Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

High Performance Tape Streaming in Tapr

Klaus Birkelund JENSEN [1] and Brian VINTER
Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. In this paper we describe the design and implementation of the Tapr high performance tape streaming system. Tapr consists of a number of basic processes interacting though message passing on CSP-style communications channels. The system is highly concurrent, uses synchronous as well as asynchronous coordination without the need for complex usage of traditional locks. The system scales to and beyond contemporary enterprise so-called *automated tape libraries* by representing each and every part of the tape library as a communicating process. This includes the robot changer, each tape drive, all clients and even the loaded tape media.

We show how such an implementation can be done in the Go programming language with relative ease by utilizing the concurrency primitives included in the base language. We also describe how complex cancellation and timeout handling can be handled directly in the language by using the concept of a surrounding context.

Finally, we present a number of benchmarks designed to show that the communicating process architecture does not impose any measurable overhead, but rather allows the system to scale to a high number of clients and devices using a simple and intuitive process-based design.

Keywords. communicating process architectures, data streaming, high-performance systems

Introduction

The use of magnetic tape for long-term archiving of data has been used in decades, an example being at CERN, where the CASTOR [1] system uses tape to archive data from experiments. The core tape technology has not changed much, but the LTO consortium has pushed out many iterations of the Linear Tape-Open (LTO) technology [2]. LTO is an open standard developed in the late 90s and is backed by the largest magnetic tape providers including IBM, HP and Quantum. The standard describes characteristics such as form-factor, capacity, wraps per band, tracks per wrap etc. The newest standard, LTO-7, has 6 TB native data capacity [3] and consists of almost 1000 meters of tape fabric.

While tapes, drives and necessary drivers are generally available, the software needed to operate these devices are dominated by proprietary products and a few freely available open source products; Bacula [4], its fork Bareos [5] and Amanda [6]. Bacula and Bareos are focused on differential and incremental backup and does not include simple archival functionality.

The proprietary products are in general a lot more advanced and geared towards the enterprise market. The most well known of these are IBM Spectrum Protect [7] (formerly IBM Tivoli Storage Manager) and Veritas NetBackup [8]. The state of available software for pure archival use prompted us to develop a new contender, Tapr, the design and implementation of which we will describe in this paper as well as the current state of the software.

[1]Corresponding Author: *Klaus Birkelund Jensen, Niels Bohr Institute, Blegdamsvej 17, DK-2300 Copenhagen S, Denmark. Tel.: +45 28 90 49 56*; E-mail: `birkelund@nbi.ku.dk`

1. Background

Because tape media are sequential in nature a number of challenges presents themselves. The largest challenge is the sequential nature in itself. To locate data on the media, costly seeks must be performed. Scanning through all data on the tape is complicated by the physical characteristics of the medium. An LTO-6 tape consists of four bands of 846 meters, each with 34 “wraps.” The bands are contained on a single spool and written in a serpentine manner. This means the tape must make 136 end-to-end passes to fill the tape. Each time the tape reaches either the beginning or end of the tape it must slow down, stop, and then accelerate in the opposite direction. At full speed, and LTO-6 tape can write up to 160 MB/s. LTO allows the tape to run at various speeds to match the incoming data rate, limiting the wear-and-tear caused by the so-called ”shoe-shine effect” that many older tape technologies suffer from. Shoe-shining happens when data cannot be delivered to the drive fast enough. When the tape runs out of input data at position A it must stop and reverse the tape to some position $B < A$. When data is again ready it must accelerate from position B and then start writing when reaching position A. While LTO handles this issue better, the drive and tape works best when running at the highest speed.

Keeping the tape at the highest speed requires special handling of slow writers, either by buffering (or staging) large amounts of data on disk or in memory before flushing it to tape. Another more efficient approach is to multiplex multiple writers onto the same tape [9] if the sum of data rates from all clients is larger than or equal to the maximum data rate of the drive.

1.1. The Automated Tape Library

Automated tape libraries are still used in many settings in enterprise as well as scientific communities with CERN being the obvious large-scale example. An automated tape library consists of one or more auto-changers or *robots*, which move media in the form of tapes around between storage slots and tape drives. When perceived as a single unit, the bandwidth of and concurrency provided by the automated tape library is directly related to the number of drives in the unit — and to a lesser degree, the number of auto changers.

Interaction with the tape library is done through low-level SCSI-commands with each drive being represented as a device file in UNIX-like operating systems. The auto changer is typically represented as either `/dev/changer` or another generic SCSI device file name. This device is used to query the library about the inventory and execute operations such as transferring a volume from a storage slot to another slot. On Linux and UNIX this is typically done through the use of the *Media Changer Tools* or MTX [10]. In Tapr, we use these tools directly to operate the changer instead of using the error-prone and potentially dangerous application of low-level SCSI commands.

1.2. Storage

Interaction with tape drives is also done through the SCSI interface On Linux and UNIX specifically through the `st(4)` SCSI tape driver and controlled through the `ioctl(2)` system call. The drive is represented as a character device (different from hard drives that are block devices) and can be opened, written to or read from and closed as any other file. Traditionally there is no file system, but the tape supports *file marks*. These are special marks on the tape that allow tools like `mt(1)` to seek to the beginning of the next or previous file or *record*. This lack of standardisation have caused the different tape systems to be incompatible with each other and they all implement their own storage strategy on the tape. `tar(1)`, the *tape archiver*, first appearing in Version 7 AT&T UNIX in 1979 [11], was not standardized until 1988 and even then, different operating system could have subtle differences in the tar archive format.

Because of that, we have chosen to use LTFS, the Linear Tape File System [12]. Primarily developed by IBM and introduced with the LTO-5 tapes and drives, LTFS consists of a user-space application that allows a tape to be mounted as a regular POSIX file system through FUSE [13], complete with directories and files. The file system is standardised and all the major LTO vendors provides a version of LTFS that is optimized for their hardware. Because the tape is formatted with a file system it becomes self-describing such that all stored files can easily be recovered if the tape can be mounted on a system that support LTFS. While the use of FUSE introduces a bit of overhead, we have not had any problems with reaching the maximum data rate on tapes. LTFS makes it vastly simpler to work with magnetic tapes because it supports the use of regular POSIX tools such as `cp(1)`, `mv(1)` and `rm(1)`. The reference implementation of LTFS is open source and available from IBM. In our case, we use a version modified by Oracle [14] because our experimental setup uses Oracle hardware. In addition to this open source version, the major vendors also have proprietary *Library Editions* [15] that allows an entire automated library to be viewed as one big file system, with the volumes automatically being loaded and unloaded as users traverse the file system.

