MikroConf: A framework for building command-line interfaces, and specific application for the administration of Linux-based network devices.

Stergiakis Alexandros <alsterg@gmail.com>

September 28, 2010

Abstract

In this paper we present MikroConf, a framework for building Command-Line Interfaces (CLI), and the application of it in the administration of Linux-based network devices. MikroConf introduces a number of innovative features that are unique even to commercial-grade products. It is written in Tcl and C, and exploits many of Tcl's capabilities to provide an a rich and consistent interface to users and developers alike.

Introduction

It is typical for large and complex software projects to incorporate a Command Line Interface (CLI) for configuration and/or administration purposes. Such interfaces exploit a more user-friendly, and robust human-computer interaction paradigm, compared to other configuration methods, such as configuration files, shell CLIs and command-line parameters. In the case of network devices, it has become a requirement for vendors to provide a CLI in conjunction with a GUI. MikroConf is the best free/open-source candidate for this application.

Strictly speaking, MikroConf is both a framework for buidling CLIs and the specific application for it in the administration of Linux-based network devices. In the rest of this paper we always use this name to refer to the latter use, however, all of the features covered herein can be applied into any other application.

MikroConf introduces a number of innovative features for the user of a network device:

- It supports (in a safe manner) in-line scripting in the Tcl Programming Language. Regular configuration commands and Tcl commands can be intertwined on the command prompt.
- It supports an event-based configuration paradigm. Tcl procedures can be registered to be invoked when certain events occur (e.g. link up/down).
- Automation of repeated tasks via parameterized macros (Tcl procs).
- Command output can be captured, filtered, manipulated, emailed etc.
- Usability features such as: ANSI terminal colors and highlights, full screen editing of configuration, and others.

These features enable administrators to be more productive, and achieve more robust device configurations.

From the point of view of the developer, MikroConf offers a number of features that eases extensibility and integration:

- It features a modular architecture. New device functionality is introduced in the form of MikroConf modules, which can be easily written in Tcl or C. Dependencies are handled automatically.
- There is clear separation between the declaration of command syntax and its enforcement code (command handler).
- XML description of command structure, syntax and hierarchy, authentication/configuration modes, running-config sections etc.
- The highest possible flexibility in specifying command syntax, described as a deterministic finite automaton.
- Server-Client architecture and transport protocol independence. Multiple concurrent user sessions are supported.

This paper describes the design and organization of MikroConf's framework, its unique features, and how Tcl has been used to implement this design.

The MikroConf Architecture

MikroConf's architecture is divided into the framework and a number of application-specific modules. The MikroConf Framework (or simply Framework) is the backbone on which MikroConf modules (or simply modules) are attached to implement the various configuration subsystems. For example, there can be a module for IP configuration, and another one for dynamic routing.

The goal of the Framework is to make life easier for the developer, by providing a clear API that he can use to easily implement new modules. The API automates most of the common tasks involved in module writing and operation. Examples of such tasks are:

- Module (un-)registration and inter-module dependency handling.
- File system monitoring.
- Command, configuration/authentication mode (un-)registration.
- Logging.
- Dependencies with external binaries, kernel, busybox, or library features.
- Input/Output and error reporting and handling.
- Running external programs and daemons.

A MikroConf Module is a pluggable component that enables configuration of a specific aspect of the system. It is universally identified by its name and its version number. It consists of a set of Tcl source-code files and a set of XML files. These two sets of files help separate the configuration enforcement code, from the aspects that pertain to the interaction paradigm. Simpler modules would most likely have a single source-code file and a single XML file that together constitute the whole module code.

The XML files, also called *Specs files*, define what we call in MikroConf, *Module Specifications*. That includes:

- New configuration modes defined by the module
- New authentication modes.
- New commands and paths.
- The structure, syntax and description of each command.
- New running-configuration sections and entries.

The exact structure of a Specs file is described by a DTD document, included in MikroConf's sources.

The Tcl source-code files, are collectively called *Configuration Enforcement code*. They are comprised mostly by callback procedures, which are called at various occasions, such as when:

- The module is first loaded
- A command that belongs to the module is executed
- Another module is requesting an exported service
- The module is unloaded or reseted.

In case where many source-code files are used, one of them is the entry-level file, also called *Module file*. This file is loaded first, and loads any other source-code files as necessary.

High-Level Design Choices

During the design and implementation of MikroConf we made a number of important decisions, that have deep and long-lasting effect on MikroConf. In this section we discuss and justify these high-level design choices.

Use of Interpreted Language

Applications for embedded systems are traditionally written in a compiled language, and in particular in C. This practice is justified by the following facts:

- C is suitable for low-level system programming, which traditionally is needed for deeply embedded systems. Such embedded systems don't feature a complete operating system, but instead the embedded software must tackle with the hardware directly.
- C produces compact code with very few library dependencies, resulting in small flash and memory footprint. The C standard library is quite small comparing to the standard libraries of other languages like C++.
- Historically C compilers produce code that runs fast. However, newer compilers for other languages produce equally fast code.
- C (like any compiled language) produces binary code that “hides” the source code on the target device. This is desirable for Intellectual Protection, as well as for minimizing exposure to patents and copyright threats of other parties.

With the advent of Linux and the increased capabilities of the hardware, many of these factors have become invalidated. In particular:

- It becomes possible now to use C for the low-level system programming, and an interpreted programming language for other parts of system programming, that don't require low-level access to the hardware.
- The extra overhead of interpreted languages becomes bearable, as embedded systems have grown in hardware resources, from a few hundreds of kilobytes of memory and flash, to a couple of megabytes.
- The processor speed has also been increased, making possible to run the slower interpreted programs with sufficient speed.
- Interpreted languages have been improved in technology. It is now possible to compile interpreted code to bytecode that runs in a comparable speed to C programs.

Based on these facts, and contrary to the tradition, we decided to use a combination of compiled and interpreted languages for MikroConf. C is used for the parts that need low-level system programming, and for the rest Tcl, a high-level interpreted language. This combination gives us the following advantages in comparison to using pure C:

- Easy debugging and patching. The code can change on the fly without recompilation

on the embedded system.

