Implementation of Multiple Key Databases for Shared Configuration

Patrick Sabin
patricksabin@gmx.at

Markus Raab
elektra@markus-raab.org

March 7, 2008

Contents

Registry, libraries and modules have been proposed as well-suited storage for software configuration. Yet so far, there's a dilemma in choosing between a system-wide configuration setting database or a configuration database optimized for the user's needs, not accessible from other software.

We demonstrate that it is possible to share configuration with different backends (i.e. different configuration formats) using the technique of dynamic linking, which fits transparently into the hierarchical key namespace. This takes place by mounting the appropriate backend to a system-wide unique key name.

We also describe how to use multiple databases at once to achieve shared configuration without losing the substantial benefits to the administrator and programmer configuration.

1 Introduction

This paper describes our work in the context of our project thesis. We added a new feature to the Elektra Project called mounting that we would like to describe here. Elektra introduced its own technical terminology, which is not well known. So we decided to explain Elektra first, before we went on to the mounting feature. Additionally, we added a glossary listing the Elektra terminology and marked all words in the text that have an additional explanation in the glossary italic.

1.1 Configuration

There are different possibilities for configuring software, from xml to databases. However, the old way to use configuration files with a basic syntax is still a very common approach. System and user configuration exists quite often on multiuser operating systems.

System configuration is typically installed during setup, shipped with a software package, and can only be changed by a system administrator. On Unix systems, system configuration can be usually found under `/etc`.

Additionally, there is a **user configuration**. Every user in the system can use his own configuration, which might be different to the other users and the system configuration. Entries in the user configuration typically supersede those of the system configuration. On Unix systems, the user configuration is most often located in the home directory of the user, quite often in hidden files. A program usually tries to read the user configuration first and then the system configuration, if it doesn't find it in the first place. This procedure is called Cascading.

Cascading must be implemented for every program by itself and so the exact procedure of reading configuration may differ in every software package. Moreover, the location of the configuration differs on every system, depending on the operating system or the distribution. This is a problem for an application developer since he has to know the exact location of the configuration files in order to access them. So he is obliged to use a lot of system dependent codes (e.g. `#ifdef` in C) to write a portable application.

The parsing of configuration files - another common task in configuration - is often done slightly differently even though it provides the same functionality.

1.2 Elektra without mounting

In this section, we want to describe what is possible with Elektra, without the mounting feature added in this work. Elektra started as a library for writing and reading configuration files.

All the configuration is mapped into an abstract, common **namespace**, that is the same in every system. Elektra abstracts every configuration in key/value-pairs, where every *key* is stored in its directory. Almost every software configuration can be represented that way. The Elektra namespace distinguishes between two kinds of keys. The name of system keys for the system configuration looks like:

```
system/directory/program/key
```

Of course, there can be as many directories as needed. The characteristic for system keys is that they start with "system". Additionally there are user keys, where the name looks like:

```
user/directory/program/key
user:username/directory/program/key
```

The `username` is optional. If you don't specify it, the current user will be assumed, e.g. the environment variable `$USER` will be used.

The part of storing your configuration is done by a so-called *backend*. Elektra supports multiple backends and you can write your own, if you like. Every backend can have different advantages. One can be very efficient in respect to execution time or it may reduce the amount of disk usage. You can choose the backend you like most, but the problem is that without mounting you shouldn't use different backends for different programs, otherwise the common namespace will be destroyed.

Without mounting, a single backend was used to supply all keys for an application. This approach works perfectly for systems where all programs use the same backend, but completely fails in other cases.

Without mounting, you need to use the Elektra library to get your keys in the abstract namespace. To use Elektra with an existing program, it has to be patched.

Thus a single backend needs to be a jack-of-all-trades, not optimized to any security, performance or syntax requirements. Thus old syntax can't be supported. Elektra applications need to be patched to prevent bypasses to the global backend. If you do not take measures yourself, you lose your old configuration method. This is very inconvenient and so mounting has been introduced.

