October 2010

Tape-bridge Requirements, Analysis and Design

Author: Steven Murray

CERN

Table of Contents

Introduction

The purpose of this document and the intended audience

This document attempts to describe in one place the tape-bridge daemon from problem statement, to analysis through to design. This document should be read by developers responsible for maintaining the tape-bridge and by developers working on the tape-gateway daemon who need a detailed description of the interface between the tape-gateway and tape-bridge daemons.

The structure of this document

The next section of this document describes the requirements that are addressed by the tape-bridge daemon. The following section analyses the problem further by describing the legacy remote tape-copy protocol that the tape-bridge daemon will need to both interact with and replace. The design section follows with the description of the new remote-tape protocol that the tape-bridge daemon will introduce into CASTOR in order to carry out its tasks. The design section then finishes by describing the internal design of the tape-bridge daemon.

Requirements

A typical CASTOR installation can be divided into four types of server: central, stager, disk and tape. 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 CASTOR daemons responsible for accessing that data. A tape-server is physically connected to one or more tape-drives and runs CASTOR daemons responsible for accessing tapes.

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 `rtcpd`. One instance of `rtcpd` 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 `rtcpclientd`. There is one instance of `rtcpclientd` per stager, meaning that one instance of `rtcpclientd` runs on one and only one of the possibly many stager-servers that make up a CASTOR stager.

The tape hardware cost of CASTOR is currently considered to be too high due to the relatively inefficient write performance of CASTOR. 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 cached 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. 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. It has been demonstrated that tape write-performance can be significantly improved by writing many small files using buffered tape-marks and then making a slower but safer flushed tape-mark.

Major modifications would need to be made to `rtcpclientd` in order to implement the idea of buffered tape-marks being enclosed between relatively longer period flushed tape-marks. The protocol used between `rtcpclientd` and `rtcpd` would need to be modified such that `rtcpclientd` would veto flushed tape-marks until a reasonable amount of data had been written to tape. `Rtcpclientd` would also have to be modified so that it only updated the CASTOR namespace for a group of small files once they had all been written to tape and most importantly a flushed tape-mark had been written behind them.

Unfortunately the code maintenance costs of `rtcpclientd` have reached the point where it is impractical to add such a new major feature. To make matters worse, it has also been requested that `rtcpclientd` be modified to keep all of its state in a database so that the daemon itself could be re-started without losing state. Given these two major feature requests and the high maintenance costs of `rtcpclientd` it has been decided to replace `rtcpclientd` with a new daemon called the tape-gateway daemon, hereafter referred to as `tapegatewayd`.

Three major constraints were imposed on the development of `tapegatewayd`. Firstly `rtcpd` should not be significantly modified due to its critical role in the data path to tape. Secondly it must be easy to switch from using `rtcpclientd` to using `tapegatewayd` and vice averse. Thirdly a tape-server should be compatible with stager-servers running `rtcpclientd` and with stagers running `tapegatewayd`. The rational behind this latter constraint is the CASTOR tape system is shared by all of stagers of a site and the stagers are upgraded one by one to reduce the risk of a site-wide failure. As a result of the problem statement and these three constraints the tape-bridge daemon, hereafter referred to as `tapebridged`, was invented. `Tapebridged` will allow a complete re-design of the remote tape-copy protocol to live alongside the legacy protocol. One instance of `Tapebridged` will run on each tape-server and will act as a bridge between the new remote tape-copy protocol of `tapegatewayd` and the legacy protocol of `rtcpd`.

Analysis

The structure of this section

This section describes the legacy remote tape-copy protocol used mainly between `rtcpclientd` and `rtcpd`. This section is divided into five subsections including this one. The next and second subsection describes the additional CASTOR daemons that participate in the legacy remote tape-copy protocol. With the daemons described the third subsection explains how `rtcpclientd` obtains a tape to write to. The tape required when recalling is known in advance. The actual work of coordinating the transfer of files from disk to tape or from tape to disk is carried out be a pair of child processes. One of the pair is called the worker process and is forked and execed by `rtcpclientd` on a stager-server. The other process of the pair is a child process forked by `rtcpd` on a tape-server. The fourth subsection explains when in the legacy remote tape-copy protocol these two processes are created. The fifth and final subsection explains how an `rtcpclientd` worker process and an `rtcpd` child process work together to coordinate the remote tape-copy of files from disk to tape or from tape to disk.

The additional daemons of the legacy remote tape-copy protocol

Before designing `tapebridged` it was necessary to reverse engineer the existing remote tape-copy protocol used between `rtcpclientd` and `rtcpd`. In order to describe this protocol it is first necessary to briefly describe the following additional daemons:

- Volume and drive queue manager daemon (`vdqm`)
- Volume manager daemon (`vmgr`)
- Remote file access daemon (`rfiod`)
- Tape daemon (`taped`)

The `vdqm` is a site-wide drive scheduler that runs on the CASTOR central-servers. The `vdqm` supports redundancy by allowing more than one instance of itself to be ran across the CASTOR central-servers. The VDQM allocates free tape-drives to compatible mount requests. Free tape-drives are matched with mount requests via device-group-names (DGNs). A DGN is a unique string identifier used to map tape libraries to physically compatible tape-drives. For a drive to be physically compatible with a tape-library it must be located within the same tape-silo and must be able to both read and write all of the tapes within that tape-library.

The `vmgr` is a site-wide tape-catalogue that runs on the CASTOR central-servers. Unlike the `vdqm`, only one instance of the `vmgr` can be ran at a site. The information stored by the `vmgr` about each tape includes the used and free space, the tape-library where the tape is located and the logical tape-pool to which the tape belongs. The tape-library of a tape indirectly gives the DGN of that tape, which in turn is used by the `vdqm` 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 `vmgr` does not store any information about the individual files stored on a tape. The CASTOR name server, which is not described any further by this document, stores this information.

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

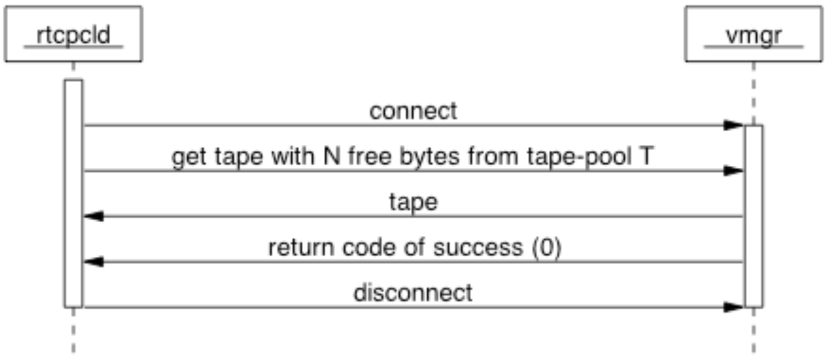
Each tape-server runs an instance of `taped`. This daemon is responsible for mounting, positioning and unloading tapes and for configuring tape-drives. The `taped` daemon listens on a TCP/IP port for mount, position and unload requests. When a request is received, the daemon forks and execs a helper process to service the request. The helper processes used by the `taped` daemon include `confdrive`, `mounttape`, `posovl` and `rlstape`.

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

Getting a tape to write to

It is the responsibility of `rtcpcld` 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 migration an appropriate tape needs to be found.

