Xerxes : an advanced search system for your desktop

Claudia Tanase
SPBA
UPB
Bucharest, Romania

Razvan Alecsandrescu
SPBA
UPB
Bucharest, Romania

Abstract

This paper presents an advanced mechanism used to organize and seach files. The system uses a combination of semantic labeling, file parsing and Virtual Folders.

1. Introduction

One of the most important things for a computer user is to be able to organize a large number of files and search for a specific one in a natural manner.

Most of the current search applications can be used to find files using information about the file name, file type or time/date information. But this only allows for simple searches.

There is a lot to learn from web search engines like Google search engine. Not only does it allow for simple searches but also for more complex searches: including content analysis.

This paper details the arhitecture of an advanced system for file organization. Section 2 describes existing applications, Section 3 details the implementation of our system and Section 4 presents the results of tests performed, conclusions and future work.

2. Previous Work

Most of the searches that we can perform using standard OS tools are limited to directory namespaces. For example in Windows we have the Search program that can perform searches filtering the results by name,type,the time it was last modified, Unix shell wildcard patterns are useful for matching against small numbers of files, and GNU locate for finding files in the directory hierarchy. The biggest issue with these applications is that to perform queries on the **data** inside a file can be a laborious procedure because of the linear complexity invoved. Fast and reliable searches require an indexing structure.

To create more advanced search systems there are 2 aproaches that can be taken. The first one which is the most commonly used is to develop applications that use the file system to obtain informations about files and then process, index and allow the users to perform queries.

The second one is to implement *file indexing inside the file system*.

There are various pros and cons to each method of implementing a search system. For example file indexing inside the file system assures that the index is synchronized with the reality of the file system. The system assures us that once a file is modified the changes will show up in the index. Also there can be a small performance boost caused by the fact that processing a file that already exists in the cache is fast.

A application separated from the file system must keep track of all changes (when files are moved, deleted, renamed, modified) which is a little more complex.

3. Implementation

There are 2 major components to our applicaton. The Kernel-space component consists in a LKM, and the User-space component consists in a C program and a Java application. In the following 2 subsections we will present the way the 2 components work and how they interconnect.

3.1. Kernel-space

The kernel modules task is to monitor system calls and select the ones that are important for our application. We are interested in system calls that let us know that a file has been modified. We want to know when and what file is edited, deleted, renamed, copied etc.

The kernel module obtains all this information and using NetSockets sends it to the user space program.

3.1.1. File-system event-monitoring mechanism

We have designed a file-system event-monitoring mechanism in the Linux kernel that checks for file-system changes like file creation, deletion, modification, and renaming. Once a watch for a directory is registered , the kernel adds all event notifications for that file in a queue, and sends them out to the server as soon as possible.

The basic idea is to intercept interesting system calls by replacing the original routine with another one that adds an item containing system call number and file path into a queue. We have built a kernel module that does the interception, queues updates and starts a kernel thread that is scheduled from time to time and sends all items found in our queue to our user-space program via netlink socket. The user-space program is responsible for calling Java code in the server(responsible with the actual indexing and searching) using JNI.

To eliminate unnecessary event processing, such as indexing temporary files that only exist for a short time or system files we will only index regular files/folders marked to be monitored.

Since the file indexing operations should not become a burden on the system, we will index files only when the system is first installed. Also, the event-monitoring mechanism works so that intercepted file-system changes will be queued and sent from time to time to the server.

3.1.2. Intercepting system calls

The easy and fast way to intercept file-system changes is to intercept file-system related system calls.

The operating system maintains a system call table that has pointers to the functions that implement the system calls inside the kernel. From the program's point of view, this list of system calls provides a well-defined interface to the operating system services. You can obtain a list of the different system calls by looking at the file /usr/include/sys/syscall.h.

Assume that we want to intercept the write system call and send a message to our user-space program when any process invokes it. In order to do this, we have to write our own fake write system call, then make the kernel call our fake write function instead of the original write call. At the end of our fake write call, we can invoke the original write call. In order to do this, we must manipulate the system call table array (sys_call_table). Armed with the sys_call_table array, we can manipulate it to make the sys_write entry point to our new fake write call. We must store a pointer to the original sys_write call and call it when we are done sending our message to user-space.

3.1.3. Using netlink socket

We have used netlink socket to send asynchronous updates from our intercepting system calls kernel module to an user-space C++ program that acts as an intermediate layer between the file-system event-monitoring mechanism and the server written in Java that does the indexing and needs these updates.

Netlink socket is a special IPC used for transferring information between kernel and user-space processes. It provides a full-duplex communication link between the two by way of standard socket APIs for user-space processes and a special kernel API for kernel modules. Each netlink socket feature defines its own protocol type in the kernel header file include/linux/netlink.h.

Why use netlink instead of system calls, ioctls or proc filesystems for communication between user and kernel worlds? It is a nontrivial task to add system calls, ioctls or proc files for new features; we risk polluting the kernel and damaging the stability of the system. Netlink socket is simple, though: only a constant, the protocol type, needs to be added to netlink.h. Then, the kernel module and application can talk using socket-style APIs immediately.

Netlink is asynchronous because, as with any other socket API, it provides a socket queue to smooth the burst of messages. The system call for sending a netlink message queues the message to the receiver's netlink queue and then invokes the receiver's reception handler. The receiver, within the reception handler's context, can decide whether to process the message immediately or leave the message in the queue and process it later in a different context. Unlike netlink, system calls require synchronous processing. Therefore, if we use a system call to pass a message from user space to the kernel, the kernel scheduling granularity may be affected if the time to process that message is long.

