Mahanaxar: Quality of Service Guarantees in High-Bandwidth, Real-Time Streaming Data Storage

Author(s) omitted
Organization(s) omitted

***Abstract*—Large radio telescopes, cyber-security systems monitoring real-time network traffic, and others have specialized data storage needs: guaranteed capture of an ultra-high-bandwidth data stream, retention of the data long enough to determine what is "interesting," retention of interesting data indefinitely, and concurrent read/write access to determine what data is interesting, without interrupting the ongoing capture of incoming data. Mahanaxar addresses this problem. Mahanaxar guarantees streaming real-time data capture at (nearly) the full rate of the raw device, allows concurrent read and write access to the device on a best-effort basis without interrupting the data capture, and retains data as long as possible given the available storage. It has built in mechanisms for reliability and indexing, can scale to meet arbitrary bandwidth requirements, and handles both small and large data elements equally well. Results from our prototype implementation shows that Mahanaxar provides both better guarantees and better performance than traditional file systems.**

I. Introduction

From the mundane to the exotic, many applications require real-time data capture and storage. Consumers wish to record television programs for later viewing, and can do so with digital video recorders. Security personnel monitor and record from cameras and sensors when guarding secure areas. Scientists must capture experimental and observational data on their first and only attempt, from seismometers to telescopes to test explosions. The base need is the same in all of these case – guaranteed real-time capture of streaming data – but with greatly differing parameters.

In television, a standard NTSC/ATSC signal provides data at around 20 MB/s [1], a rate easily recorded by any standard consumer grade hard drive. By contrast, the Large Hadron Collider at CERN generates data on the order of 300 MB/s after filtering [2], utilizing a large backend system and global network. The data rate of one-shot large scientific experiments may be enormous, limited only by the ability of a specialized data recording system to capture a burst of data all at once.

Sometimes this data is vitally important, at least for a time, and cannot be safely discarded. However, a large subset of this data has the curious property of being mostly "worthless" over the long term. A security camera positioned to watch over a door does not generate a steady stream of "useful" data. If somebody has attempted a break-in, then the data is useful. Otherwise, there is rarely any practical use in retaining a record summarized by "nothing interesting happened." Many types of sensor data follow the same model, often summarized by "nothing interesting here" with sporadic bursts of data worth saving. Unfortunately for data storage purposes, it is often impossible to determine which data is worth saving until well after the fact.

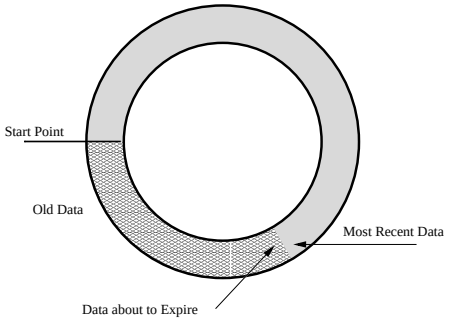

Fig. 1. Ring Buffer Diagram

The storage system for this model is best described by "write-once, read-maybe," or perhaps "write-once, read-rarely." All data needs to be captured and (temporarily) stored in real time, but the odds are good that it will never actually be needed, and can safely expire after a period of time. This is easily conceptualized as a ring buffer (figure 1): if the data is not declared "interesting" within a set amount of time, it is automatically discarded to make room for new data. This is not a difficult problem on small scales, but presents a challenge when dealing with large amounts of data.

We created a prototype system, Mahanaxar, to address this problem. Our first priority is to provide quality of service guarantees for incoming data streams, ensuring that the process of saving and retrieving old data does not interfere with the real-time capture of new data. We also have mechanisms for reliability and indexing, and discuss the problem of scalability. We will first present our design for this class of problem, and then provide results that show superior performance to other methods of managing this type of data.

II. Background

This project was first conceived as a storage system for the Long Wavelength Array (LWA) project [3]. The LWA is a distributed radio telescope currently under construction in southwestern New Mexico. The initial plan is for 53 separate stations scattered widely over the desert. Each station generates approximately 72.5 MB/s of data, for an overall data rate of slightly over 3.75 GB/s. This data is generated continuously and without letup over the lifetime of the project.

Radio astronomy is one of several observational sciences which generates large amounts of "useless" data: in this case,

apparently random radio noise. Since over a petabyte of new data is generated in just over three days, it is fortunate that we can safely throw most of it away. However, it may not be immediately apparent whether the data is useful or not until much later, and we are required to retain it for some time until an outside observer has time to decide whether the data is interesting, and whether it should be preserved.

As we explored this concept, we realized that there were many other applications which generate lots of "useless" data, but deem some of it interesting from time to time. Therefore we decided to develop a generalized model to address all such problems. Broadly speaking, we focused on two canonical real-world problems at opposite ends of our spectrum of possibilities, with other example problems being derivatives and combinations of our two primes.

1) **Fixed-size, non-indexed data:**
   Fixed-size, non-indexed data is generated by the LWA project, and by many types of sensor systems in general. It arrives at an absolutely fixed rate, never varying, and is only indexed on a single variable: time of generation. Oftentimes such data is generated at too high a rate to be captured on a single storage device, and must be broken into multiple streams. Such streams need to be correlated with each other in order to regain the entire data picture. Any command to preserve data will be given according to timestamp only.
2) **Variable-size, indexed data:**
   Variable-size, indexed data describes a data source where the data elements arrive at variable rates and have variable sizes. Such events may also be indexed by time, but also by other attributes as determined by the exact data type. Searching and preserving this data may be done according to any of the indexed attributes. This is a more difficult problem due to the non-fixed sizes and data rates, in addition to the difficulties of a complex index.

While no existing system yet addresses this specific problem, the use of a ring buffer to gather sensor data is not new: both Antelope [4] and Data Turbine [5] use that approach. However, neither system offers quality of service guarantees, only best-effort data recording. Other systems like the network traffic capturing "Time Machine" [6] deal with the problem only by classifying and prioritizing data streams, and dropping what they can't handle. Even then, there are no real time guarantees in the system, and it promises only that it will record data at best-effort capacity, arranged by priorities.

