The Kosmos distributed FS

KOSMOS FILES

Distributed filesystems effortlessly juggle enormous files in the gigabyte and terabyte ranges. The Kosmos filesystem plans to impress its competitors. **BY TIM SCHÜRMANN**

Modern computer programs handle increasingly large volumes of data. Whereas data-mining applications are content to sift through mountains of existing data, Internet search engines constantly horde new information. Users who access this data regularly encounter files of several gigabytes or more.

Legacy filesystems soon reach their limits with this kind of data and throughput. Consequently, organizations that manage huge volumes of data need an alternative solution for fast and safe access. Having redundant data storage is useful; after all, who wants to lose the valuable data gained by several days of number crunching because of a banal disk error?

Distributed filesystems fulfill these requirements. A distributed filesystem splits the data into manageable chunks and stores the chunks on a scalable cluster of computers. By virtualizing storage on the cluster, the filesystem then tricks applications into believing that they are talking to an enormous hard disk.

Into Space

The Kosmos filesystem (KFS) [1] is a promising new entry into this field. Kosmix Corporation developed KFS and released the source code under the Apache license. The first alpha version 0.1 appeared in September 2007. KFS's relative youth shows when setting up the filesystem: KFS requires 64-bit Linux. If possible, the Linux version and distribution should be identical on all the computers involved in data storage.

KFS is up against a number of renowned competitors, including Google filesystem (GFS), which Google uses as the underpinnings for its search engine, and Hadoop project's HDFS [2]. The KFS developers lifted much of the structure and functionality from Google, but they have removed a number of limitations. KFS – like GFS – is optimized for scenarios in which many large files are created once but read many times [3].

Job Descriptions

The Kosmos filesystem consists of three components:

- one or multiple chunk servers that store the data on their own hard disks,
- a metaserver that keeps an eye on the chunk servers, and
- an application that quickly gets rid of a single large file.

KFS thus works much like a database that resides between a computer pro-

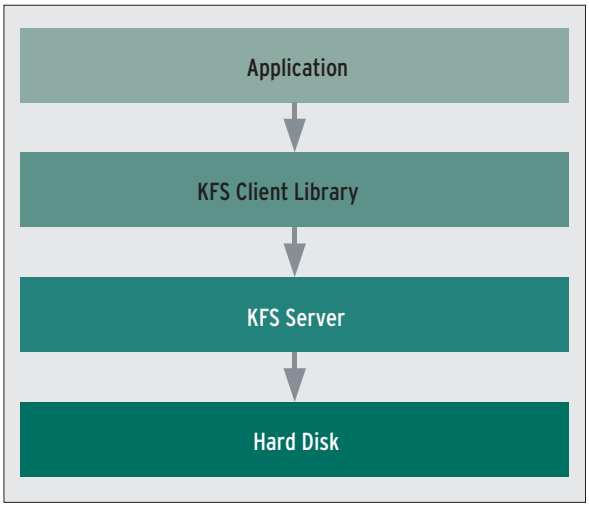

Figure 1: The Kosmos filesystem resides between the existing hardware and the application, just like a legacy database. A client library handles access to the virtual filesystem.

gram and the traditional filesystem (see Figure 1).

Chunkwise

KFS first splits a file into handy 64MB blocks. The filesystem distributes these chunks evenly over all attached servers, aptly referred to as block or chunk servers. The servers store the blocks on normal filesystems that belong to the host operating systems.

If the chunk servers start to run out of storage capacity, the administrator can simply add a new computer to the cluster. KFS automatically adapts the new storage node, which keeps the whole system scalable and helps it keep pace with increasing storage demands.

KFS mitigates hardware errors by storing the blocks from every single file redundantly on multiple chunk servers; typically, three instances of each file placed in storage exist.

This safety net allows administrators to deploy standard PCs as cheap, but reliable, data repositories. Google FS proves that this works day after day. If a disk or server fails, you just replace it with a new one. KFS detects the replacement and automatically integrates the newcomer into the cluster.

As another preventive measure against data loss, each block has both a version number and a checksum. KFS evaluates the checksum on each read operation. In case of irregularity, the distributed filesystem deletes the defective chunk and replaces it immediately with an intact copy (re-replication).

Version numbers help to identify obsolete chunks: If a poor Internet connection temporarily separates one server from the cluster, it can identify obsolete chunks quickly when the connection is reestablished and retrieve the more recent variant from the other servers in the cluster.

