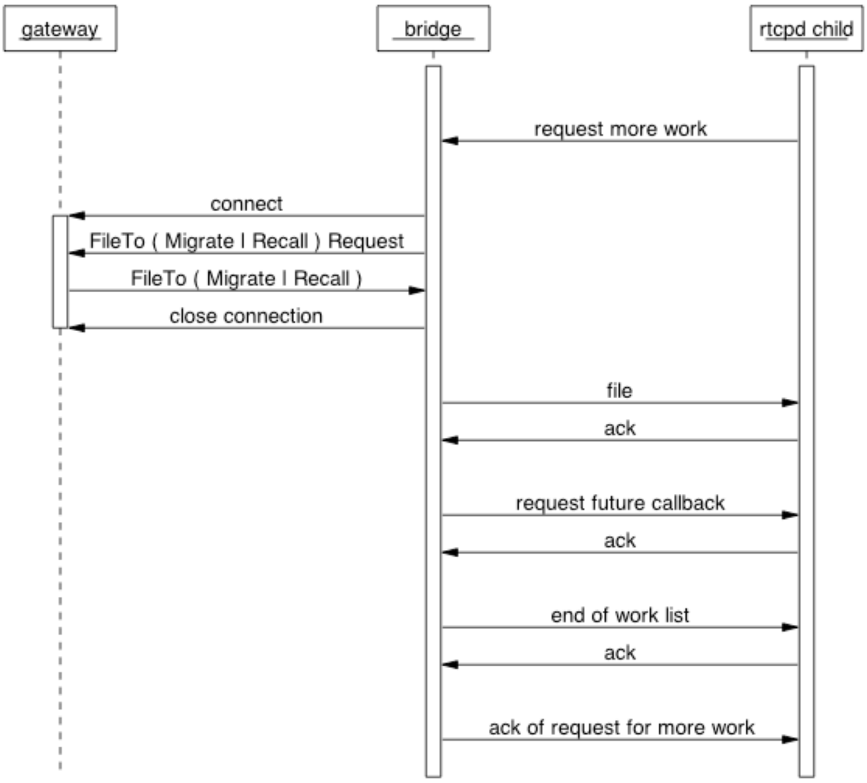

bridge
rtcpd child
create control-connection for tape IO thread
create control-connection for disk IO thread
create control-connection for disk IO thread

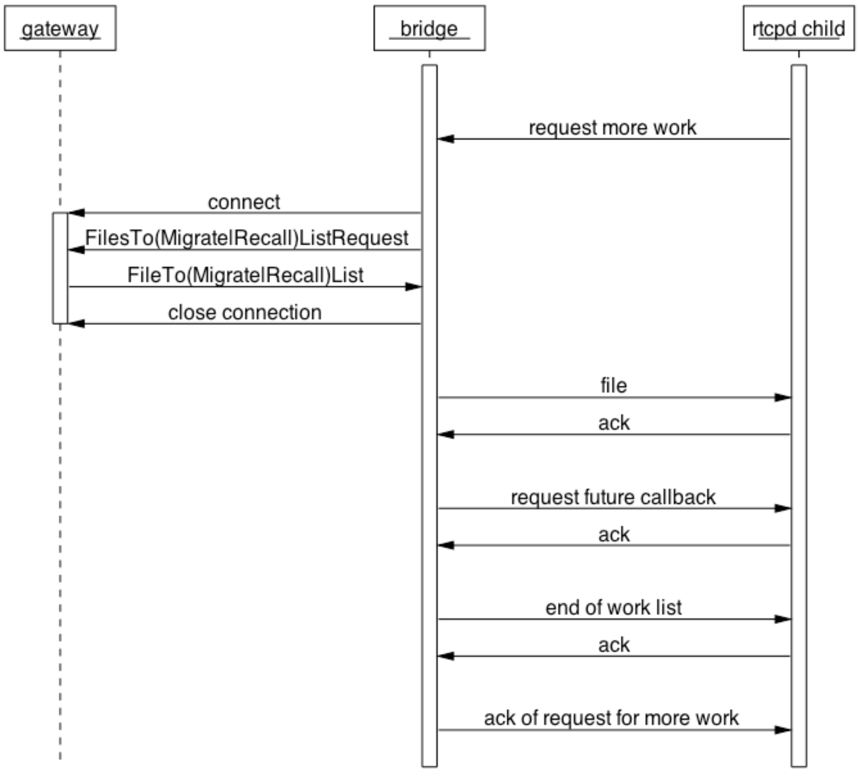
The multiple connections between the rtcpd child process and the tapebridged daemon are purposely hidden from the tapegatewayd daemon. Without going into too much detail the tapebridged daemon achieves this by using a select() loop over many connections and separating the sending of messages from the reception of their replies.

With the disk/tape IO control-connections made, the rtcpd child process enters the main loop of asking for the next file to be transferred.

The main loop of the new remote-tape copy protocol can be divided into two message sequences. The first, hereafter referred to as the “what’s next?” sequence, is the sequence of messages that control which file should be transferred next and when the transfer session of all the files to be transferred is finished. The second sequence of messages, hereafter referred to as the “transfer feedback” sequence, is the sequence of messages used to give feedback on the files being transferred to tape or to disk.

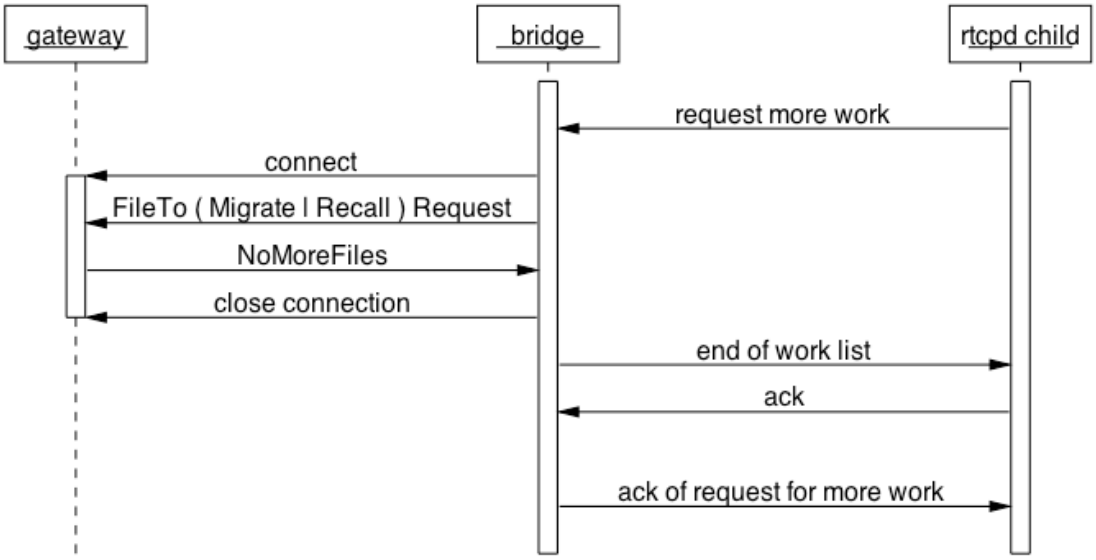
The following two sequence diagrams show the main "what's next?" loop, firstly implemented using the deprecated single-file protocol and secondly using the bulk-file protocol. Even though the current version the `tapebridged` daemon uses the bulk-file protocol, it uses the bulk-file request messages `FilesToMigrateListRequest` and `FilesToRecallListRequest` to request one and only one file at a time.

