Katie

Geoffrey D. Bennett
NetCraft Australia
g@netcraft.com.au

January 6, 2003

1 Introduction

Katie is an open-source (GPL-licensed) revision control system which has been modelled on, and intended to be a look-alike of, the proprietary ClearCase Software Configuration Management System. Simplistically, ClearCase can be thought of as a cross between CVS and NFS. That is, it provides revision control (as you would expect), but the repository is presented as a filesystem with a number of interesting features, which will be described in this paper. The technologies used in Katie, and plans for the future will also be addressed.

2 Concepts

The terminology and concepts used by Katie are rather different to CVS, so for people familiar with CVS, here is an introduction to the major differences.

2.1 Versioned Object Base

A *VOB* is a *versioned object base*, analogous to a CVS repository. Except for the actual contents of the file versions (which are stored separately), the contents of each VOB are stored in a PostgreSQL database. In the same way that you would not usually directly interact with a CVS repository, but rather use commands such as `cvs checkout`, and `cvs commit`, you do not directly interact with a VOB. A daemon (`katieserver`) handles requests for access to the VOB from clients.

2.2 Views

A *view* is the primary means of accessing a VOB, and is the equivalent of the CVS workspace that gets created when you run the `cvs checkout` command. Rather than creating a local copy of the files in the repository, a Katie view is mounted as a filesystem

in a directory using the `katie-mount` command. Once a view is mounted, the elements in the VOB are visible through the newly-mounted filesystem.

In general, each Katie user will have their own view in the same way that each CVS user would usually have their own workspace.

2.3 Elements

VOBs primarily contain versioned files and directories, which are called *elements*. Unlike CVS, directories in Katie are versioned in the same way that files are versioned, which means that file renames, moves, and deletions are fully supported. For example, a file deletion would result in a new version of the directory which would contain one less file than the previous version.

All file elements within a VOB appear as read-only files in a view, and in order to modify an element, the `katie checkout` command must first be used. There is no CVS equivalent to this command, as CVS doesn't require you to inform it before you start modifying a file.

2.4 View-private Files and Directories

To support each Katie user having their own private workspace, any new files or directories that get created within a view using standard Unix utilities are not visible to any other view. These are called *view-private* files or directories.

For example, if an element `test.c` were visible in a view, then running the command `cc -c test.c` would presumably result in `test.o` being created. Until or unless you actually check `test.o` into the VOB with the `katie mkelem` and `katie checkin` commands (equivalent to `cvs add` and `cvs commit`), it will not be visible in any other view.

2.5 Branches

The naming of element versions and *branches* is done in a tree-like manner. Figure 1 shows a VOB element tree with three directory elements and two file elements. Each of these elements has its own version tree, and the version tree for one of those elements (`doc/README`) is shown. The first version of an element is always named `/main/0`, and for file elements is always empty (directory elements would only contain `.` and `..`). Subsequent versions of elements are named `/main/1`, `/main/2`, and so on.

Additional branches on an element can be created using the `katie mkbranch` command, but the branch name (known as a *branch type*) must be predefined with the `katie mkbrtype` command.

Figure 1 shows a branch called `/main/v1.0`. Similarly to the `main` branch, versions are numbered starting at 0 (ie. `/main/v1.0/0`, `/main/v1.0/1`, and so on).