The COSS Storage System from Squid [7] utilizes a ring buffer based model, but also functions solely on a best-effort basis in terms of bandwidth. The mechanism for "preserving" data is simply to rewrite it again at the top of the buffer, which is suitable for cache purposes but not scientific data capture. Larger storage systems such as Lustre do not make quality of service guarantees from moment to moment [8], which is problematic in running a system where the data generation rate is very close to the maximum sustainable bandwidth. Larger systems also have no convenient and automatic mechanism to expire old data when capacity is low.

There has been some quality of service work focused on providing guarantees of a certain service level from the storage system, as in RT-Mach [9] and Ceph [10], but only to the degree of categorizing traffic for an appropriately "fair" level of service. Data streams can be guaranteed to receive a certain portion of system resources in both the short and long term, but the guarantee is of the form "you will get X% of the time every Y time units," rather than an explicit "you are guaranteed a bandwidth of Z."

The disk request scheduling system Fahradd [11] is capable of providing QoS guarantees within certain constraints. Fahradd allocates a certain amount of disk head time to a requesting process, and lets each process spend the disk head time as it sees fit. Unfortunately for the purposes of this problem, that guarantee is not quite strong enough: a percentage of disk head time does not necessarily translate directly into bandwidth guarantees, and we need to guarantee the latter rather than the former.

Because we need to make firm quality of service guarantees, we cannot work with standard file systems or databases. These systems have the benefit of simplicity, but are not designed to work at near-full capacity, and suffer significant performance degradation in such circumstances. A standard file system is capable of handling this class of problem in certain rigidly defined circumstances, but cannot do it well in the general case, and can never offer explicit quality of service guarantees without additional modification.

Since this problem involves constant and uninterrupted writing, we assume that any solution will need to remain based on conventional rotational disk drives for the foreseeable future. Solid state storage devices promise to become prominent in future years, but despite their potential bandwidth improvements, we do not believe that it is wise to use a device with a limited number of write cycles in this task. Write endurance for one of the latest top-rated Intel SSDs is rated at only 1-2 petabytes [12], an upper limit which our system would exceed in months. The use of SSDs for indexing purposes is viable in some circumstances, but we anticipate that standard mechanical hard drives will continue to be necessary for main data storage.

III. Example Use Cases

Two example use cases were briefly described in the last section, standing at opposite extremes of our problem space. The first example use case is based on the type of data which the LWA generates: continuously streaming fixed size sensor data. It arrives at an unchanging bandwidth, needs no indexing, and is uniformly "large." The second example use case is described by the problem of monitoring network traffic: each element is fairly small (often several thousand bytes or less), and non-fixed in size. Each data element must be indexed on multiple variables other than time alone.

We can easily imagine other combinations of data size, size variability, indexing requirements, and arrival rate variability.

However, in addressing the two extreme cases, we should be able to handle anything in between.

A. Continuously Streaming Sensor Data

This type of data arrives at the same rate forever, never varying. The size and layout of each data element is known in advance, or perhaps we only treat it as a stream of bytes, arranging it in whatever manner is most convenient. Interaction with this type of data is extremely limited: we take it and store it with a sequence number (timestamp) and need not worry about it again until it comes time to overwrite it.

If an external process decides that the data is interesting and should be saved, it only needs to tell the storage system that "timestamps X through Y should be preserved" and it is done. The data is marked as preserved on the storage medium, the ring-buffer recording is logically rearranged to bypass the newly-preserved region, and operation continues normally.

This model is relevant in a broad variety of scientific fields because it may take some time to determine whether the data is interesting. A radio telescope may be capture a sudden burst of activity, registering some cosmic event, but scientists also need to know what was happening in the time leading up to that event.

This is perhaps the most basic use case possible in this problem space, but covers a wide variety of systems.

B. Variable-Rate Indexed Network Traffic

In order to detect intrusion attempts into a system, we may wish to monitor network traffic on a particular router. The basic concept of the problem is the same: a firehose of data, most of which is unimportant, but which may become important based on future detection results. However, the specifics of this problem are quite different from continuously streaming sensor data. "Variable" best describes all the major parameters.

First, there is a natural ebb and flow of network traffic depending on several factors, some of which can be predicted, and some of which cannot. For example, we can predict traffic amounts based on the time of day in certain regions of the world. However, we cannot necessarily predict the state of society at any given time, as many things lead to increased or decreased activity: news, sports, disasters, etc. This changes moment to moment, and we can only make rough guesses at it.

The size of network traffic is also variable, as data elements do not have a single fixed size. An IPv4 packet may range in size from a few bytes to tens of thousands of bytes. The data rate may hold steady at X MB/s, but the number of individual elements to consider and index can differ by a few orders of magnitude at extreme ends of the spectrum.

The number of indices per data element is also variable. In the case of an IP packet, time alone is not a sufficient index. To be useful, we must also be able to index and search on aspects such as the source and destination addresses, the protocol, the size, and similar characteristics. This extra indexing poses a further complication when constructing the initial index and when performing subsequent searches on the data.

There are several commercial products which provide network traffic monitoring ability, but without the quality of service guarantees that we desire (aside from the guarantees provided by brute-force overprovisioning). The strategies we need to solve this problem are also useful in many other types of data collection where the rate or size is variable, or where many indexes are required.

IV. Design

We designed Mahanaxar to meet three primary goals:

1) **Provide a quality of service guarantee**
   Our first priority is to provide a quality of service guarantee for the incoming data stream, up to a declared bandwidth. If the incoming data stream requires X MB/s of write bandwidth, we need to make sure that it has X MB/s no matter what. If it exceeds that amount, we'll do the best we can, but make no guarantees. All other activity on the disk must have lower priority, and be carefully managed so that it does not interfere with the recording. We must not lose a single byte; all other tasks, including reading the data back off the drive, must wait.
2) **Use commodity components**
   We want our system to run on commodity hardware in a variety of locations. In the case of the LWA project, the physical location may be a small outpost in the desert. We cannot assume a high-end network infrastructure or storage backend, or highly reliable (and expensive) disk drives. Conversely, if we do have a dedicated machine room available, it would be foolish not to take advantage. In no case do we want to attempt to solve the problem by "throwing more disks at it" until it works.
3) **Never lose data**
   The data that we collect can never be regenerated. If there is a hardware failure, and there are always hardware failures, we need to be able to retrieve the data on demand. However, any reliability mechanism we use must not compromise the first goal, which is providing a quality of service guarantee.