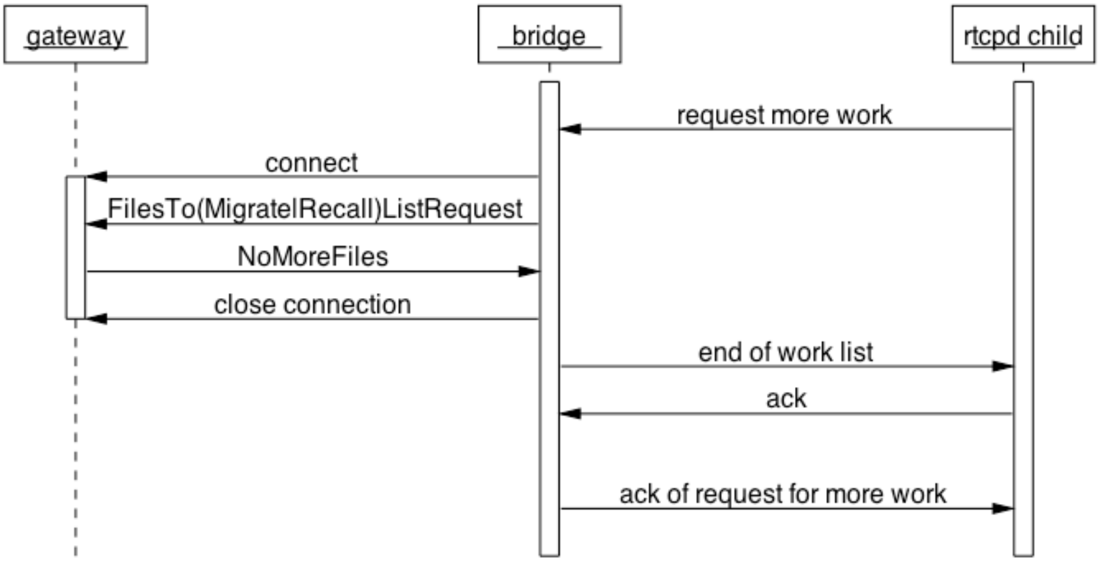




The rtcpd child process effectively loops requesting the next file to be transferred until there are no more files or an error is encountered, which as stated earlier could be generated by reaching the end of tape during a migration session. The tapebridged daemon translates each of these requests into a connection to the tapegatewayd daemon, followed by a request for either a file to migrate or a file to recall.

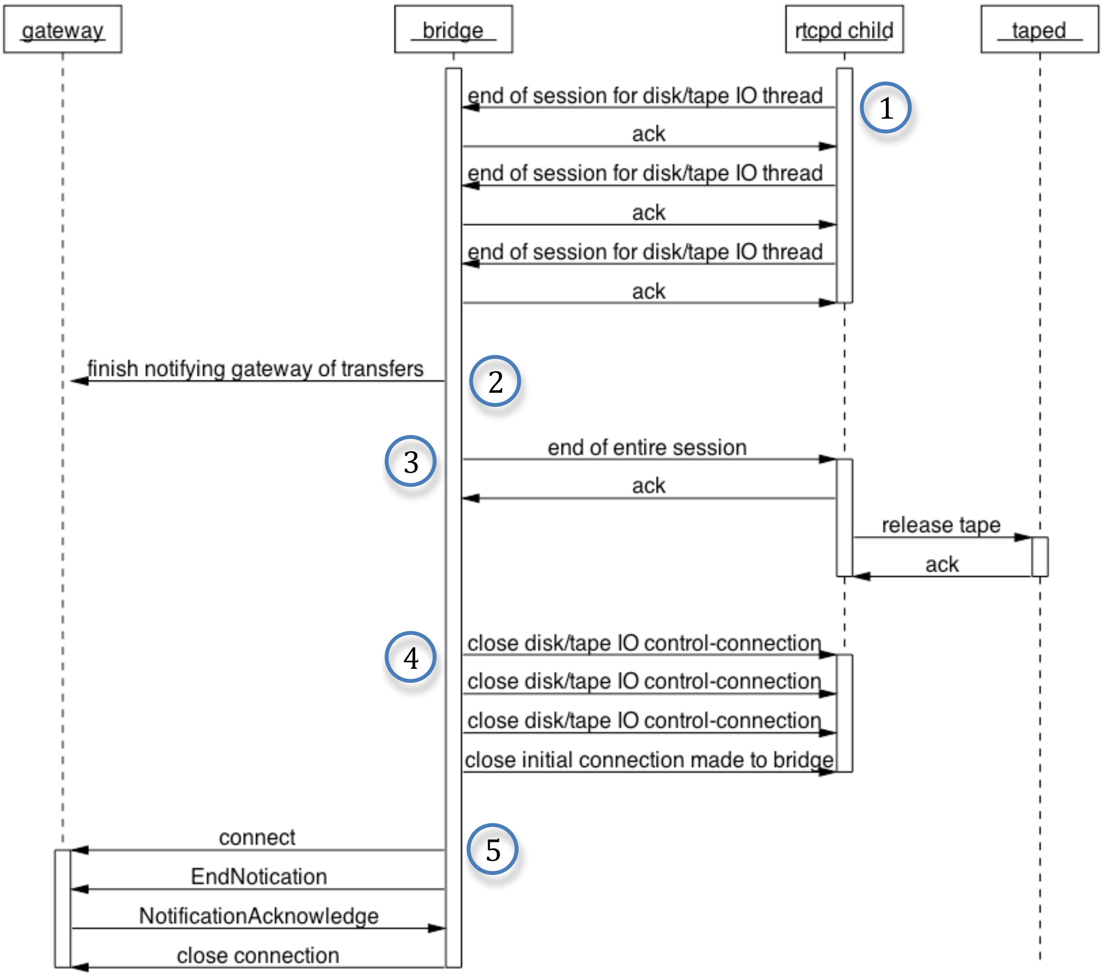
The following two sequence diagrams show how the shutdown sequence is started when the tapegatewayd daemon has no more files to be transferred. The first diagram shows the single-file protocol and the second shows the bulk-file protocol.





The rtcpd child process involuntarily starts the shutdown sequence by requesting more work, the tapebridged daemon relays this request to the tapegatewayd daemon, which having no more files to transfer, replies to with a NoMoreFiles message. The tapebridged daemon then sends an empty work list back to the rtcpd child process and acknowledges the request for more work.

The next sequence diagram shows that once the shutdown sequence is started, each of the disk/tape IO threads of the rtcpd child process send their own individual end of session messages as they each finish the work they were assigned to do.


gateway
bridge
rtcpd child
taped
end of session for disk/tape IO thread
ack
end of session for disk/tape IO thread
ack
end of session for disk/tape IO thread
ack
1
finish notifying gateway of transfers
2
3
end of entire session
ack
release tape
ack
4
close disk/tape IO control-connection
close disk/tape IO control-connection
close disk/tape IO control-connection
close initial connection made to bridge
connect
5
EndNotication
NotificationAcknowledge
close connection

The above sequence diagram has been divided up into 5 steps in order to help explain the details.

At step 1 the tapebridged daemon receives the end of session messages from each of the disk/tape IO threads of the rtcpd child process.