Metastases

Unfortunately, chunk servers do not bother remembering which parts of which file are stored on which member server. For this reason, a metadata server (or metaserver, for short) is deployed to monitor a number of chunk servers (the Google filesystem refers to these metaservers as masters). As the name suggests, the metaservers store the metadata, including details of which chunk server has which part of a file, the corresponding file sizes and file names, and information on which processes are currently accessing each file.

At regular intervals, the metaserver checks the capacity of the chunk servers assigned to it. If necessary, it will migrate chunks from a server with a heavy load to a less busy machine (rebalancing). This optimizes use of available capacities, thus improving the performance in general.

Clients

Applications use the client library to access this infrastructure (Figure 2). The library includes a complete filesystem API that allows clients to store (large) files on KFS and to manipulate and read existing files in the normal way.

In contrast to its competitor HDFS, KFS supports writing to multiple arbitrary positions in a file or appending data to existing files.

Unfortunately, the client library is the only door to the distributed filesystem, except for a couple of minimal tools (see the box titled "Toolbox"). Consequently, there is no escaping modifying your own programs, and the choice of programming languages is restricted to C++ or Python. Java programmers can use the JNI native interface. In a clever move, the KFS developers have added an API for the HDFS filesystem, a competitor to KFS; programs written for HDFS can be ported easily to KFS.

Quickstart

Kosmos FS is provided in the form of a handy source code archive that you can only build on a 64-bit system. Apart from this, Kosmos is fairly frugal in its requirements: besides CMake, you just need the log4cpp and Boost libraries. After fulfilling the requirements, just unpack the archive and open the *CmakeLists.txt* file.

By default, the compiler will build the KFS programs and libraries with debug information. If you prefer to do without debugging, change the value in quotes that follows *CMAKE_BUILD_TYPE* from *Debug* to *Release*. If you need FUSE support (see the "Toolbox" box for details), uncomment the

```
# set (Fuse_LIBRARY_DIR "")
```

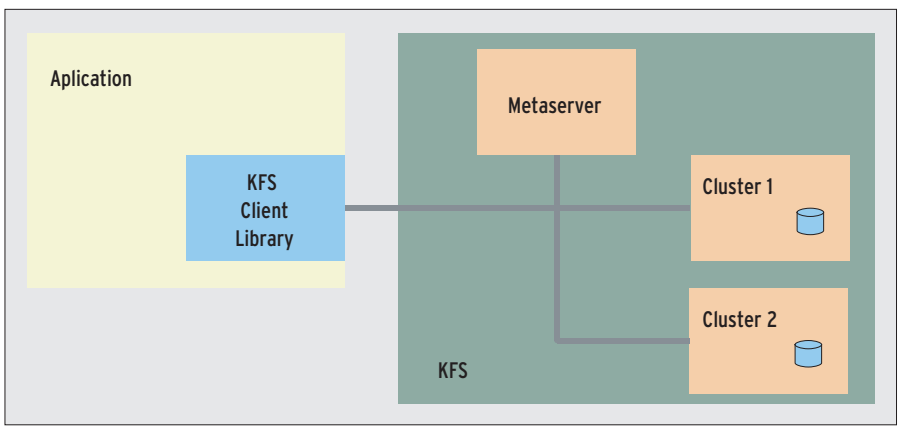

Figure 2: An application wanting to access a file first turns to the client library. The library queries the metaserver to discover which cluster servers the file resides on and then retrieves the file from the servers.

line and add the path to the FUSE library in quotes.

The administrator needs to enter a couple of commands to build and install KFS. To start, change to the KFS source code directory, which is *~/kfs-0.1.1* in this example. When you get there, enter the following commands:

```
mkdir build
cd build
cmake ~/kfs-0.1.1
gmake
gmake install
```

The last command suggests a system installation, but what actually happens is that the programs created in the previous step are moved to *~/kfs-0.1.1/build/bin* and the corresponding libraries to *~/kfs-0.1.1/build/lib* or *~/kfs-0.1.1/build/lib-static*.

If you need a Java interface, you can change to the KFS directory, *~/kfs-0.1.1*, and launch *ant jar*.

If everything has worked out okay, the *kfs.jar* file should be in the *build* subdirectory. This package contains everything you need to develop Java programs that use KFS.