1.3 Elektra with mounting

As stated in the previous chapter, using Elektra without mounting has some problems, which we want to summarize:

1. An existing piece of software needs to be patched and linked against the Elektra library to take advantage of it.

2. You need the same backend for every software program to keep the common namespace.

3. Getting a configuration file like `/etc/passwd` in the Elektra namespace requires patching all the tools that may access this file, which can be a lot.

The mounting feature solves all the problems listed above. With mounting, you can specify for every *key directory*, which backend should be used for it and all its keys in subdirectories. So you can use a different backend for every program . This backend can be specialized for a single application. So instead of patching old software it is possible to write a new backend for this application. With mounting, there is no need to dig around in an application's source code, instead you simply add an Elektra backend. So the first item is solved.

With the mounting feature it is easy to specify a backend for an application. You can choose a backend for a program that is linked against the Elektra library and you don't lose the common namespace. So the second point is solved.

For configuration files like `/etc/passwd`, you can write a simple backend. This file can still be used and is mapped into the Elektra namespace. If you make changes to `/etc/passwd`, they are visible in the Elektra namespace. Respectively, if you make changes using Elektra the password file is updated. So the third problem is solved as well.

2 Use Cases

2.1 Views

It is possible to define different roles in Elektra. Every role has its own view. For instance there is the view of the **user or administrator**. He has the advantage of the standardized namespace. Applications exist to edit the Elektra namespace (e.g. kdbedit, kdb commandline tool). The user can use one tool to configure all his applications and he doesn't have to search for the configuration files.

Another view is the one of the **application developer**. He can take advantage of Elektra as a library for reading and writing configuration or he can write a specialized backend for his own application. Another advantage is that he can easily access the configuration of another program or system configuration without portability problems.

The last view is the one of the **package maintainer**. His installation scripts can use Elektra to configure each application individually.

2.2 Elektra without mounting

Oyranos is a color management application and one of the first using Elektra. The developer decided to use Elektra when he started writing his application. He didn't want to write the code for parsing the configuration on his own. And he wanted his program to be cross-platform. Making the configuration system portable on one's own is tedious work. As Elektra is in quite an early stage he did not have much use of the common namespace, because there were almost no other applications that used Elektra at this point, but the Oyranos user can use tools that are shipped with Elektra (e.g. kdbedit), to change the configuration. Since the decision to use Elektra was made at the beginning of the project, rewriting of code was not necessary.

Another project that already used Elektra before mounting was introduced was Samba. Samba already had a configuration system, before someone had made the decision to rewrite it using Elektra. There was a patch around for Samba to use Elektra, but it was hardly used, because it interfered with the old configuration system. If a system administrator used Samba with Elektra he couldn't use his old configuration files, but instead had to use the Elektra tools. Another way is to understand where the configuration is stored, which depends on the backend.

2.3 Elektra with mounting

We want to look at a use case for Elektra with the mounting feature. In this use case we have a **system administrator** of a Unix or Linux system. He wants to add new users to his system. For this purpose he has to edit the `/etc/passwd` configuration file. He can edit this file by hand, but it is easier to use a system management tool. In our example, he uses the program **useradd**. This tool modifies the passwd file. In his system, there is **Elektra** with the **passwd-backend** installed. The passwd backend maps the passwd file into the Elektra namespace and the other way round. So he can use a tool like kdbedit, that is linked against Elektra to add a new user. Although useradd is not linked against Elektra, changes done using kdbedit are seen in useradd and vice versa. If he wants to update his system and installs a version of useradd that is linked against Elektra, let us call it **useradd-elektra**, this is still possible without problems. The tool useradd-elektra simply edits the Elektra namespace, that is written to the passwd file. So he still can edit `/etc/passwd` manually using a text editor.

2.4 Other use cases

1. There are many configuration files in every system that can't be replaced for various reasons. These files can be faded into the global Elektra namespace without applications using these files taking notice. Backends for `/etc/fstab`, `/etc/mtab`, `/etc/passwd` and `/etc/hosts` exist so far.

2. Users or administrators might be accustomed to certain files or syntax without wanting to change the whole configuration using Elektra. The mounting technique allows them to choose.

3. Specific programs may have a very complex and large configuration. Binary files with an index may give them a better performance without missing the connection to the global namespace.