1.2.1. Partitions

An LTFS formatted volume is partitioned in two. The first *index* partition is located at the beginning of the tape and holds the index of where files and folders are located on the tape. This means that a volume can be mounted and the contents quickly listed without scanning through the entire volume. This is an advantage because moving through an entire LTO-7 tape takes around 100-110 seconds [16].

The second partition holds the data and is append-only. Thus, file deletions or modifications always writes to the end of the tape and the index is simply updated. This implicitly makes LTFS a versioned file system and the tape can be restored to any point in time by loading an older index. The index partition is also append-only, so it essentially functions as a copy-on-write data structure. For efficiency reasons and to limit tape wear-and-tear due to excessive seeking, the index is generally held in memory and written subject to a policy. The current reference implementation includes three policies: a time-based policy, where the index is written every *n* minutes, a policy where the index is written every time a file is closed and finally a policy where the index is only written when the file system is unmounted. In Tapr, we use the unmount policy for efficiency reasons. Note, that if power should fail while the tape is mounted, a utility tool `ltfscheck` can scan the volume and update the index.

2. System Description

For a large-scale tape library to be effectively used for long-term archiving of data streams, there are some requirements.

- **A minimum of two drives dedicated for writing.** This requirement stems from the relatively large mount/unmount duration of tapes. In an automated library, the robot must first unmount the tape from a drive and return it to its storage slot. Before this can be done, the tape must also rewind and be ejected from the drive. Then, a new so-called "scratch" volume must be located and moved from its storage slot to the drive. The tape is then made ready for writing by the drive. This whole process may take minutes, but this latency can easily be hidden by having a another drive with a scratch volume ready to take over the stream. This requires that Tapr be able to handle multiple drives, and an *automated library* in general.
- **Enough data to keep the drive at high speed.** LTO drives have the ability to control the speed of the mounted tape, but to keep it at maximum bandwidth, the system

must be able to deliver enough data to the drive. The easiest way to achieve this, is to interleave writes from multiple streams.

- **Disaster recovery management.** As a long-term archiving system, Tapr should include some form of disaster recovery. Databases may be lost or media may be shuffled and repositioned in the library. These inconsistencies must be handled and managed properly.
- **An extensible client API.** Because Tapr is intended to be used as a storage solution for many different types of I/O systems, the interface to the system should be easily extendible and adaptable for whatever setting it is used in. Currently, we have only implemented an HTTP/2 API, but the component is completely disjoint from the rest of the server code, so other interfaced can be added easily. Another valuable interface is a FUSE layer similar to what is provided by the proprietary solutions discussed in Section 1.2 that allows easy navigation of the files in the entire library.
- **Scalable.** Tapr must be designed as highly scalable and concurrent. It relies on representing all objects in the tape system as communicating processes. This includes the media changer, the drives and even the volumes themselves. There is not a single mutex exposed in the code, though the underlying runtime naturally makes use of them.

These requirements make Tapr suitable for use in many settings and by requiring a scalable base, it allows the system to manage today's libraries consisting of hundreds tape drives and tens-of-thousands of storage slots. Figure 1 shows the overall design that we propose.

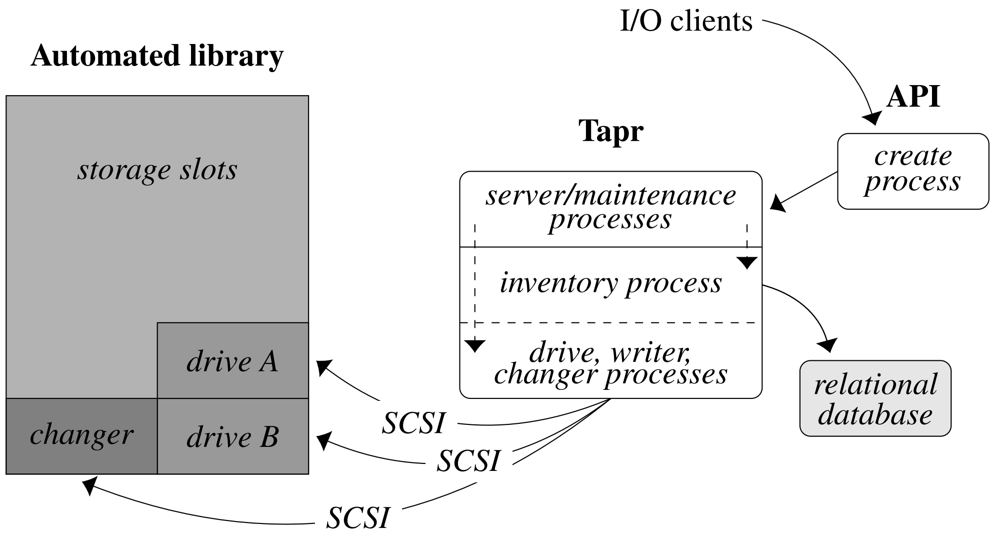


Figure 1. Overview of the Tapr design. The system and maintenance processes communicate with the other processes in the system and acts as coordinators when needed.

In this section we will keep the discussion on a theoretical level of abstraction. While Go is often discussed in the context of CSP [17], it has more in common with the π-calculus [18] and this discussion uses the concept of named channels in addition to communicating processes.

The different processes in Tapr communicate using channels. The channels are used in two different ways. Firstly the channels are used for communication in the form of requests with replies if necessary. Secondly they are used as a means to achieve streaming mechanics without explicit replies. The channels are unbuffered and communicating on one requires a ready sender and receiver for the processes to proceed.