These goals guided our thinking when designing Mahanaxar. We will now present the specifics of our design, along with the rationale and subsidiary goals behind each of them.

A. Staying Close to the Hardware

One of our first design decisions was that we needed to stay very close to the hardware. In order to assure quality of service, we need to know what the underlying hardware is capable of, and more importantly, what it is actually doing at any given moment. This is particularly important in rotational disk drives, as performance can differ by several orders of magnitude based on the access pattern. We need to carefully map out hardware capabilities before organizing our own layout. It may be that we need to avoid certain regions of the disk drive which cannot guarantee the data rate we need.

As an example of why we need this mapping of the hardware, consider one of the hard drives we used for testing: a 1.5 TB drive from Western Digital (model number WD15EARS). The first quarter of the drive (measured from the outermost track) provided a constant minimum write bandwidth of 68 MB/s or better. The last quarter of the drive (innermost tracks) could manage a consistent minimum write bandwidth of only 52 MB/s. The graph of its performance is shown in figure 2. Other disk drives we tested showed similar patterns, with higher capacity drives showing a sharper dropoff towards the "end" of the disk.

This information differs from hard drive to hard drive, even on those of the same make and model. In fact, an identical hard drive to this one was approximately 2 MB/s faster over most of the drive, and significantly slower near the end. Since we want the best possible performance from our hardware, it is critical to have this information for each drive. Continuing with the hard drive above, we can safely advertise a bandwidth of perhaps 50 MB/s over the entire drive (allowing a bit of slack for other drive activity). However, if we use only 80% of the drive in the uppermost region, we can advertise a bandwidth of around 65 MB/s instead, a significant improvement.

To take advantage of this knowledge of hardware, we must use the disk without any interface layers. We envision turning our prototype system into a specialized file system in the future, but for current purposes in our prototype, we treat the disk as a raw device and manage our own specialized layout.

B. Chunk-Based Layout

In order to take maximum advantage of our hardware knowledge, we must restrict the data layout. Modern filesystems are generally good at data placement, but prone to fragmentation over time. This fragmentation problem is dramatically worse when operating in a system at 99%+ capacity at all times, as we intend. Unless file placement is rigidly planned out in advance, fragmentation will quickly add up. Bandwidth is very difficult to guarantee when related data is scattered all over the surface of a disk rather than clustered together.

To solve this problem of data layout, we take a cue from the traditional 512-byte disk block, and declare that no data may be written in a segment smaller than the chunk size. Chunk size is customizable based on the exact sort of data that the system is storing, but as a general rule of thumb, the bigger the chunk, the better. The time required to write 1 KB to a disk drive is most often dominated by the seek time and rotational delay as the disk head moves to the correct portion of the drive. These same factors are diminished into near-insignificance when writing a single 50 MB chunk to a sequential area of the drive, where the actual writing time dominates.

It is well known that data sequentiality has a very large impact on overall bandwidth [13], and we attempt to exploit this factor as much as possible. There are certain disadvantages in dealing only with very large chunks, but what we lose in flexibility and packing efficiency, we make up on raw bandwidth. As long as we follow a basic "don't be stupid" rule (for example, we should not use a chunk size that is slightly smaller than twice the data element size), there is minimal inefficiency.

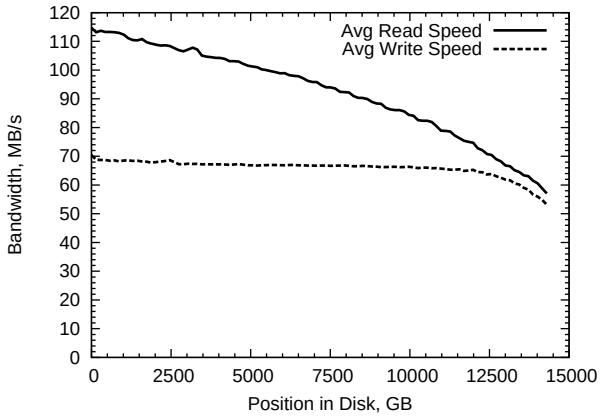

Fig. 2. Average read and write speeds on a particular disk

By strictly maintaining this chunk size and forcing all incoming data to conform to it, fragmentation problems are practically non-existent. The worst case scenario possible is this: there are only two "free" data chunks in the system, and every other chunk is marked as preserved. These two chunks are at exactly opposite ends of the disk, the outermost and innermost tracks, and the disk head must constantly jump back and forth between the two. Even in this scenario, since chunk sizes are large and immutable, seek time between them is only a tiny part: on the order of a few milliseconds, compared to about a second for the chunk as a whole.

The worst case scenario for a less strictly controlled filesystem might scatter the chunks over the entire surface of the disk, anywhere there was spare room, in the 99%+ full system. This behavior drastically increases the total writing time because of the large number of seeks. We wish to avoid this scenario at all costs.

This approach presents no problems with fixed-rate continuously streaming data, since we can easily pick the ideal chunk size based on the incoming data. If one "data element" is the same size as one "data chunk," we have no indexing difficulties and no packing inefficiency.

When data elements are small and variable in size, we must pack multiple elements into each chunk. This may create packing inefficiency as portions of each chunk are left unfilled, too small to hold an additional data element. If chunk sizes are chosen particularly unwisely, up to 50% of the drive may be unutilized. However, this is easily mitigated by carefully choosing the chunk size, or by splitting elements into two portions if necessary. A greater problem is indexing, which we address in a later section.

C. Disk Structure and Consistency

Standard file systems store their indexing information on the disk itself for two main reasons. First, holding the entire

disk index in memory is at best inconvenient, and at worst impossible, depending on the amount of RAM. It also is not necessary most of the time because large portions of the file system are not accessed for large periods of time. The second reason is far more important: in the event of a system crash, it is far easier to recover file information from known portions of the disk than it is to traverse the *entire* disk partition at mount and reconstruct the entire file system anew each time.