4. It is possible to extend Elektra by writing specialized backends. One important backend is daemon which sends the configuration to a network daemon instead of storing it locally. Doing so you can cache configuration and notify applications.

5. Configuration provided by daemons can't be used for every program. Configuration related to the boot process needs to be available without them, but should also be accessed by applications needing configuration from network.

3 Vision

So far we have learned that Elektra introduces a global namespace which can be accessed independent of the operating system and programming language. That is, at first glance, a nice feature for developers. It can save a lot of time in projects invested in fixing problems of particular operating systems. It also sounds nice to save time for writing support for parsers and generators for every programming language used in their projects.

But there is more about it that can be summarized as integration of software. Applications should share appearance, keybindings, language, proxies and much more. There are user, administrator and operating system preferences describing what should be used. Applications often already have the feature to appear differently and show messages in a different language, but often don't invest enough time to find the perfect configuration for that user and system.

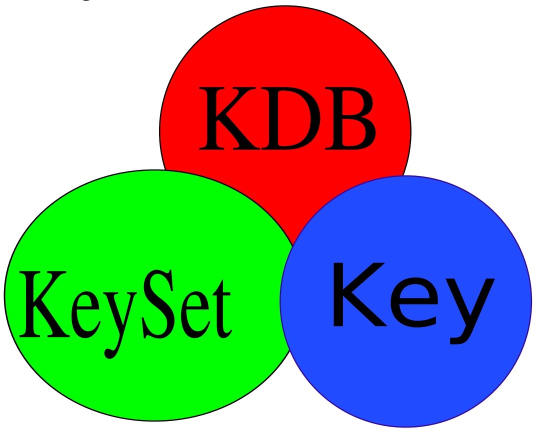
That's where Elektra comes in. Elektra exactly provides a place to access this information easily. It allows applications to tie together in a way which was not possible before giving great power to express what behavior is expected. It is up the programmer to standardize where, in that namespace the information resides.

4 Problem and choices

This section discusses the problems that we encountered when we designed the interface for mounting backends and explains how the internals work together but leaves out implementation details.

4.1 Classes

The API uses an object-oriented design and there are only three classes as shown by the figure:



KDB introduced more and more methods in the past to implement links and to remove or rename keys. With some tricks, we could reduce them to four core functions, allowing for the implementation of mounting in a single place.

*Key*s may now be marked to be removed and be `stat()`ed only. Doing so you gain more flexibility and we were able to provide the same functionality in a new interface with `kdbOpen()`, `kdbClose()`, `kdbGet()` and `kdbSet()`.

4.1.1 Kdb

The following text will try to distinguish between `kdbGet()` and `kdbGet_backend()`. Substitute `_backend` with the name of a specific backend to receive the correct method name of the backend. For example, `kdbGet_filesys()` is the name for the `kdbGet_backend()` method of the backend filesys. `kdbGet()` is the interface for applications to Elektra and `kdbGet_backend()` will do the work and construct the *keyset* out of permanent storage supported by backends. So `kdbGet()` calls `kdbGet_backend()` and the information is passed back to `kdbGet()` to be filtered out and combined together.

`kdbSet()` works the other way round by filtering and splitting the information before passing it to `kdbSet_backend()`.

`kdbOpen()` and `kdbClose()` work respectively without information exchange.

Error Conditions While using `kdbGet()`, error conditions are rare and you see where the error occurs by looking at the last keys you got. But `kdbSet()` may fail at any point. This was solved by marking the key where the problem appeared.

4.1.2 Keyset

The kdb functions described above need a datastructure for efficient information flow. The previously used link list caused some problems. It did not scale well for large keysets, because `ksLookup()`, a very common operation for finding a key in a keyset took $O(n)$ on average. Sorting was also very inefficient, realized by constructing an array out of the list, running `qsort()` and transforming it back to a linked list.

This was solved by using an extendable array as datastructure, looking up with binary search but sorting with `qsort()` remained the same.

Another main problem of the previous implementation was the impossibility to store a key in more than one keyset because the next pointer was directly in the key datastructure. While this was solved with the array, a new problem appeared because of `ksDel()` calling `keyDel()`. That caused a problem that a key might be deleted which was used in another keyset.

