Integrity Checking and Monitoring of Files on the CASTOR Disk Servers

Author: Hallgeir Lien

CERN openlab

17/8/2011

Contents

Abstract

For storing physics data from the experiments at CERN, a hierarchial system of tapes and disks are used where tapes are the main media of storage, while disks are used as a cache. Making sure that the files on the disks are not corrupted is important in order to prevent bad files from being written back to tape. Typically this is done by computing a checksum of the files and comparing it with a stored checksum. In my project I implement a file integrity checking daemon based on previous work that aims to continuously check files for corruption and provide logs for external applications, with minimal impact on performance of the disk servers. I also implement monitoring through Lemon (**L**HC **E**ra **Mon**itoring) that gives general statistics, e.g. files checked, etc., and I also develop a monitoring application in Django for more detailed statistics on a per-error basis.

1 Introduction

With an increase of storage space requirements exceeding 15 peta bytes per year at CERN, the number of disks currently deployed at CERN is measured in the thousands, which means that data corruption is not something that is unheard of. At the same time, performing integrity checking on production servers means sharing resources with the users of the servers, which may result in performance hits for the users of the system.

The goal of this project was first and foremost to develop a robust integrity checking daemon for the disk servers at CERN with minimal performance impact on current users, while at the same time performing better than the previous attempt on such a daemon, which was unacceptably slow. I developed improved an improved method for choosing which file to check next, based on which mount point the file resides, as well as improving the way the daemon checks if the system is busy or not. I also implemented threading for increased performance. The daemon outputs the checksum errors to the syslog and an error log chosen by the user, and it also produces summary statistics for how many files were checked, how many bad checksums were found, and so on.

In addition to the checksum daemon itself, I developed a Django web application for monitoring events (i.e. checksum errors) as they happen, allowing the administrators to locate which server or which cluster, or which machine model causes problems in order to replace faulty hardware, etc. As an additional feature, this monitor also parses logs of resolved checksum errors, as well as files lost due to disk and server crashes, etc. A Lemon metric instance was also configured to parse the checksumd summaries.

1.1 Background

In November 2009, the first version of a checksum daemon called "checksumd" was developed by the DSS group in the IT department at CERN. This daemon is used as a basis for my work, although it's heavily modified. For completeness, I will here give a brief overview of the CASTOR system and Django, both of which are relevant to this project.

1.1.1 CASTOR Overview

CASTOR (**C**ERN **A**dvanced **STOR**age manager) is a hierarchical storage management system used at CERN[1] designed to handle the enormous amounts of data produced every year (as per 2011, the storage requirements grow by about 15 petabytes every year).

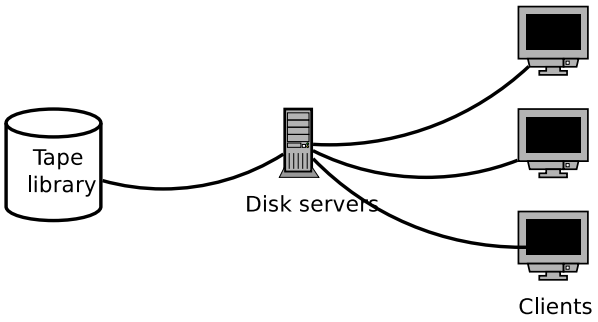

Figure 1: Simplified view of the hierarchial storage system at CERN

There are two kinds of storage media mainly in use at CERN: Magnetic tapes and disks. Tapes are cheap and energy efficient, but slow; disks are more expensive and always use energy, even when not in use, but they are fast. This has been combined at CERN into a hierarchical system where tapes are used as the main media of storage, but disks are used as a cache. When a user requests a file, and the file is not on any of the disks it will be retrieved from the tapes, put on one of the disk servers, and served to the user. If the same file is requested by another user shortly after, it can be fetched directly from disk, eliminating the need to scan the tape drives.