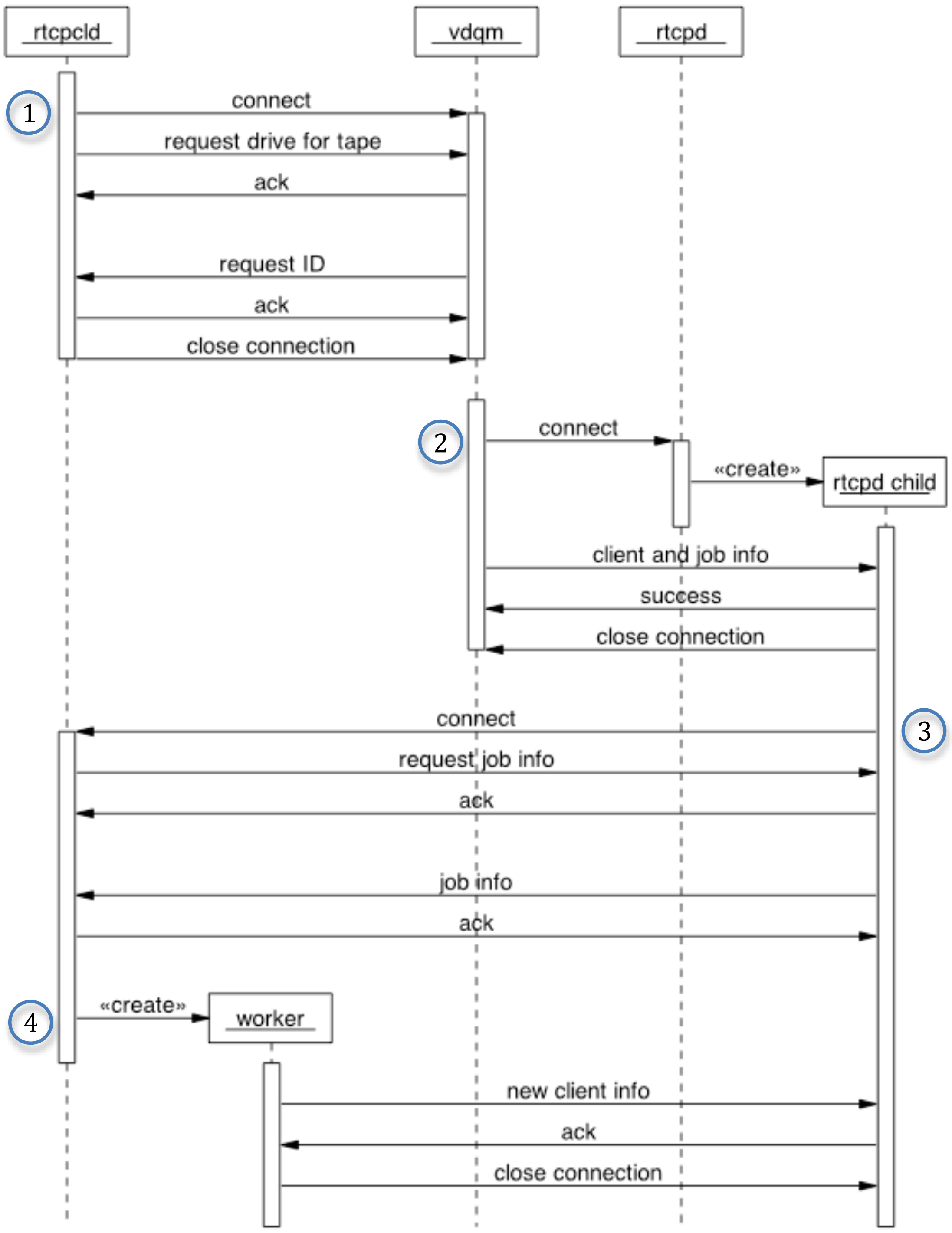
The following sequence diagram describes how `rtcpclientd` asks the `vmgr` for such a tape to write to.


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

`Rtcpclientd` connects to the `vmgr` via TCP/IP and sends a message requesting a tape that can be written to and that satisfies two conditions. The tape should have a specified amount of free space and it must belong to a specified tape-pool. In response to the request, the `vmgr` 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 `rtcpclientd`.

Starting an rtcpclientd worker process and an rtcpd child process

The following sequence diagram shows how `rtcpclientd` asks for a tape to be mounted and when `rtcpd` and `rtcpclientd` will fork their child processes to coordinate the remote tape-copy of one or more files.



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

At step 1 rtcpclientd has determined that a tape needs to be mounted for either migration or recall. As a result, rtcpclientd sends a mount request to the vdqm for a free and compatible tape drive. The request includes the volume request identifier and the DGN of the tape. In response the vdqm queues the request from rtcpclientd and replies with a unique request identifier.

At step 2 the vdqm has asynchronously found a free and compatible drive that it can allocate to the pending mount request received earlier from rtcpclientd. The vdqm subsequently connects via TCP/IP to the rtcpd instance running on the tape-server connected to the free tape-drive. The rtcpd instance immediately forks a child process to handle the request and passes it the accepted connection. The vdqm sends to the child rtcpd process, the TCP/IP connection details of the rtcpclientd instance that requested the 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 individual drive-unit needs to be identified because a single tape-server and its single instance of rtcpd may be associated with more than one physical drive. The volume identifier of the tape to be mounted is not passed from the vdqm to rtcpd.

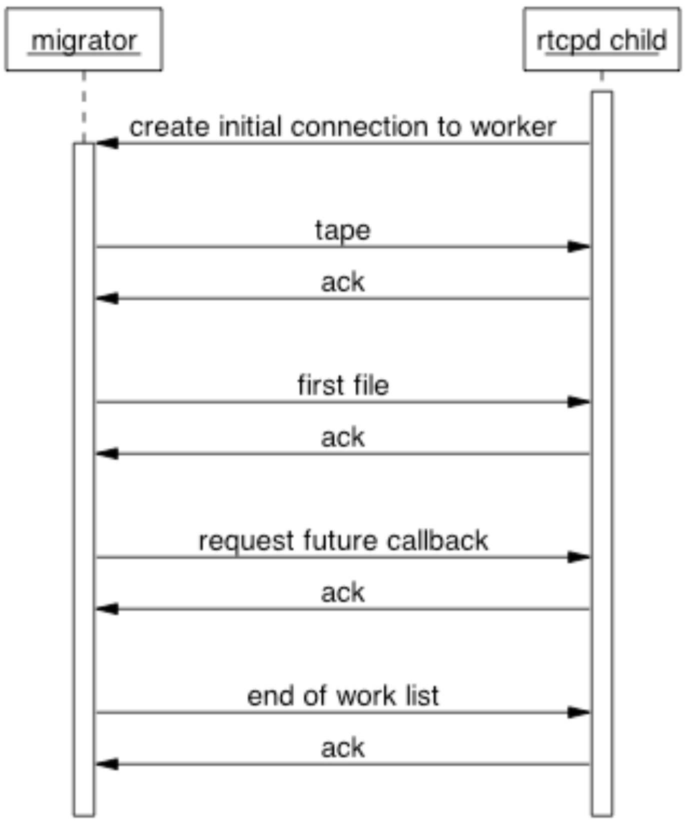
At step 3 the rtcpd child process connects to rtcpclientd, which immediately sends back a request for the remote tape-copy job information. The rtcpd child process sends the information to rtcpclientd, which in turn uses the information to determine the volume identifier of the tape to be mounted and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

At step4 rtcpclientd forks and execs a worker process in order to coordinate the reading or writing of the tape. When reading from a tape the worker process used is called the recaller process and when writing to a tape the name of 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