- Tcl is a glue language. This promotes re-usability of existing system-level components (external programs), by glueing them together with a thin interpreted language code layer.
- Speeds-up development process. Programming in Tcl is much more easier and efficient than in a compiled language. Studies have shown that there is a 1-10 correspondence between Tcl and C code, in terms of lines of code. In other words, one line of code of Tcl, would take ten lines of code if it was written in C (in average).
- Since less code is being written, it is easier to spot logical bugs in the code.

Use of Tcl

It turns out that Tcl is particularly suitable for the task, for the following reasons:

- Its syntax naturally resembles the customary user interface paradigm used in CLAIs, which is: word1 word2 word3 ... So Tcl code and device configuration commands look the same. Which means that both can coexist naturally on the same command prompt.
- The majority of Linux system configuration is performed via text commands or text files, which makes Tcl a good choice, because everything in Tcl is text and it has superior text processing capabilities.
- It’s easy to learn, both for developers (writing modules) and users (in-line scripting).
- The standard Tcl interpreter is quite small comparing to some other interpreted languages, around 700kb, and there exist a number of different minimalistic TCL implementations, which are less than 100kb.
- It turns out that the question mark and tab characters don't have any special meaning in Tcl, and hence they can be used by MikroConf without breaking interoperability with Tcl.

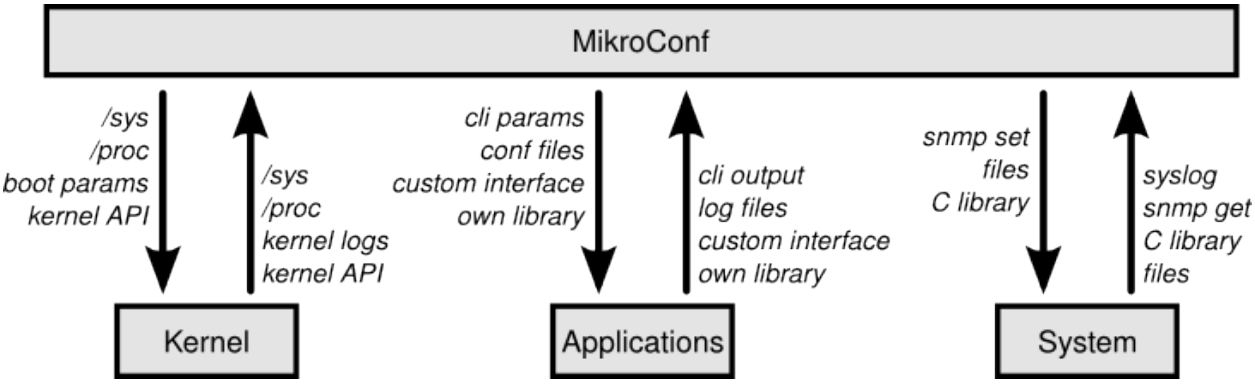
We found in Tcl a number of build-in features that were particularly helpful for the development of MikroConf:

- The ability to rename and delete commands.

- Variable, command and execution traces.
- The build-in mechanism for finding and loading Tcl modules.
- The Tcl Virtual File System.
- Tcl threads and interpreters.
- The ability to customize error handling.
- The catch-all mechanism for commands that are not recognized.
- It's standard library and many other libraries and extensions.
- The Safe interpreter security mechanism.

Maximizing Outsourcing

To minimize development effort, one important design goal for MikroConf was to maximize exploitation of what it is already offered by the FOSS community. The idea is to utilize existing FOSS projects and mechanisms that are included in a typical Linux-based firmware, both for configuration enforcement (set) and information retrieval (get). And since each individual FOSS project uses radically different approaches for set and get, a glue language like Tcl, comes very handy to integrate all these heterogeneous components together.



Drawing 1: Set/Get mechanisms MikroConf uses to interact with its software environment.

Set Method	Examples
External Program Execution (exec)	Interface configuration (ifconfig)
Kernel Virtual Filesystems (/sys, /proc)	IP forwarding, Security options
Kernel Boot Parameters	Console settings, Password recovery, Module parameters
CLI program arguments	Busybox DHCP Client (udhcpc)
Application Configuration files	DNS Server (dnsmasq)
System Configuration files	Timezone (/etc/TZ), Static IP-Hostname mappings (/etc/hosts), DNS Settings (/etc/resolv.conf)
SNMP SET	Not currently used

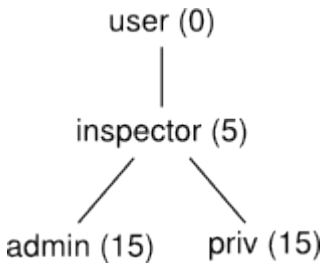
Table 1: Configuration enforcement methods.

Get Method	Examples
Syslog	Debug messages (`debug ...`)
Program Output	Interface Configuration (`show interfaces`), Logged-in Users (`show users`)
Kernel Virtual Filesystems (/sys, /proc)	OS info
Low-level APIs through Tcl binary extensions	Checking POP3 mailbox (`mail check`)
SNMP GET	Not currently used

Table 2: System information retrieval methods.

Authentication Modes

MikroConf supports arbitrary number of authentication modes organized in a tree hierarchy, with increasing privilege level from the root to the leafs. Therefore the authentication mode on the root is the less privileged of all. The side figure gives an example tree of authentication modes. It consists of four modes: user, inspector, admin and priv, with corresponding privilege levels: 0, 5, 15 and 15 respectively.



Drawing 2: A tree with four authentication modes.

When a User logs-in to the system, he automatically enters the root of the tree. In order to move to an other authentication mode, he has to traverse the complete path that connects the two modes.

Certain commands can be made accessible only to authentication modes of certain privilege level or higher. Due to the organization of the authentication modes in a tree structure, this is the same like saying that certain commands can be made available to subtrees of the complete tree of authentication modes. Switching mode requires the execution of a MikroConf command, which of course must be visible in the authentication mode we are switching from.

Different MikroConf Sessions can be artificially confined within different subtrees of the complete tree of authentication modes. When this feature is used, then a session enters directly on the root of its subtree, and cannot escape to authentication modes above its artificial root. We will see why this feature is useful when we talk about Background Sessions in a later paragraph.

An authentication mode is declared by some MikroConf module which is called the owner for this authentication mode. Thereafter, other modules can make use of this authentication mode, as long as the owner module remains loaded.