The most visible change is that `ksNew()` now takes a parameter to give a hint for the size of the keyset. Afterwards, there is a variable number of pointers to keys that will be added to the freshly allocated keyset. The advantage is that the user can create an arbitrary keyset with a single C-statement, which is heavily used by `ksGenerate()` and `keyGenerate()`, that both allow to output the particular C-code for a keyset and a key.

`ksDel()` is used to `keyDel()` all keys, which are in the keyset. This principle conflicts with a key in multiple keysets. To avoid this problem, a reference counter for keys was introduced. Whenever a key is added to a keyset it is incremented and it is decremented when it is removed. The key itself will be deleted when the last keyset holding it is deleted.

4.2 Mountpoints

From the very beginning, it was clear that the backend selection criterion should be the key name. Doing so, you can get logically grouped keys together in the same backend as desired. There are different possibilities to achieve mounting, each with its own strengths and weakness.

In this section, we will look at different approaches we evaluated to implement in Elektra and present our choice afterwards.

4.2.1 Special Type Approach

In Elektra, every key has its own type describing the kind of information it holds. This approach introduces a new key type `KEY_TYPE_MOUNT`. Whenever such a key is noticed by `kdbSet()` or `kdbGet()`, the backend, named by the value of this key, will be used from then on for all subkeys. The subkeys themselves also can be of the type `KEY_TYPE_MOUNT`.

+ There is no limitation where the mountpoint is. Any backend being capable of representing a key of type `KEY_TYPE_MOUNT` may have arbitrary other backends mounted without any other external information needed. Using this possibility one network backend may delegate to another one without any local information about that.

+ Performance impact only takes place when actually reaching keys containing mountpoints. It can be implemented without any significant loss compared to not using mountpoints, thus `kdbOpen_backend()` can be omitted for backends not visited.

  You could even avoid having a data structure containing any information about mountpoints. Doing that every `kdbGet()` and `kdbSet()` would produce the same expense but reduces initial effort in `kdbOpen()` since no backend needs to be loaded.

- `kdbGet()` and `kdbSet()` need to walk through every part of the keyname from root to the requested leaves. This requires some effort without backends, but could be very expensive using network backends redirecting to local backends.

- Every user must have a local key from the *default backend* even if another backend is used for `user/`. This requires the user to have a home directory and would not be very convenient when using a network backend for users.

- The implementation effort is higher because every backend must be modified to support the type `KEY_TYPE_MOUNT` and there is no central place where a data structure can be built. This approach can't be used as a solution for *backend configuration*.

Mainly because of the last issue we decided not to implement mountpoints with a special type.

4.2.2 Static Configuration Approach

Another method turned out to be central storage for all information where each backend should be mounted. This could reside in an external configuration file, but the much easier and convenient way is to store it in the key database itself.

The predefined place storing the configuration is `system/elektra/mountpoints`. This key directory contains configuration, i.e. mountpoints, name of the backends and their configuration.

+ Users are not required to have any files related to configuration in their home directory. Root directories can be mounted in a central place controlled by the administrator.

+ The performance impact depends on the number of backends mounted but not on the number of keys, leading to good overall performance.

- You need an extra implementation for users to make their `user/elektra/mountpoints` work.

+ After building an in-memory data structure of all mountpoints, the changes to the rest of Elektra are easily understandable and there is no need to modify the backends.

4.2.3 Hybrid Approach

The two possibilities fit together well. While the configuration approach makes the administrator's life easier, the type approach provides an ad-hoc solution for users and applications to use a specific backend. Backends do not notice if they are mounted by configuration or a special type.

In this scenario, the configuration approach is only used for mounting `user/`, network backends and backends not supporting `KEY_TYPE_MOUNT` type. Keeping it to a minimum, reduces the backends which must be opened at `kdbOpen()`.

One drawback is that it adds some more complexity inside Elektra, but not visible to the programmer. The other is that it needs a lot of optimization techniques to handle both scenarios well.