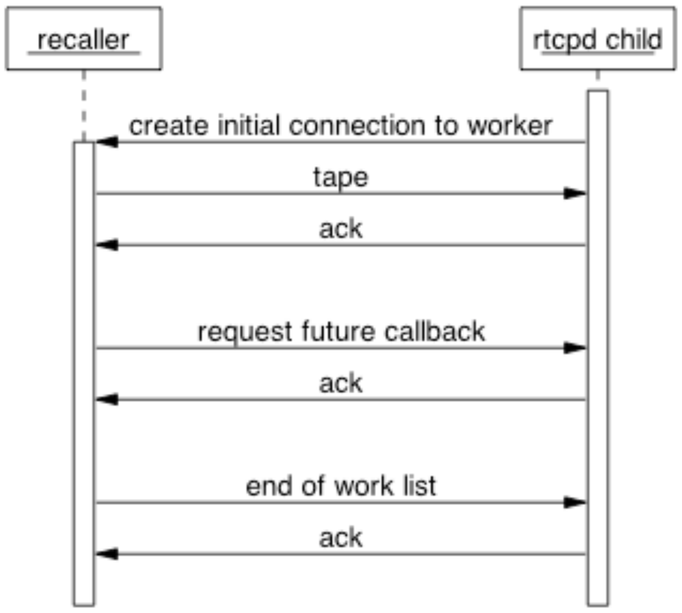
The `migrator` worker process exchanges more initial messages with the `rtcpd` child process than the `recaller` worker process does. When migrating files from disk to tape there is the opportunity to start copying data from disk into the memory of the tape-server whilst the tape is being mounted. Such an optimization is not possible when recalling files from tape to disk.

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

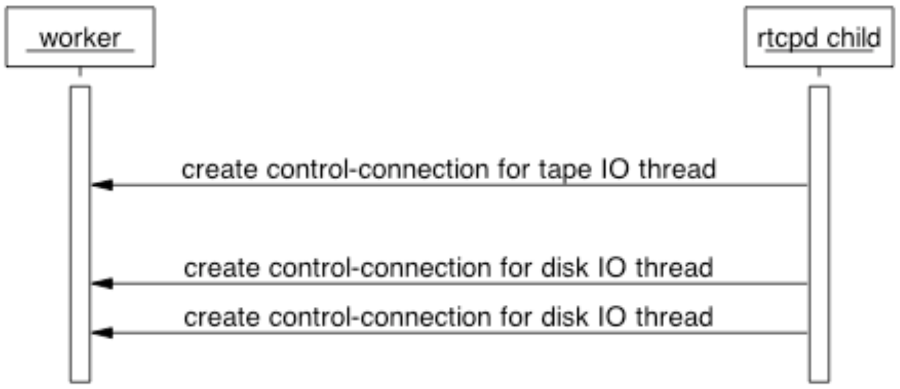


The `rtcpd` child process connects to the `migrator` process that was waiting for the connection after sending the details of its own newly created TCP/IP listening port. The newly created connection with the `migrator` process will hereafter be referred to as the initial connection to the worker, because 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 legacy remote-copy protocol. Once the initial connection to the `migrator` 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 that the `rtcpd` child process must callback the `migrator` process again for more work.

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.



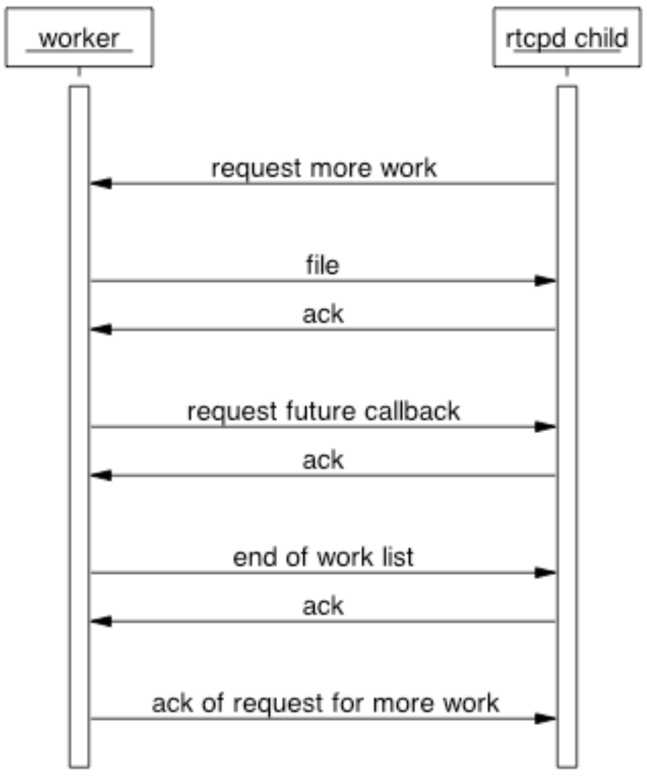
Once the `rtcpd` child process knows which tape to mount it internally creates one tape IO thread for reading or writing files to tape and one or more disk IO threads for remotely reading or writing disk files using TCP/IP connections to the `rfiod` instances running on the disk-servers. A separate control-connection 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 has 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 legacy remote-tape copy protocol can now begin given the fact 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 legacy 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 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.

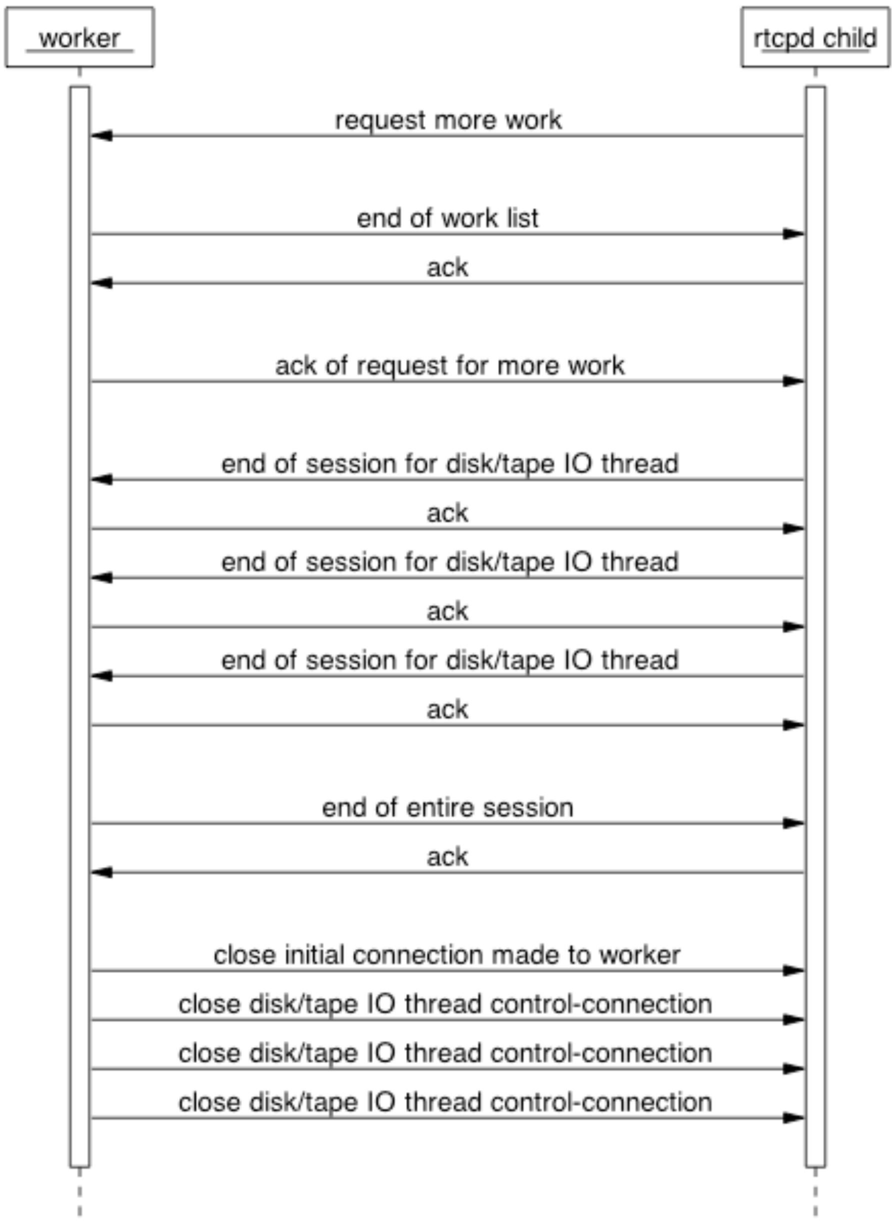
The following sequence diagram describes the “what’s next?” message sequence.