Configuration Modes

Configuration Modes are in many ways similar to Authentication Modes. They are organized in a tree structure, and moving around along the tree paths requires execution of commands. Sessions can be confined in sub-trees of the whole tree of configuration modes, and upon session establishment they always enter to the root of their tree.

Every configuration mode dictates the minimum privilege level required in order for its commands to be executed. Individual commands within a configuration mode can optionally impose a more restrictive privilege level, but not a less restrictive one.

The module that initially declares a configuration mode is the owner of this mode. Other

modes can then expand it with their own commands. Hence, a configuration mode can be collectively defined by many MikroConf modules, each of which defines a segment of it. However, the owner of a mode should always remain loaded, as long as other modules rely on it.

Configuration Enforcement

In the previous section we enumerated the methods that we are using to apply configuration directives in a MikroConf-enabled device. In this section we further elaborate, and discuss the assumptions that we make, the overall design of our configuration enforcement approach, and some of the internals of the implementation.

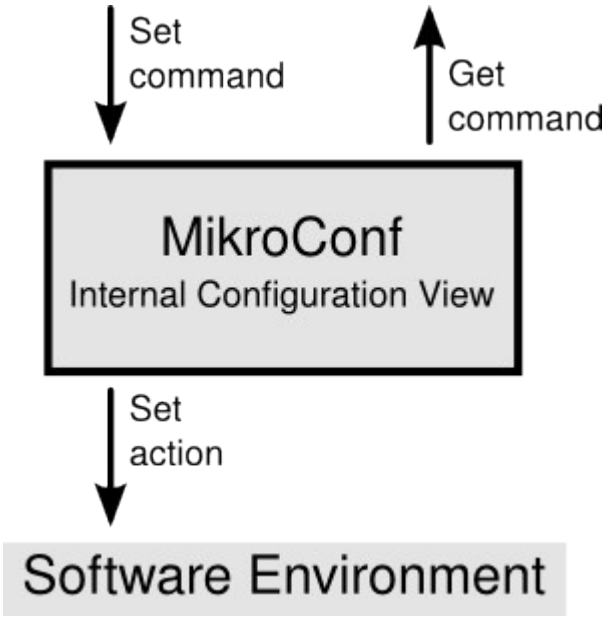
Design Choice: One way flow of configuration

In MikroConf, when setting configuration, actions are taken to enforce it on the surrounding software environment. However, when showing configuration with the `show running-config` command, the software environment is generally not queried. The answer will be based on what MikroConf things is the case, which of course depends on the configuration commands that have been successfully applied so far.

Conversely, if there is a bug in MikroConf and a configuration command is accepted but not enforced, or if the underlying system configuration changes via other means, then `show running-config` will show a configuration that does not correspond the “real” system configuration. We can say that the running-config shows the intention of the administrator, and not the actual system configuration. In the optimal case where there is no software bugs, the two should be match. This is not necessarily the case for other show commands, that typically display the output of external programs, which accurately reflect the actual system configuration.

This one-way flow of configuration greatly simplifies the design and implementation of MikroConf modules. Otherwise, module writers would have to query the underlying system and/or parse complex configuration files in order to derive the necessary information to generate the running-config. Running a program with some arguments or writing a complex file, is far easier than parsing the output that a program generates or a complex configuration file. Parsing would

most certainly need regular expression matching and/or parsing based on grammars (with scanner and parser).



Drawing 3: The flow of running configuration directives.

A side effect of this decision is that all configuration must go through MikroConf. **No manual configuration is allowed, or by any other means**, at least for a subsystem that a MikroConf module exists and is loaded.

MikroConf's *Internal Configuration View* is essentially Tcl variables, arrays and memory objects that store the running configuration in the format that is most convenient to MikroConf module. The running-config is generated based on this data. The startup-config on the other hand is merely a regular file on the file systems.

Design Choice: Active generation of running-config

One approach in generating the running-config, is to merely remember the issued (and successfully executed) commands, and listing them when the running config is requested. We call this passive generation of the running-config, because it doesn't involve any Tcl code that generates dynamically the running-config.

This approach has many flaws. One problem is that it is difficult to match the positive and

negative forms of a command. Remember that the two forms don't need to have the same syntax, and a negative command can cancel many positive ones at once. Therefore, merely recording commands and listing them, has the problem that it is not possible with the information available to match positive to negative commands and vice verse.

In any case, an approach must give satisfactory answers to the following questions:

1. One question is in what order to list the commands in the running-config. Merely listing them in reverse order of execution is not enough.
2. A second issue is how to handle side-effects that the execution of a command can have on other, previously executed commands. For example, turning off RIP dynamic routing, should also remove all other RIP-related commands.
3. Sometimes the negative form of a command must be displayed, instead of the positive form. An example is the `no shutdown` command under interface configuration mode.
4. Sometimes a positive or a negative command takes effect by default, but is not necessarily listed in the running-config.
5. And finally, any MikroConf module must be able to include commands for listing in any place in the running-config, and at run-time.