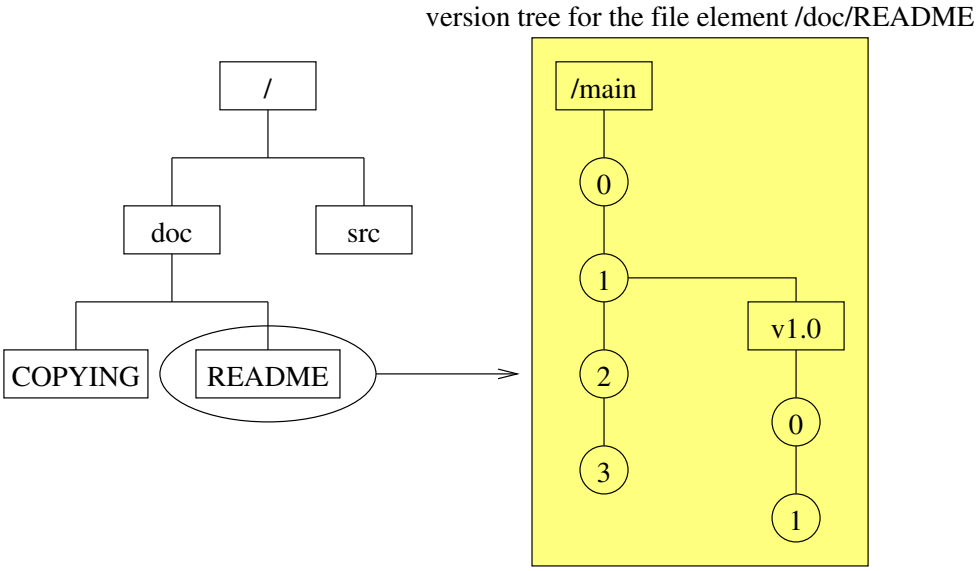

Figure 1: VOB and Version Trees

2.6 Labels

Katie *labels* are the same in concept as CVS tags, except that label names (known as *label types*) must be predefined using the `katie mklbtype` command. The predefined label type `LATEST` always refers to the latest version of any particular element. It is usual practice to name labels in all-uppercase, and branches in all-lowercase.

2.7 Configuration Specifications

To determine what particular versions of elements from the VOB get shown in a view, a short text file called a *configuration specification* (*config spec*) is associated with each view. The current config spec can be seen with the `katie catcs` command, and edited with the `katie edcs` command.

The simplest config spec is the one line `element . /main/LATEST` which would show the latest version (on the main branch) of each element, but config specs can be more complicated and include other branch and label names to select versions, along with regular expressions to match filenames. The following example config spec would show in a view the first match of:

- the checked-out version of any checked-out element
- the latest version on the `/main/testing` branch of any element within `/doc` that has that branch
- the version of any `*.[ch]` element within `/src` with the `V1.0` label
- the latest version on the main branch of elements that don't match any of the above

```
element . CHECKEDOUT
element ^/doc($|/) /main/testing/LATEST
element ^/src/.*\.[ch]$ /main/V1.0
element . /main/LATEST
```

The equivalent functionality within CVS is the concept of sticky options.

3 Implications of Dynamic Views

Since the view is itself a filesystem, and not a static copy of the files from the repository, there are a few mostly-obvious implications:

- No additional disk space is consumed by additional views beyond some housekeeping data and the view-private files that are created within that view.
- If your config spec selected the latest version of a file, and someone else checked in a new version of that file, the changes would be immediately visible in your view (ie. no `cvs update` equivalent is required).
- `katie-mount` (the equivalent of `cvs checkout`) runs in little time (not proportional to the size of the repository) as it is a mount of a filesystem rather than a copy.
- Changing your config spec (the equivalent of `cvs update -r mytag`) also takes little time as it again doesn't require any files to be copied around.

A less-obvious implication is that by extending the semantics of the filesystem, it is possible to allow direct access to any version of any element within the VOB using standard Unix tools. This is done with an *extended namespace*, where every element within a VOB has a hidden directory through which its entire version tree is accessible.

The extended namespace is accessed by appending `@@` to the end of any element name, so for the file element `doc/README`, the path would be `doc/README@@`. Within that hidden directory, the version naming described earlier can be used. Therefore, to access the `/main/2` version of `doc/README`, the path `doc/README@@/main/2` could be used.

Since standard Unix tools can be used to access old versions of files, the equivalent of `cvs diff -r 1.2 -r 1.3 doc/README` would be `diff doc/README@@/main/[23]`.

Labels can also be used within the extended namespace, so you could see the differences between two versions with labels `V1.0` and `V1.1` respectively by using a command such as `diff doc/README@@/V1.[01]`.

Slightly more out of the box, a command such as `grep regex doc/README@@/main/1?` could be used to search versions `/main/10` through `/main/19` of `doc/README` for instances of a particular string.

4 Technologies

4.1 NFS

To develop a native Linux filesystem with the properties previously described would require significant work, and be beyond the author's patience. Using NFS as a basis for Katie's filesystem seemed to be an obvious choice for a few reasons:

- No kernel code would need to be written.
- As the server would be entirely userland, it would be easily portable.
- As NFS is the Networked File System, Katie views would be accessible across a LAN without any additional work required.
- The client had already been written.

The initial proof-of-concept for Katie was developed by taking the Linux userland NFS server and replacing the filesystem-accessing code with database-accessing code. This was able to demonstrate that dynamic views with their hidden directories could be made to work, however the performance was hugely lacking due to no caching of database queries being implemented.