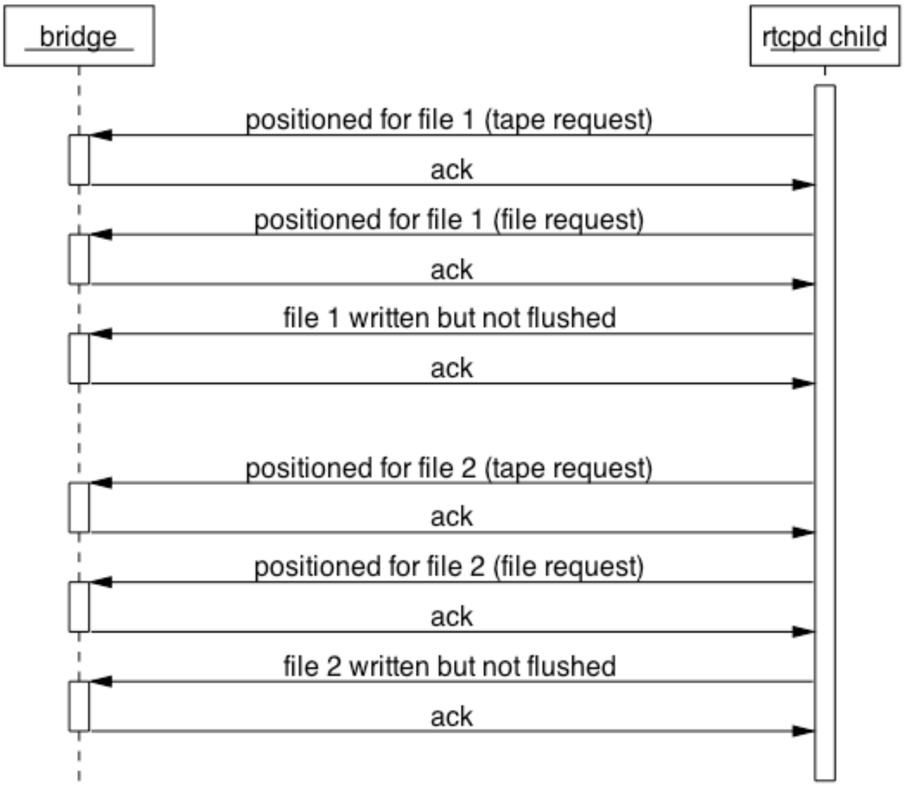
At step 2 the tapebridged daemon makes sure it has notified the tapegatewayd daemon of all of the file transfers that have taken place. This may take some time if the tapegatewayd daemon is slow for any reason.

At step 3 the tapegatewayd daemon has been notified of all file transfers and the tapebridged daemon can go ahead and send the rtcpd child process the end of entire session message. In response the rtcpd child process tells the taped daemon to release the tape. It is very important to hold back the release of the tape until the tapegatewayd daemon has been notified of all the file transfers. The tapegatewayd daemon may take a relatively long period of time to record all of the file transfers and therefore take a relatively long period of time to acknowledge the end of the tape session. By releasing the tape right at the end of the session avoids the vdqmd daemon being notified by the taped daemon of the tape-drive becoming free before the tapegatewayd daemon considers the session complete. The tapegatewayd daemon does not consider a session complete until it has finished processing all of the file transfer notifications and has received and acknowledged the end of session message (EndNotification) from the tapebridged daemon.

At step 4 the tapebridged daemon makes sure all of its connections with the rtcpd child process are closed.

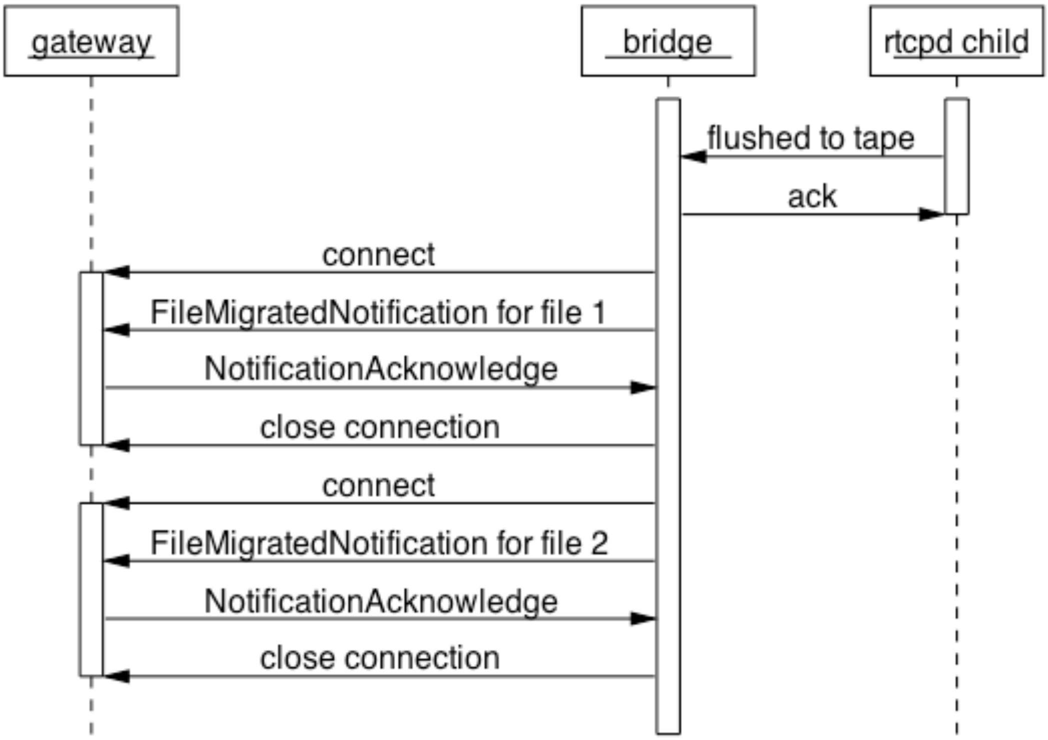
At step 5 the tapebridged daemon notifies the tapegatewayd daemon that the end of the session has been reached. The tape-session is now finished for all three parties: tapegatewayd, tapebridged and the rtcpd child process.

The following sequence diagram shows the start of the "transfer feedback" message sequence when migrating files from disk to tape.


bridge
rtcpd child
positioned for file 1 (tape request)
ack
positioned for file 1 (file request)
ack
file 1 written but not flushed
ack
positioned for file 2 (tape request)
ack
positioned for file 2 (file request)
ack
file 2 written but not flushed
ack