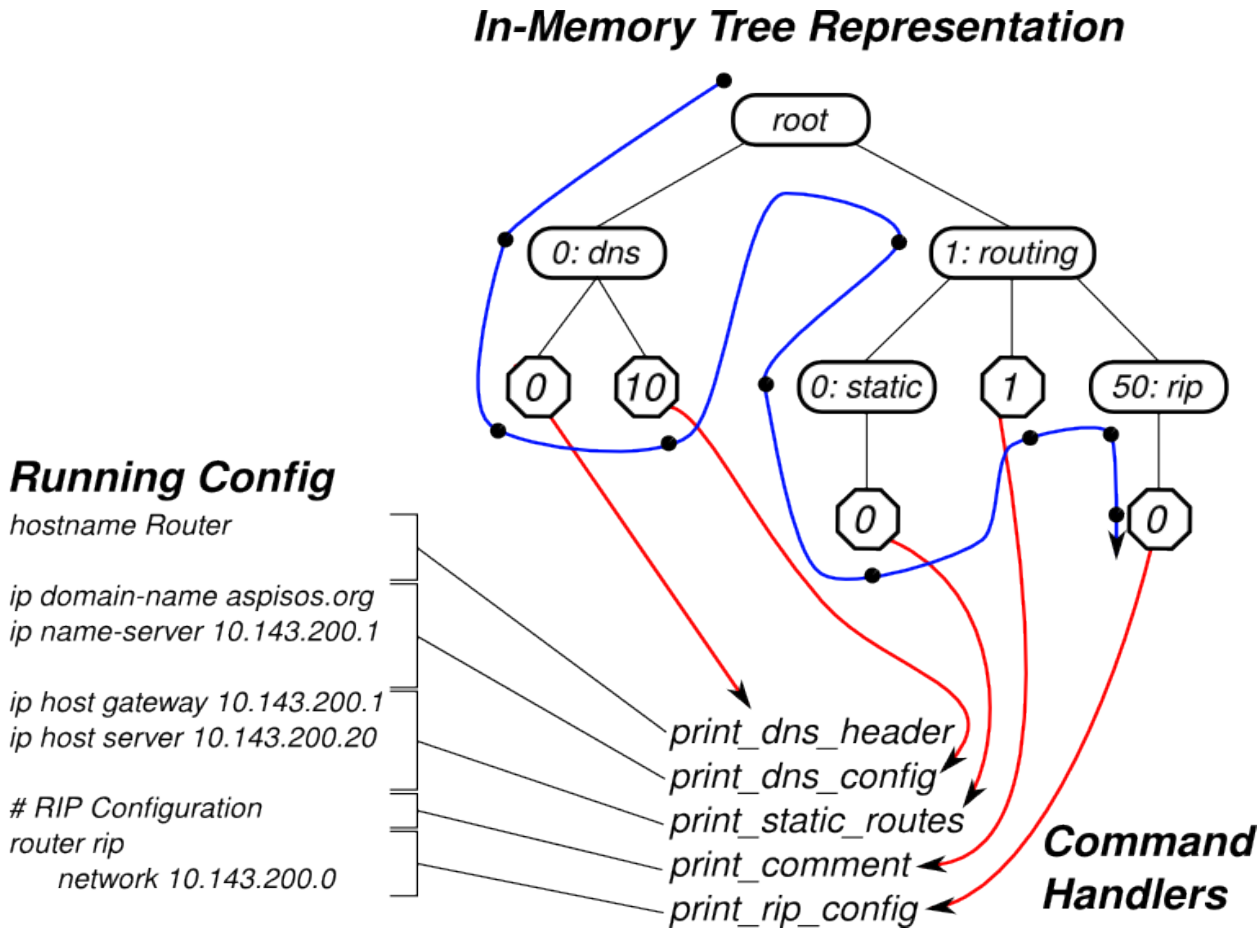
The approach that we follow in MikroConf enjoys all this versatility.

There are two elements that together construct the in-memory representation of the running-config: The *Running-config Section* (or simply section) and the *Running-config Entry* (or simply entry). The in-memory representation is a tree structure, consisting of sections that include as children other sections and/or entries. At each branch level on the tree all sections and entries are ordered based on an *Order Number* assigned to them.

Each section has a name, a parent section, and an order number. Each entry has a parent section, an order number, and a callback Tcl procedure. The callback Tcl procedure when executed returns text that is displayed verbatim on the running-config.

When the running-config is requested, the algorithm visits all sections and entries in the tree, starting with the root and progressing to the leafs in a depth-first fashion. At each branch the

section or entry with the smallest sequence number that hasn't been visited before is selected first. When an entry is encountered, its callback procedure is executed and the returned value is concatenated at the end of the "working" running-config. When all the tree nodes are visited the algorithm terminates, and we have the complete running-config.



Drawing 4: Generation of running-config from in-memory tree representation. The curved blue arrow shows the order with which the Depth First Search algorithm visits the tree nodes. The red arrows point to the event handlers for the respective running-config entry.

Albeit versatile, this design has one disadvantage: Normally the syntax specifications for MikroConf commands reside on a separate XML file (or many) that accompanies a MikroConf module. With this design, a module's Tcl code also needs to be aware of the syntax of the individual commands, since the commands have to be generated back from the Internal Configuration View.

Design Choice: Apply anew on boot

As configuration commands are enforced on the software environment, files get written and initialization scripts can change. These system files reflect the running configuration, but they are not kept for next reboot. Instead, the startup-config is applied on boot and all configuration enforcement operations are preformed again.

At first glance, this doesn't seem a good design, as the boot time increases. Keeping those files would reduce boot time, because we wouldn't have to go through the normal command-execution/command-enforcement loop of MikroConf.

However, there is a very good reason for doing so, which also simplifies MikroConf's design. The problem arises when the hardware settings of the underlying device change between two boots, or a MikroConf module is removed or get updated. When the system reboots, some configuration files on the file system will not reflect the changed environment, and this can cause hidden configuration or misconfiguration on the device.

On the other hand, if the startup-config is always applied on boot, and no system-level configuration files are kept between reboots, then such changes in the hardware or software environment will most likely cause an error in the execution of some of the commands in the startup-config. MikroConf will reject the erroneous commands, and the system will remain in a stable state.

Design Choice: Incremental Configuration

From what we have discussed so far, it must be clear that the running-config and the underlying system configuration must be synchronized at all times. A configuration command must always be reflected back to the operating system. This is an Incremental Configuration approach, because it allows for single independent commands to be executed and take effect immediately.

We also considered a non-incremental approach that seems to simplify the design, but we ruled it out on the basis of failing user's assumptions. In the non-incremental approach we would enter configuration commands at some point in time, and at some later point we would enforce

them all at once by entering the special command `submit`. Then the internal view would be synchronized with the operating system. This approach makes module coding easier, because we don't have to worry about the interactions between different commands. All command handlers assume factory settings before they are executed.

For example consider the case: (it does not apply to MikroConf, but demonstrates the point)

```
Router(config)# service password-encryption
Router(config)# enable password foobar
```

In this simple example we enable password-encryption before we specify a password to be encrypted. When the `enable password` command is later entered, it has to take into consideration the possibility that password-encryption was previously requested, and hence the password must be encrypted. With the non-incremental approach the `enable password` command would always be executed first, and hence such conditional checks would not be necessary. There are other more complex examples where the benefits of non-incremental configuration become more apparent.