We can take substantial advantage in this area by noting that our chunk sizes are both uniformly large and deterministically placed. The only information that Mahanaxar requires in order to understand physical disk layout is the chunksize, the number of chunks, and a possible list of skipped sections within the disk. This information may be thought of as akin to the superblock in a standard file system, and is the only disk structure information which must be stored on the disk itself – and even that may be skipped, if the information is provided by an external source prior to mount.

The chunk index itself is only a list of sequence numbers (timestamps) and a few state variables (for example, marking whether the chunk is preserved), and must be kept in memory at all times in order to know which data is the next to expire. It might also be kept on disk in event of a crash, but that approach would mean frequent seeks to an index area, a waste of bandwidth.

The implications of these observations are that we can hold the entire index structure in memory, and need *never* commit it to disk. We gain measurable performance advantages by only writing the data itself, rather than constantly updating an index or on-disk metadata. The only real disadvantage is in reconstructing the index if it is ever necessary, due to a failure of some sort.

However, we also observe that this is a system which is never supposed to go offline. If it does go offline, there has been a problem of some sort (perhaps through a power failure), and there must be a backup plan available to ensure that data is not lost. Because of this, the startup time of a failed system is much less of an issue, even if it happens to take a few more minutes than usual.

The greater reconstruction time for the index is a small price to pay for increased overall performance. This is particularly true since in the event of a system crash, the disk drive would need to be rescanned for consistency anyway. We do not anticipate this type of system ever shutting down in normal conditions.

D. Reliability and Recovery

Storage systems fail from time to time, both from recoverable crashes and via outright hardware failures. When this happens, we must take two things into account: the ongoing data collection must not be disrupted, and we must be able to recover lost data if the failed drive is entirely dead.

This problem is easily addressed by redundant drives in a smaller system. For example, each LWA station generating data at 72.5 MB/s may be best backed up by a redundant drive, or a second computer entirely if funding is available. Since each station is independent from the others and possibly not connected to a network, the simplest solution is probably the best, and we need not consider it further, other than ensuring the drives are matched in their capabilities.

The far more interesting case is a large system, where a total mirroring of drives is inefficient and uneconomical both in terms of monetary cost and power consumed. A far more elegant solution is available, and is an old familiar one: RAID.

A conventional RAID system provides fault-tolerance and even certain performance advantages with the proper workload, but is disadvantaged when a drive has failed and the system must operate in degraded mode. Read times often increase dramatically since the data from an entire series of drives must be reassembled for every single read. Writing an entirely new stripe of data into a degraded RAID system will often not hurt performance, and ironically may even increase it slightly due to one less disk being involved.

Recalling that our system can be characterized as “write once, read maybe,” it becomes apparent that the disadvantages of a RAID system may never actually come into play. When a disk fails, it is entirely possible that none of its data is “interesting” and we *never* need to reconstruct it. In fact, all data stored on that disk will expire within a matter of hours unless the system is specifically instructed to preserve a section. We may need to regenerate a portion of the data, but almost never will there be a case in which we have to regenerate an entire disk’s worth of data.

This technique works best when the data chunks in a RAID group are all related to each other. For example, an “ideal” RAID group might be a single 300 MB/s stream broken up into five 60 MB/s streams going to five different drives. In this case, an order to preserve data would be given to all drives simultaneously, and there would be no need to preserve unwanted data. Even reconstruction of data for storage elsewhere would be easy, since the same chunks from the working disks would be read in either case.

Unfortunately, if the data chunks are not related to each other, there is a potential downside. If there are five separate streams of data, the preservation of any given chunk in a stream would require that four other “unneeded” chunks be saved for redundancy purposes. Collection of data would never be impaired and quality of service guarantees would be unaffected, but total capacity of the buffer would be reduced unnecessarily. For this reason, it is preferable that chunks in a RAID group be highly related.

We are not limited to standard RAID alone, as any erasure-correcting code would work equally well. Reed-Solomon codes (as an example) are not often used in high-performance storage because of a high computational overhead for encoding and decoding. Because of our coordinated chunks and write-intensive workload, such codes have a lower performance penalty, and may be worth considering in future work.

E. Indexing

It is difficult to design a general solution for the problem of indexing. If we only need to index a timestamp for each (large)

data element, there are few problems. If we need to index four different factors of a twenty byte data packet, indexing is a problem no matter how we try to solve it. Nonetheless, we must be able to index data at its arrival rate, and search it efficiently upon request.

We address the simple problem first. If data elements are large and indices are few, we can keep an entire searchable index in main memory. This describes the type of searching we must do with many types of continuously streaming sensor data, including the LWA, where the only required index is time. For an example calculation, assume that our data elements are 50 megabytes in size, and indexed by an 8-byte sequence number (timestamp). The entire index is only a few hundred kilobytes in size when using an entire 1.5 terabyte drive. Reduce the data element size down to only a few kilobytes and the size of the entire index is only a few gigabytes in size, easily held in memory for standard commodity systems of 2010.

It is a far more complex problem when data elements are tiny and there are multiple factors which must be indexed. Consider the problem of storing IP packets that are indexed on source and destination addresses (4 bytes each), protocol (1 byte), and data length (4 bytes). Furthermore, assume that each of these data packets are tiny for the worst-case scenario: 20 bytes each. The indexing in such a scenario would run to hundreds of gigabytes. In fact, the indexing in such a case would be 13/20 of the data itself. While this is an unlikely scenario, a more reasonable scenario may still include an index large enough such that it cannot be stored entirely in main memory.

If there is no room in main memory for the index, we must clearly divert at least a portion of it to secondary storage of some sort. We have developed two ways of doing this, and implemented the first into Mahanaxar while we consider how to best implement the second.

Out first solution is to attach an “index” segment to each chunk and commit it to disk alongside that chunk. We maintain a bird’s-eye index in main memory, but details are stored on disk. If nobody ever inquires about the data in that particular chunk, the index segment expires at the same time as the data segment. If a search is performed, we can narrow down the potential chunks as much as possible with our bird’s-eye view, then read the necessary index segments for a more detailed search.

Unfortunately, this search is necessarily quite slow because our ability to read from the disk is limited by the quality of service guarantees we make for incoming data. It is entirely possible to miss the opportunity to preserve data because our search is far slower than the speed at which incoming data overwrites old data. We can mitigate this effect, partially, by marking all chunks currently being searched as temporarily preserved.