CASTOR provides transparent access to the tape storage by only letting users store and read files to and from the disk cache. Although the fact that there is a tape back-end is not completely transparent due to magnitudes of difference in latencies, depending on wether the file is in the disk cache or not, no user has to explicitly store data to the tapes as this is handled by the CASTOR Stager.

Another important component is the name server, which keeps track of the directory structure and file names. The name server may also contain checksums for the files contained on the disk servers, which may be used as an alternative way of checking the integrity for files that do not have a checksum stored in their extended attributes.

The need for integrity checking The files on the tapes and disks may for various reasons become corrupted. It could happen due to an error while transferring from the tape to the disk, while it's sitting on the disk due to a physical flaw on the disk, and so on. It's important to catch these errors so that if a file stored in one of the disks is getting corrupted, it can be restored from a good tape copy. It's also useful to see if there are patterns of corrupted files, e.g. if there is a specific machine or cluster that has a higher than average error rate due to hardware errors, so that this hardware can be replaced.

CASTOR already stores an Adler-32 checksum for all files. Typically this checksum is stored in extended file attributes, although it can also be stored in the name server. There is already some integrity checking in place, e.g. when a file is transferred from tape to disk; however, there is currently no system running for detecting checksum errors that occurs during the time a file remains on the disk server. With literally thousands of disks, such errors may not be uncommon.

1.1.2 Django

Django is a web application development framework written in Python that aims to handle many of the tedious tasks related to web development, such as URL parsing, session management, database queries and so on.[2] It is based on a model-template-view architecture, where a model is a class with fields mapped to columns in a relational database, the template is the HTML code with support for a special template language used to insert data from views, and the view is the component that retrieves data from the model, and serves it to the template, possibly with additional logic added. The models and views are both written in Python.

1.1.3 Lemon

Lemon stands for LHC Era MONitoring and is a system used for monitoring computing devices at CERN. The system is built up by sensors (which in turn have metrics), agents and a server (or a

Central Measurement Repository). Sensors retrieve data using their defined metrics which can be a single value (e.g. number of files in /tmp), a tuple (e.g. statistics from checksumd) or a matrix. The agent then retrieves the data from the sensors and sends it to the server.

2 Project Overview and Functionality

The project consists of two main parts:

- checksumd is the checksum daemon, written in Python.
- checksum-monitor is a monitor written in Django that parses log files from checksumd to generate statistics and display details about checksum errors.

In addition there is the Lemon integration, but this was purely a configuration task. Please see section 3.3 for details.

2.1 Overview of architecture

An overview of the project can be seen in figure 2. The checksumd program runs as a separate application on each disk server. For each file, it computes a checksum, and if there is an error it outputs this to an error log that resides in a folder in AFS (Andrew File System, a distributed file system used at CERN[4]). These entries consist of the filename, file size, the checksums, the path returned from the name server, the specifications of the machine (hardware model, hostname, etc.) and more. The same entry is also put in the syslog.

The checksum-monitor Django application parses the error logs that were stored on AFS. This can be set up as a cron job, or done manually through the web interface. The application will then parse the logs, organize the entries in a relational database, and display it to the clients according to their query preferences.

With regular intervals (e.g. every 24 hours) a summary is also written to the syslog. This summary consists of, among other things, the aggregated numbers for the number of files checked, bad checksums, good checksums, etc. This is then parsed by the Lemon sensor, which is passed to the agent which passes it to the central Lemon database.