Information Retrieval

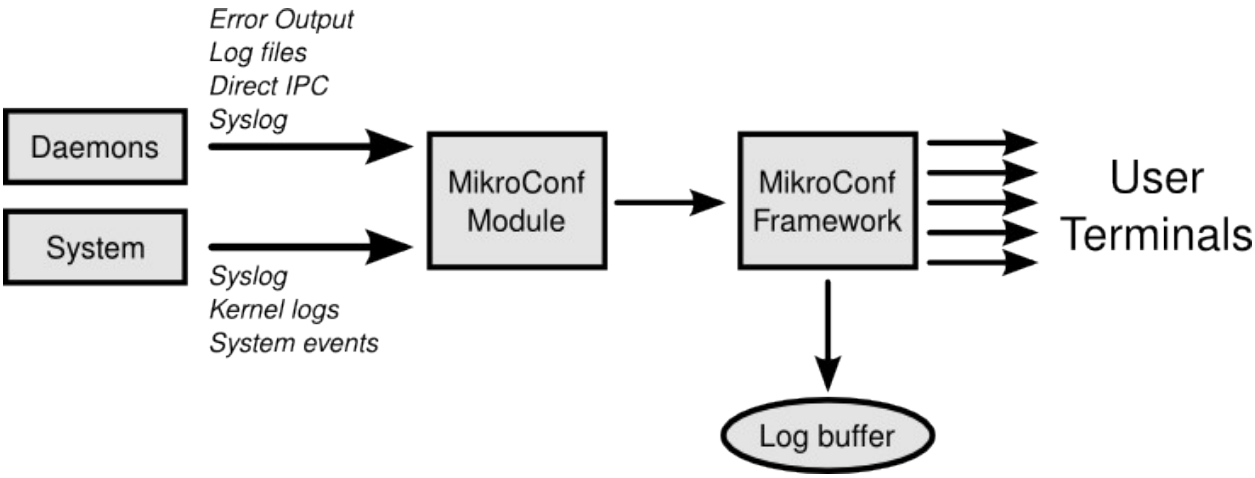
Most of the times the show commands will simply print verbatim the output of external programs. Other, less often used methods, are to parse configuration files, query the /sys or /proc kernel filesystems, or get the required information via a system-level API with a binary Tcl extension. Last, it is also possible to use SNMP GET where-ever applicable.

Except for the `show running-config` command, all other show commands should always reflect the actual configuration of the system, and not the perceived one.

Debugging Information

The `debug` command enables logging of extra debugging information, via the standard MikroConf logging mechanism. This information is most often used for troubleshooting purposes. Responsible for collecting the debug information is the module that owns the particular debug command executed. For instance, the `debug kernel` command owned by the *base*

module is responsible for collecting extra kernel debug information and sending them over to MikroConf's logging buffer. From there these messages will be sent to all active user sessions that have their logging threshold to *debug*. A module logs debug info via the standard MikroConf logging mechanism, which sends the logs to MikroConf's log buffer, as well as to the terminal of every active user sessions (granted that their log threshold permits it).



Drawing 5: Flow of debug information.

Getting extra debug messages, might require patching of external applications, or special compilation flags, or command line arguments. In any case, these messages can be sent from within the application to syslog and from there MikroConf will pull them, or alternatively they can be sent directly to MikroConf via a IPC mechansim, such as message queues.

Incidentally, MikroConf is using a message queue to pull the syslog messages directly from the syslogd busybox applet.

User-Interface Independence

Since MikroConf modules already go into the length of implementing configuration enforcement procedures for the CLAI, it is desirable to make these procedures accessible also to other possible front-ends, via a appropriate generic interfaces. Three such possible administration interfaces or front-ends are: Web-Gui, NMS, and SNMP. MikroConf is designed to allow all three of them to be layered on top of MikroConf's configuration enforcement code, so that much

of the work done for the CLAI can be reused for these interfaces as well.

One important design feature of MikroConf that enables this abstraction, is the separation of the code that handles the user interaction (which is specific to the interaction paradigm), from the code that enforces the actual configuration directives. Each MikroConf command has a corresponding command handler. Command handlers do not receive the words that make-up the command-line one-by-one and in the order they appear. Instead the command-line is broken down into its components, these components are then recognized and passed-on to the handler in the form of a dictionary. The handler can then check the dictionary for the desired information in a convenient manner, without the need to do any parsing of its own. This makes possible to make syntactic changes to a command, without updating the respective command handler, as long as the dictionary remains the same.

An RPC mechanism can be used to export the command handler functionality to remote processes, that follow different interaction paradigms.

Process Architecture

In this section we outline the architecture of MikroConf on a process level. In the following discussion we assume that the reader is familiar with many of the terms used in the context of Unix-like Operating Systems, such as: Process, Daemon, Shell, login process, Interprocess Communication (IPC), Unix Domain Socket, and FIFO (or named pipe).

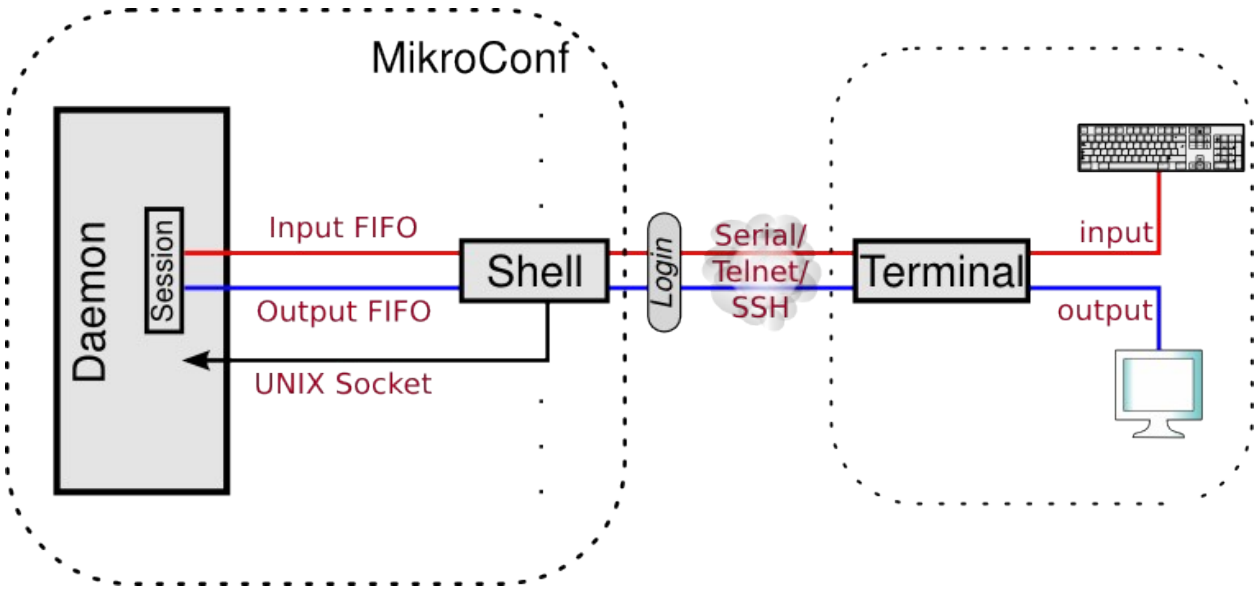
MikroConf, on a process level, is divided on the *MikroConf Daemon* (or simply the Daemon) and the *MikroConf Shell* (or simply the Shell). At any given moment, only one MikroConf Daemon can be running, while multiple MikroConf Shells can connect to it.