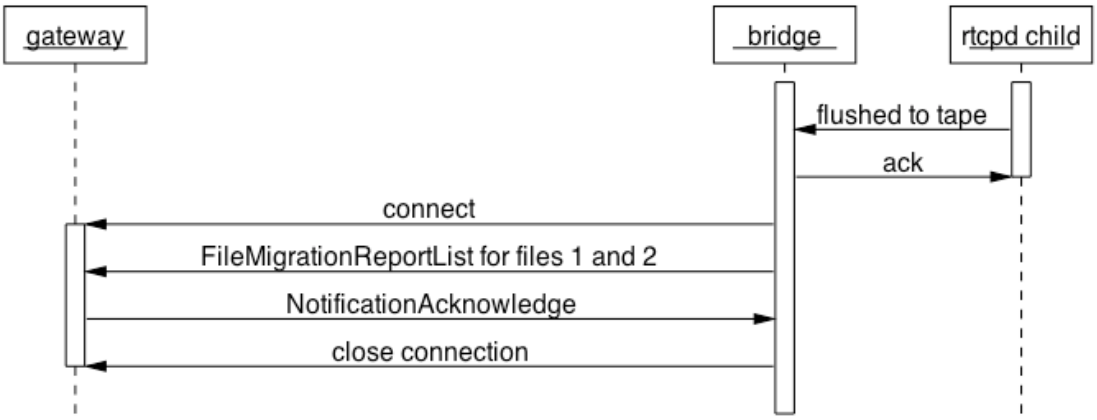
The `rtcpd` child process sends three messages to the `tapebridged` daemon for each file it writes to the tape-drive but has not yet flushed to tape. The first two messages indicate the tape was positioned and are semantically the same. The two messages are structurally different; one is based on a notification message concerning a tape and the other on a file. The third message states the file was written to the tape-drive. The `tapebridged` daemon acknowledges the two tape-positioned messages. If either of the position messages contains an error report then that error report is relayed to the `tapegatewayd` daemon. If the position messages do not contain any error reports then they are not relayed to the `tapegatewayd` daemon. The "file written but not flushed" message of each file is held back within the `tapebridged` daemon until a flushed to tape message is received from the `rtcpd` child process. The `rtcpd` child process flushes the data buffers between itself and the tape based on its run-time configuration. A tape operator can specify both the maximum number of files and the maximum number of bytes to be written to the tape-drive before the data buffers are flushed. The maximum that is reached first will cause the `rtcpd` child process to trigger the flush. Please note that in the case of the maximum number of bytes triggering the flush, the flush will occur on a file boundary and therefore more bytes than specified will normally be written to tape before the actual flush occurs.

The following sequence diagram shows the end of the "transfer feed back" message sequence of the singe-file protocol when migrating files from disk to tape.



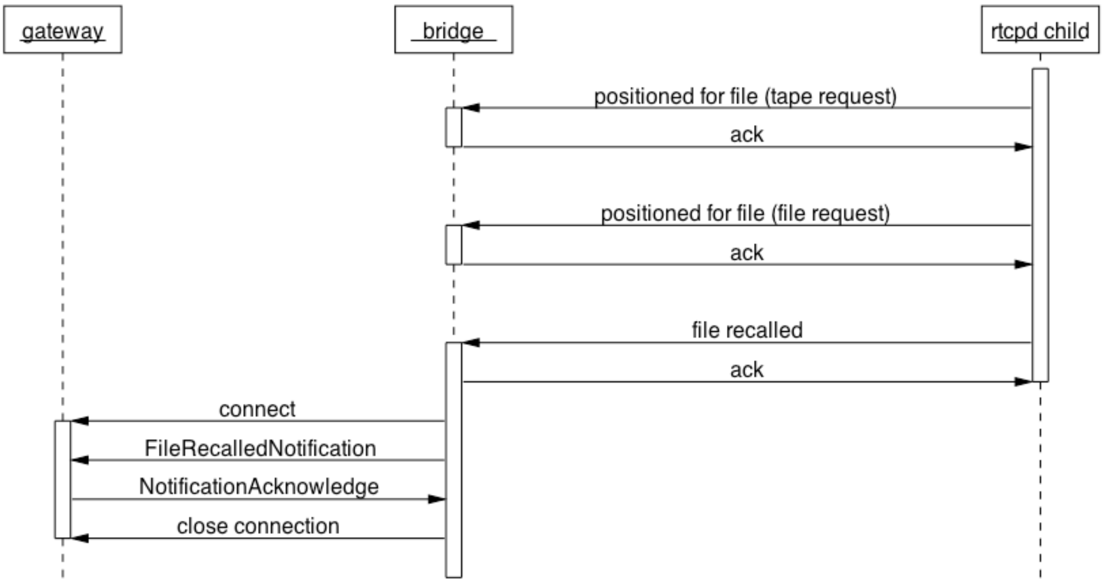
When the `tapebridged` daemon has acknowledged the flush of the data buffers to tape, the daemon starts sending the list of files associated with the flush to the `tapegatewayd` daemon. With the single-file protocol, the list is sent as a series of `FileMigratedNotification` messages with one message for each file written and flushed to tape. The `tapebridged` daemon purposely serializes the individual file transfer notifications it sends to the `tapegatewayd` daemon and keeps the notifications in ascending order of tape-file sequence number. The tape-file sequence number gives the position of a file on tape. Keeping the file notification in ascending order of tape-file sequence number enables the `tapegatewayd` daemon to update the tape-file sequence number in the `vmgrd` daemon on a consecutive file-by-file basis.

The following sequence diagram shows the end of the "transfer feedback" message sequence of the bulk-file protocol when migrating files from disk to tape.



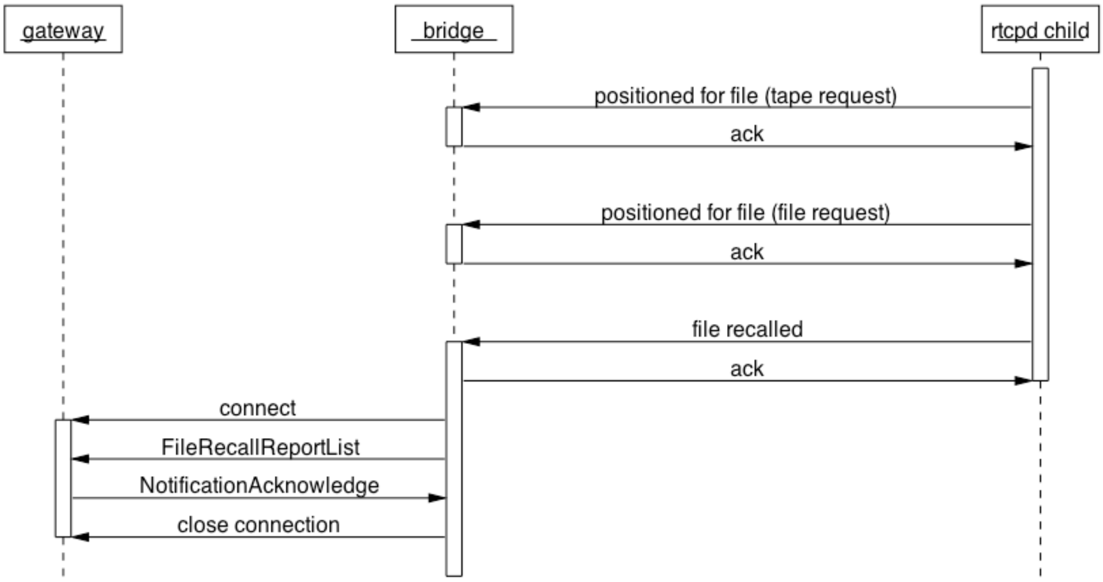
The bulk-file protocol differs from the single-file protocol in that the tapebridged daemon does not send an individual notification message to the tapegatewayd daemon for each individual file written and flushed. Instead the tapebridged daemon sends a single FileMigrationReportList message for all of the files associated with the same flush of buffers to tape. If there is more than one FileMigrationReportList message to be sent to the tapegatewayd daemon then the tapebridged daemon ensures they are sent one after the other and in ascending order of tape-file sequence-number. There maybe more than one FileMigrationReportList message to be sent because a backlog of these messages will effectively build up in the tapebridged daemon if the tapegatewayd daemon is slower to process the messages than the rtcpd daemon is able to write and flush files to tape.