4.2 Perl, SunRPC, and Inline.pm

Katie was then rewritten as a C program, using an embedded Perl interpreter to do most of the "real" work. The common wisdom that Perl is faster to develop with than C was immediately obvious, but surprisingly, this version outperformed the pure-C version so much so that the filesystem now ran at an acceptable speed. The speedup was entirely due to the ease of putting appropriate caches around the relatively slower database calls.

The major sticking point in converting Katie to a Perl program was the lack of any SunRPC/XDR library for Perl at the time (SunRPC is the *Sun Remote Procedure Call* protocol which NFS uses as its basis; XDR is the *External Data Representation* Standard which SunRPC uses for encoding data). This meant either writing Perl code to talk XDR, or learning how to call the already-written C SunRPC/XDR handling routines from Perl.

Learning how to call C from Perl seemed to be the easy way out. XS and SWIG were investigated, but the Perl `Inline` module seemed to be a good example of the Perl motto of "making easy things easy". The below listing shows a complete Perl program using `Inline` to link in a C function at runtime.

```
#!/usr/bin/perl
use Inline C;
print "6 * 9 = ", multiply(6, 9), "\n";
__END__
__C__
int multiply(int x, int y) { return x * y; }
```

By passing additional options to `use Inline`, the external library required for SunRPC was also easily linked in. The next step was learning how to translate to and from Perl arrays, hashes, and references from within C. This required reading `perlguts(1)` more than once, along with a bit of trial and error.

The Katie non-filesystem-related RPCs such as checkin and checkout were initially defined as additional SunRPC functions (on top of the standard NFS ones such as read, write, lookup, etc.) as this resulted in the easiest initial implementation. It also meant that every additional RPC (creating a VOB, creating a view, creating an element, checkin, checkout, and so on) involved writing more XDR definitions and tedious C code to convert in and out of Perl data structures.

The next implementation replaced SunRPC for non-NFS purposes with the a module based on PlRPC, a Perl-specific RPC library, unrelated except in name to SunRPC. This allowed for arbitrary Perl data structures to be passed from client to server and back again without needing to predeclare the RPC arguments or return values as SunRPC required.

PlRPC works by using the `Storable` module to convert an arbitrary Perl data structure to a string representation which gets transferred over the network. The server side then converts the string representation back to the original data structure and calls the appropriate function. The return value is similarly encoded into a string by the server and decoded by the client.

5 Status and Future Work

Katie is still in a pre-alpha state. All the features described so far work very nicely, and it is capable enough that it is now self-hosting, but there is still much work to do. There are general issues to deal with such as improving its robustness and rewriting code that wasn't intended to last long. The more specific issues include many small "must-have" features that are necessary for it to be a useful tool (to name two: access control and importing CVS repositories). There are also a number of more-interesting larger features that are planned, including replication, attributes, hyperlinks, and `katiemake`.

5.1 Replication

Due to the use of NFS, the Katie server must be "close" (network-wise) to the client. A server-server communication protocol needs to be designed and implemented so that Katie is useful over lower-bandwidth, higher latency networks. The idea is that a Katie user would run their own server locally, and the local server would maintain at least a partial copy of the VOB. This would also allow for disconnected operation.

5.2 Attributes

Attributes allow arbitrary data to be assigned to particular elements, branches, or versions. This could be used, for example, to assign a `tested` attribute to note whether a

particular version of an element has been tested. Config specs would also support attributes so that you could see in your view the latest version of each file which has the `tested` attribute set to `True`.

5.3 Hyperlinks

Hyperlinks allow relationships between VOB *objects* (such as element versions) to be recorded. This would be required to track the history of merges between branches so that when doing a merge, the most appropriate common ancestor version can be identified.

5.4 katiemake

Another result of using dynamic views is that a `make` program specific to Katie would be able to determine what files were read during the build of a target.

By using this information, implicit dependencies (such as `#include` directives within C source code) would not need to be listed in a `Makefile`. For example, if `katiemake` notices that `gcc -c foo.c` results in `foo.h` being read, it would know that if `foo.h` changes, `foo.o` would need to be rebuilt without specifically being told about the relationship between `foo.o` and `foo.h`.

In a similar way, by tracking what versions of elements went into the build of a particular file, a developer who repeats the build of an object that another developer has built could be given the already-built version of the file, which would save time.

6 Conclusion

Katie is not yet suitable for real-world use, but it is steadily progressing, and already implements some useful features. For further information and to download Katie, see `http://www.netcraft.com.au/geoffrey/katie/`.