4.2.4 Dynamic Mounting Approach

A completely different approach breaking the global namespace of Elektra is `kdbMount()` and `kdbUnmount()` to let software using Elektra decide where to have which backend not visible by other processes.

This might look like nonsense at first sight, but could be used by a daemon backend to extend the namespace on a local machine. Below the path where the daemon backend is mounted, the daemon process chooses dynamically which backends to use. Though all applications using that path send the keyset to the daemon, they all see the same keys below, restoring the global namespace.

4.2.5 Elektra Approach

The decision was made in favor of static configuration for `system/` only and provides dynamic mounting. The use of type `KEY_TYPE_MOUNT` and configuration of mounting for users may be added in future.

4.3 Locking

Elektra does not claim to fulfill either all requirements of a database or a filesystem. The key database is stateless, that means after `kdbOpen()` a random sequence of `kdbGet()` or `kdbSet()` can occur with only `kdbSet()` influencing a consecutive `kdbGet()`. The actions take place at once and there is no more information present than what exists in the database.

Backends should support read/write locks. Because of this, concurrent calls only impact in a sequential order.

Because of an imbalance use in terms of frequency towards `kdbGet()`, the issue that `kdbSet()` locks a whole hierarchy does not influence negatively. But substantial benefits arise because deadlocks can't occur and implementing backends becomes easier.

4.4 Capability

Some backends fulfill the whole specification of `kdbGet_backend()` and `kdbSet_backend()` and support all Elektra features. Other backends have principle limitations and do not implement all features described in the specification but are useful nevertheless.

One reason may be that some parts are simply not implemented because of lack of time or the software did not need more features. The other reason is because of problems with syntax to represent all features. Some file formats just lack a way to express a comment or a key type.

To handle this problem we created a data structure describing what capabilities a backend does not have. With that technique, you can use the testing framework for developing the backend from early stages. To do so, just declare your backend has only implemented minimal features and delete step-by-step missing capabilities while your backend evolves.

4.5 Access types

Elektra used to have a 3 bits access representation similar to the filesystem bits. While the read write bits have their usual meaning, they control if value/comment may be changed, the executable bits had no meaning.

Semantics changed concerning directory keys. One executable bit is sufficient to allow the backend to create a directory. User, group or world need their appropriate executable bit set to access the subdirectories.

4.6 Memory allocation

A design decision is that everything allocated by Elektra will be freed, which avoids unreproducible bugs triggered by the use of different libc or implementations of `malloc()` and `free()`.

4.7 Thread safety

To achieve thread safety, backends are not allowed to use any global variables. Instead, they should use `kdbhSetBackendData()` to store backend related data.

5 Implementation

Elektra provides a global namespace. The idea behind backend mounting is that you can choose, with limitations, which backend you use to read or write a key. Backend selection is done by the full key name. So you have to provide a table of directories mapping the Elektra namespace with the corresponding backends. Elektra then chooses for each key the longest directory entry in the table, that is a subdirectory of the absolute keyname. If this is not available, it uses the default backend.

5.1 Backend configuration

`kdbOpen()` loads the default backend. Then it reads its own configuration, using this backend. This contains the mountpoints for the various backends and its configuration and is usually located under `system/elektra/mountpoints/`. The configuration will be passed on to the backends. Because we introduced the *backend configuration* with the backend mounting all together, we can now use it different ways, e.g. to mount it to a different mountpoint or to use a different file to store data, because the backend may be used more than once.

5.2 Backend selection

There is a need to determine for a single key which backend it belongs to. This can be simply solved by looking up the table of all mountpoints and choosing the longest entry, that is completely a substring of the key name. Therefore, each mountpoint in the table must end with a '`/`' to make the lookup correct. Otherwise, two mountpoints that have the same name, except for one of them which has some additional characters at the end, could not be distinguished.

The table lookup is a very common operation and should be as fast as possible. The creation of the table is only done once and so time efficiency is not so important. To solve this problem, two data structures for the implementation were considered.