2.1. The Primary Communicating Processes

The design includes three main types of communicating processes. Their purpose and basic behaviour is described below.

Clients Each client is represented as a process. It is responsible for reading from the network, assembling data into chunks of a certain size and sending this chunk to a drive to be written to a tape. The process also handles timeouts and cancellations, which will be described in detail in the implementation section of this paper.

Drives A process of this type contains the state of a single drive. Clients communicate directly with it, by shipping assembled data chunks to it. The drive process handles media error, unloading and loading of media, movement of clients from one drive to another as well as keeping statistics such as current data rate, number of attached clients etc. It is by far the most complex process in the system and the mechanics it uses to handle media failure and drive allocation requests from clients are described in detail in the implementation section.

Writers The writer process is the simplest process in the system. It performs only basic communication and primarily receives data chunks on two data channels and writes them to the media it is currently responsible for. The process transparently interleaves multiple write streams onto a single tape as data chunks from multiple clients arrive on the channel in FIFO order. It has two incoming data channels because one is a channel dedicated to single-drive writes and the other to parallel transfers where multiple writers are grabbing chunks to be written from this single aggregated one-to-many channel shared among all writers.

Additionally, multiple short-lived processes are routinely created to perform asynchronous operations such as statistics updates and doing process communication. These are part of the “system and maintenance” supporting processes shown in Figure 1.

2.2. Communication Overview

When a client connects to the server the `Stream` data structure is created. The stream contains a channel dedicated for error reporting and a reference to a data channel. The data channel is pointed towards the data channel of the drive that the stream is currently attached to or a special aggregated channel if parallel or *group writes* are being performed. Whenever the client assembles a full chunk it is sent on the data channel and a reply is expected on the error channel. The chunk goes directly to the writer process without involving the drive process. In the absence of an error at the writer process the writer simply replies directly to the client with a non-error value indication success. On the other hand, if an error is encountered in the writer process while writing, the error is sent to the drive process instead (without replying to the client) and the writer process terminates immediately. The drive can then choose how to deal with the error. Depending on the error, this may include retrying the write, marking the media as faulty or offloading the stream to another drive while mounting a new volume. At some point, the chunk that was to be written when the error was encountered will be successfully stored and another writer process will reply to the error channel of the client, indicating success, at which point the client will continue to assemble chunks and send them to the new writer. If for some reason the chunk cannot be written, the drive process may send an error back to the client process which can then send the error as an response with an error code and terminate with failure.

This design has the benefit of providing a *fast track* whenever the execution is successful. It is also worth noting, that because the main data channel is allocated in the drive process and not the writer, the client will always send chunks on a legal channel. There may not be a receiver, but the channel is never closed or otherwise unavailable. Figure 2 shows the basic relations of processes and how they communicate in the system.

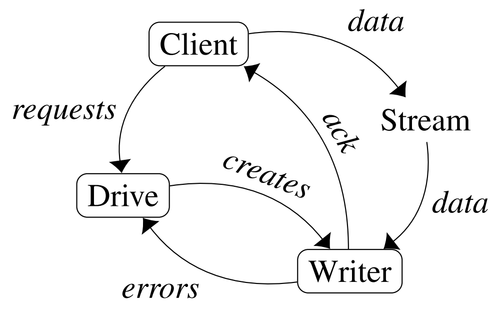


Figure 2. Channel communication overview. Boxed entities are processes, Stream is a structure of the Client.

Note that, depending on the durability guarantees of the client, it can choose to not require an acknowledgement from the writer on each write. In that case, the client will just continue to assemble chunks and send it to the writer process. It will only require an acknowledgement, when writing the last chunk in the stream. In Section 4 we shall see why this strategy can be beneficial in some cases.

3. Implementation

Tapr is implemented in the Go programming language [19]. While one cannot say that Go is built for developing certain kinds of systems, Go includes language primitives that makes it easy to implement highly concurrent systems. Go also includes good support for writing servers that speak and understand HTTP/1.1 [20] as well as HTTP/2 [21]. We have chosen HTTP as the initial interface to Tapr because it can be used from standard tools, is easy to implement and use in Go and provides the necessary streaming mechanics that Tapr requires.

The primary concurrency feature in Go is the *goroutine*, a light-weight serial process of which a Go program can, in principle, have an almost endless amount of. The processes do not have a traditional stack. The stack of a goroutine is allocated on the heap and grows and shrinks as needed, making it very memory efficient.

Any function can be started as a goroutine by simply prepending the `go` keyword to the function invocation. The function will then run concurrently and control is immediately returned to the calling function. Channels are created with the builtin function `make`. Channels are typed and garbage collected as everything else in the language. Calling `make(chan int)` creates a channel that can be used for sending and receiving integers. When doing channel communication in Go, the `go` keyword is often used to do the communication asynchronously, for instance by calling `go func() { ch <- "message" }()`. When doing this, the problem becomes how to cancel that operation. If at some point there is no receiver on the other side of the `ch` channel, that goroutine will stay in memory forever until the program is terminated.

3.1. Timeouts and Cancellations

There are several places in the Tapr source code, where processes must invoke an I/O request. An example is the acquisition of a drive for use by a client. If there is no preferred choice on which drive to acquire, we can send an acquisition request to each drive.

In Go, this is done by invoking separate goroutines (sharing a *reply*-channel) for each drive process. The main (or surrounding process) reads once from the reply-channel which will complete as soon as a drive process has acknowledged the acquisition upon which the separate goroutine will send information about the drive on the reply-channel. A problem with this approach is that if a read is not done for each of the separate goroutines, they will leak and remain in memory. A naïve solution is to start `len(drives)-1` goroutines to read the remaining messages on the channel, but what we really want is the means to signal the remaining goroutines and have them shut down by themselves. A common way of signalling on a channel is to close it by using the builtin function `close()`. This works because a receive on a closed channel will always yield the channel type's *zero value*. However, it is *not* allowed to perform a send on (or closing) a closed channel and this will cause a *panic*, that will crash the program unless explicitly handled. While the panic *can* be recovered from, it should not be in this case. Sending on a closed channel is almost always due to a deficiency in the design and the Go developers explicitly choose this behaviour to promote good program design [22]. Because of this behaviour, it is not possible to simply close the reply-channel after the first message have been received as this will result in a panic across the sending goroutines.