The following sequence diagram shows the complete "transfer feedback" message sequence of the single-file protocol when recalling a file from tape.



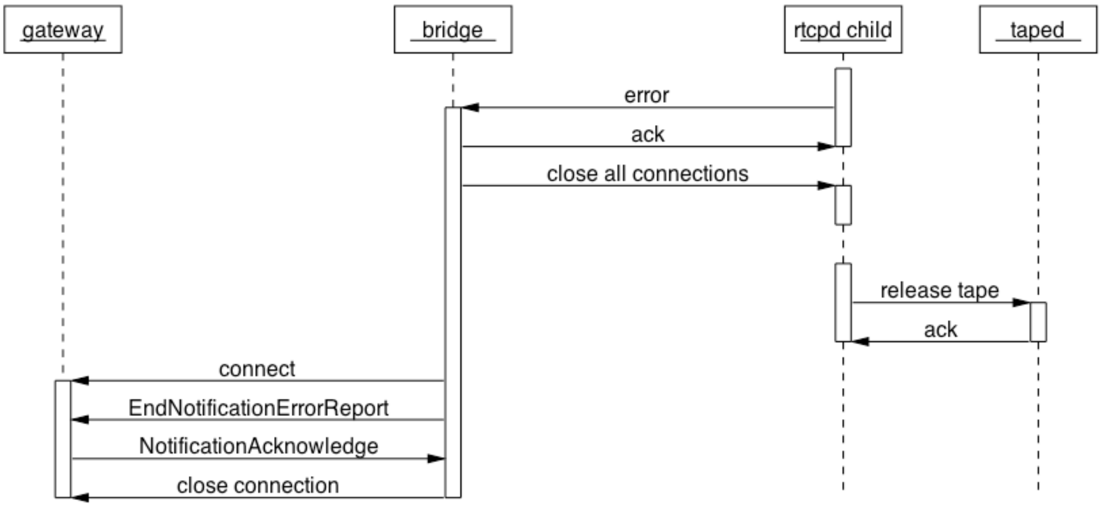
When recalling files from tape there is no concept of flushing data buffers to tape. The tapebridged daemon therefore does not hold back any file-recalled notifications from the `tapegatewayd` daemon. When the `tapebridged` daemon receives a file-recalled message from the `rtcpd` child process it immediately notifies the `tapegatewayd` daemon.

The following sequence diagram shows the complete "transfer feedback" message sequence of the bulk-file protocol when recalling a file from tape.



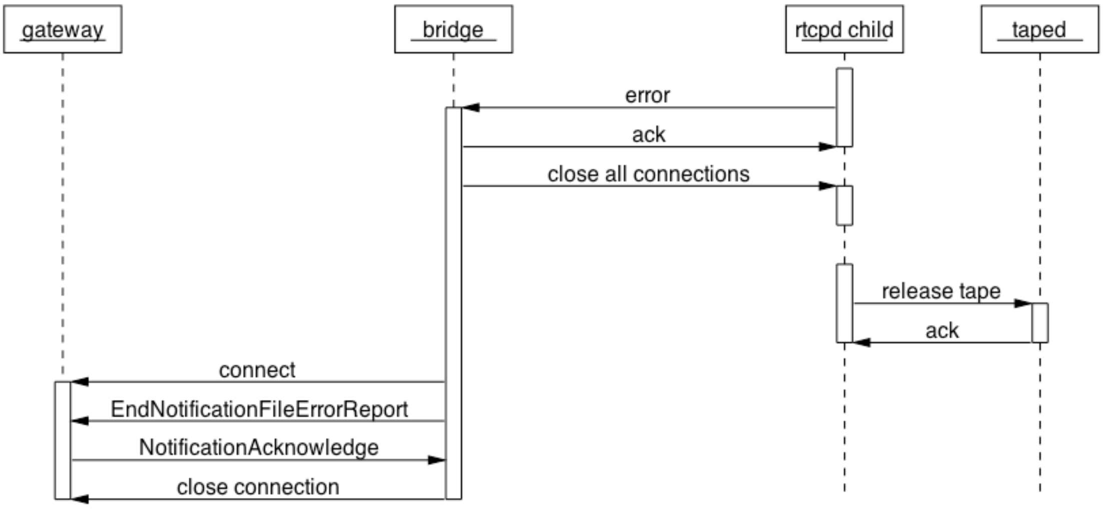
The bulk-file protocol is the same as the single-file protocol except for the fact that the `tapebridged` daemon sends a `FileRecallReportList` message to the `tapegatewayd` daemon instead of a `FileRecalledNotification` message. The current version of the `tapebridged` daemon sends one and only one file per `FileRecallReportList` message.

The following sequence diagram shows the shutdown sequence of the single-file protocol in the event of a non file-specific error when migrating files from disk to tape.

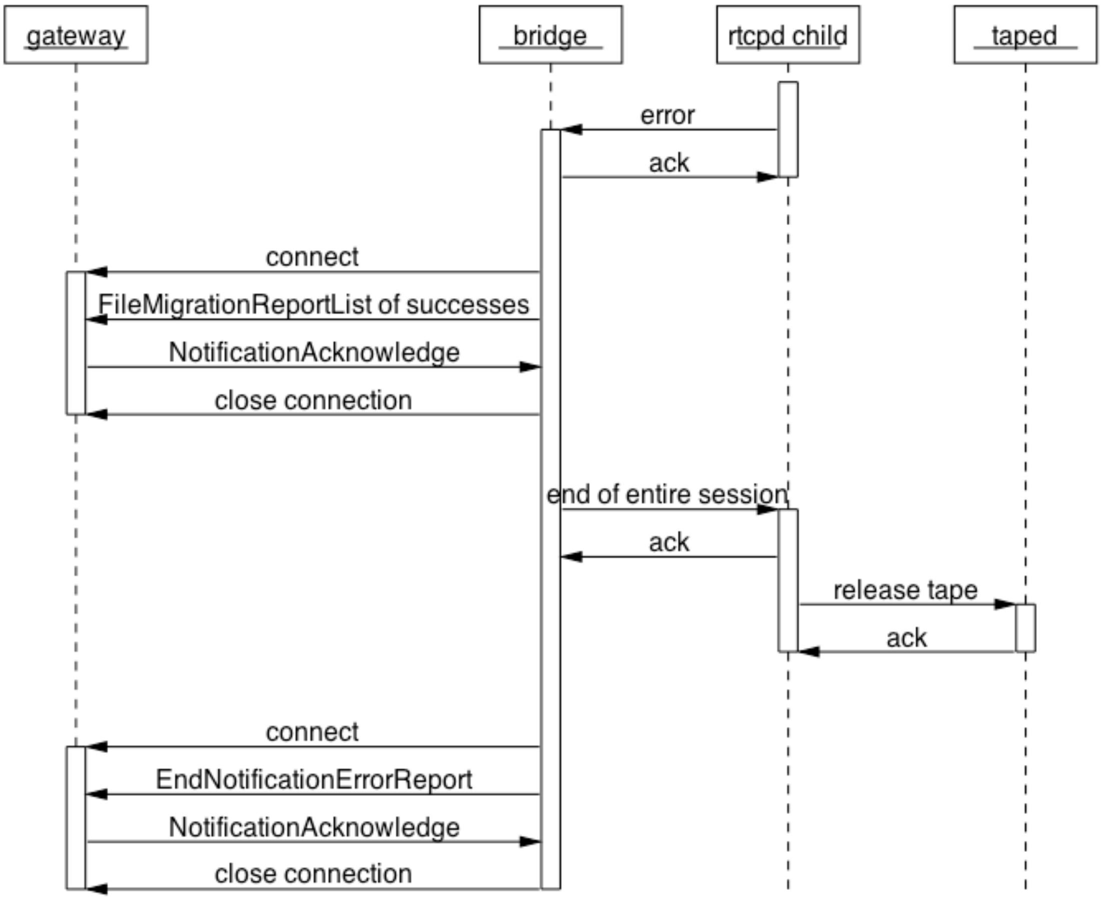


When an error is sent to the tapebridged daemon or the daemon detects an error within itself, the daemon immediately closes all of its connections with the rtcpd child process and notifies the tapegatewayd daemon. The rtcpd child process detects the closing of its connections with the tapebridged daemon and starts shutting down. During the shutdown logic of the rtcpd child process, the rtcpd child process requests the taped daemon to release the tape from the tape-drive.

