Clearing the Obstacles to Backup Pebibyte Size Filesystems

Automating Backups On A 4 Pebibyte Filesystem

Andy Loftus, Alex Parga
National Center for Supercomputing Applications
University of Illinois
Urbana, USA
Email: aloftus@illinois.edu
Email: aparga@illinois.edu

***Abstract*—The NCSA Blue Waters supercomputer provides users with two 2 PiB filesystems. Despite RAID mitigating data loss for most hardware failures, the only real guarantor of data availability is daily, automated backups. Currently, no existing products scale easily to backup filesystems in the multi-PiB range. A few Lustre specific solutions exist; however, they depend on access to the underlying hardware, which is not only risky, but unnecessarily ties the solution to a specific implementation. A more generic solution is needed. Building a successful backup solution requires taking advantage of existing filesystem features, namely parallelism. By distributing the workload of backups across the many nodes already available in the cluster, backups become a simple problem of workload management. NCSA designed and prototyped a distributed backup solution that addresses these issues.**

Keywords-backup, disater recovery, Lustre, parallel filesystem

I. PROBLEM

The increasing size of disk drives and the advancement of data storage technologies have enabled filesystems to grow to sizes larges enough that existing backup solutions are inefficient. Filesystems in the PiB range are common now and EiB size filesystems are being planned. Despite advanced algorithms to detect and prevent data loss resulting from hardware failures, the only real guarantor of data availability is a backup. A backup, at the simplest level, is a copy of every file in a different location. The main problem with this is the sheer volume of data to be scanned and copied. It simply is not feasible to copy all the data to a remote location in a reasonable amount of time. Typically, a backup should take no longer than 24 hours; thereby allowing a backup of user data on a daily basis.

II. MOTIVATION

To date, there are a few backup solutions for filesystems in the PiB range, however they are tied to a specific architecture (ie: IBM GPFS and IBM Tivoli) or they are tied to a particular implementation (ie: mount the ext3 filesystem that Lustre is built on top of). These solutions are suboptimal as they are tied to a specific version or implementation of a filesystem and therefore do not upgrade easily or port to other systems or sites.

III. GOALS

The primary goal is speed. The backup solution should be fast enough to complete in less than 24 hours to provide daily backups. Additionally, the backup solution should be transparent to users of the filesystem, which means it should not induce significant load on the filesystem. The final goal is portability to different filesystems and different implementations. To be portable, the solution must not rely on any internal features or knowledge of the underlying filesystem. The solution should interact with the filesystem through either normal user interfaces or programmer APIs.

IV. APPROACH

The difference between a pebiscale filesystem and a usual desktop backup is size. Breaking the backup of a pebiscale filesystem into many small backups will allow any standard backup tool to be used for archival purposes. The main job of the software will be creating and managing parallel backup tasks.

V. CHALLENGES TO PARALLEL BACKUPS

There are four main challenges to automating parallel backups. The first challenge is to split up the filesystem into small chunks that can be backed up in parallel. Second is parallelizing the execution of the individual backups. The third challenge is transferring the backups in parallel to a secondary location and the final challenge is dealing with full backups.

A. *Splitting Up The Filesystem*

The primary requirement for splitting up the filesystem into multiple chunks is that the chunks must not overlap. The natural layout of the filesystem into home directories and project spaces provides a natural structure of non-overlapping directories. Automating the splitting activity is accomplished by creating a list of top level directories to serve as launching points. The top level directories, referred to as topdirs, are listed manually in a configuration file. The software then scans one layer below each topdir and all directories found are recorded as backup targets, referred to as basepaths.

B. *Parallel Backups*

With the filesystem splits defined, each basepath must be backed up daily. Performing these backups in parallel is a relatively standard exercise in job management. On a typical cluster, it would seem reasonable that the existing job scheduler could be utilized for running these backup jobs in parallel, but that is not possible since user jobs on a cluster typically cannot run as root due to security practices. The backup job requires root permission for access to all files on the filesystem. Since the standard scheduler cannot be used, a custom solution is needed. Distribution and execution of backup jobs is accomplished using RPyC [1]. The RPyC python module provides a framework for connecting to and running tasks on remote hosts. The status of each basepath is maintained in a file on the shared filesystem. Using a file allows any backup to run from any node participating in the backup solution.