This problem cannot be solved so long as indexing information resides on the same disk as the data itself, which led us to a second solution: a secondary disk designed to store indexing information only. Indexing information is (usually) much smaller than the full data element, which would allow a single indexing disk to hold the indexes from several data disks at once. This is not a perfect solution since it depends on a secondary disk being available, and creates potential new reliability issues if the index is only stored on that one drive. However, it allows a very large speedup in search speed, which may be worth the extra cost in some situations.

We have considered using an SSD for the secondary index drive. As previously discussed, SSDs are not suitable for data drives in our model, but may be for the far smaller indexes. The vastly superior read bandwidth available also contributes to speedy searches on large datasets.

If we are only indexing a few well-ordered indices, we have elected to handle the search within our own system. If we need to search on multiple variables which are not well ordered amongst each other, we determined that it would be best not to reinvent the wheel, and pass the problem to another mechanism well suited to the task: a database. We create a database anew with each search, using the indexing information from whatever data chunks are needed. Following the results of the search, the database is dropped entirely, never being used for more time than it takes to complete the search. This “lazy search” allows us to optimize data storage according to our own bandwidth needs, but pass the search problem to a mechanism better-suited to handling it.

V. Scaling

Our prototype system is mainly concerned with the problem of guaranteeing quality of service from single data streams onto single disks. We can take multiple data streams and route them to different disks and create RAID groups within the same system, but have not yet addressed the larger scaling problem involving multiple sites and systems. The LWA project involves only 53 stations at the start, but what if it were to expand to 500 stations with more interconnectivity? We need to understand how to best scale upward.

Because our model is tied so closely to the hardware, we can easily scale up the data capture portion. Each disk is bound to a single data stream and need only concern itself with putting that data stream on that disk, and reading other sections as requested. An external process is responsible for giving instructions to preserve or read data, and from the disk point of view, the only logical connection it has with any other data stream is when it happens to be in a RAID group.

From a control point of view, hundreds or thousands of streams may be tied together, and a controller process may need to preserve data over a thousand different disks at once. We have treated this as a communications problem thus far and not focused on it, but we intend to fully explore it at a later time. For the moment, we prefer to focus on the quality of service and individual indexing issues.

VI. Prototype Architecture

Our long-range intention is to create a specialized filesystem and interface layer, but for prototype and testing purposes, we first created Mahanaxar. It is a multithreaded process which

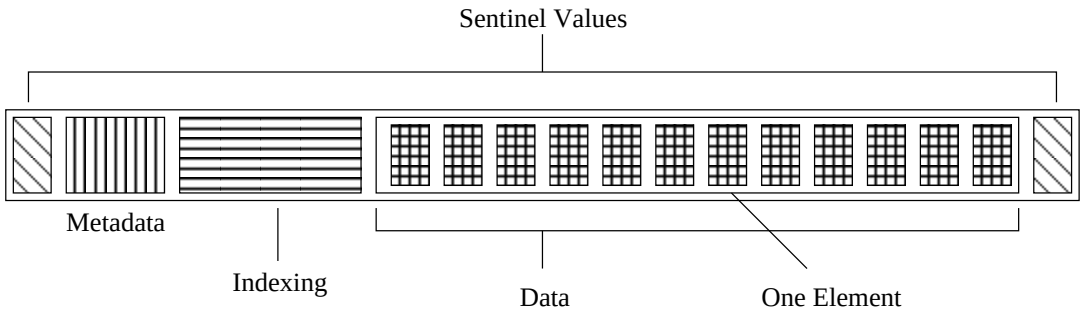

Fig. 3. Data chunk layout

runs in userspace and accesses disk drives as raw devices. Multiple processes may run on the same machine, one process per disk, and an additional process may be configured to accept multiple data streams and combine them for RAID reliability purposes. All access to a given disk *must* be through its associated process in order to manage the bandwidth. Each process/disk is governed by a configuration file that specifies, among other things, the chunk size, the element size range, and the method of indexing. State can be restored from a shutdown or crashed state by scanning and re-indexing the disk.

Each process runs several threads, of which the two most important are data processing and I/O. The data processing thread is responsible for arranging the data elements into chunks and indexing it. An example data chunk layout can be seen in figure 3. The components are:

1) **Sentinel Values** are intended to ensure that a chunk is marked inconsistent if the system crashes in the middle of a write.
2) **Metadata** is present for system restart purposes, and describes the exact layout of the chunk index and set of data elements.
3) **Indexing** is described in a previous section.
4) **Data and Elements** are where the actual data itself is stored.

The second main thread is intended for I/O. Our I/O model works on a simple priority scheme: if there is a chunk of data ready to be put on disk, it gets first priority and is written as soon as possible. Data is only read if there is no chunk which currently needs to be written.

This method produces a jagged access pattern, especially for reads, since nothing may be read for some time, and then an entire chunk is read and delivered all at once. This is an unfortunate but necessary effect of making quality of service guarantees and maximizing bandwidth, since we do not wish to fritter away disk head time seeking back and forth over the disk in pursuit of a smoother curve on a bandwidth graph. Long-term trends average out the bandwidth into smooth curves, which we feel is acceptable when considering the sheer amounts of data we are attempting to deal with. In the short-term view, while a read request is never entirely starved, it may be forced to wait some time.

As each data element arrives at Mahanaxar, it is immediately indexed and placed into a chunk. If element sizes are large, one element may be equal to one chunk. If sizes are small, hundreds or thousands of data elements may be placed into a single chunk. Default primary indexing is based entirely around chunks, and is made up of a pair of timestamps. The first marks the time at which the first element started arriving, and the second marks the time at which the last element has fully arrived. Other indexing schemes are possible, and we use only a single sequence number for LWA-type data.

The chunk size may be configured according to system requirements, but we have found that larger chunks provide the best performance. We prefer to use a rule-of-thumb that one chunk should be approximately the amount that the disk takes one second to write to disk. In order to minimize wasted space, chunk size should be arranged to be a close multiple of the typical element size, and including a margin for metadata and indexing information. If elements are X MB each, a chunk size of $2X - 1$ MB would be a very poor choice.