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 callback again for more work.

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. Reaching the physical end of tape when migrating files from disk to tape is considered and error, though an expected one at that.

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).

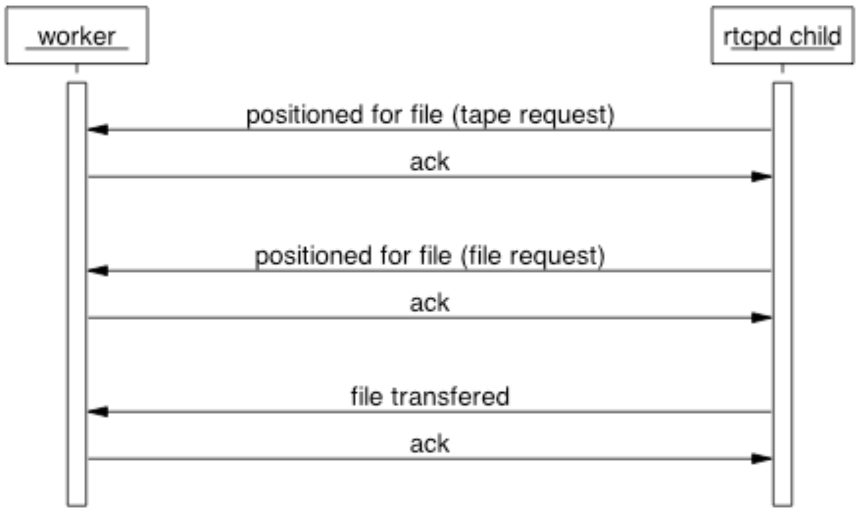

worker
rtcpd child
request more work
end of work list
ack
ack of request for more work
end of session for disk/tape IO thread
ack
end of session for disk/tape IO thread
ack
end of session for disk/tape IO thread
ack
end of entire session
ack
close initial connection made to worker
close disk/tape IO 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. 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 occurs on the rtcpd child process side of the protocol, then the rtcpd child process sends an error message to the worker process. As mentioned before, reaching the end of a tape when migrating files to tape is considered an error, though a normal and expected one. On receiving an error message the worker process usually tries to start an abort sequence. This sequence has been seen to fail and for the purpose of designing tapebridged the simpler solution of just closing all TCP/IP connections with the rtcpd child process has been proven to be safe and adequate. This document therefore does not describe the broken abort message sequence between a worker process and an rtcpd child process.

If an error occurs on the worker process side of the protocol, then the worker process starts the broken abort message sequence with the rtcpd child process. Again, this abort sequence will not be describes because the design of tapebridged incorporates the simpler solution of just closing all TCP/IP connections with the 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

For each file transferred the rtcpd child process sends three messages. The first two signal that the tape has been positioned for the file transfer and the third states that the file in question has been successfully transferred. 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, these two messages are only acknowledged by tapebridged and are not relayed to tapegatewayd. Tapebridged is only really interested in the file-transferred message.

Design

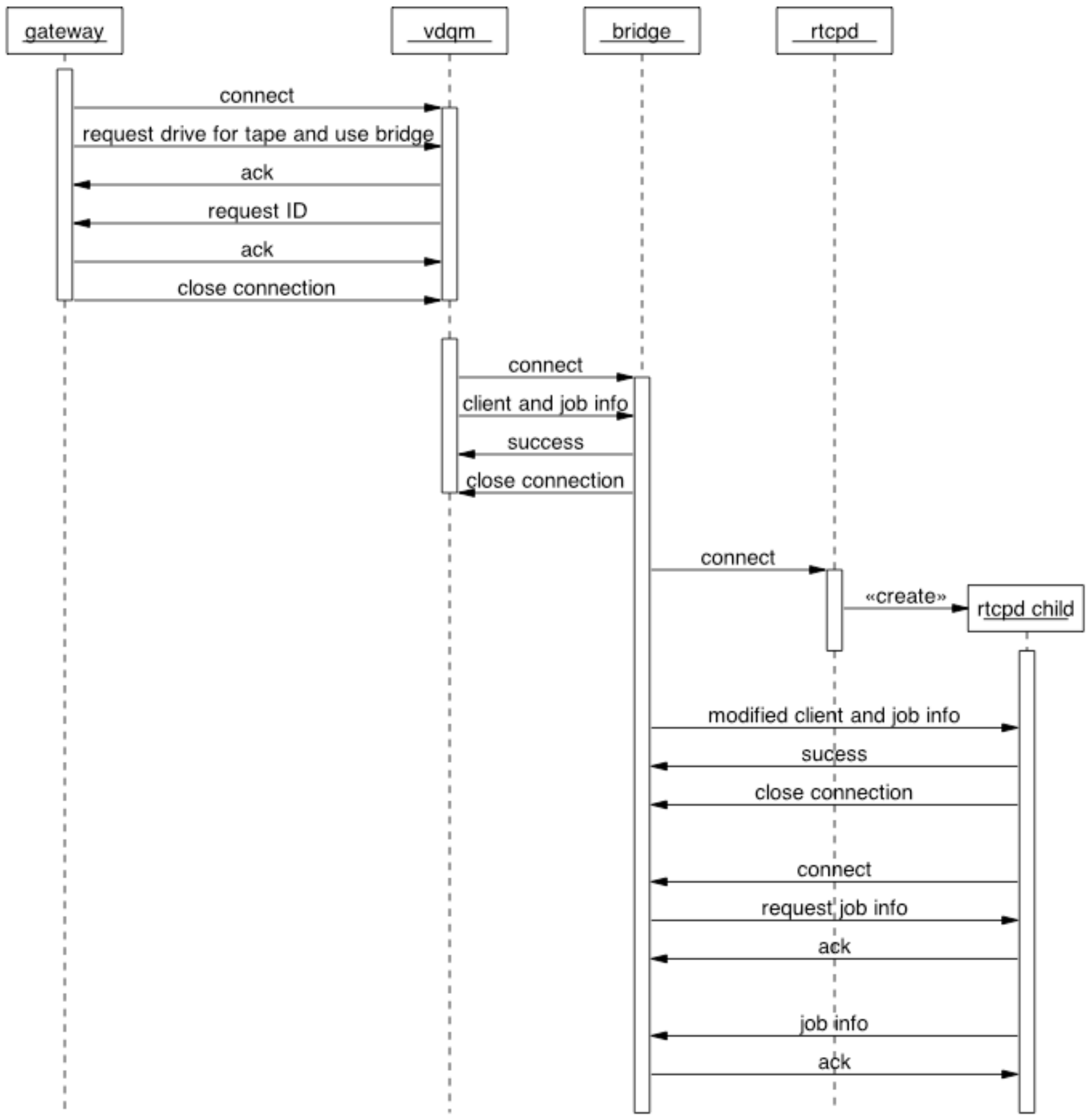
The structure of this section

This section is divided into five subsections including this one. The next and second subsection describes the message sequences that make up the new remote tape-copy protocol to be used between `tapegatewayd`, `tapebridged`, `vdqm` and `rtcpd`. The third subsection gives a summary of the message sequence used between just `tapebridged` and `tapegatewayd`. This summary is targeted at developers of `tapegatewayd` who are only interested in the interfaces of `tapegatewayd` as opposed to how `tapebridged` works. The forth subsection describes the format of the messages sent between `tapebridged` and `tapegatewayd`. The fifth subsection gives the hierarchy of the C++ classes that represent the messages sent between `tapebridged` and `tapegatewayd`. The sixth subsection gives a brief overview of the internal design of `tapebridged`. The idea of the sixth subsection is not to give all the low-level details, but to give the reader enough information to be comfortable navigating through the source code, which is has doxygen comments in the C++ header files.