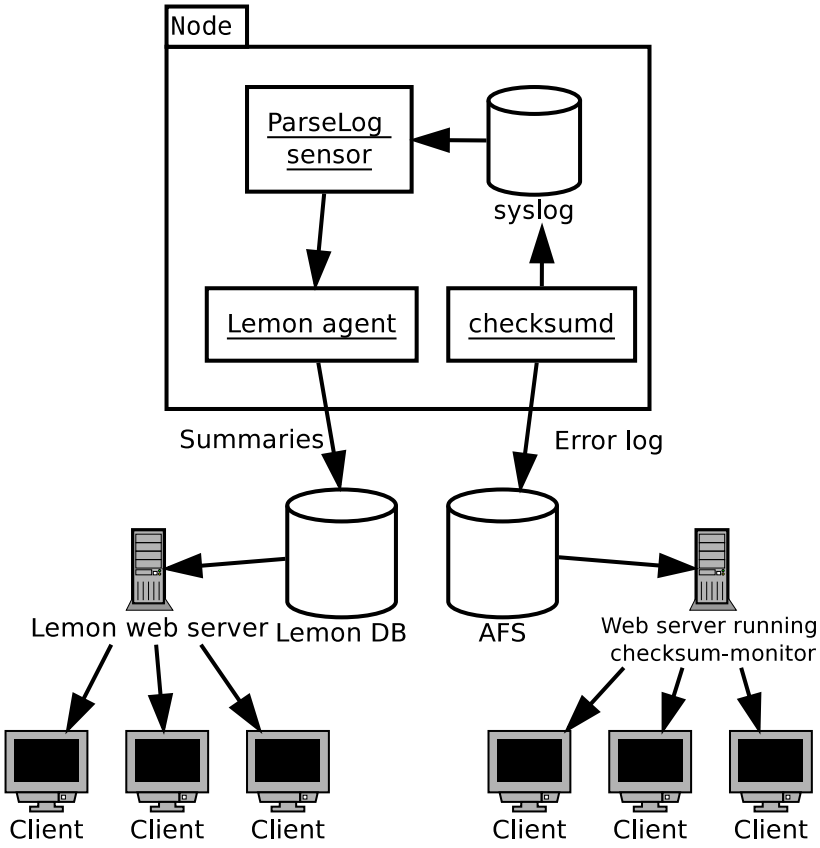

Figure 2: Architecture overview

2.2 checksumd

The checksumd daemon is implemented as a standalone application in Python. In principle, it can be used for other types of machines than CASTOR disk servers, but that is not within the scope of this project. This section gives an overview of the daemon.

2.2.1 Daemon overview

The process by which the daemon checks files is first by building a queue of all the disks. Each disk has a list of files to be checked (all files under the specified directory, that are not recently modified), and each has an index that points to the next file on the disk. Each of the mount points is assumed to be on a separate disk. At regular intervals, the main thread refreshes the file lists by querying for all files again, adding new ones and removing deleted ones.

checksumd is built with threading in mind. There are two types of threads: the main thread, and worker threads. The worker threads do the actual integrity checks, while the main thread is responsible for housekeeping (e.g. keeping the file list updated and writing summaries to the log). The number of worker threads can be specified on the command line.

Until every file on each disk has been checked, each thread will grab one disk, check if it is busy, and if it is not, compute the checksum of the files. If the disk becomes busy while checking, the thread working on that disk will complete the current file it is working on and move on to the next disk. If no more disks are remaining in the queue, the threads will sleep for a specified amount of time before attempting to check again, continuing where they left off. The process is modelled in figure 3.