5.2.1 Hash Table

A hash table is a very efficient data structure that has constant-time ($O(1)$) key-value lookup on average. This sounds good at first sight, but the problem is a bit more complicated than that. Instead, you search for the longest prefix for the key-value. To use a hash table, we must compute the index for every substring of the key and look for the associated value. From there, the complexity is proportional to the length of the string. That is why a hash table doesn't seem to be optimal for this kind of problem.

5.2.2 Trie

Instead of a hash table one can use a tree, that has $O(\log(n))$ complexity for lookup, where n is the number entries in the tree. The number of entries is the same as the number of mountpoints and should not be very high. Even if someone for an unexpected usage uses millions or billions of backends, the $O(\log(n))$ complexity would scale quite well. In such a case, memory consumption is likely to present a bigger problem than time complexity. But for the intended use of Elektra, the number of backends will be quite small, maybe up to a few hundred at most.

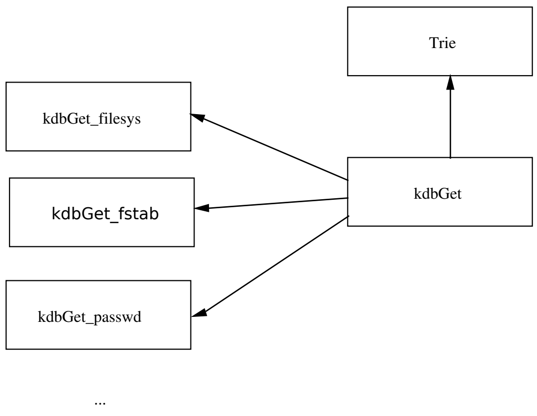
A requirement is to make the lookup as fast as possible, so optimizations need to be considered. One possibility is to use a kind of tree that is often known as trie[5] in literature. For each character, there is a possible branch in each node. To choose a path to walk down, the algorithm uses the value of a character as an index for the array of subnodes. This simple array lookup can be done in $O(1)$. If the trie branches at every character, it will be very loose. To compress it, all nodes with only one child will be compressed into their parent node. So the trie only branches if there is more than one to walk down. The compression also quickens the lookup time, because the lookup-functions use `strcmp()` instead of comparing each character on its own.

5.3 kdbOpen

After `kdbOpen()`, the default backend is opened. Then we read the configuration and create a data structure that contains all mountpoints and corresponding backend handles. This data structure should be able to look up the mountpoints fast. We decided to use a trie to speed up the lookup. So we have to create a trie in `kdbOpen()` and delete it in `kdbClose()`.

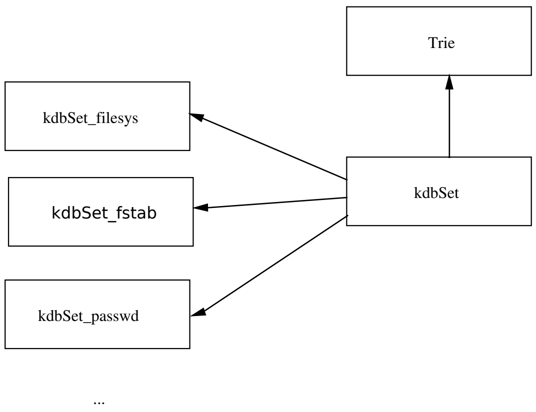
`kdbOpen()` returns a pointer to a newly allocated handle. This has to be done for every thread using Elektra.

5.4 kdbGet



The function `kdbGet()` is the only interface to read activities of backends. The parameters are a handle from `kdbOpen()`, a keyset, a parent key and an integer for options. The function first fetches the parentKey by looking up the backend for it and running `kdbGet_backend()` for it. The received keyset will be post-processed by sorting hidden keys out and considers other options. The directory keys will be handled by a recursive call of `kdbGet()`, making sure that mountpoints are considered.

5.5 kdbSet



The function `kdbSet()` has the same parameters like `kdbGet()` but it works the other way round. Here you already have the full keyset and the job is to separate keys to their backends. To do so, the keyset is pre-processed by grouping keys for the same backend together. When at least one key has changed the keyset will be transferred to persistent storage by looking up the backend and running `kdbSet_backend()` for it.