C. *Parallel Transfers To Long Term Storage*

A successful backup yields an archive file, which must be transferred to a secondary location. Many existing solutions are available to efficiently transfer numerous, large files. Blue Waters makes use of Globus [2], which is already configured for both the online filesystem and the nearline archive system. Globus is a third party file transfer service that connects to predefined endpoints and initiates a transfer directly between the endpoints on behalf of the requesting user. Automated usage of the service is possible through use of the provided software APIs.

D. *Handling Full Backups*

When backups run for the first time, every basepath will need a full backup so all data will necessarily have to be scanned and transferred. There is no way around this; fortunately, it only has to be done once. However, with a policy in place that dictates a full backup every 30 days, the problem turns into a recurring issue. This problem is addressed using a scheme of rolling full backups. The idea behind rolling full backups is to spread out the load of full backups over the total cycle length of 30 days, so only a few full backups run each day. The implementation involves assigning a distinct integer identifier to each basepath (basepath-id). Using the modulus operation on the basepath-id and the cycle length determines what cycle day the particular basepath receives a full backup. The current cycle day is calculated by modding the day of year (julien date) with the cycle length. For each basepath, the decision for which type of backup to do is then given by:

```
backup_type = INCR
full_cycle_day = basepath_id % cycle_length
today_cycle_day = day_of_year % cycle_length
if full_cycle_day == today_cycle_day :
    backup_type = FULL
```

The reason for a policy of new fulls every 30 days is to avoid complex restores requiring many, potentially costly, archive retrievals from long term storage.

VI. AUTOMATING BACKUPS

The next step in automating backups is to design software to implement the proposed solutions to each of the challenges. The main job of the software is to parallelize backups. Parallelizing backups is accomplished by distributing the work among many nodes, therefore the software must be fault tolerant of problems such as network outages, filesystem outages, and node failures. To meet the overall goal of portability, the software must be also scalable and flexible.

VII. SOFTWARE DESIGN GOALS

A. *Fault Tolerance*

Fault tolerance is addressed through two features: basepath status files and small, restartable tasks. A task here refers to any piece of work that is requested of a worker node. When a worker node receives a task request, the worker node checks if the task has been started. The check is facilitated by a status file saved on the filesystem. If this is a new task, then a new file, or entry in the file, is created and the task proceeds. If the task was previously started, but interrupted, the task is either restarted, or if possible, resumed. When the task is completed, the status file is updated accordingly. Since the filesystem is shared any node has access to the status file. Any worker node can be assigned any task. Lockfiles are used to prevent simultaneous access to status files. This is sufficient to handle network failures and node failures. Monitors and timeouts are used to halt non-progressing tasks when there are filesystem failures. Tasks are either resumed or restarted when the filesystem failure is remedied.

B. *Scalability*

Scalability refers to adding more nodes to handle larger filesystems and improving performance. One goal of scalability is to ensure that the addition of more worker nodes does not significantly increase network traffic. This goal is met through both the use of the command and proxy design patterns and the core design of an event driven architecture. In an event driven architecture, the events are an implementation of the command design pattern. The remote services on each node are instances of the proxy design pattern. These will be discussed in more detail later, but the main issue is that network traffic is limited to simple message passing. The messages are either task requests from the manager to a worker node or a task result passed from a worker node to the manager. A second goal of scalability is that a new node should require minimal setup and configuration. The software must be copied or accessible on the new node and an accessible IP address must be added to the software configuration file. It is implied that the new node has access to the shared filesystem. Storing the software on the shared filesystem eliminates the need to install any software on worker nodes.