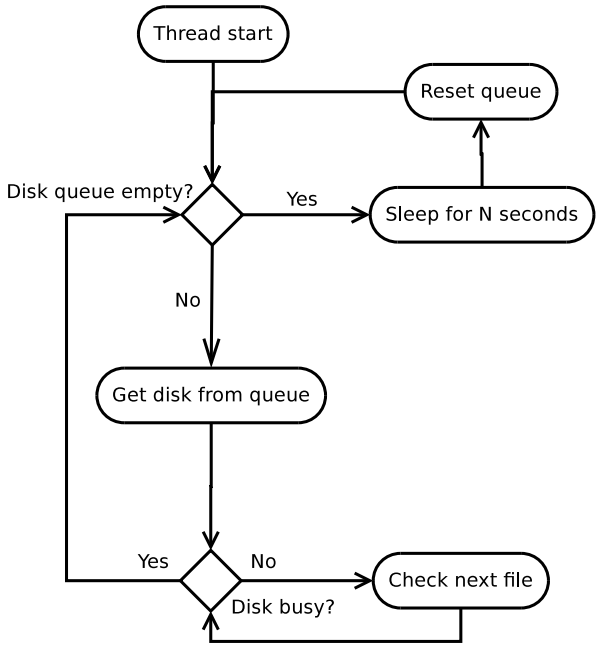

Figure 3: Process for selection of files for checking

2.3 checksum-monitor

A monitor for individual events (errors) and statistics over multiple events has been developed. In addition to displaying the checksum errors from checksumd, it may also display a list of resolved events (e.g. events that the user has been notified about but that were impossible to fix), as well as "lost" files, files that were lost on disk servers due to crashes or other events. The monitor also supports, in addition to listing every event, searching over a specified time frame, on host name, hardware model,

cluster name, and so on. It also displays statistics like counting events per host name, hardware model, and user group. The monitor was fully developed in Python using Django.

A screenshot from the web interface is shown in figure 4. Statistics are shown to the right, grouped on hostname, hardware model and user group. Events can be filtered based on age (e.g. less than 24 hours, less than a week, etc) on hostname (by clicking each hostname), on hardware model, or on group. Since there may be a lot of errors (in e.g. the Lost files category), paging was also implemented, where only a subset of the events are displayed at once. Up to 1000 events can be shown on one page. To switch categories, one can click "From checksumd" for checksumd errors, "Fixed" for resolved errors and "Lost files" for files from the lost files logs.

The entries can be sorted by the fields with a blue color in the header. Clicking the header will sort by that field, and clicking it again will reverse the sorting direction.