Instead, a second channel is introduced, dedicated to signaling that the operation is done or cancelled by being closed and then immediately willing to communicate the zero-value of the channel's type. A channel used for signalling by closing it can preferably be of type `<-chan struct{}`, which denotes a receive-only channel of type *empty struct*. The empty structure is used because it has zero memory allocation overhead. There is only a single empty structure allocated in the entire Go program no matter how much it is used.

This pattern is so often used in well-designed Go programs that the Go developers has developed the *Context* package included in Go version 1.7[2]. The technique requires the process to perform external choice on multiple channels. In Go, the `select` statement is used for this purpose. The statement chooses between a set of send and receive operations that can be performed. If multiple operations are ready to be performed, it chooses one of them pseudo-randomly.

3.2. The Context Package

A context is a special type that carries deadlines, timeouts and cancellation signals across function calls and between processes. It also supports adding request-scoped values. Weighing in at 256 lines of code the `context` package enables complex deep cancellation and timeout handling by exploiting the relatively simple channel semantics discussed in the previous section. Context values can be *derived* from other context values to form a tree in which the cancellation of a context causes all derived contexts to also be cancelled. To correctly use a context it must be propagated across the function call chain. In the case of Tapr this chain begins with the HTTP server receiving an HTTP request. The *root* context is then created by the `context.Background()` function. This special context is not cancellable and does not carry a deadline or timeout. If the client specifies that it requires exclusive access to a tape drive it may specify a timeout to not wait forever if all drives are currently busy servicing other client requests. A new context is thus derived from the root context by calling `context.WithTimeout(rootctx, timeout)`. The function returns the derived context (now carrying a timeout) and a *cancel* function to be used if the context should be explicitly cancelled. Figure 3 shows the tree structure that derived contexts create.

Regardless of whether a timeout is specified a context is now propagated as the first parameter to all functions that deal with this request. In this case, the first function to be called by the HTTP server is the `server.Store()` method in Tapr. The example thus far discussed (acquisition of a drive) is from this function. Because `Store()` takes a context as the first parameter, it has a *surrounding* context. To use the context correctly, whenever a function has a context parameter it should honor it by returning early if the context is cancelled or times out. Monitoring for cancellation is done by selecting on the context's `Done`-channel. Listing 1 shows how the context package is used.

[2]Release August 15th 2016.

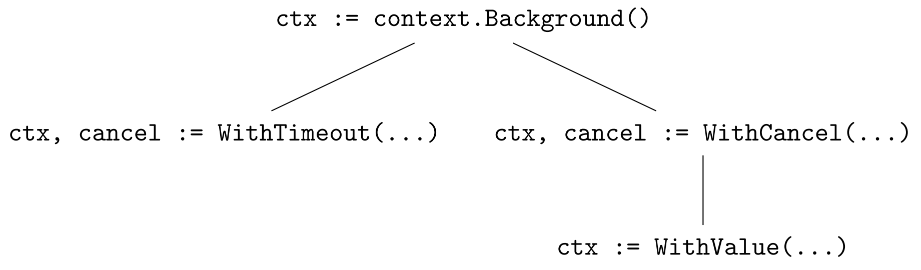

ctx := context.Background()
ctx, cancel := WithTimeout(...)
ctx, cancel := WithCancel(...)
ctx := WithValue(...)

Figure 3. The tree formed by deriving contexts. Cancelling a context will cause the Done channel to be closed for the context and all derived (child) contexts.

```
func (s *Server) Store(ctx Context, ...) {
    derived, cancel := context.WithCancel(ctx)
    ch := make(chan drive)
    for _, drv := range drives {
        go func(drv *drive) {
            if err := drv.Use(derived); err != nil {
                return
            }

            select {
            case <-derived.Done():
                drv.Release() // we didn't need the drive anyway
            case ch <- drv:
            }
        }(drv)
    }

    select {
    case <-ctx.Done():
        // timeout
    case drv := <-ch:
        cancel()
    }
}
```

Listing 1: Example use of the context type

Note that the `drv.Use()` function also takes a context parameter. Instead of simply returning without a value, the function returns an error if cancelled or if the drive is unavailable. Using the derived context, this can happen either if the context is cancelled implicitly by the parent context's timeout or explicitly by the call to `cancel()` by the `Store()` function. Note that cancelling an already cancelled context has no effect.

The code used in Tapr for acquiring a drive is a bit more complex than the code outlined here. This is due to the fact that drive selection is not as straight-forward as simply selecting the first responding drive. For efficiency reasons we want to select a drive such that bandwidth is fully utilized, but we do not want to send more streams than can be handled by the device if another less used device is also available. Tapr also supports *write groups* that provides parallel writing by splitting the stream across multiple drives as well as exclusive access to drives if the client requires the data to be written in a contiguous manner.

3.3. Stream Interleaving and Parallel Writes

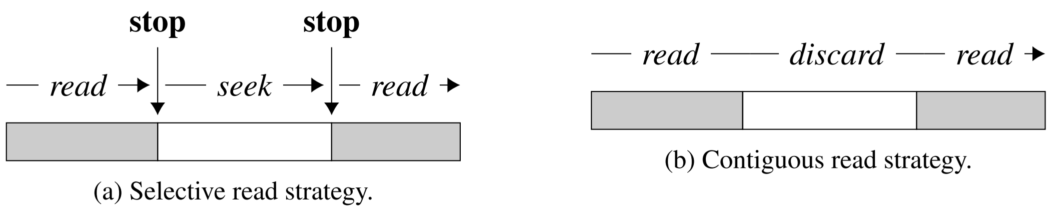
An essential technique for efficient use of magnetic tapes is to keep them operating at their highest data rate. In many cases a single stream is insufficient to achieve this. The two most prominent methods to achieve a high sustained data rate is staging and stream interleaving. Stream interleaving is used heavily in Veritas NetBackup [8]. The point is to multiplex streams onto the same medium. If a large enough data rate is available, the drive can be kept at maximum data rate while still guarding against *slow writers*. A disadvantage is that the data streams are being mixed together, complicating retrieval. For most long-term archival scenarios this is rarely a problem though as we shall see in Section 3.3.1.