The MikroConf Shell is a normal shell like bash or ash, only that it gives the MikroConf CLAI to the connected user. It is spawned by login after a successful authentication over the network (telnet, ssh, ...) or a local (serial, ...) connection.

When a MikroConf Shell first starts, it connects to the MikroConf Daemon via a Unix Domain Socket, in which the Daemon is constantly listening at. Via this connection it requests

for a new session to be allocated to handle the incoming connection. If the Daemon grands the request, it sends back an acknowledgment (with some additional information) via the same socket connection.

The socket connection is maintained throughout the existence of the session. If it breaks for any reason, then both parties know that the session has been terminated, and the Daemon will clean-up any session information, whereas the Shell will simply exit.



Drawing 6: Daemon - Shell Architecture.

After the Daemon sends back the acknowledgment, it creates two FIFOs to handle input and output from/to the Shell. The Shell redirects its standard input (stdin) to the Input FIFO, the Daemon reads the input from there and processes it. The Daemon writes any session-specific output to the Output FIFO, from which the Shell reads it and redirects it to its standard output (stdout). This way we have a continuous stream-oriented connection from the remote user to the Daemon, for both input and output.

It might sound complex, but in actuality this is a very simple and elegant design, involving simple and standard IPC mechanisms. Most importantly it makes MikroConf connection-protocol unaware. MikroCon doesn't need to “speak” the various communication protocols that can be used to connect remotely to a device. It will work with any remote shell protocol, so long as there is a server implementation for it in the firmware.

Thread Architecture

In the previous paragraph we outlined the architecture of MikroConf on a process level. Now we will see how a MikroConf Daemon process is organized internally in threads. In this discussion we assume that the reader knows the non-exclusive list of terms: Event loop, Threads, Multi-threading, Synchronous and Asynchronous message passing, and Unix pipes.

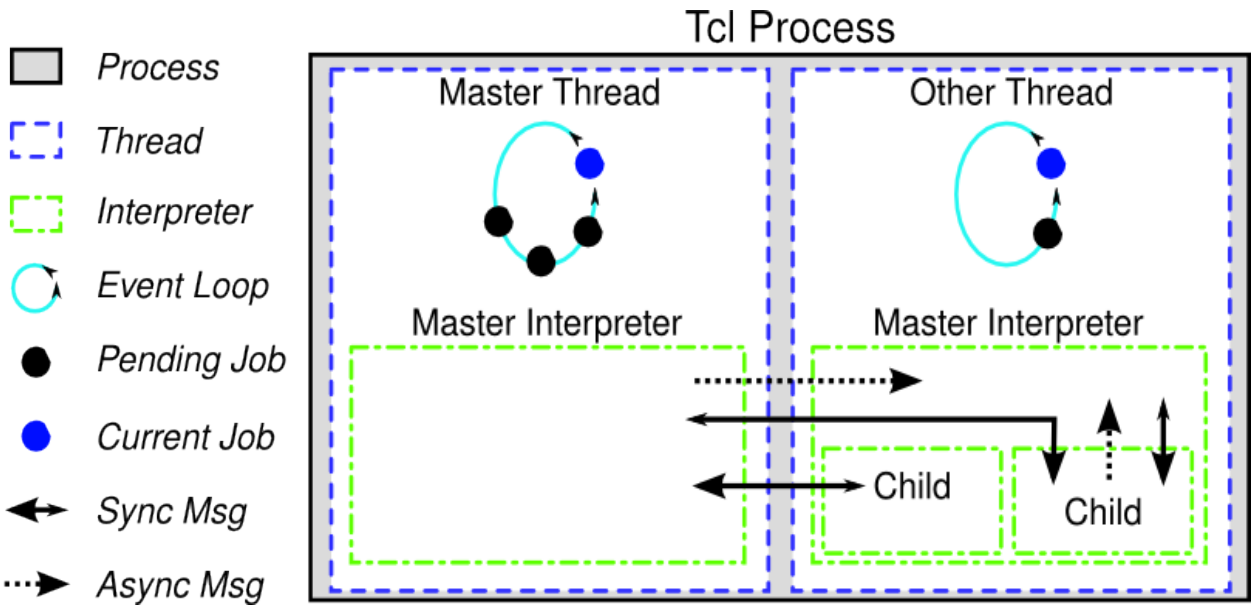
Processes, Threads, Interpreters

MikroConf is a multi-threaded application, and it builds heavily on the threading infrastructure provided by Tcl. Therefore, before we go ahead and talk about MikroConf's internal architecture, we need to summarize Tcl's threading model. For a more in-depth discussion of Tcl's threading model, please consult external resources.

A Tcl application can initialize and use many Tcl interpreters at the same time, organized in a tree hierarchy. They can communicate by message passing, either synchronous (returned value is waited upon) or asynchronous (returned value is not waited for). Initially, there is only one interpreter, the Master interpreter, which is the root of the tree and can create other child interpreters as needed.