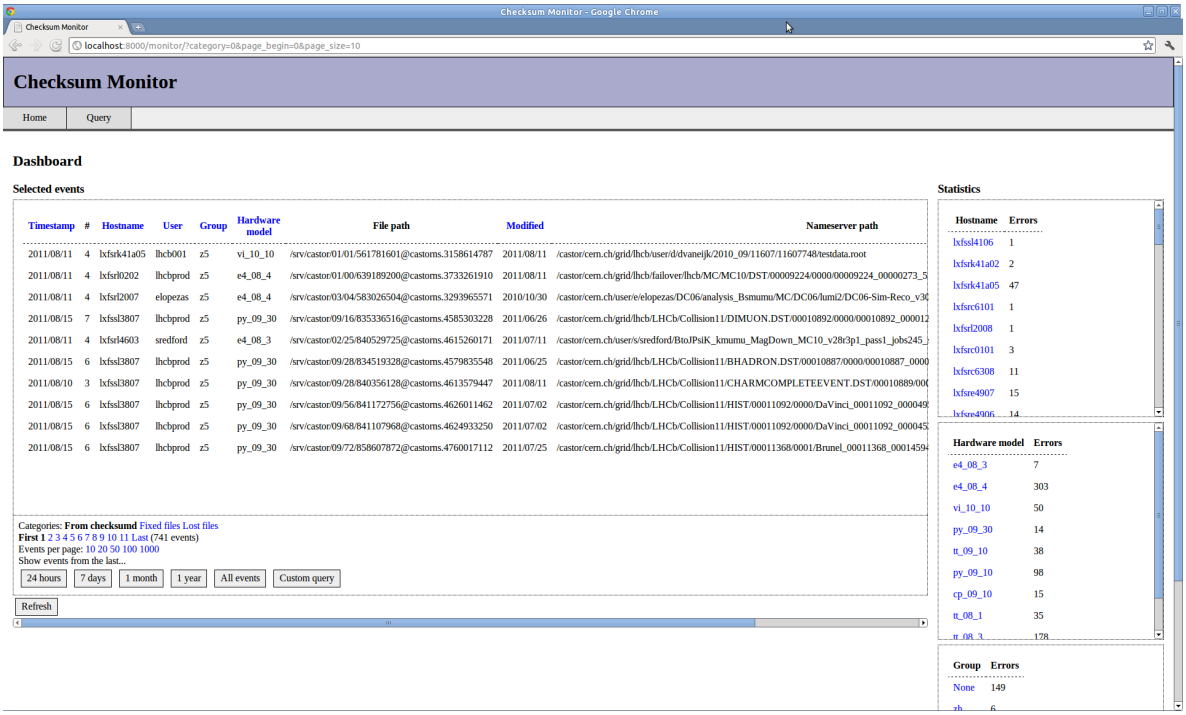

Figure 4: Checksum monitor main page

In addition to the main page, there is a search page which can be reached either by clicking "Query" in the menu or "Custom query" near the timespan buttons. The search form is shown in figure 5. All the fields that are searched on are AND-ed, forming the final query. So specifying e.g. hostname "lxfsj4107" and category "fixed" searches for all events from host lxfsj4107 that are in the fixed category.

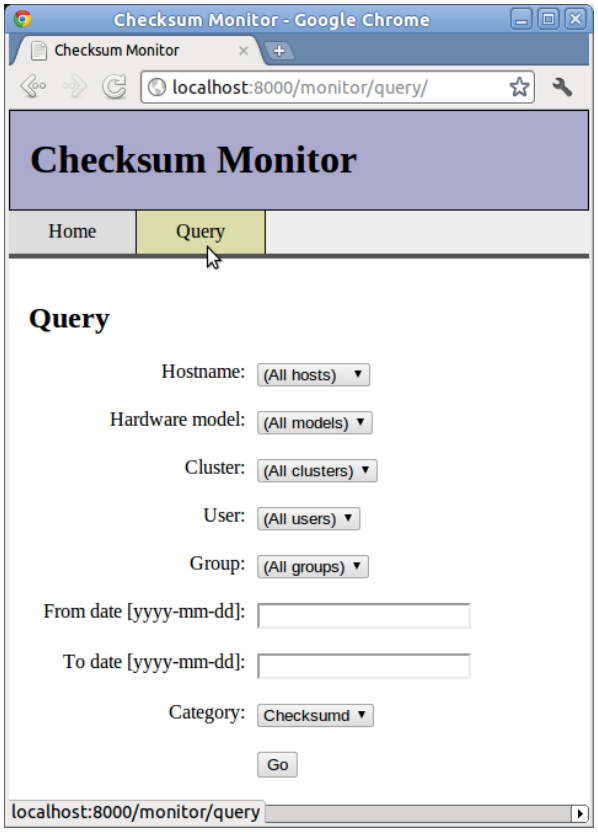

Figure 5: Checksum monitor search page

3 Implementation

This section goes into detail about the implementations of checksumd and checksum-monitor: classes, design patterns, etc. In addition, the integration with Lemon is briefly described.

3.1 checksumd

3.1.1 Disk and DiskQueue objects

As mentioned in section 2.2, disks are kept in a queue, and each disk has a list of files as well as indices pointing to the next file. The root directory of the scan, specified when running the daemon (default /srv/castor if nothing is specified) is assumed to contain folders that are mount points for different disks. A Disk object is then created for each of these mount points, which contain the mount point, an

index to the next file and a list of absolute paths of files on that disk. Whenever the next file on the disk is requested (by calling `Disk.get_next_file()`), the file at the current index is returned, and the index is incremented.

At certain intervals, the file lists are refreshed for each disk. When the file list is refreshed, a cleanup is performed first, which traverses the current file list and removes all files that no longer exist or that have been modified recently, after they were added to the list. Then, the Disk objects first get a full list of all the files on the disk (by recursively traversing the directories). Files that don't match certain criteria, e.g. files that were modified recently, are not included. This new list of files is then traversed. For each file, if it is in the file list already, it's ignored. If not, it's added. A hash map is used to keep track of which files are currently in the file list; this is due to the much faster lookup in a hash map than a list.