A Python interface is slightly more complex. Start by changing directory to *~/kfs-0.1.1/src/cc/access*, then open the file *kfs_setup.py* in an editor and modify the include paths.

Next, give the *python kfs_setup.py ~/kfs-0.1.1/build/lib build* command. This creates *kfs.so* in the *build* directory, which you can then integrate with your Python system by typing *python kfs_setup.py ~/kfs-0.1.1/build/lib/ install*.

Launching KFS

The next step distributes the binary files to the meta- and chunk servers. A Python script in the *~/kfs-0.1.1/scripts* directory takes care of this, creating a customized program package for each server and then securing the installation with SSH.

To allow this to happen, all of your servers should run the same Linux environment, or at least the distributions should not be wildly different. Configuring SSH with keypairs removes the need to keep entering multiple passwords.

Topology

The only thing missing now is the configuration file that tells the script which computers on the network will be handling which task. Listing 1 shows a sample configuration file.

The file has a separate section for each server involved, headed by the server name in square brackets. The minimal requirement is a *[metaserver]* section.

Following is a section for each chunk server, which typically takes the form of *[chunkserver1]* through *[chunkserverN]*. The KFS cluster in this example comprises a metaserver and two cluster servers. Each section contains the settings for one server.

node: is followed by the name of the IP address for the server. *rundir:* is followed by the directory in which the binaries will be stored (in the example in Listing 1, this is the home directory for the *tim* user account on each server). The *baseport:* keyword specifies the TCP port that the server will use to communicate with the other nodes.

The computer names do not need to be different. In fact, Kosmos FS will let you run all the servers on a single machine – and this can be *localhost* – but in cases like this, you must assign unique TCP ports to your metaservers and cluster servers.

Each chunk server has a *space:* option that specifies how much disk space the server will use to save data. In the example here, the first chunk server provides 30GB, the second slightly less, 18,000MB. Sample configuration files are available in the *conf* directory below the source code archive.

Command Center

Now that the configuration file is complete, the next step is to change directory to *scripts* and enable the following:

```
python kfssetup.py -f ➥
configuration_file.cfg ➥
-b ../build/bin
```

Thanks to the configuration file, all the servers and SSH can be launched centrally from the current machine:

```
python kfslaunch.py -f ➥
configuration_file.cfg --start
```

The following call shuts the system down:

```
python kfslaunch.py -f ➥
configuration_file<.cfg --stop
```

Specifying the configuration file is important and lets users manage different KFS clusters from a single console.

Listing 1: Kosmos FS Sample Configuration

```
01 [metaserver]
02 node: 192.168.1.100
03 rundir: /home/tim/kfs/
   metaserver
04 baseport: 20000
05 [chunkserver1]
06 node: 192.168.1.101
07 rundir: /home/tim/kfs/chunk1
08 baseport: 30000
09 space: 30 G
10 [chunkserver2]
11 node: 192.168.1.102
12 rundir: /home/tim/kfs/chunk2
13 baseport: 30000
14 space: 18000 M
```

Toolbox

The client library gives applications convenient access to filesystem functionality, but to check the content of a directory would mean programming a tool for the task. The KFS package has a special Shell to remove the need for extra programming. The Shell provides counterparts to popular Unix tools, including *ls*, *cp*, and *mv*. Thanks to the Shell, users can navigate the KFS tree in the normal way. To launch the Shell, you need to execute a script in the *scripts* directory below the source code archive:

```
python kfsshell.py -f ➥
Konfigurationsdatei.cfg -b ➥
~/kfs-0.1.1/build/bin/KfsPing
```

KfsPing is an advanced *ping* that provides a useful service monitoring KFS servers. Typing *KfsPing -h* displays help. Other useful tools are located in the *build/bin/tools* directory.

If you do not like the idea of special commands, your alternative on Linux is FUSE support (Filesystem in Userspace), a kernel module that migrates a filesystem driver to user mode. FUSE allows users to mount KFS like a normal hard disk partition and then deploy the full range of Linux tools.

Now that the servers are running, users can start moving data onto the enormous new filesystem using either the special KFS Shell (see the box titled "Toolbox" for more details) or via the API. A simple example of a C++ program that stores its data in KFS is given in Listing 2.