With the staging approach as used by IBM Spectrum Protect [7], incoming streams are initially written to a random access disk array and only flushed to tape when enough data is available to sustain the drive for an extended period of time. This approach has the benefit of allowing the software to reorder and co-locate data on the tape for better retrieval performance, but comes at the cost of an expensive high-performance disk array to handle the incoming data rate.

Because Tapr is designed to handle high-volume streams of data to be archived long-term we use a *direct-to-tape* technique with aggressive stream interleaving. The semantics of the data channel automatically queues data chunks in FIFO order towards the tape. The writer process will not make any attempt to reorder chunks.

3.3.1. Stream Demultiplexing

When an archive needs to be retrieved from tape *demultiplexing* must be done. Here, the correct archive chunks must be cherry-picked from the tape and reassembled into a data stream. Because Tapr will always write stream chunks in sequential order, the tape will never need to perform random access, but the amount of forward seeks that must be performed is directly related to the number of write streams that were active at the time of writing.



(a) Selective read strategy.

(b) Contiguous read strategy.

Figure 4. Two possible read strategies.

There are two methods to achieve this (see Figure 4). A naïve approach is to simply seek to the beginning location of a chunk, read the chunk and then seek to the next chunk. Because tape drives cannot be instructed to prefetch, the drive will slow down, seek, then speed up while reading the chunk. This read/seek cycle will severely impact the data rate at which an archive can be retrieved but the impact varies with the chosen chunk size. If the chunks are several gigabytes the impact will be less severe.

The other approach keeps the tape at full data rate by reading all chunks sequentially and simply discarding the chunks that do not belong to the archive being retrieved. With this approach the choice of chunk size is less of a factor as the drive does not care about chunk borders. On the other hand, in this approach the number of write streams active at the point of writing directly impacts the perceived bandwidth.

3.4. Stream Teeing and Splitting

Single-drive stream interleaving makes it relatively easy to retrieve the data, but if the incoming data rate exceeds the maximum rate of a single drive the client can request to write to a *write group* instead (see Figure 5). Currently, write groups are statically defined in the Tapr configuration file. A drive can belong to any number of write groups and the purpose is to provide different levels of parallelism. If a client knows the data rate at which it will write, it can request exclusive access to a parallel write group and have the incoming data stream transparently split over multiple drives, effectively multiplying the perceived bandwidth. This complicates retrieval because multiple drives must now participate to extract the chunks in correct order if non-disk operation is desired. If a staging disk is available, chunks can be extracted to disk from each tape in sequence and then assembled back into a data stream. The strategy should be chosen according to the environment in which Tapr is deployed and the expected use. If retrieval is a rare occurrence it may be beneficial to use parallel writing by default for the larger bandwidth.

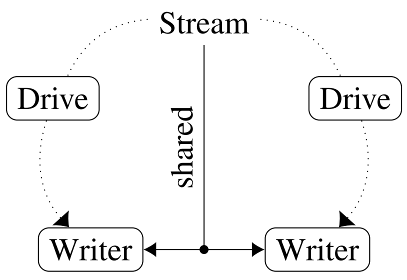


Figure 5. When writing to a parallel write group, the stream is writing directly to a shared channel that writes receive on. When writing to a single drive, the stream is associated with a channel allocated in the drive, but does not go through the drive (pictured with dotted line).

Because Tapr is designed as an archiving system for raw data from scientific experiments, it includes a novel feature that we call *stream teeing*, named after the UNIX utility `tee(1)` that copies standard input to standard output while making a copy in zero or more files. The tool itself is most likely named after the letter "T" that graphically represents a stream forking or splitting into two. Stream teeing is used to direct a data stream towards an alternative data sink such as an analysis engine for immediate consumption. The idea is to do analysis while data is flowing and, if at all possible, never touch a hard drive. The result of the analysis can be directed back into Tapr for long-term storage.

3.5. Stream Hand-off

Tapr allows drives to hand off a stream to another drive if the stream cannot be written to the currently assigned drive. This most commonly happens when the mounted media fills up, but can also happen as a result of another transient or permanent error such as a faulty media or drive. Hand off may also be initiated by directly requesting the drive to do it. This can be done by the server process if it wishes to release a drive for some reason.

In most cases stream hand-off is transparent to the client if there is another drive ready. If not, then the client will see a drop in perceived bandwidth. Hand-off is initiated when a writer process informs the drive of an error. If the error is related to lack of space, a new cancellable context is created (no timeout). The code to acquire another drive is then executed in a separate goroutine. The goroutine sends the drive on a channel once acquired. Concurrently, a goroutine is started that unmounts the LTFS [12], unloads the media from the drive and returns it to a storage slot, loads a fresh scratch volume into the drive, format the volume with LTFS and finally mounts it. When done, it signals completion by sending a new writer process on a channel. After starting the two goroutines, the drive process enters a `select` statement and awaits completion of either one. If another drive is acquired the stream is immediately handed off, by sending a `Takeover` request to the acquired drive. The drive taking over the stream then proceeds to write the chunk that failed to be written on the filled media and continues writing chunks from the stream transparently to the client. If another drive was not available, the unmount/unload/load/format/mount process will eventually complete. At that point the stream continues to write on the new media and the context that was previously created is cancelled to cause the goroutine currently requesting a drive to terminate. The process is shown in Figure 6.

This process shows, that the code in the drive process does not care if it is being requested by a client or another drive. When requested by a client the context may carry a timeout which will be acted upon if needed. When requested by another drive it is only explicitly cancelled if not already terminated. The process also shows the generality of using contexts for cancellation.