A DiskQueue object contains a list of disks, and also implements locking for thread safety. There is one main disk queue shared by all threads and each thread may call `DiskQueue.get_next_disk()` to get the next disk in the queue. The queue will then first be locked, the index will be incremented, the lock is released, and a Disk instance will be returned to the caller. In addition to providing queue functionality, a DiskQueue object also has wrapper functions for counting all files on the disks, refreshing all disks and so on.

3.1.2 Threading

checksumd is mainly an IO bound application; most of the time it simply waits for IO, and the typical CPU usage when processing just one file is about 20-30% on one core for a modern CPU. Having more than one thread on one disk will hurt performance due to more non-sequential reads, but processing multiple disks at once with one thread per disk will utilize more of the available processing power. As DiskQueue objects are thread safe, each thread may request a disk and be sure that this disk is not given to any other thread, so the effort of implementing multithreading is trivial in this case.

Each worker thread is represented by a ChecksumThread object which inherits the threading.Thread class. A worker thread lives as long as there are files left to check; once all files have been checked, the worker threads will die and the main thread will sleep for a specified number of seconds, before the threads are spawned again to do another check. Each worker thread follows the following logic:

1. Get a disk from the disk queue

2. Until disk becomes busy, or all files have been checked, perform integrity checks on the files.
3. Release the disk, and get a new one. Go to 2. until there are no more disks in the queue.
4. If all files have been checked, then exit. If not, sleep for a certain number of seconds before resetting the queue and checking each disk again to see if they are now free.

3.1.3 Logging

checksumd may output log entries to multiple logging destinations, like the syslog, standard output (if running in non-daemon mode) and a flat file with file errors. Because of this, a unified log manager was developed that handles output to each log. One class, LogManager, is responsible for taking in a log entry and storing it in each log. Loggers may be added to the LogManager by calling `add_logger()`. Figure 6 shows an overview of these classes.

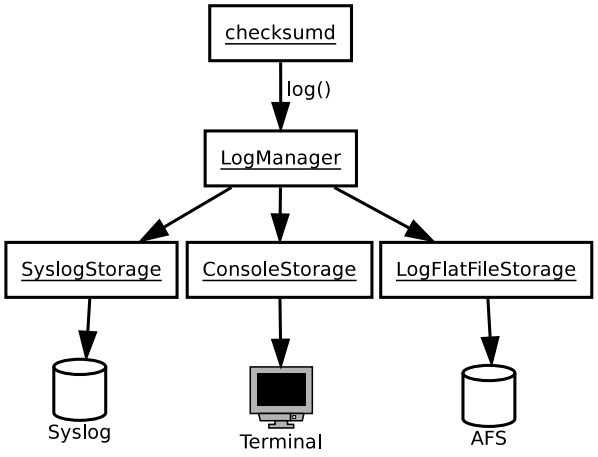

Figure 6: Logging in checksumd

3.2 checksum-monitor

The checksum monitor application was developed using Django, which uses a model-view-template architecture where the model represents the actual data, the view serves data to the template for viewing, and the template formats the actual output. The access to the database is made transparent by Django.

There are three views:

index is the default view where events and statistics are shown. Query variables are taken in via HTTP GET or HTTP POST that specifies what fields the events are filtered on. After filtering, queries are made to find the count of events grouped by host, hardware model or user group. Finally the event list is reduced by limiting the number of results to the page size.

refresh is not a real "view" but is used to call the refresh command for parsing the logs.

query displays the search form.