If the main memory is of sufficient size and elements are sufficiently large, each element can be individually indexed by timestamp and possibly other "primary ids." Mahanaxar does its best to store as much indexing information as possible in main memory, so that searches can be performed quickly, but in certain scenarios, it may not be able to index anything more than timestamp ranges in main memory.

Unfortunately, in order to maintain rigid chunk size and available bandwidth, we must save an entire chunk at a time, even if only a single small element of that chunk is actually desired. We are currently working on a way to accumulate individual elements on the fly as they pass under the disk head for later storage in dedicated chunks.

VII. Testing Procedure

We designed our tests to focus on the raw bandwidth under several different workloads. We used several different disks on the same machine and achieved similar results for each disk, adjusted for its available bandwidth. All the results presented here are based upon one particular disk (profiled in figure 2) so that we make fair bandwidth comparisons between tests.

Our testing machine used an Intel Core 2 Quad processor clocked at 2.83 GHz, with 8 GB of main memory, and an operating system of Debian 5.0 ("lenny"). The particular disk used for these results was a Western Digital Caviar Green of 1.5 (decimal) TB advertised capacity. Our own reported

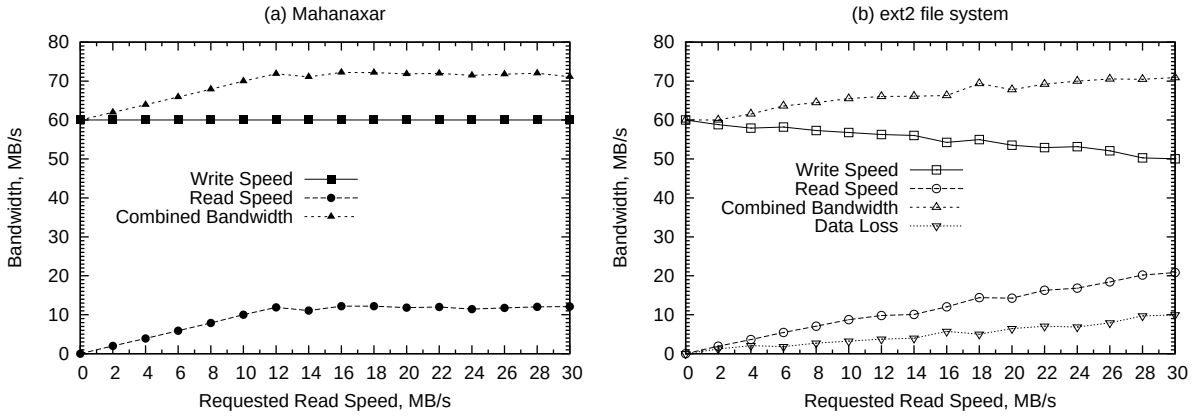

Fig. 4. Comparison of Mahanaxar and ext2 filesystem with an incoming data rate of 60 MB/s and increasing requested read speeds

measurements shall be understood to use the binary convention of KB, MB, etc.

The raw write bandwidth of this particular disk averaged from 50 MB/s to 70 MB/s, and its read bandwidth from 57 MB/s to 115 MB/s. Upon profiling the drive, we determined that the read bandwidth decreased in an approximately linear fashion from its peak at the outermost tracks of the drive at 115 MB/s to about 70 MB/s at a point about 80% into the drive. Bandwidth dropped sharply in the last 20% of the drive to a low of around 50 MB/s. Meanwhile, the write bandwidth only dropped from 70 MB/s to 65 MB/s in that same 80% of the disk, and sharply decreased in the last 20%.

Due to this behavior, we elected not to use the lower/innermost portion of the drive. This allows us to offer a sustained write bandwidth of 60 MB/s with 5-10 MB/s left available for reading. This gave us a usable disk size of about one (binary) terabyte.

We disabled disk write caching for our testing so that we could be (reasonably) sure that the data was on disk when we thought it was, and we ensured that the disk was fully synchronized with each chunk that was written. Interestingly, disabling the write cache slightly improved overall write bandwidth on our disks.

Our primary comparisons were made against the ext2 file system utilizing flat files. We also tested against ext3 and XFS, each of which had worse performance than ext2, which we attribute to their journaling nature. Journaling file systems impose unnecessary bandwidth on the disk for problems of this nature. In order to give the file system write caching an opportunity to reorder its disk access patterns as necessary, we refrained from explicitly sync'ing to disk after each chunk, as we did with our own system. Instead, we only explicitly synchronized to disk every few hundred megabytes, which was several seconds worth of writing. We would prefer to have tight sync'ing to disk to keep the same consistency as Mahanaxar, but we discovered that ext2 performed much better when explicit synchronization was rare.

We had also intended to compare against a pure database model, but quickly discovered that performance was extremely poor as we approached the limits of the disk. The initial population of our database on the first cycle of data was of comparable speed to our ext2 based system, but performance quickly dropped to less than a third of ext2 when constantly expiring old data in favor of new elements. We therefore stopped testing against a database model and focused on our comparisons against the regular filesystem.

Our primary testing procedure was to select various element and chunk sizes, then measure the bandwidth in both writing and reading. We partitioned the "uppermost" 80% of the drive and ran tests utilizing the entire space for some of the results here. For others, we created smaller partitions within that space in order to gather data of a finer grain. All results are gathered from an "aged" system which has been in operation for several full cycles over the space of the disk, unless otherwise noted.

We present here only those results for which our comparison ext2 filesystem achieved stable performance. Certain of our tests led to an ever-decreasing performance over time as the entire system continued to age, and continued that decrease over many full disk cycles. For example, when dealing with highly variable element sizes, the standard filesystem had to constantly delete a variable number of elements and create new elements of different size. Because the file system was operating at 99%+ of capacity (as intended), fragmentation problems built up very quickly and data locality was destroyed.

Because of this characteristic, we hypothesize that fragmentation would continue until blocks belonging to the same element would only be physically consecutive by happenstance. Therefore, we discontinued these tests and instead used a constant element size which we overwrote in-place on the standard file system in order to give it as much of an advantage as possible.