The following sequence diagram shows the shutdown sequence of the single-file protocol in the event of a file-specific error when migrating files from disk to tape. The sequence is the same as that of a non file-specific error except the tapebridged daemon sends the tapegatewayd daemon an EndNotificationFileErrorReport message instead of an EndNotificationErrorReport message.

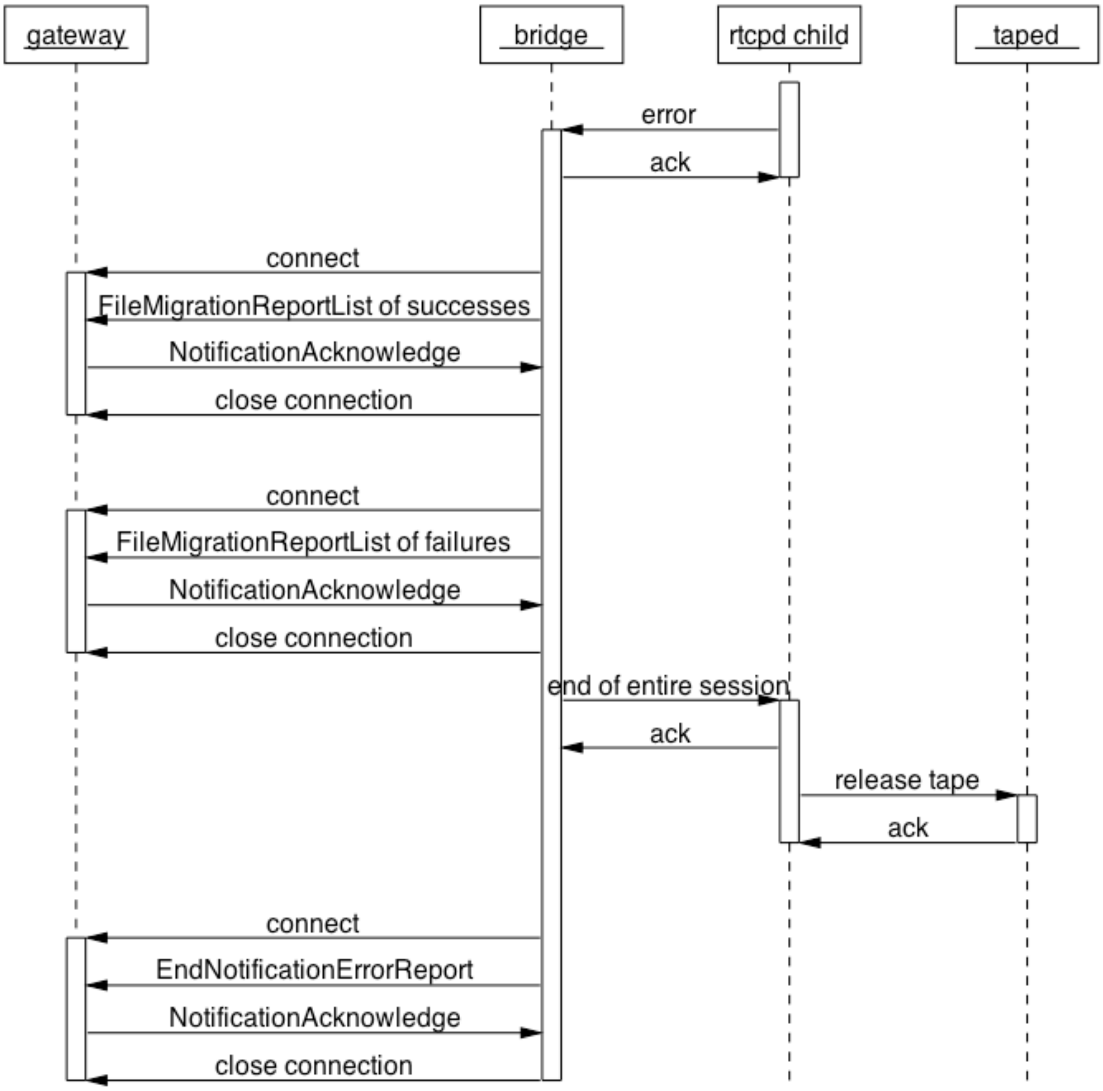


The following sequence diagram shows the shutdown sequence of the bulk-file protocol in the event of a non file-specific error when migrating files from disk to tape.



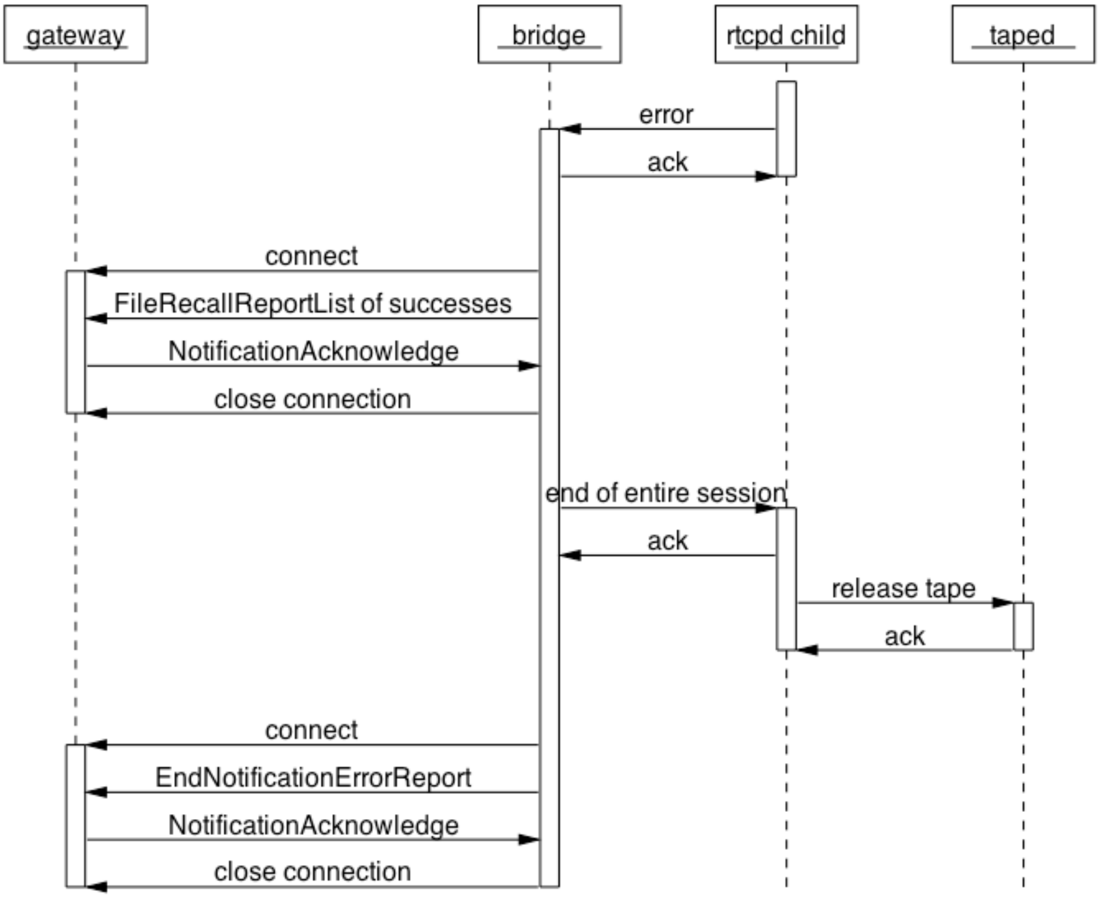
After receiving and acknowledging an error from the `rtcpd` client process, the `tapebridged` daemon sends as many `FileMigrationReportList` messages as is necessary to report on all of the files that have been successfully written and flushed to tape. Once the `tapegatewayd` daemon has been notified of all the successful transfers, the `tapebridged` daemon sends the “end of entire session” message to the `rtcpd` child process. The `rtcpd` child process in turn requests the `taped` daemon to release the tape from the tape-drive.