The models of this application are rather simple. There are two entity classes: Host and Event. A Host represents one host, with a hardware model, hostname and serial number. An Event represents one event, which can be a lost file, a checksum error, or a resolved checksum error. Each event has the following fields:

host is the foreign key to the Host model.

timestamp is the time of the event (from the log, rather than when it was put in the database)

filepath is the path on the file server for the file that this event applies to.

nspath is the name server path of this file.

cluster is the cluster that the host was in at the time of this event. The reason this is not in the Host model instead is due to the fact that hosts may be moved between clusters.

category describes the type of event (lost file, checksum error or resolved checksum error)

group is the user group for this event, if given.

All templates are stored in <django_project_path>/templates. There is a base template, base.html, which contains the base layout (title, menu, ...) as well as including the stylesheet, style.css. Static files, like images, stylesheets and so on, are served using the view django.views.static.serve as described by the Django documentation[2]. These files are kept in the directory <django_project_path>/static.

3.2.1 Implementation of the index view

The index view is the main view showing the entries, statistics and everything else of interest. First, the query string from the URL is parsed, which contains the current state of the web page (e.g. which

page have the user browsed to, the page size (in # entries), the different filters used, sorting information (which field to sort on, ascending/descending), ...). The filters that are not None are then applied to the Event database. The events are then grouped so that there is only one entry per (hostname, path) tuple; this is done because the same error may be found several times. The number of times the error was found, along with the latest timestamp, is retrieved from the database.

Finally the links on the page (e.g. the category links, page size links, sorting links, ...) is generated, and everything is then passed to the template.

3.2.2 Log parsing

When the refresh command is issued, the log directories are scanned and the events are put in the local relational database. In addition, some information is retrieved using "nsls" and similar queries.

First, all the existing hosts are retrieved from the database, so that there is no need to query the host table for every entry to check if we need to add a new host. Then, a dictionary is set up that maps the existing entries, by selecting all the objects from the event table. The tuple (timestamp, hostname, path) is assumed to be unique. This is very fast compared to doing a select operation per entry in the log files. This way, the log parser can continue where it left off if it is interrupted, and adding new log entries will take less time once the bulk has been added already.

For inserting into the database, threading is utilized because of rather large delays due to the queries to external servers. Each thread will be given a portion of the rows from the logs. For every entry in the thread's set of rows, the thread will do a lookup in the dictionary. If the entry is there (i.e. the entry's (timestamp, hostname, path) tuple exists in the dictionary), it will skip the entry. If not, it will do an nsls query to find the time it was modified, the user and group of the files and so on. Finally, the event will be added to the database.

3.3 Lemon integration

Monitoring of checksumd summaries are done using Lemon. More specifically, the metric ParseExtract in the sensor ParseLog was used. ParseExtract parses a log file, attempts to find a match for the regular expression given to the metric, and extracts specified values from each line.

In the case of checksumd, the following strategy is used: Every hour, the sensor will get the matching entries from the last hour, which should be either 0 or 1 entry as long as the reporting rate from the daemon is larger than one hour. It extracts the key numbers (number of files scanned, number of bad checksums found, and so on) and stores them in the same order that they are found. This is not a problem because the key values will always be output in the same order. This data is then sent to the Lemon server by the agent.

4 Configuration

4.1 checksumd

All of checksumd's configuration is done by command line arguments. checksumd can be run in two modes: as a daemon, or as a regular command line tool. The available command line arguments are listed in table 1.

A default configuration file, /etc/sysconfig/checksumd is shipped with the software package containing the daemon. This is a basic shell script that puts the command line arguments into an environment variable $CHECKSUMD_OPTIONS that is used by checksumd's runscript (typically /etc/init.d/checksumd). E.g., setting $CHECKSUMD_OPTIONS to --maxwait 90, causes the runscript to run "checksumd --maxwait 90 --daemon" (daemon mode is implied by the init.d script).