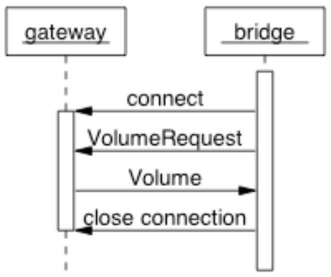
Remote tape-copy protocol between bridge, gateway, vdqm and rtcpd

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

`Tapebridged` acts as both the `vdqm` and `rtcpclientd` to `rtcpd` and its child process. The following sequence diagram shows how `tapebridged` effectively wraps `rtcpd` when `tapegatewayd` requests 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.

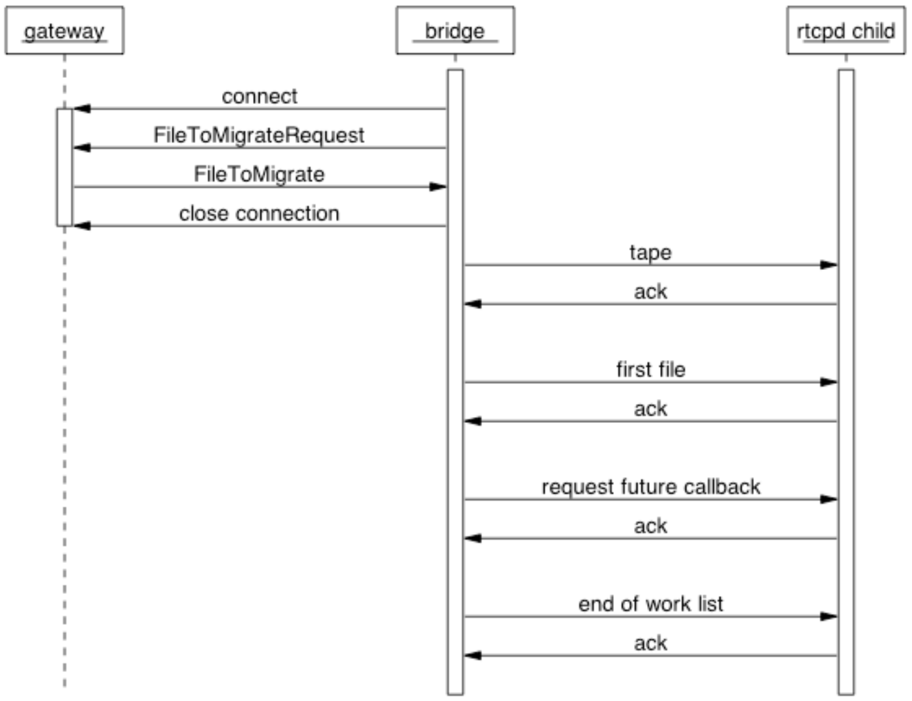


Once `tapebridged` has effective wrapped the `rtcpd` child process, it asks `tapegatewayd` for the volume to be mounted and whether it should be mounted for read or write. The sequence of messages exchanged is as follows.

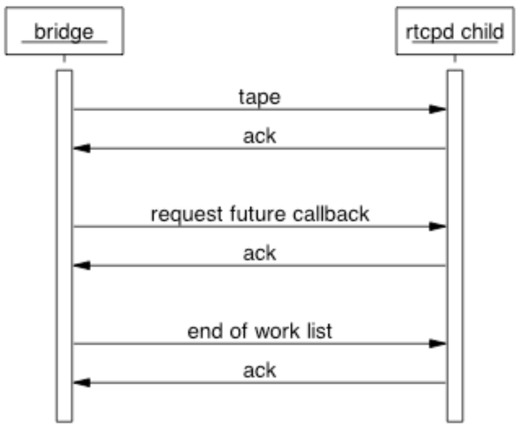


When designing the new remote tape-copy protocol it was decided to create a new connection for each request made from `tapebridged` to `tapegatewayd`. This decision was taken to keep open the possibility of supporting redundancy by being able to load balancing requests across many instances of `tapebridged`.

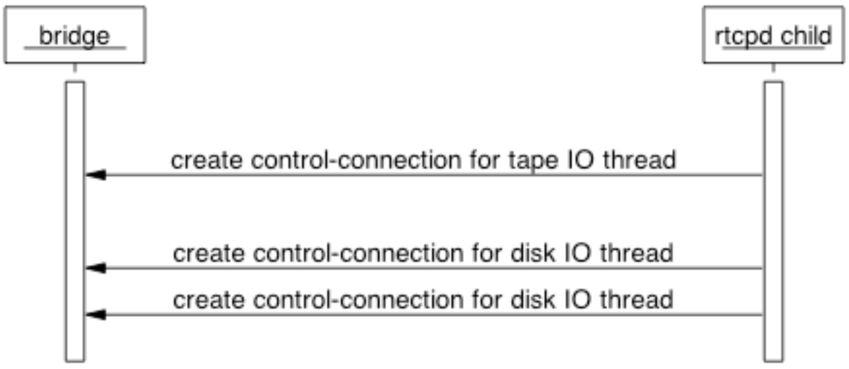
The following sequence diagram shows how in the case of migrating a file from disk to tape, `tapebridged` asks `tapegatewayd` for the first file to be migrated before sending the work list of tape-volume to be mounted, first file to be migrated and request for a future callback to the `rtcpd` child process.



In the case of recalling a file from tape to disk, `tapebridged` 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` instances running on the disk-servers. A separate control-connection between the `rtcpd` child process and `tapebridged` 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.