The following sequence diagram shows the shutdown sequence of the bulk-file protocol in the event of a file-specific error when migrating files from disk to tape.

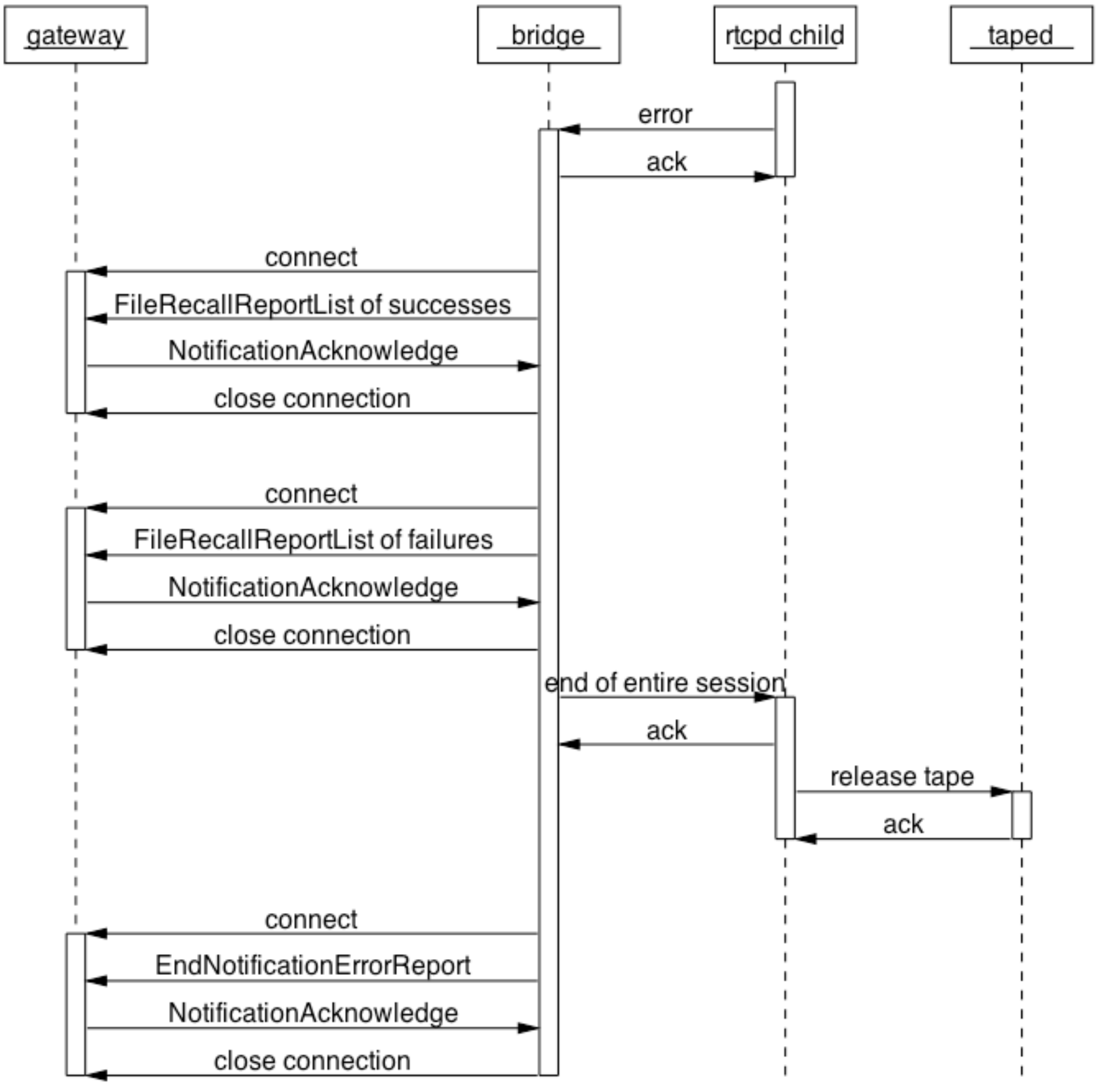


There is one difference in the bulk-file protocol between the shutdown sequence caused by a non file-specific error and one caused by a file-specific error. In the latter case there is the additional step of sending a `FileMigrationReportList` containing the file error that caused the shutdown sequence to begin. This message is sent by the `tapebridged` daemon to the `tapegatewayd` daemon immediately after the `FileMigrationReportList` message containing the last set of successful migrations and immediately before the `tapebridged` daemon sends the "end of entire session" message to the `rtcpd` child process. The `tapebridged` daemon only sends the first file error that was detected.

The following sequence diagram shows the shutdown sequence of the bulk-file protocol in the event of a non file-specific error when recalling files from tape to disk.



The following sequence diagram shows the shutdown sequence of the bulk-file protocol in the event of a file-specific error when recalling files from tape to disk.



The request and response messages of the tapegatewayd protocol

The following table lists each message of the single-file `tapegatewayd` protocol that can be sent from the `tapebridged` daemon to the `tapegatewayd` daemon together with the possible reply messages.