C. *Flexibility*

The goal of software flexibility derives directly from the overall goal of portability. Portability requires that the software does not rely on any internal feature or specific implementation of the filesystem. That does not mean that it cannot or should not utilize filesystem specific features that are useful to backups. Here, the point is that any part of the software that uses specific features should be modular so that the software part can be easily replaced or removed. Modularity in software design is a well-known and widely accepted best practice. The major advantages of a modular software design are simplicity of design and ease of understanding, debugging and maintenance. These advantages promote an event driven architecture, but they are not the primary reason it was chosen.

VIII. Event Driven Architecture

The primary reason an event driven architecture was chosen is because it easily fits the model of distributed computing. As tasks are farmed out to remote hosts, they will complete at different times. A central manager must be notified and follow-up steps will need to be scheduled as each task is completed. The completion of a task is encapsulated in a software representation as an event. The event is sent to the central manager, where it is processed and may in turn generate one or more new events. All notifications, requests and responses are events.

A. *Event Manager*

The core piece of event driven architecture is the event manager. The event manager operates like a post office. It receives events and delivers them to the appropriate software modules. The modules, upon receiving notification of an event, perform whatever action is appropriate for that event, and if any new events are generated as a result of the action, send those new events to the event manager. The event manager has four responsibilities: accept events, notify modules of events, accept module registrations, and process events. The first two responsibilities have already been discussed. A module registration is the way a module informs the event manager which events it needs to know about. The event manager does not know ahead of time which modules will need to know about which events. This is an important point as it allows new modules to be added without any modification to the event manager. It also allows new events to be created simply by defining them. The event manager also does not know which events or modules exist. It simply accepts events as they are posted and looks in a table of registrations for which modules to notify, if any. The last responsibility of the event manager is processing events. This is mostly straight-forward, except for special consideration to ensure events are processed in the correct order when successive events are spawned from an initial event. Event processing is initiated by a special ticker event. The timer module generates ticker events at regular intervals that cause the event queue to be processed. Events are processed from the queue until the queue is empty. Events arriving after the queue is emptied are held until the next ticker event is received, upon which the whole process is repeated. The design of the event manager is based on a model described in "sjbrown's Guide To Writing Games" [3]. As mentioned earlier, the events are an implementation of the command software design pattern. The event manager does not know any detailed information about the event, it just passes the event to all modules that requested notification. Any specific information relating to the event is encapsulated inside the event.

B. *Distributed Computing With RPyC*

RPyC is a python module that provides a framework allowing transparent access to remote hosts as if they are local. RPyC defines a service class, from which a custom class can be derived. This new custom class will encapsulate the methods that define the tasks for worker nodes. These methods include: scanning a topdir for new and deleted basepaths, scanning a topdir for basepaths that need a backup, and performing a backup for a particular basepath. The service module is an example of the proxy software design pattern that was referred to earlier, since it provides a simple interface to request the tasks without requiring detailed knowledge of the tasks themselves.

IX. Steps Of A Backup

As all the main parts of the backup system have been defined, the backup process itself can be demonstrated. When the software starts, various initializations occur. Eventually, the topdirs are scanned for new basepaths and if any are found, a *start-backup* event is generated. The event will contain information indicating the basepath name and the type of backup, a full backup in this case. A full backup will also occur if the rolling full check succeeds or if the last full backup is older than the cycle length. Outside of these instances, the *start-backup* event will indicate an incremental backup. The event manager passes the event to a job scheduler module, which will find an available worker node and pass the task to that worker node. The worker host will then begin the backup. After verifying that no other backups are in progress or were previously interrupted, the backup can be started. The actual backup processing is done using dar [4]. Dar scans the basepath for changed files and copies them to an archive file. When completed, dar then extracts a catalog (index of files in the archive) to a separate file. It should be noted that these two steps are separate from each other and if catalog creation is interrupted, it can be restarted without having to rerun the archive step. This is part of the fault tolerance built into the design. At the end of processing, the status file is updated. A *backup-completed* event is created and sent to the event manager. In processing the *backup-completed* event, another module will generate a *start-transfer* event, which is sent to the appropriate module for communicating with Globus to transfer the archive file to long term storage. Likewise, when the transfer is completed, an event will be generated to cause the cleanup activity to start. Cleaning up consists of removing temporary and lock files and finalizing the status in the status file. If an error occurs at any point during the process, an appropriate event is generated to cause a new action to occur.