5.6 Dynamic mounting/unmounting

The usual way to mount a backend is to use the Elektra configuration. But sometimes, you don't want to close and reopen afterwards just because there is a little change in the mounting structure. So we implemented dynamic mounting. You can use the function `kdbMount()` to mount a new backend or you can unmount it using `kdbUnmount()`. This function only changes the current *mounting table*, but not persistently. Closing and reopening kdb will drop all changes made by `kdbMount()` or `kdbUnmount()`. To change the mounting table persistently, someone has to edit the configuration of the mountpoints.

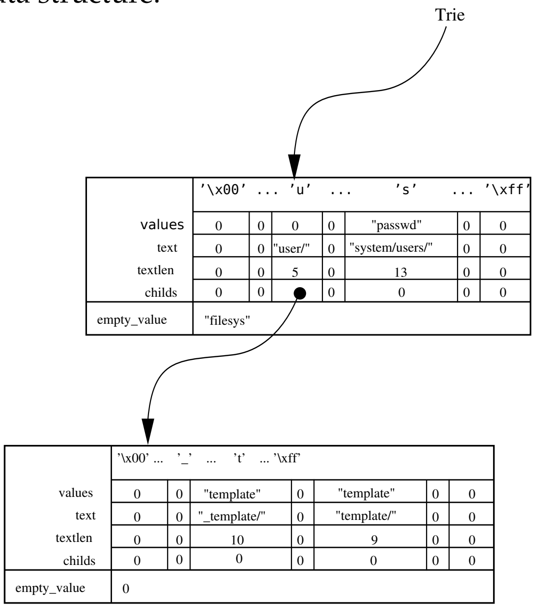
These functions come in handy if a program wants to use a modified mounting table that is only used by it. A sample application would be a configuration program that allows you to preview the changes before they are applied to the configuration.

5.7 Trie

For the trie implementation, we created a data structure:

```
struct _Trie {
     struct _Trie* children[MAX_UCHAR];
     char *text[MAX_UCHAR];
     unsigned int textlen[MAX_UCHAR];
     void *value[MAX_UCHAR];
     void *empty_value;
};
```

We have an array entry for each character. The element's children hold the pointers to the subnodes. The `text` element holds the string that is stored in the node, `textlen` is the corresponding length of this string and `value` contains the backend information. The item `empty_value` is used to store the *root backend*. The following graphic illustrates the use of this data structure.


Trie
'\x00' ... 'u' ... 's' ... '\xff'
values 0 0 0 0 "passwd" 0 0
text 0 0 "user/" 0 "system/users/" 0 0
textlen 0 0 5 0 13 0 0
childs 0 0 ● 0 0 0 0
empty_value "filesys"
'\x00' ... '_' ... 't' ... '\xff'
values 0 0 "template" 0 "template" 0 0
text 0 0 "_template/" 0 "template/" 0 0
textlen 0 0 10 0 9 0 0
childs 0 0 0 0 0 0 0
empty_value 0

5.8 KeySet

Keys can be joined together to a set and need to fulfill various requirements. Applications typically need one to a hundred thousand keys. To provide an efficient set for all of these requirements, we used an extensible array storing pointer to keys.

Starting with `ksNew()`, an application can give a hint how large the keyset is at initialization. Using that, the keyset is not likely to grow too often.

The allocated space is doubled each time because most allocators work quite well with power of 2 storage sizes.

```
struct _KeySet {
    struct _Key **array;
    size_t        size;
    size_t        rsize;
    size_t        alloc;

    struct _Key  *cursor;
    size_t        current;
    uint32_t      flags;
};
```

`array` contains an array of pointers to keys. While `size` is the size of all keys, `rsize` is the size of keys having the `keyRemove()` flag set, used to provide a sort order needed for `ksLookup()` and `kdbSet()`. The allocation size `alloc` implies the actual allocated size of the array.

The attributes `cursor` and `current` together with the functions `ksSetCursor()` and `ksGetCursor()` provide a cursor to save and restore the state of the keyset. Internal `current` will be used as index for the array and changed by `ksSetCursor()`.