Argument	Description
`--daemon`	Run checksumd as a daemon.
`--loopcount N`	Run for N loops. Default: -1 (infinite) in daemon mode and 1 otherwise.
`--nodlf`	Disable output to syslog.
`--checkns`	Also check the content checksum against the one stored in the name server.
`--throttlesleep N`	Sleep for N seconds if all disks are found to be busy (or done), before doing another busy check. Default: 10 sec.
`--maxruntime N`	Terminate the program after N seconds. Default: infinite
`--maxwait N`	If disks are busy and have M open files, force a check after $N \cdot 2^{M-1}$ seconds. Default: -1 (infinity)
`--minrepeattime N`	If checking all files takes T $<$ N seconds, sleep for N-T seconds before checking them again. Default: 24 hours
`--setchecksumifnone`	Enable in order to set the checksum of files that do not have one.
`--modifyignore N`	Ignore files that have been modified less than N seconds ago. Default: 24 hours
`--errorfile DIR`	Save error logs under the directory DIR. Filenames are given as `checksum_errors_<hostname>.log`.
`--nthreads N`	Use N worker threads for integrity checking. This does not include the main thread. Default: 2
`--refreshrate N`	Refresh the file queues every N seconds. Default: 1 hour
`--verbose, -v`	Enable more output.
`--clear`	Clear the checksum of the file if it's wrong.
`--recoveryscript FILE`	If a checksum error was found, run this script. The character $ represents the filename as an argument, so e.g. "fixchecksum $" would run "fixchecksum" with the bad files as the argument. Default: None

Table 1: Command line arguments

4.2 checksum-monitor

The checksum-monitor main settings file is "settings.py" in the root directory of the checksum-monitor project. The important configuration variables are

DATABASES has the database information: type of database (e.g. SQLite or MySQL), database name and path, username/password and so on. As default, it is set up with MySQL on localhost with username root without a password.

MONITOR_ROOT is the root directory of the checksum-monitor project. This must be set.

log_path is the (log path, extension) tuple for the checksumd error logs.

fixed_checksums_path is the (path, extension) tuple of the logs containing resolved checksum errors.

lost_files_path is the (path, extension) tuple of the logs containing lost files.

stager_hosts is a list of the stager hosts used for checking if a file is lost.

After the database is set up and the settings are updated, the tables can be created automatically:

```
<MONITOR_ROOT>/manage.py syncdb
```

All the required tables will then be set up with the proper keys and datatypes. The web server can then be set up to listen on port 80 using

```
<MONITOR_ROOT>/manage.py runserver 0.0.0.0:80
```

However, the recommended way is using a third-party web server like for instance apache, as the built-in web server is not intended for production use.

4.2.1 Refreshing the database

When new log entries are added, the database must be refreshed (logs must be parsed again) for the entries to show up in the monitor. There are two ways:

1. Using the web interface's "Refresh" link.

2. Calling the refresh command directly through manage.py, e.g. through a cron job.

The second method may be the preferred one because it ensures that many people don't call the refresh command at once. The syntax is

```
<MONITOR_ROOT>/manage.py refresh <category>
```

where <category> can be "checksumd", "fixed" or "lostfiles". The log directory for the specified category will then be scanned and new entries will be added.

5 Conclusion

I improved upon a script for integrity checking files on the disk servers used by the CASTOR storage manager. I implemented a more robust way of checking if a disk is busy or not, more options to override busy checks after a specified amount of time, implemented threading and improved the configuration script used for deployment.

In addition, I implemented a Django web-based monitor for doing more detailed analysis of the errors encountered by checksumd. The web interface shows errors grouped by the user's group (where applicable), hostname and hardware model, as well as providing search over many different parameters like a timespan, hostname, cluster, and so on. For monitoring of the general progress and status of the checksumd checks, the ParseLog sensor in Lemon was adapted for use with the logs written by checksumd.

References

[1] CASTOR web site
http://castor.web.cern.ch/

[2] Django web site
https://www.djangoproject.com/

[3] Lemon web site
http://lemon.web.cern.ch/

[4] Andrew File System
http://services-old.web.cern.ch/services-old/afs/afsguide.html