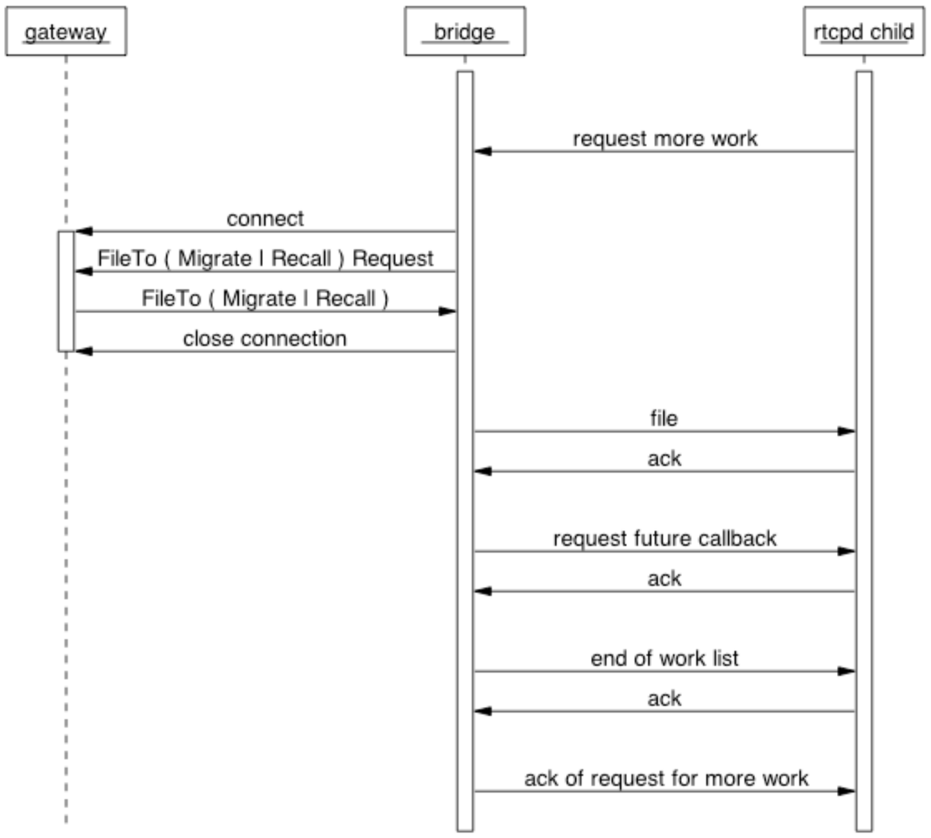

bridge
rtcpd child
create control-connection for tape IO thread
create control-connection for disk IO thread
create control-connection for disk IO thread

The multiple connections between the `rtcpd` child process and `tapebridged` are purposely hidden from `tapegatewayd`.

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

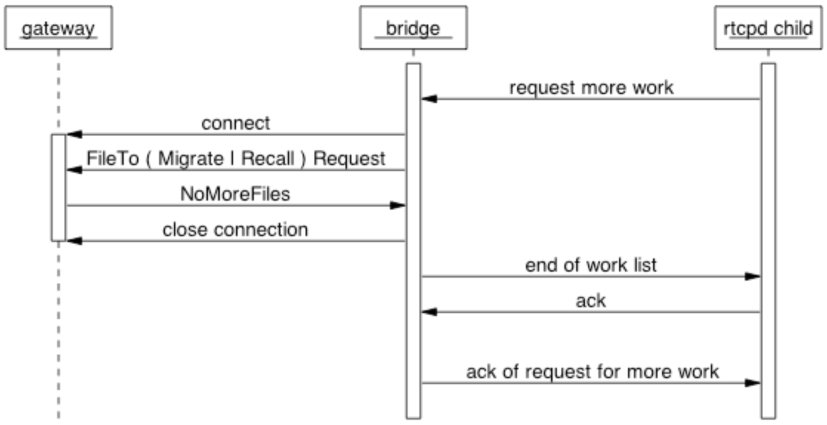
The main loop of the new remote-tape copy protocol can be divided into two message sequences. The first, hereafter referred to as the "what's next?" sequence, is the sequence of messages that control which file should be transferred next and when the transfer session of all the files to be transferred 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 individual files being transferred to tape or to disk.

The following sequence diagram shows the main "what's next?" loop.



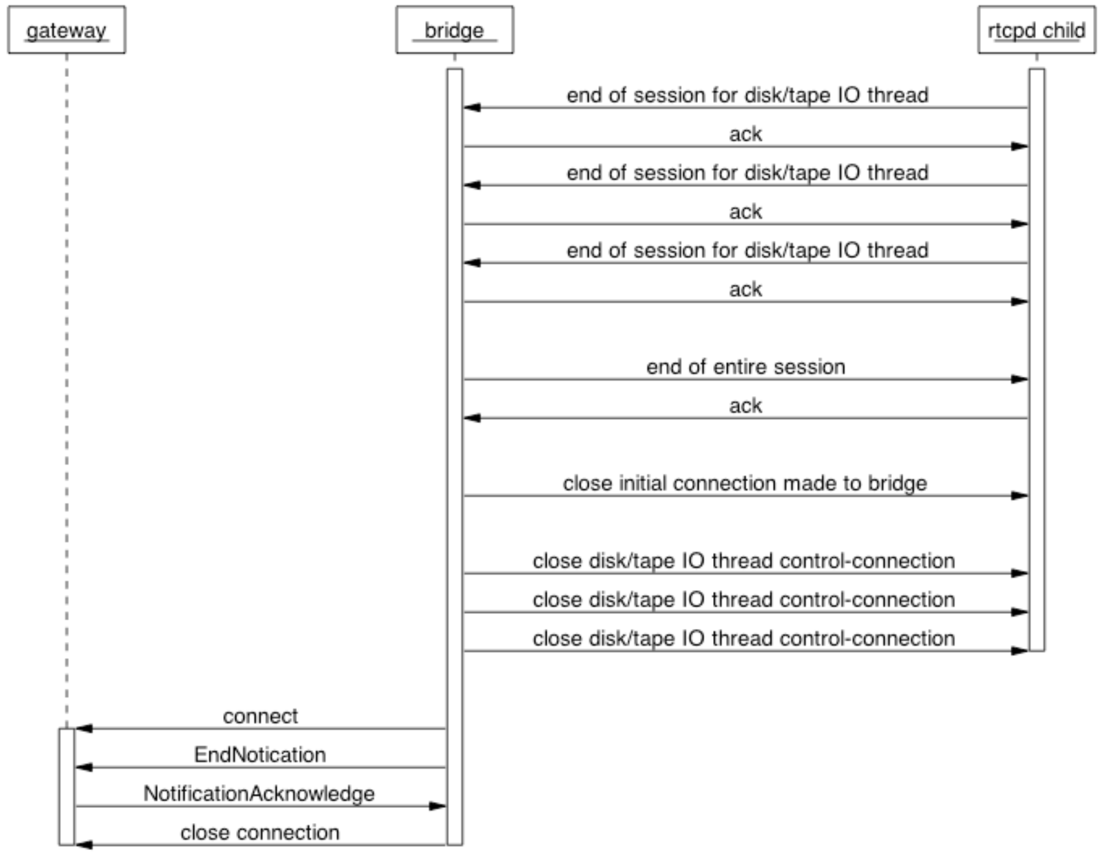
The `rtcpd` child process effectively loops requesting `the` next file to be transferred until there are no more files or an error is encountered, which as stated earlier could be generated by reaching the end of tape during a migration session. `Tapebridged` translates each of these requests into a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

The following sequence diagram shows how the protocol shutdown sequence is started when `tapegatewayd` has no more files to be transferred.



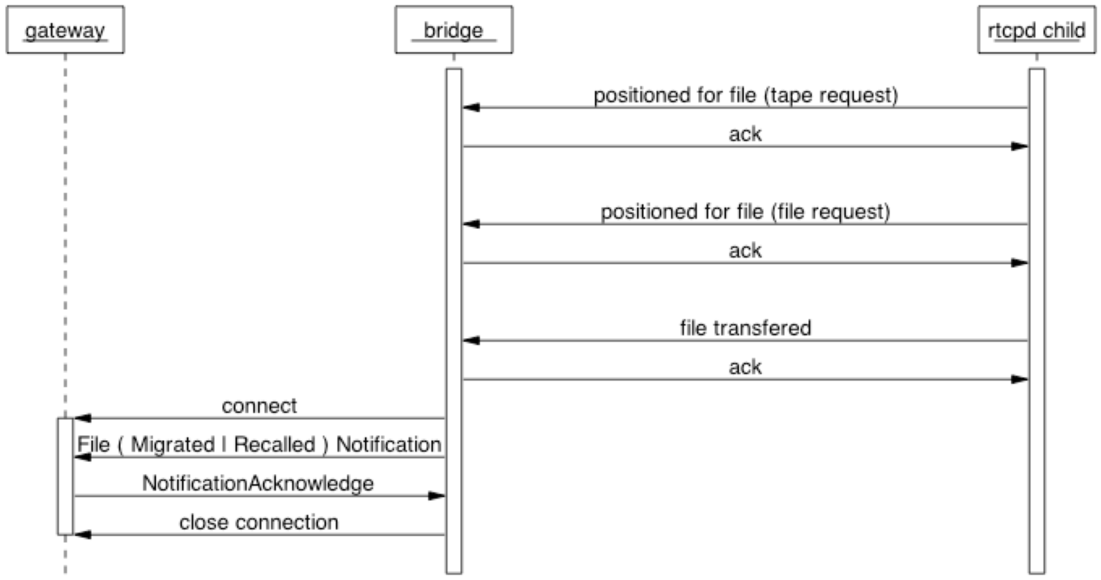
The `rtcpd` child process involuntarily starts the shutdown sequence by requesting more work as usual, `tapebridged` relays this request to `tapegatewayd`, which having no more files to transfer, replies to `tapebridged` with an explicit `NoMoreFiles` message. `Tapebridged` then sends an empty work list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