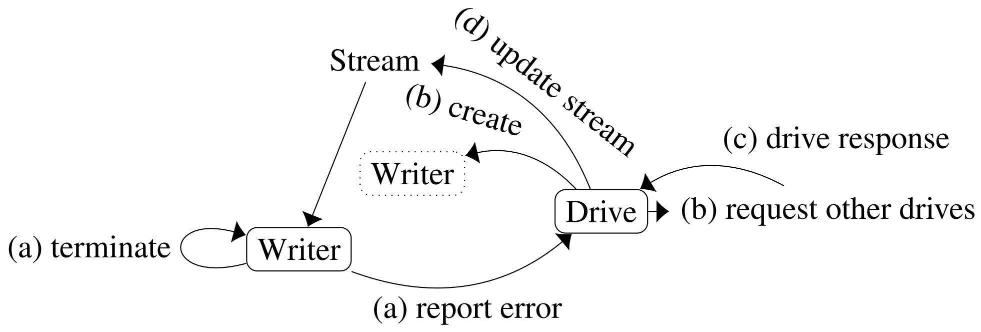


Figure 6. Handing off a stream. If a Writer encounters an error it will (a) send the error to the drive along with the chunk that failed to be written and the Writer will terminate immediately. Then (b), the writer simultaneously start the process of starting a new writer and requesting hand off on another drive. Whatever of (b) or (c) that finished first determines how the stream is updated (d).

4. Experimental Results

We have performed a number of experiments to measure the impact of building Tapr with a highly concurrent design. Due to the lack of an experimental setup with enough LTO drives we have designed our experiments such that writing to tape is simulated. Specifically, the tape is assumed to write at full speed (160 MB/s), which at a chunk size of 64 MB would require it to write for 400 milliseconds. For accuracy this is not scaled down, so the writer process will sleep for 400 milliseconds (or a different duration depending on the chunk size) for each chunk received. Importantly though, data is still read from the network and sent on channels to writers.

In all experiments it is assumed that enough data to keep the drives at full speed is available and that the drives were already running at full speed. This is done because we want to measure if the communicating process architecture (and the Go runtime) is capable of handling incoming data on a 10 Gigabit link. We have thus also limited our experiments to 8 virtual drives which equals an aggregated bandwidth of 1280 MB/s (or 10 Gb/s).

Our experiments are performed on a Dell PowerEdge R620 server with two quad-core Intel Xeon E5-2603 v2 CPUs running at 1.8 GHz and 64 GB of system memory. The system is running CentOS 6.7 on a 2.6.32 Linux kernel. Tapr is compiled with the Go compiler, version 1.6.

We perform each experiment five times, and average the results.

4.1. Write Throughput

Our first two experiments measure the write throughput of Tapr. The experiment has been run with 1, 2, 4, 16, 32 and 64 concurrent write streams. First, we measure the throughput when clients requests their data to be placed on a single drive and not split over multiple drives. This is done with and without *chunk acknowledgement* as discussed in Section 2.2. Clients are simulated using the tool `curl(1)` [23] with multiple instances started in parallel.

The bandwidth is calculated from the time it takes for the server to process the entire request (from request is received at the HTTP server to the response has been sent to the client).

Table 1 shows the raw results from the experiment. As expected we achieve the highest throughput when the client does not require an acknowledgement for each chunk. The difference in throughput is very clear for a concurrency level of 1, 2, 4 and 8 with the acknowledged writes reducing the throughput to around 85% of the non-acknowledged write. However, as the number of concurrent streams gets higher than the number of drives in the system the results are inconclusive.

Currently, the implementation does not continue to assemble chunks while waiting for an acknowledgement, and in the absence of acknowledgements can only assemble a single one while waiting to send it to a drive. This has the benefit of limiting memory consumption, but leads to lower bandwidth utilization as seen in Table 1.

Table 1. Write throughput with and without chunk acknowledgement. A stream only writes to one drive of the eight drives. Streams are allocated fairly to all eight drives. Chunk size is 64 MB.

Number of streams	Chunk acknowledgement	Close acknowledgement
1	133.936	155.818
2	263.633	311.664
4	523.933	619.181
8	1027.019	1210.366
16	1233.000	1225.364
32	1232.339	1235.077
64	1230.134	**1242.576**

In Table 2, the results for the second experiment are listed. Here, the stream is allowed to be written in parallel to all drives in the system. Again, there is a large difference in achieved bandwidth until the concurrency level goes above the number of drives. As the drives are oversubscribed the cost of acknowledging each chunk write is amortized away and the aggregated bandwidth reaches 1270 MB/s, very close to the theoretical limit of 1280 MB/s.

Table 2. Write throughput with and without chunk acknowledgement. Streams are written in parallel to all drives. Chunk size is 64 MB.

Number of streams	Chunk acknowledgement	Close acknowledgement
1	134.650	518.513
2	263.249	969.893
4	519.310	1168.921
8	1016.759	1217.439
16	1247.587	1241.749
32	1262.326	1260.930
64	1270.142	**1270.864**

The parallel version is faster because drives that have written a chunk do not have to wait for the client to assemble another full chunk. If any client has a chunk ready, the drive can grab it and write it to its media. And, as the concurrency level rises, the competition on I/O becomes higher for the clients.

The results from the two experiments are summarised in Figure 7. The results shows, that in general Tapr handles oversubscribing of the tape drives well. It should (and can without disadvantages) allow the clients to choose if they want parallel writes and whether they require chunk acknowledgements because the aggregated bandwidth remains stable and close to the theoretical limit. If a client expects the stream to be retrieved in the near future, it might not be worth the extra throughput of a parallel transfer for the potentially slower retrieval it would lead to.