A multi-threaded Tcl application has many threads, each one of them hosting a interpreter tree. Threads can also communicate with each other via synchronous and asynchronous message passing. Such a message has as source an interpreter in one thread, and as target the Master interpreter of another thread. A multi-threaded Tcl application initially starts with one thread, the Master thread, that can later create other threads. The thread organization is flat, not tree-like as the interpreters.

Each Tcl thread has a single event loop which is shared among all hosting interpreters. Any interpreter can send a job to be executed on the event loop of its thread, or to that of other threads. If the job is sent synchronously, then the sender waits for the job to be completed before execution resumes in that interpreter. If it is sent asynchronously, then the execution continues immediately, and result is not waited for. At any given moment (assuming that the event loop is not empty), there is a job being executed, while the remaining ones are waiting for their turn.



Drawing 7: Tcl's threading model.

Master, Session, and Slave

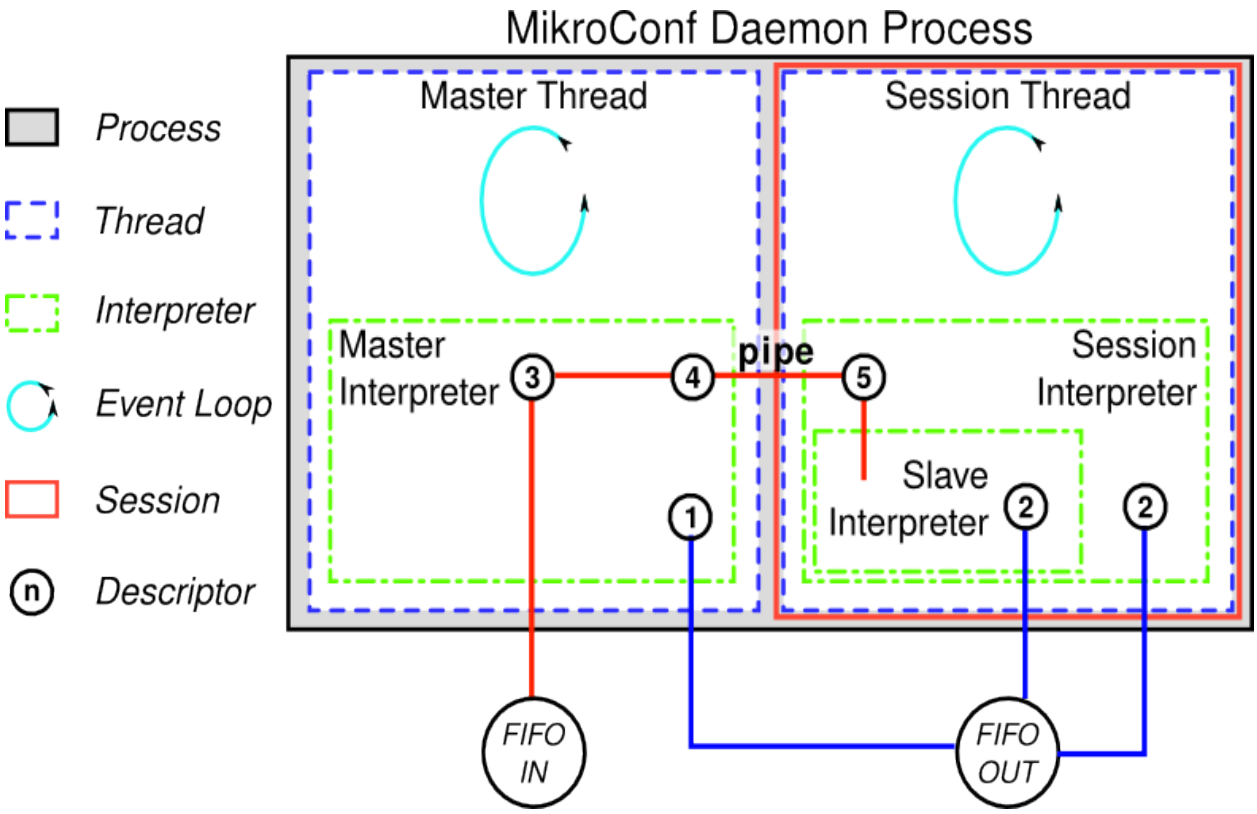
As aforementioned, MikroConf relies heavily on Tcl's threading model and interpreter architecture. We start with a *Master Thread* that hosts the *Master Interpreter*. These two alone implement the MikroConf framework.

For every incoming user connection, a new Tcl thread is created to handle it. This per-session thread is called *Session Thread*, and its initial (or master) interpreter is called *Session Interpreter*.

A Session interpreter handles user interaction for its session, among other session-specific things. Each MikroConf Session is identified by a unique *Session ID*, which is essentially the thread id that Tcl returns for the respective Session thread.

Each Session interpreter creates a child interpreter, which we call *Slave Interpreter*. User commands and scripts are safely evaluated within this Slave interpreter.

The next Figure visualizes the information flow. User input that comes through the Input FIFO is read by the Master interpreter and passed on to the respective Session interpreter with the use of a Unix pipe. The Slave interpreter can only access the input stream indirectly via the Session interpreter.



Drawing 8: MikroConf's threading model.

As far as the output stream is concerned, both Master and Session threads maintain a direct connection with the Output FIFO. The Session interpreter shares the file descriptor for the Output FiFO with the Slave interpreter.

Next we give the rationale for the most important aspects of this design, in a FAQ fashion:

Why we need a per session thread

We need a separate Tcl thread for each user session, because we need a separate event loop for each session. The reason we need a separate event loop is that we don't want user actions in one session to block the whole MikroConf application (including other parallel user sessions). This would be the case if we had a single thread (and therefore event loop) for the whole MikroConf application.

For example, with the current design this simple endless loop executed in a user session does not block the MikroConf framework itself or any other parallel user sessions, even though it blocks the Session thread it is executed in (or better say, it puts it in runaway state):

```
Router> while {1} {puts “endless”}
```

Why we need a Slave interpreter for each session

We need the Slave interpreter for two reasons:

1. To enforce restrictive security policies on what Tcl code the user can execute on the command line. The Session interpreter cannot be used for this purpose because it needs full Tcl access, as it handles user interaction and other low-level stuff.
2. To hold any user defined procedures, variables and any other session state created by the user. We don't want to contaminate the Session interpreter with user defined state.