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


gateway
bridge
rtcpd child
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 bridge
close disk/tape IO thread control-connection
close disk/tape IO thread control-connection
close disk/tape IO thread control-connection
connect
EndNotication
NotificationAcknowledge
close connection

Once `tapebridged` has seen the end of session messages from each of the disk/tape IO control-connections, it sends the final end of `rtcpd` session message to the `rtcpd` child process. With the session now completed between `tapebridged` and the `rtcpd` child process, `tapebridged` connects to `tapegatewayd` and sends an `EndNotification` message that is acknowledged by `tapebridged`. The transfer session is now finished for all three parties: `tapegatewayd`, `tapebridged` and the `rtcpd` child process.

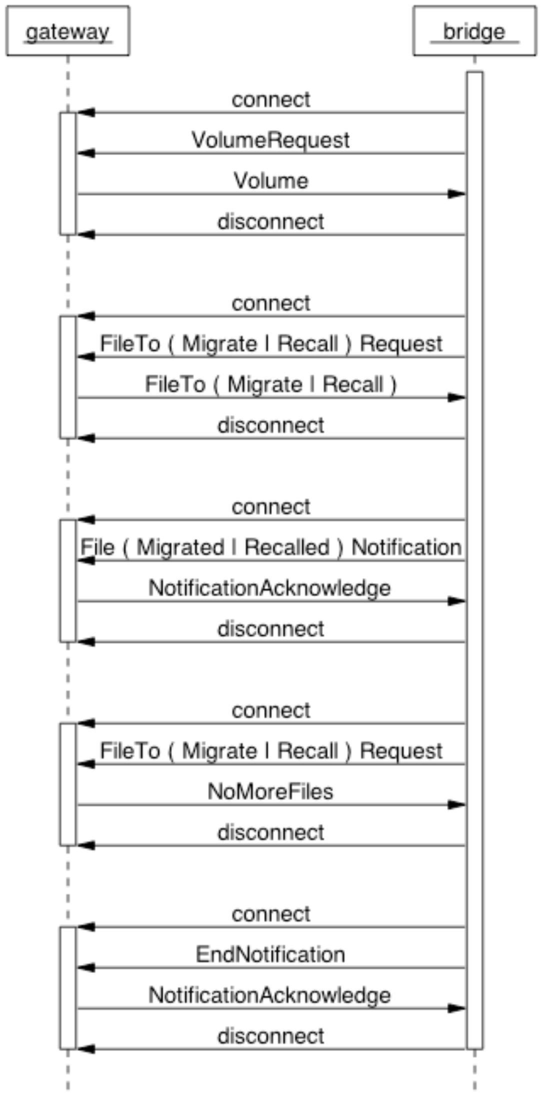
The following sequence diagram shows the “transfer feedback” message sequence.


gateway
bridge
rtcpd child
positioned for file (tape request)
ack
positioned for file (file request)
ack
file transfered
ack
connect
File (Migrated I Recalled) Notification
NotificationAcknowledge
close connection

The `rtcpd` child process sends three messages to `tapebridged` for each file successfully transferred. The first two messages indicate the tape was positioned and the third and final message signifies the successful transfer of the file. `Tapebridged` acknowledges the two tape-positioned messages, but does not relay them to `tapegatewayd`. `Tapebridged` is only really interested in the file-transferred. `Tapebridged` relays the file-transferred message to `tapegatewayd` in the form of either a `FileMigratedNotification` or a `FileRecalledNotification` message.

Remote tape-copy protocol between the bridge and gateway

The following sequence diagram shows only the messages sent between tapebridged and tapegatewayd. This diagram is a convenience for developers and maintainers of tapegatewayd.



Not shown on the diagram is the fact that the vdqm has just sent tapebridged a remote tape-copy job. Tapebridged connects to tapegatewayd in order to request the tape-volume to be mounted and whether the tape-volume should be read or written. The above sequence diagram describes the scenario of a single file being transferred, therefore tapebridged gets to ask for work twice, the first time it gets a file to transfer and the second time it receives a NoMoreFiles message. Tapebridged sends tapegatewayd a FileMigratedNotification or a FileRecalledNotification message when the transfer of the file has completed successfully. When tapebridged has determined that there are no more files to be transferred and all files have been transferred (in this case there is only one), tapebridged sends an EndNotification message to tapegatewayd. The file transfer session is now complete.

The following table lists each message that can be sent from `tapebridged` to `tapegatewayd`. For each message the table gives the possible messages with which `tapegatewayd` can reply.

Bridge to Gateway	Gateway to Bridge	Comment
VolumeRequest	Volume	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileToMigrateRequest or FileToRecallRequest	FileToMigrate or FileToRecall	Good day
	NoMoreFiles	Good day
	EndNotificationErrorReport	Bad day – Abort mount
FileMigratedNotification or 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
DumpNotification	NotificationAcknowledge	Good day
	EndNotificationErrorReport	Bad day – Abort mount

Format of messages sent between bridge and gateway

The following tables describe the format of the messages sent between `tapebridged` and `tapegatewayd`.

VolumeRequest message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
Unit	String	

Volume message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
vid	String	
mode	Enumeration	READ =0 WRITE = 1 DUMP = 2
density	String	
label	String	
dumpTapeMaxBytes	Integer	
dumpTapeBlockSize	Integer	
dumpTapeConvert	Integer	
dumpTapeErrAction	Integer	
dumpTapeStartFile	Integer	
dumpTapeMaxFile	Integer	
dumpTapeFromBlock	Integer	
dumpTapeToBlock	Integer	

FileTo Recall / Migrate Request message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers

FileToRecall message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
fileTransactionId	Signed 64-bit integer	Uniquely identifies the file transfer within the namespace of the mountTransactionId
nshost	String	
fileid	Unsigned 64-bit integer	CASTOR names server file ID
fseq	Signed 32-bit integer	CASTOR tape file sequence number
positionCommandCode	Enumeration	TPPOSIT_FSEQ, TPPOSIT_FID, TPPOSIT_EOI, TPPOSIT_BLKID
path	String	
blockId0	Unsigned 8-bit integer	
blockId1	Unsigned 8-bit integer	
blockId2	Unsigned 8-bit integer	
blockId3	Unsigned 8-bit integer	

FileRecalledNotification message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
fileTransactionId	Signed 64-bit integer	Uniquely identifies the file transfer within the namespace of the mountTransactionId
nshost	String	
fileid	Unsigned 64-bit integer	CASTOR names server file ID.
fseq	Signed 32-bit integer	CASTOR tape file sequence number
positionCommandCode	Enumeration	TPPOSIT_FSEQ, TPPOSIT_FID, TPPOSIT_EOI, TPPOSIT_BLKID
path	String	
checksumName	String	
checksum	Unsigned 64-bit integer	
fileSize	Unsigned 64-bit integer	