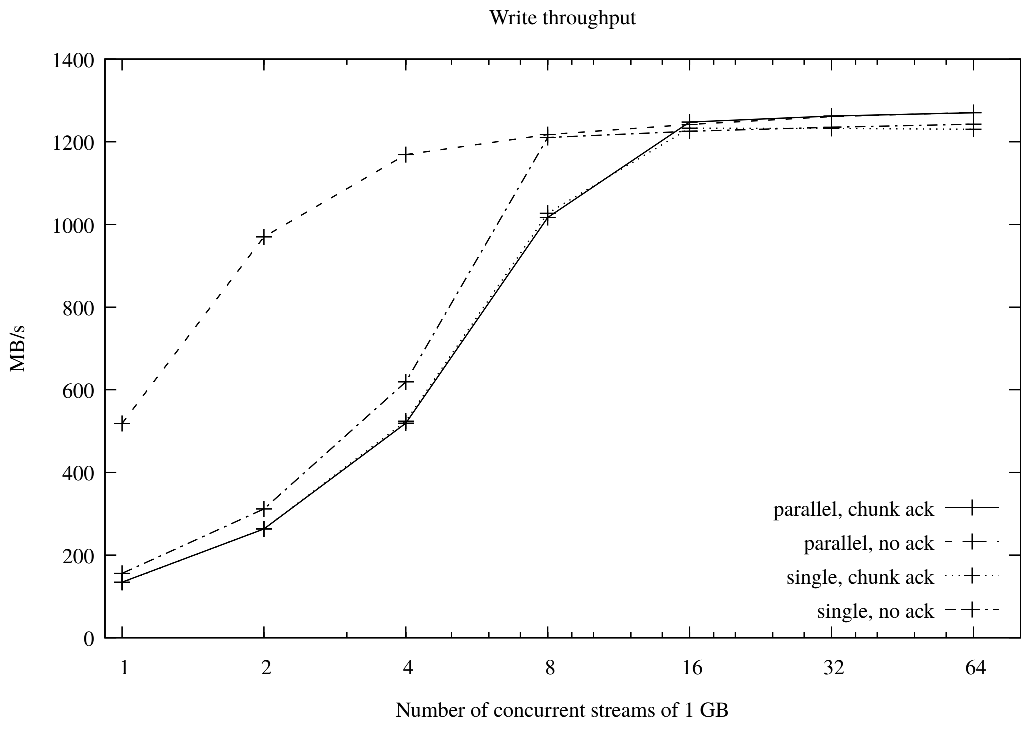


Figure 7. Write throughput

4.2. Chunk Size

Our final experiment measures the impact of different chunk sizes. Again, the experiment is performed for 1 to 64 streams. The chunk size is varied between 4, 32, 64 and 128 MB. The experiment is based on the "best" results from the first two experiments and therefore use parallel writes and unacknowledged chunks. The results are listed in Table 3 and Figure 8. Interestingly a chunk size of 32 MB yields a higher aggregated bandwidth at 64 concurrent write streams of almost 1274 MB/s. In general, the smaller chunk sizes benefit the lower concurrency levels.

Table 3. Write throughput with various chunk sizes. Streams are written without chunk acknowledgement and in parallel to all drives.

Number of streams	4 MB	32 MB	64 MB	128 MB
1	777.090	590.973	518.513	415.088
2	1213.508	1085.203	969.893	809.663
4	1243.269	1196.506	1168.921	1106.889
8	1256.086	1235.506	1217.439	1168.370
16	1263.616	1259.102	1241.749	1207.612
32	1267.660	1268.058	1260.930	1241.533
64	1268.802	**1273.849**	1270.864	1261.567

While the results initially point towards using a smaller chunk size than 64 MB there are a number of trade offs. This is related to retrieval. While a 4 MB chunk size lowers the latency induced by the assembling of chunks from a stream, it complicates retrieval if a high number of concurrent streams were written to the tape. As discussed in Section 3.3.1

this would perform very bad for a selective read approach. It would not impact a contiguous approach though.

There is nothing that requires Tapr to use a fixed chunk size however. Thus, this result suggests that it could be very beneficial to vary the chunk size with the concurrency currently imposed on the drives if combined with a contiguous read strategy.

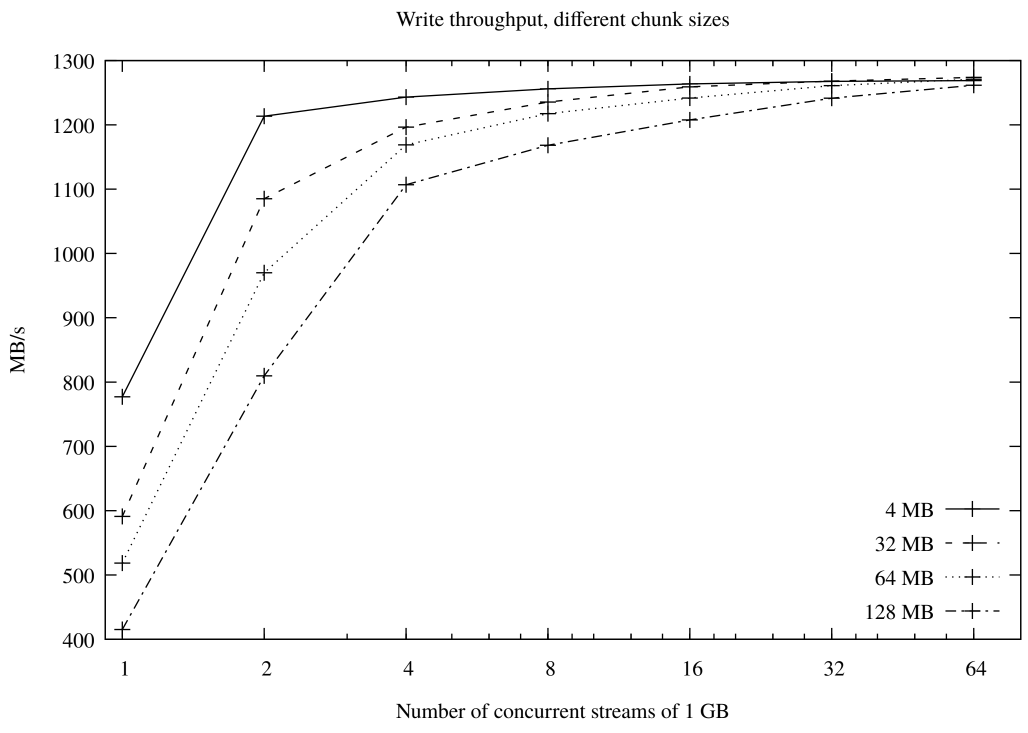


Figure 8. Write throughput

5. Conclusions and Future Work

In this paper we have described the design and implementation of Tapr, a highly concurrent network server for long-term archival of data streams. The basic functionality is already in place, but Tapr remains a prototype, though we expect it to go into production at the University of Copenhagen for specialized use in long-term archival of research data. Because of this prototype status, there remains a lot of challenges as well as opportunities for future work.