6 Status

The new interface `kdbOpen()`, `kdbClose()`, `kdbGet()` and `kdbSet()` works properly. There is a testing framework checking all functions with valgrind for memory leaks. The keyset works, keys have a reference counter to automatically get freed and the sorting will happen whenever needed.

There are 4 backends working with some limitations: fstab, passwd, hosts and filesys. You can mount backends statically by adding the requested entry in `system/elektra/mountpoints`, but you can also mount backends dynamically by `kdbMount()`.

7 Further Work

The stable version has not been released yet and we are endeavoring to weed out bugs and to extend the test framework to achieve full coverage.

`kdbSet()` has an awkward solution for splitting the keysets requiring walking through the keyset multiple times and not taking benefit of the sorting. Furthermore, the parent key is not taken into account and error scenarios are in a very premature state and we expect a lot of work there too.

Not-yet implemented other mounting strategies are of further interest like the special type approach. `kdbMount()` works, but the actual use case, the daemon, just uses a single backend not capable of building its own namespace out of configuration.

8 Conclusion

We showed that it is possible to implement a global configuration namespace with any desired backends. With that solution, efficient backends can be used to handle applications needing a large configuration or a small startup time. Furthermore, old configuration files can still be used. This and more can be achieved without losing the ability to access the whole configuration of the system by any application using Elektra. Because of a faster implementation of keyset, the performance impact is tolerable.

This work certainly gives a perspective on what future configuration libraries can offer. Additionally, it gives a basis for the next stable Elektra release.

9 Glossary

backend Elektra has to store the key-value pairs somehow. Many different ways exist, e.g. to store it in an xml-file or in a database. Elektra is not tied to a single method and every technique to save the key-value pairs is implement in a backend. Before this work, it was not possible to have multiple different backends at runtime, but with the backend mounting feature you can choose the backend of your choice for each key-value pair.

backend configuration Every backend can be configured. Typically parameters are to use different files or servers for network connections. A keyset will be passed to the backends holding this information. This was introduced during our project thesis.

binding Elektra is written in C, but it should be possible to use Elektra in other programming languages. A language binding maps the Elektra API to a different programming language.

mounting table There is a table to choose a backend which contains the mountpoints associated with the backend that is used for this key and all it's subkeys, unless there is no other entry in the table for this key. If multiple different entries exist for a key, the longest will be chosen.

default backend Someone can use Elektra without the new mounting feature. In this case the default backend is used. The backend that is used as default backend is chosen at compile time. The default backend is also used to read the *backend configuration*, especially the mounting table. At this point Elektra doesn't know anything about the existing backends, so it needs a backend that can be used for this task. The default backend is also used if there is mounting for a key, if there is no entry in the mounting table for the key name.

root backend In contrast to the default backend, the root backend is only used in combination with mounting, i.e. a loaded mounting table must exist. Otherwise, no root backend is present. The root backend is the backend used if there are no other backends for a key in the mounting table available. If mounting is loaded, the root backend supersedes the default backend, although the default backend is used as a root backend if the latter is not specified.

key As Elektra deals with key-value pairs, a data structure was introduced to represent these pairs. Each key has a name, a value, an optional comment and some metadata. There are a lot of functions available to access or modify key data. The key data structure is designed so that it can be implemented as a class in an object-oriented programming language.

keyset Since dealing with many single keys has a negative effect on performance, a container exists for a set of keys or, to be exact, a bag of keys. Like the key data structure, the keyset is designed for implementation as an object in an object-oriented language binding.

key directory Each key can have subkeys. If there are subkeys for a key present then this kind of key is called a directory key.

mountpoint A mountpoint is a key directory. Below that point, another backend will take care of those keys.

References

[1] Jon Louis Bentley. *Writing Efficient Programs*. Prentice Hall, first edition, 1982.

[2] Helmut Herold. *C-Kompaktreferenz*. Addison-Wesley, first edition, 2002.

[3] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Prentice Hall, second edition, 1988.

[4] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, first edition, 2000.

[5] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen*. Spektrum, Akad. Verl., fourth edition, 2002.

[6] Gary V. Vaughn, Ben Ellison, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake and Libtool*. New Riders, first edition, 2000.