A. *Advantages of Dar*

The dar tool was chosen for its external cataloging feature. The external catalog contains the metadata of a backup archive file and can be used as the reference for an incremental backup. The catalog file is much smaller than its base archive and can be saved on disk allowing the large archive file to be transferred off the filesystem. The catalog files are also used for restores, since they can be scanned to determine the minimum set of archive files to be retrieved from long term storage for a particular restore. Additional features of dar include cache directory tagging [5] support for excluding entire directories and xattr support.

X. RESULTS

The current implementation of the software shows that, on a 4PiB filesystem with 683 basepaths and 1.6PiB used, daily backups finish, on average, in 9.6 hours. 28 worker nodes are currently in use with one of them doubling as the manager node. The worker nodes have 192 GiB of RAM and run a patched version of the Lustre 2.3.0 client. The worker nodes also double as Globus endpoints for the Lustre filesystem and may be running one or more gridftp transfers at any given time. The filesystem is hosted on a Cray Sonexion 1600 running a patched version of the Lustre 2.1.0 server with 36 SSUs fully populated with 2TB SATA drives. Table 1 shows additional relevant statistics.

XI. FUTURE ENHANCEMENTS

Possible future enhancements include multiple files per Globus task, automated deep scan of basepaths, multiple tasks per worker node, and using the changelog to generate filelists for backup.

A. *Multiple Files Per Globus Task*

The most significant performance bottleneck at this time is the assignment of a single dar archive file per Globus task. The Globus transfer service is intended to transfer multiple files in a single task and only allows a few tasks per user. This results in a low level of parallelism and more time spent waiting for new task slots.

TABLE I. CURRENT STATISTICS

Average time to complete daily backups	9.6 hours
Average full backup time	2.41 hours
Average incremental backup time	9 minutes
Average number of inodes scanned daily	404.5 million
Average number of inodes backup up daily	51.3 million
Number of topdirs	10
Number of basepaths	683

B. *Automated Deep Scan Of Basepaths*

The existing scheme for splitting the filesystem into small parts works well when the basepaths reference directories limited in size, such as user home directories, which are quota limited to 1TiB. Some project directories, however, have grown to 50TiB and larger, which take multiple days to complete a full backup and more prone to failure from filesystem problems. One solution is to scan the basepath to split it into multiple backups based on size thresholds. These multiple backups could then run in parallel.

C. *Multiple Tasks per Worker Node*

Allowing multiple tasks per worker node could increase performance of the backups and possibly reduce the number of physical worker nodes required to complete backups in 24 hours. Since the NCSA worker nodes are also Globus transfer nodes, the software could use current system stats such as system load or size of free RAM to dynamically determine if another task should be allowed on a given node.

D. *Use Changelog To Generate Backup Filelist*

Using a changelog provides multiple advantages. First, a predetermined filelist provides the obvious benefit of precluding a scan of the basepath to determine which files have changed. This is significant since scanning requires a stat of every file and since every basepath is scanned each day, the entire filesystem is scanned each day. Second, the list of files could be used to estimate the resulting size of the dar archive and determine if the backup should be split into multiple parts for parallel execution. A potential problem with this enhancement is that cache directory tagging for excluding directories would no longer be effective, since dar would not be scanning the basepath and thus would not find any cache directory tag special files. Another problem with changelog is that it necessarily ties the solution to a specific filesystem.

REFERENCES

[1] http://rpyc.readthedocs.org

[2] http://www.globus.org

[3] http://ezide.com/games/writing-games.html

[4] http://dar.linux.free.fr/

[5] http://www.brynosaurus.com/cachedir/