The high bandwidth provided by modern tape systems and the possibility of splitting streams to write to multiple tape devices in parallel can effectively provide endless amounts of bandwidth, limited only by the availability of drives in the library. This horizontal scaling allows a system to be provisioned that can handle most bandwidth requirements of industrial imaging applications without ever using disk as an intermediate staging area.

Our results in this paper show that while the effect of employing chunk acknowledgements are amortized away when many streams are interleaved onto the drives, it could be beneficial to allow the client to continue to assemble chunks from the network and queue them up for writing. While it is possible to implement a standard CSP-like *buffer process* in Go, the language has direct support for this; the data channel must be changed to a buffered version with a certain size[3]. But as the results show, this is currently not critical as long as

[3]A buffered channel has different semantics, i.e. it does not require a ready receiver for a value to be sent.

Tapr has *enough* work to do. We note that these results are done in a simulated environment, so while the bandwidth of real tape drives will probably fluctuate slightly, the results still show that the Tapr design is capable of handling a data stream of 10 Gigabits. In a real-world setting, the use of buffered channels could be beneficial, and the design still allows for this to be introduced.

While writing of data coming from other durable storage systems can easily be retried in the presence of failures, this is not possible when data is streaming from scientific equipment. To better support this, a sliding window as used in TCP [24] could be used to limit the number of chunk acknowledgements and limiting the need for staging of important data. The data could either be retained (or buffered) at the equipment or directly at the archive system on fast durable storage (solid state disks or a high-performance disk array) until the data is known to be written to durable (tape) storage.

The results suggesting that Tapr use varying chunk sizes also presents a challenge. Because Tapr uses LTFS as the file system on the tapes and because LTFS is a regular POSIX file system a 6 TB LTO-7 tape would end up with over 1.5 million files in a single directory in the current implementation of Tapr when using 4 MB chunks. For a chunk size of 64 MB, this number would still be around 100,000 so in any case it might be prudent to explore ways to nest directories appropriately. While this suggests that Tapr should use its own native format on the tape, the advantage of an open and well-supported standard, as well as the completely self-describing media is very important to the goals of Tapr. Currently the naming scheme used for chunks written to the tape does not easily support a nested structure, but this is something that we are confident can be solved.

It should be noted that we have not regretted the choice of Go for this high performance application. We have made relatively few optimizations to achieve maximum throughput, the only optimization being a tight network socket read-loop and the use of a chunk pool to limit pressure on the garbage collector. Instead, we have extensively used the concurrency features available in Go to rapidly build the communicating process architecture. While the CPU usage is still relatively low, it is good to know that Go will automatically utilize all available cores as the concurrency level of the application increases when more devices are added to the system.

Acknowledgements

This work is part of the CINEMA [25] project and financial support from CINEMA: the allianCe for ImagiNg of Energy MAterials, DSF-grant no. 1305-00032B under The Danish Council for Strategic Research is gratefully acknowledged.

References

[1] Giuseppe Lo Presti, Olof Barring, Alasdair Earl, Rosa Maria Garcia Rioja, Sebastien Ponce, Giulia Taurelli, Dennis Waldron, and Miguel Coelho Dos Santos. Castor: A distributed storage resource facility for high performance data processing at cern. In *MSST*, volume 7, pages 275–280. Citeseer, 2007.

[2] Linear tape open consortium. http://www.lto.org/.

[3] Lto roadmap. https://www.spectralogic.com/features/lto/.

[4] Bacula. http://www.baculasystems.com/.

[5] Bareos - backup archiving recovery open sourced. https://www.bareos.org/.

[6] Amanda, the advanced maryland automatic network disk archiver. http://www.amanda.org/.

[7] Ibm spectrum protect. http://www-03.ibm.com/software/products/en/spectrum-protect.

[8] Veritas netbackup. https://www.veritas.com/product/backup-and-recovery/netbackup.

[9] Xianbo Zhang, David Du, Jim Hughes, Ravi Kavuri, and Sun StorageTek. Hptfs: A high performance tape file system. In *26th IEEE Symposium on Massive Storage Systems and Technology*, 2006.

[10] Leonard Zubkoff and Eric Lee Green. Mtx: Media changer tools. https://sourceforge.net/projects/mtx/.

[11] K. Thompson. Unix time-sharing system: Unix implementation. *The Bell System Technical Journal*, 57(6):1931–1946, July 1978.

[12] David Pease, Arnon Amir, Lucas Villa Real, Brian Biskeborn, Michael Richmond, and Atsushi Abe. The linear tape file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–8. IEEE, 2010.

[13] File system in user space. https://github.com/libfuse/libfuse.

[14] Oracle's storagetek linear tape file system, open edition.

[15] Storagetek linear tape file system (ltfs), library edition.

[16] Ibm lto ultrium 7 tape drive performance white paper. http://www-01.ibm.com/support/docview.wss?uid=tss1wp102594&aid=1.

[17] Charles Antony Richard Hoare. *Communicating sequential processes*. Springer, 1978.

[18] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[19] The Go Authors. The go programming language. https://golang.org/ref/spec, January 2016.

[20] T. Berners-Lee et al. Hypertext transfer protocol – http/1.1. https://www.ietf.org/rfc/rfc2616.txt.

[21] M. Belshe et al. Hypertext transfer protocol version 2 (http/2). https://tools.ietf.org/html/rfc7540.

[22] Rob Pike. Go design detail rationale question - channel close. https://groups.google.com/d/msg/golang-nuts/_Q6FCjWIr18/J5Js1VogjIoJ.

[23] Daniel Stenberg. curl. https://curl.haxx.se/.

[24] University of Southern California Information Sciences Institute. Rfc: 793, transmission control protocol. https://www.ietf.org/rfc/rfc793.txt.

[25] Cinema: the alliance for imaging and modelling of energy applications. http://www.cinema-dsf.dk/.