The code implementing a system call in the kernel is linked statically to the kernel in compilation time; thus, it is not appropriate to include system call code in a loadable module, which is the case for most device drivers. With netlink socket, no compilation time dependency exists between the netlink core of Linux kernel and the netlink application living in loadable kernel modules.

Netlink socket supports multicast, which is another benefit over system calls, ioctls and proc. One process can multicast a message to a netlink group address, and any number of other processes can listen to that group address. This provides a near-perfect mechanism for event distribution from kernel to user space.

System call and ioctl are simplex IPCs in the sense that a session for these IPCs can be initiated only by user-space applications. But, what if a kernel module has an urgent message for a user-space application? There is no way of doing that directly using these IPCs. Normally, applications periodically need to poll the kernel to get the state changes, although intensive polling is expensive. Netlink solves this problem gracefully by allowing the kernel to initiate sessions too. We call it the duplex characteristic of the netlink socket.

Finally, netlink socket provides a BSD socket-style API that is well understood by the software development community. Therefore, training costs are less as compared to using the rather cryptic system call APIs and ioctls.

3.1.4. Java Native Interface

We have used JNI to send messages from the kernel module that intercepts the file-system related system calls to the server that does the indexing.

JNI is a programming framework that allows Java code running in the Java virtual machine(JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly.

3.2. User-space

3.2.1. User interaction

To interact with the user, we chose a shell interface rather than GUI interface since we feel that the shell is more expressive and more powerful, while with a GUI interface the operations that can be performed are restricted to those that are explicitly implemented.

You can use the user-space program intercept to control the server configuration, loading the kernel module and for managing our semantic file system by creating tags, adding them to files, monitoring interesting folders and searching.

Also, the usual commands are modified to reflect the new flattened file system put in place by our semantic file system and virtual folders. In order for the user to interact efficiently with the system, we had to rethink the classical shell paradigm. Since folders are virtual, there is no longer a real notion of being inside of a directory. With Virtual Folders there no longer exists a clear, predefined hierarchy .

3.2.2. Document processing and search

The Java application uses Lucene (a high performance text search engine library) to mantain the index and search through it.

At the core of Lucene's logical arhitecture is the idea of a document containing fields of text. This flexibility allows Lucene's API to be independent of file format. Text from any kind of file can be indexed as long as textual information can be extracted from that file.

To allow our application to index different types of files we needed to be able to parse them. In order to do this we have developed modules that parse mp3,txt,pdf,open office documents and other files can be added later. The modules are loaded dynamically so anybody interested in writing a parser for a new file is able to do so easily.

The modules must implement an interface and announce the kind of files that they can handle and when the application encounters a file that the module can understand it will send the file to the module for parsing.

Because the relevant information differs from one type of file to another (for example in a mp3 file we will be interested in the meta-information stored in ID3V1/ID3V2 tags while in a txt file we will be interested in the textual content) when a file is being processed different fields are being created in the index.

To these fields we add tags. Tags can be inserted into the index by the user to allow for easy acces to desired files. For example an user may add a tag named Paris to all the documents,pictures,mp3s on his computer that are linked to Paris.

The application is also connected to a database(MySQL) where it stores what files it is monitoring and if the files need to be reparsed. The idea is that when a file is modified the kernel module will send the information about the change to the User-space C program which in turn will call a Java method that marks the record of that file in the DB as tainted. At a certain interval the Java app will make a query in the DB which returns a number of tainted files and it will begin to process them.

When the user wants to perform a search a query is sent to the Java app which in turn uses the Lucene framework to parse that query and perform the search. Because of the way the information is stored in the Lucene index searches can vary in complexity from a simple "Windows-like" search that filters the files by name, type, date to more complex searches that take into account tags, meta-information, content of files etc.

The queries can be very complex also because the Lucene framework allows for logical operators and other operators to be inserted in the query. For example we can search for "tag:Paris " where the " " character signifies that all records which are "like Paris" should be returned.

3.3. Virtual Folders

Once a query has been performed and the user has a list of files displayed it is almost certain that the user will be interested in performing some action on the files(copying them, archiving them, deleting them, etc). But what do we do when there are 50 files spread allover the file system.

In order to deal with this we will be using Virtual Folders. VFs act as abstract containers of search queries. Internally a VF can store a logical set of files that are the result of a query. To the user however a VF is represented by a single folder-like data structure. A Virtual Folder could for example be called "recent pictures" and store all the ".jpg" files with a creation date withing the past n days.

A file can have multiple labels so it can appear in more than one search query which means that a certain file could be part of more than one VF which should be considered an advantage over hierarchichal file-systems where users often tend to copy or symbolically link files which fit two or more nodes of their existing directory hierarchy.

4. Conclusions and Future Work

One of the ways the application can be expanded is to allow it to perform queries on more than the local machine. An interesting possibility is to perform queries on the whole network or on a number of linked machines. This can be achieved fairly easily by taking advantage of the Lucene framework. The only problem is that a mechanism for downloading the files selected by the user to the local machine must be implemented.

Another improvment would target the Virtual Folders. As we mentioned when we explained the VF concept the idea behind it is to group a number of files in a logical manner. What would be more useful to the user would be that once a VF is created based on a a query that VF should be updated from time to time based on the stored query to include a more recent answer to the query. Coming back to the example used for VFs once a folder ”Recent photos” has been created it should be updated to include jpgs added after the original query has been performed.

Acknowledgments

bla bla bla

References

[1] Mukund Gunti, Mark Pariente, Ting-Fang Yen, Stefan Zickler, *File Organization and Search using Metadata, Labels, and Virtual Folders*, December 15, 2005

[2] Srinath Sridhar, Jeffrey Stylos and Noam Zeilberger, *A Searchable-by-Content File System,* May 11, 2005

[3] Erick Hatcher,*Lucene in Action*, Manning Publications 2004.