Unfortunately, the header files are hidden away in the depths of the source code archive in *src/cc*. This also applies to the libraries, which are located in *build/lib*:

```
g
++ test.cpp -I ➥
~/kfs-0.1.1/src/cc -L ➥
~/kfs-0.1.1/build/lib/ ➥
-lkfsClient -lkfsIO ➥
-lkfsCommon
```

Before calling the results, *LD_LIBRARY_PATH* has to be set:

```
export LD_LIBRARY_PATH=➥
~/kfs-0.1.1/build
```

To save the linker the trouble of searching for the dynamic libraries, you can link your own programs with the static variant, which is located in *~kfs-0.1.1/build/lib-static*.

To handle huge volumes of data, a KFS application simply opens a new file via the client library.

Buffers

First, the library buffers the incoming write operations and waits for the cache memory reserved for this purpose to fill or for the application to issue a flush command before pushing the data to the chunk servers.

Immediately after the data arrive, they become available for further operations.

Besides the outgoing data, the client library also buffers any metadata that are requested for 30 seconds. This helps to avoid unnecessary, repeated server contact.

If a client is running on a chunk server, it retrieves the data locally rather than using up network bandwidth. If a chunk server suddenly fails during a read operation, the client library automatically switches to another chunk server. All of this is completely transparent for the application.

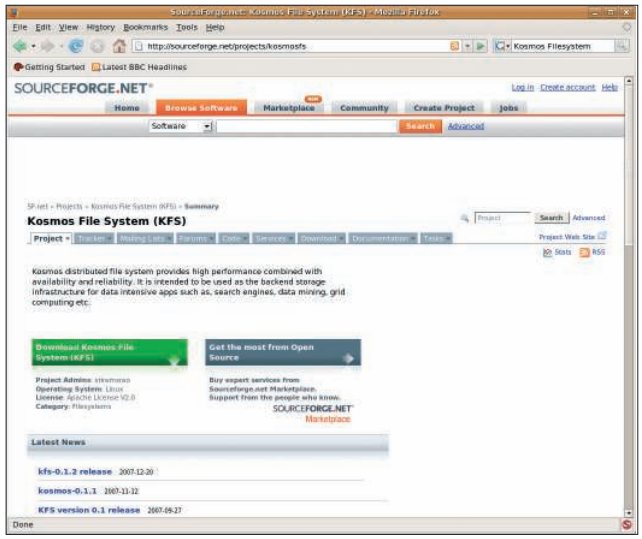

Figure 3: Kosmos FS is available at SourceForge.

Conclusions

Kosmos FS is an interesting alternative to HDFS and Google FS, but it is still at an early stage of development. Currently, one weak point is the metaservers. They need to be able to deliver metadata quickly. After all, to be able to process the file, a client needs to know which node the file it requires is stored on. If the metaserver fails completely, the files on the chunk servers it manages are also unreachable.

Because the metaserver additionally handles load distribution, it is responsible for the performance of the KFS network it manages. Unfortunately, there is currently no replication plan for metadata, in contrast to the scheme used by the chunk servers. Administrators need to take care of this manually and back up the data regularly.

Another issue is the lack of access controls. Currently, users can store any data on the distributed filesystem and read any data stored there. For this reason, KFS should only be deployed in trusted environments until a more mature version is released. ■

Listing 2: Creating a File

```
01 ...
02 #include "libkfsClient/
   KfsClient.h"
03
04 using namespace KFS; // KFS
   Namespace:
05
06 int main(int argc, char
   **argv)
07 {
08     string serverHost =
   "localhost";
09     int port = 20000;
10
11     KfsClient *gKfsClient;
12
13     // Get access to
   filesystem:
14     gKfsClient = KfsClient::
   Instance();
15     gKfsClient->Init
   (serverHost, port);
16
17     // Create subdirectory:
18
   gKfsClient->Mkdirs("testdir");
19
20     // Open file, "fd" is the
   handle:
21     int fd = gKfsClient->
   Create("testdir/foo.1");
22
23     // Write junk:
24     int numBytes=2048;
25     char *buffer = new
   char[numBytes];
26     gKfsClient->Write(fd,
   buffer, numBytes);
27
28     // Flush changes:
29     gKfsClient->Sync(fd);
30
31     // Close file:
32     gKfsClient->Close(fd);
33 }
```

INFO

[1] Kosmos filesystem: *http://kosmosfs.sourceforge.net*

[2] HDFS and the Hadoop project: *http://lucene.apache.org/hadoop*

[3] Paper on Google filesystem (GSF), on which KFS is based: *http://research.google.com/pubs/papers.html*