Our own system, Mahanaxar, is designed to deal with this variable element size without changing its mode of operation and thus its performance never declined over time. Because its design packs variable element sizes into fixed chunk sizes, the graphs presented in the next section are identical to those from our variable element size testing, except in "packing efficiency."

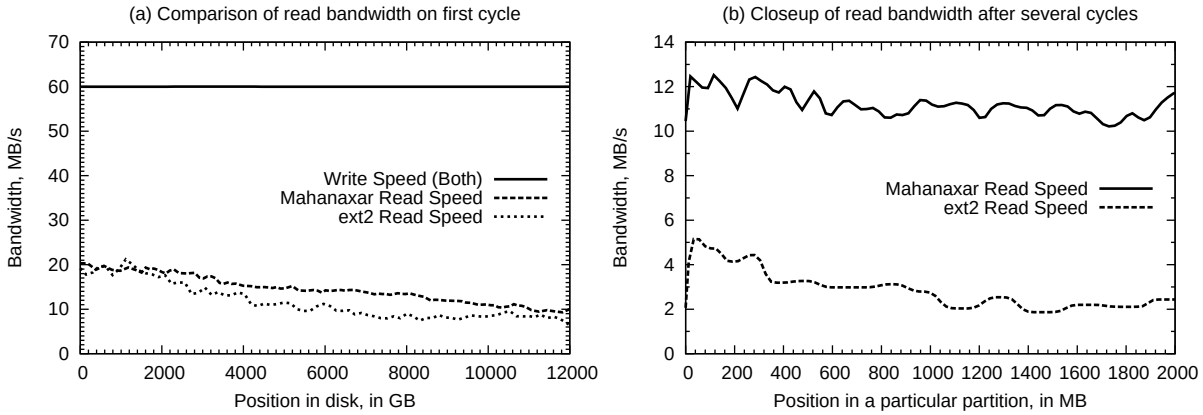

Fig. 5. Performance of Mahanaxar versus a regular filesystem with strict priorities, overall view on first cycle (a), and closeup view of a single partition after multiple cycles (b)

VIII. Results

Figure 4 is a basic comparison of Mahanaxar versus a normal ext2 filesystem with no modifications. The incoming data stream has a bandwidth of 60 MB/s. The element size is also set to be 60 MB. An external process is attempting to read data from the disk at increasing speeds along the x-axis. The total available bandwidth of the disk drive in this region of the disk has ample capacity to write at 60 MB/s, and read at up to 12 MB/s if the writing and reading are properly coordinated.

The values shown on the graph are the average read and write bandwidths of Mahanaxar and the ext2 filesystem over an entire partition. The partition is already populated with existing data (for several cycles) at the start of the test. Mahanaxar maintains a constant write bandwidth of 60 MB/s no matter how much bandwidth the external reading process requests. Up to the physical limitations of the disk drive, Mahanaxar can also provide the requested read speed.

By contrast, the ext2 based filesystem starts falling behind the required 60 MB/s write bandwidth, even when the read process is only attempting to read at 2 MB/s. By the time the read process is attempting to read at 10 MB/s (which the disk can easily handle if managed correctly), over 5% of the incoming data is lost due to insufficient bandwidth. Even at that point, the reading process still can't reach the requested 10 MB/s read speed, being held to about 8.5 MB/s.

The reason for this disparity is that a standard ext2 filesystem manages its bandwidth "fairly" rather than managing it in a way to provide quality of service. Mahanaxar is able to throttle the read requests and prioritizes writes entirely. Because of this disparity, we decided to introduce a similar mechanism for the standard filesystem which ensures writing always has priority.

Figure 5 shows two different views of the comparison against a filesystem with strict priorities introduced, mimicking Mahanaxar. The element size remains at 60 MB/s for this test. Part (a) shows the initial populating of the disk. In other words, this is a "fresh" system on its first round. Both systems are able to maintain a 60 MB/s write speed here. Although Mahanaxar has a slightly higher bandwidth on reading, the two systems are largely equivalent for the first "cycle" through.

Part (b) shows what happens after several cycles, and focuses on the read performance only (write performance remains at 60 MB/s for both systems). This test takes place within a single partition in order to limit the region of the disk which is used. Here, Mahanaxar maintains a read performance of 10-12 MB/s, while the ext2 system drops down to about 2-4 MB/s before it stabilizes. As mentioned before, all elements in this test are overwritten in place. We also wish to note that we used the same in-memory index for the ext2 system as we did for Mahanaxar. If we let the ext2 system rely on filesystem metadata only, in determining which data elements expire, performance continues to decrease steadily over time.

For both the graphs in figure 5, the x-axis has a slightly different interpretation for each system. For Mahanaxar, the x-axis represents the literal position of the data within the disk or partition. In the ext2 filesystem approach, the x-axis only represents the position in the cycle for that particular set of data. The literal position within the disk is determined by the filesystem's data placement techniques, though the units are identical. We focused on a smaller partition for part (b) mainly to ensure that the ext2 approach stayed in the same small region of the disk for a more limited test.

Figures 4 and 5 were both carried out with a 60 MB element size, which is what a regular filesystem can handle best: large contiguous files written and read as one unit. Figure 6 shows the results when we reduce the element size to 1 MB, but leave in the other enhancements to the regular filesystem with regards to indexing and priorities. This results of this test are shown on a single partition for detail, rather than over the whole drive.