NotificationAcknowledge message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers

NoMoreFiles message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers

EndNotification message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers

FileToMigrate message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
fileTransactionId	Signed 64-bit integer	Uniquely identifies the file transfer within the namespace of the mountTransactionId
nshost	String	
fileid	Unsigned 64-bit integer	CASTOR names server file ID.
fseq	Signed 32-bit integer	CASTOR tape file sequence number
positionCommandCode	Enumeration	TPPOSIT_FSEQ, TPPOSIT_FID, TPPOSIT_EOI, TPPOSIT_BLKID
fileSize	Unsigned 64-bit integer	
lastKnownFileName	String	Last known filename as cached in the stager database. This maybe days old.
lastModificationTime	Unsigned 64-bit integer	Number of seconds since the EPOCH GMT. The timestamp for the lastKnownFileName
path	String	

FileMigratedNotification message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
fileTransactionId	Signed 64-bit integer	Uniquely identifies the file transfer within the namespace of the mountTransactionId
nshost	String	
fileid	Unsigned 64-bit integer	CASTOR names server file ID.
fseq	Signed 32-bit integer	CASTOR tape file sequence number
positionCommandCode	Enumeration	TPPOSIT_FSEQ, TPPOSIT_FID, TPPOSIT_EOI, TPPOSIT_BLKID
fileSize	Unsigned 64-bit integer	
checksumName	String	
checksum	Unsigned 64-bit integer	
compressedFileSize	Unsigned 64-bit integer	
blockId0	Unsigned 8-bit integer	
blockId1	Unsigned 8-bit integer	
blockId2	Unsigned 8-bit integer	
blockId3	Unsigned 8-bit integer	

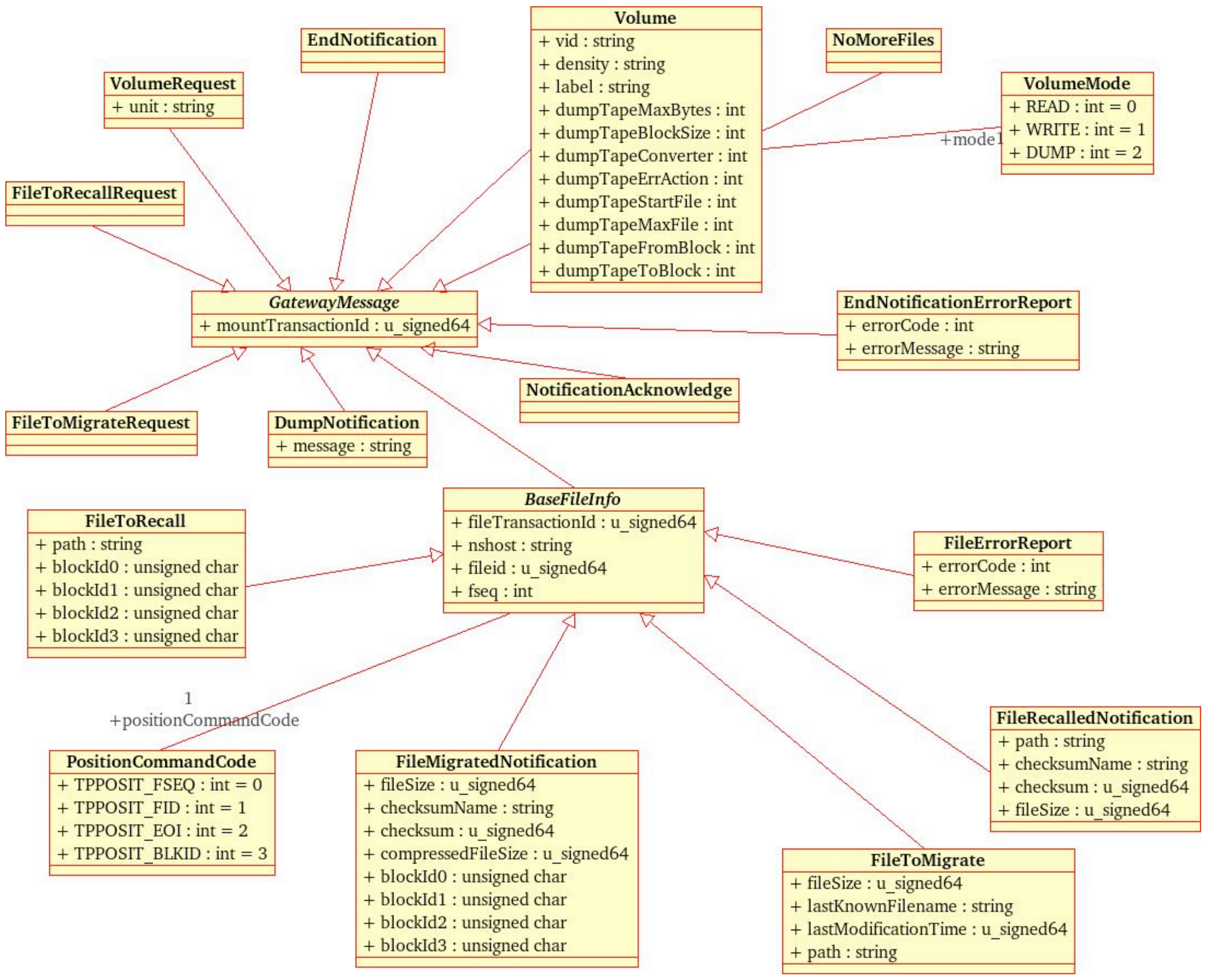
FileErrorReport message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
fileTransactionId	Signed 64-bit integer	Uniquely identifies the file transfer within the namespace of the mountTransactionId
nshost	String	
fileid	Unsigned 64-bit integer	CASTOR names server file ID.
fseq	Signed 32-bit integer	CASTOR tape file sequence number
positionCommandCode	Enumeration	TPPOSIT_FSEQ, TPPOSIT_FID, TPPOSIT_EOI, TPPOSIT_BLKID
errorCode	Signed 32-bit integer	
errorMessage	String	

EndNotificationErrorReport message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
errorCode	Signed 32-bit integer	
errorMessage	String	

DumpNotication message		
Member	**Type**	**Description**
mountTransactionId	Unsigned 64-bit integer	Uniquely identifies the mount across all tape servers
message	String	The message string sent to the bridge by RTCPD via a GIVE_OUTP message

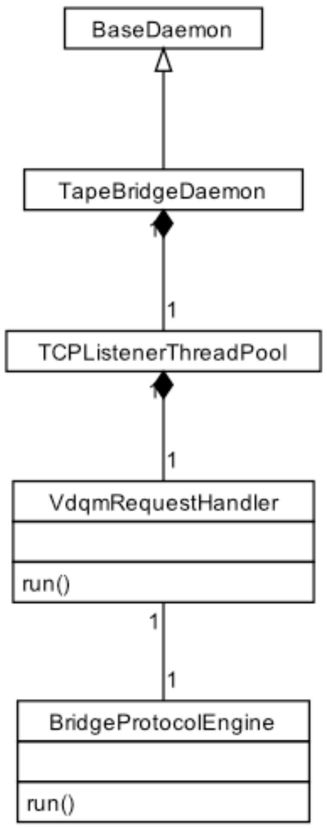
Inheritance hierarchy of bridge and gateway message classes

The following class diagram shows the inheritance hierarchy of the C++ classes representing the messages sent between `tapebridged` and `tapegatewayd`.



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.