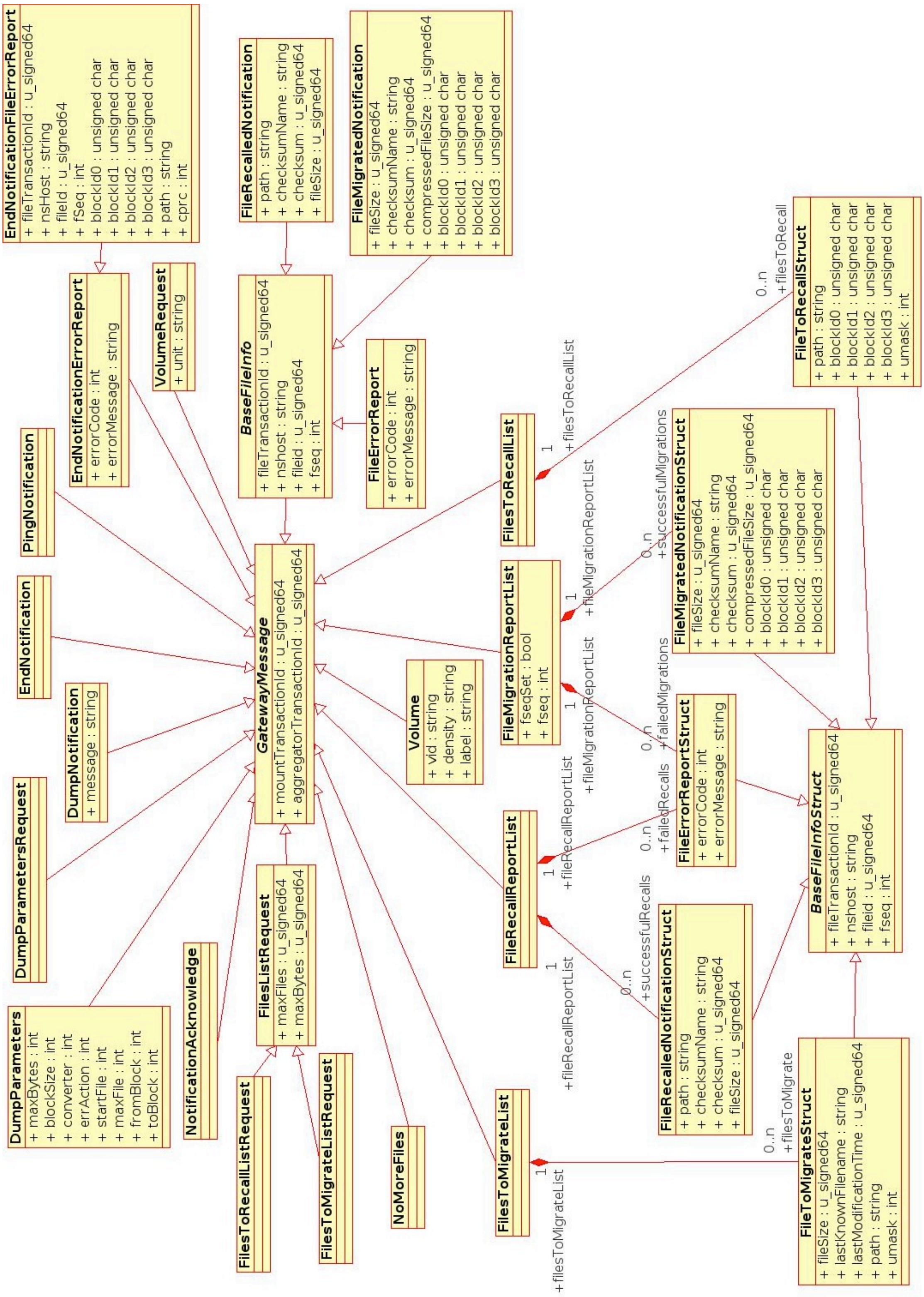
Bridge to Gateway	Gateway to Bridge	Comment
VolumeRequest	Volume	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileToMigrateRequest	FileToMigrate	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileToRecallRequest	FileToRecall	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileMigratedNotification	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileRecalledNotification	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount
EndNotification	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileErrorReport	NotificationAcknowledge	Bad day – Abort file
	EndNotificationErrorReport	Bad day – Abort mount
EndNotificationErrorReport	NotificationAcknowledge	Bad day – Abort mount
	EndNotificationErrorReport	Bad day – Abort mount
EndNotificationFileErrorReport	NotificationAcknowledge	Bad day – Abort mount
	EndNotificationErrorReport	Bad day – Abort mount
DumpNotification	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount

The following table lists each message of the bulk-file `tapegatewayd` protocol that can be sent from the `tapebridged` daemon to the `tapegatewayd` daemon together with the possible reply messages.

Bridge to Gateway	Gateway to Bridge	Comment
VolumeRequest	Volume	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileToMigrateListRequest	FilesToMigrateList	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileToRecallListRequest	FilesToRecallList	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileMigratedReportList	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileRecallReportList	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount
EndNotification	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount
EndNotificationErrorReport	NotificationAcknowledge	Bad day – Abort mount
	EndNotificationErrorReport	Bad day – Abort mount
DumpNotification	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount

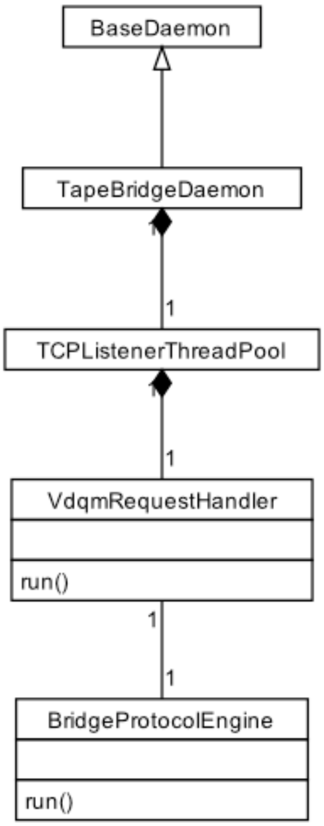
Class diagram of the tapegatewayd message classes

The following class diagram shows the C++ classes that represent the messages of the `tapegatewayd` protocol. Both the single and bulk file messages are included.



The internal design of the tape-bridge daemon

The following class diagram shows the main classes making up the static structure of tapebridged.



The C++ class named TapeBridgeDaemon represents tapebridged. The class uses the CASTOR framework by inheriting from BaseDaemon.

TapeBridgeDaemon assigns one thread to each concurrent tape mount, as a single tape-server may be connected to more than one tape-drive. TapeBridgeDaemon implements this threading strategy by containing a TCPListenerThreadPool of VdqmRequestHandlerThreads. The number of threads in the pool is set to the number of drives found in the following configuration file on the tape-server:

```
/etc/castor/tpconfig
```

When the vdqm connects to tapebridged, the TCPListenerThreadPool gives the newly accepted connection to the run() method of one of the contained VdqmRequestHandler threads. The thread in question then communicates with both rtcpd and the client (tapegatewayd, readtp, writetp or dumptp) to the point where tapebridged has just got the tape-volume to be mounted from the client.

Now that the VdqmRequestHandler thread being used for the mount knows the tape-volume to be mounted it then calls the run() method a helper BridgeProtocolEngine object to iterate through the main loop of the new remote tape-copy protocol. The run() method effectively results in the VdqmRequestHandler thread spinning in a TCP/IP select() loop until the file transfer session is completed by either tapegatewayd indicating there are no more files to be transferred or by an error occurring which includes the physical end of tape being reached in the case of files being migrated from disk to tape.

The BridgeProtocolEngine uses the select() loop to decouple the sending of requests for more work to tapegatewayd from reading back the replies. This decoupling is crucial in gaining good performance with the design choice of one VdqmRequestHandler thread per ongoing tape-mount. Requests from the individual disk/tape IO threads of an rtcpd child process must not be blocked by the relatively slow responses of tapegatewayd to requests for more work.

The end of this document has now been reached as the main classes making up the static structure of tapegatewayd have been described.