Mahanaxar retains a write bandwidth of 60 MB/s, and has an available read bandwidth of nearly 20 MB/s in this partition, for a total I/O bandwidth of around 80 MB/s. This performance is practically identical to when it was working with 60 MB elements because it combines those elements together into 60 MB chunks. This same pattern holds for any element size, as

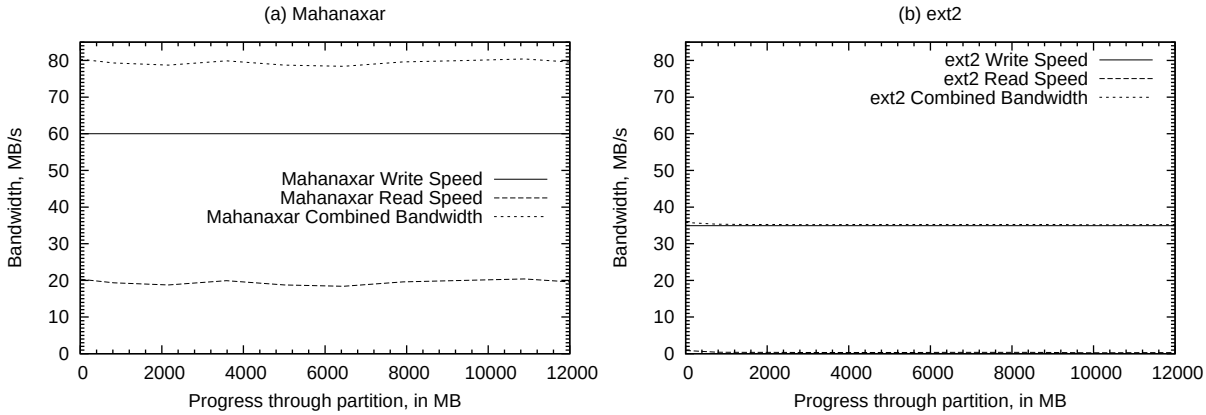

Fig. 6. Comparison of Mahanaxar and a regular filesystem with strict priorities, 1 MB element size. The lines in (b) are extremely close to the lines at 35 and 0, respectively.

we tested in going down to a mere 20 byte element size.

However, the performance of the ext2 filesystem is drastically reduced with 1 MB elements. If we do not explicitly synchronize the write cache to disk, the filesystem can ”pretend” to keep up for quite some time, but it eventually collapses under its own weight. Synchronizing after every element is unrealistic, however, and we only forced a sync to disk every few hundred elements (megabytes) to keep it honest.

We found that the maximum sustainable write bandwidth for the ext2 filesystem was about 35 MB/s. At 38 MB/s and above, it slowly starts falling behind over time and eventually loses data as it runs out of buffer space. At 35 MB/s it loses no data over the course of many cycles, having stabilized, and it can read off data at around 1 MB/s. This combined bandwidth of less than 36 MB/s compares very poorly with Mahanaxar's performance of nearly 80 MB/s combined bandwidth. The performance difference between Mahanaxar and flat files only increases as the element size shrinks even further.

When we tested variable element sizes on the ext2 filesystem, performance decreased steadily over time without appearing to stabilize at any point, and thus we do not have a proper comparison to make against the steady performance of Mahanaxar under the same circumstances. However, the performance graph of Mahanaxar when run with variable element sizes is identical to that of 6 (a).

IX. Conclusion

The performance of Mahanaxar shows that is has a clear edge over standard filesystems in the high-bandwidth "write once, read rarely" workload. By staying very close to the physical hardware and aligning our workload to match, we are able to provide real quality of service guarantees to meet a set of hard real-time deadlines in a high-turnover, high-bandwidth environment. We are able to reach performance levels on par with the tested maximum of individual hard drives, though this depends on generating disk profiles on a per-drive basis in order to maximally exploit the hardware.

Even when standard filesystems are adapted to prioritize data streams and enhanced with a more appropriate indexing capacity, they cannot maintain as high an overall bandwidth as Mahanaxar. Even with the ideal large element sizes, standard filesystems can only come "close" to Mahanaxar's performance. When element sizes are smaller or variable, performance of standard filesystems drops drastically, and they cannot handle variable element sizes in a 99%+ full system sizes at all.

Our future intentions are to turn this project into a full specialized file system, develop an API to interact with it, and develop an interface allowing an arbitrary number of such systems to operate in concert to capture arbitrarily large data streams. We also need to run performance tests on various types of rebuilding after hardware failure, and experiment with using separate "index" drives to improve search performance. Lastly, we need to address the problem of preserving individual data elements within a chunk, and develop a system for scalability.

However, we feel that the raw performance numbers are sound, and promise a substantial improvement over the current systems which cannot offer any quality of service guarantees for this type of problem.

References

[1] *A/53: ATSC Digital Television Standard, Parts 1-6, 2007*, Advanced Television Systems Committee, Inc., 3 January 2007.

[2] L. C. Grid, "Gridbriefings: Grid computing in five minutes," August 2008.

[3] "http://www.phys.unm.edu/˜lwa/index.html."

[4] *Antelope: ARTS configuration and operations manual*, Boulder Real Time Technologies, Inc., 3 November 1998.

[5] S. Tilak, P. Hubbard, M. Miller, and T. Fountain, "The ring buffer network bus (rbnb) dataturbine streaming data middleware for environmental observing systems," in *e-Science*, Bangalore, India, 10/12/2007 2007.

[6] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a time machine for efficient recording and retrieval of high-volume network traffic," in *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. Berkeley, CA, USA: USENIX Association, 2005, pp. 23–23.

[7] A. Chadd, "http://devel.squid-cache.org/coss/coss-notes.txt," 2005.

[8] DataDirect Networks, "Best practices for architecting a lustre-based storage environment," DataDirect Networks, Tech. Rep., 2008.

[9] A. Molano, K. Juvva, and R. Rajkumar, "Real-time filesystems. guaranteeing timing constraints for disk accesses in rt-mach," in *The 18th IEEE Real-Time Systems Symposium*, December 2-5, 1997 1997, pp. 155–165.

[10] J. Wu and S. Brandt, "Providing quality of service support in object-based file system," in *24th IEEE Conference on Mass Storage Systems and Technologies*, 24-27 Sept. 2007 2007, pp. 157–170.

[11] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient guaranteed disk request scheduling with fahrrad," in *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. New York, NY, USA: ACM, 2008, pp. 13–25.

[12] *Intel X25-E SATA Solid State Drive Product Reference Sheet*, 2009.

[13] W. W. Hsu, A. J. Smith, and H. C. Young, "The automatic improvement of locality in storage systems," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 424–473, 2005.

[14] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *IEEE International Conference on Multimedia Computing and Systems*, 7-11 June 1999 1999, pp. 400–405 vol 2.

[15] R. Rangaswami, Z. Dimitrijević, E. Chang, and K. Schauser, "Building mems-based storage systems for streaming media," *Trans. Storage*, vol. 3, no. 2, p. 6, 2007.