Therefore, command line is evaluated in Slave, where security policy is enforced and user state is isolated. User interaction and other stuff are handled by Session.

Why input stream has to go through the Master interpreter

First of all we need to remind the reader that it does not make sense to open a FIFO for read by many threads, as it becomes non-deterministic which thread receives what part of the input. However, a FIFO opened for write by many threads is ok, since all written data will eventually end-up on the other end of the FIFO, albeit possibly mixed-up.

Having the above in mind, the reason why we needed the input to go first via the Master interpreter is because we want the Master to intercept special keystrokes and enforce them in his execution context. The primary keystroke that is intercept is the break key sequence, which breaks any executed command. If the user prompt is blocked due to evaluation of a user script, Master is the only interpreter/thread that can interrupt the evaluation of the script prematurely. The same applies for other bocking cases.

Why we need to have two direct connection to Output FIFO

We need one connection from the Master in order to print messages and logs asynchronously. And we also need one from the Session, obviously for allowing the user and the Session interpreter to print messages to the terminal.

Background Sessions

One special type of session in MikroConf is the *Background Session*. This is like a user session but without any user connected to it. Instead of being user-driven, it is event-driven. MikroConf uses backgrounds sessions to evaluate event-handlers and to apply the startup configuration on boot.

At any given time there is one background session active, that is used to evaluate event handlers. This background session is called *Events Session*.

Security Policy

As aforementioned, a Tcl *Security Policy* is enforced within the Slave interpreter of each session. A security policy defines what Tcl commands are accessible to the user of the session, and with what functionality each. The policy depends on the authentication mode the user is currently in. More privileged modes have access to more Tcl commands and of less-constrained functionality.

We define three security policies:

- **Background**: This is for background sessions.
- **Privileged**: This is for user sessions in privileged mode.
- **Default**: This is for user sessions in user mode, and for any other session type for which a security policy is not explicitly defined. This is the most constrained polity.

For each security policy we make some changes on the Slave interpreter, which constitute an offset from Tcl's Safe Interpreter Base. The Tcl Safe Interpreter Base is a security policy build-in to Tcl that is considered (by the Tcl developers) safe for most applications. As of Tcl 8.5.1 it is defined as depicted in the next table. Of course it is subject to change in future versions of Tcl.

Command state	Build-in Tcl commands or procedures
hidden	file socket open unload pwd glob exec encoding fconfigure load source exit cd
visible	tell subst eof list pid time eval lassign lrange fblocked lsearch gets case lappend proc break variable llength return linsert error catch clock info split array if concat join lreplace fcopy global switch update close for append lreverse format read package set binary namespace scan apply trace seek while chan flush after vwait dict continue uplevel foreach lset rename fileevent regexp lrepeat upvar expr unset regsub interp puts incr lindex lsort string
visible alias	clock

Table 3: The Tcl Safe Interpreter Base.

The actions that we take to modify Tcl's safe security base are:

- **Hide**: We hide additional Tcl commands.
- **Expose**: We expose previously hidden Tcl commands.
- **Alias**: We redefine the functionality of a Tcl command, usually by adding constraints in it or special functionality.

Policy	Action	Commands/Procedures
Background	hide	clock chan interp rename package
	expose	
	alias	unknown proc gets read fileevent fconfigure flush
Privileged	hide	clock chan interp package
	expose	cd exec file glob load open pwd socket
	alias	rename proc unknown gets read fileevent fconfigure flush
User & Default	hide	clock chan interp package
	expose	
	alias	rename proc unknown gets read fileevent fconfigure

Table 4: MikroConf's Security Policies.

You will find the rationale for these decisions documented inside MikroConf's sources (policy.tcl).

Note that we disallow renaming, deleting, or overwriting build-in Tcl commands. These actions are only possible for user-defined commands. User-defined commands must start with a

capital Latin letter, underscore or number. This is to avoid name collisions with MikroConf commands, which could cause a lot of trouble to the administrator. This could also have security implications (e.g. install malicious versions of MikroConf commands).

Logs

MikroConf has a central logging mechanism, which aggregates logs from many sources, and handles, stores, and reports them centrally. Namely the event sources are:

- Syslog, the GNU/Linux system log facility. (From syslog we also get kernel log messages that are sent by klogd)
- Output from external applications (like daemons).
- Messages logged in external log files, belonging to other applications.
- Messages logged by the user from on the command line.
- Messages generated by MikroConf itself.

Each log message is associated with the usual syslog properties: the time it was arrived, a severity level, a facility, and a process name. Except for the messages that were pulled from syslog, all the other messages are reflected to syslog as well. This is done so that we can use busybox's syslogd applet to send over the messages to a remote syslog server.

Log entries are stored in a memory mapped file which resides on the file system. The memory mapped region is written in a cyclic manner: when writes reach the end of the region, they continue from the start of it. These two properties (regular file & cyclic writes) gives us a persistent cyclic log buffer. Such a buffer offers two advantages:

1. The logs are persistent across reboots. If a MikroConf router crashes unexpectedly, then after boot we can still find the error messages associated with the crash.
2. The log buffer always holds the newest/latest messages. Therefore, we don't have to deal with log rotation manually.

Of course, for the log buffer to be persistent, the corresponding log file must reside in a persistent file system. If this is not the case then the firmware vendor must write an event handler

for the LOG_BUFFER UPDATE tag/event, that is generated when the log buffer changes. This event is rate limited. The event handler should copy the log file to a safe place, in a persistent file system. The firmware vendor must also create an event handler for the SERVER BOOT tag/event, to restore the log file from the safe place, back to its original place where MikroConf expects to find it. This event is generated on boot, before the log buffer is loaded.

Events

MikroConf's event architecture allows for the following:

- A style of aspect oriented programming which makes the source code more understandable and clear.
- Easy and straightforward customization from firmware developers, with no need for knowledge of MikroConf's internals.
- Facilitates the implementation of MikroConf's event-based configuration paradigm.

An *Event* is a combination of two strings: a *Tag* and event *Name*. It can be emitted from anywhere within the Framework or MikroConf Modules.

A generated event is received by the *Event Dispatcher*, which looks for any registered *Event Handlers* for this tag/name combination. If handlers are found, they are executed in the order they were registered. The dispatcher also passes on to each handler any parameters provided at the moment of generation. Multiple event handlers can be associated with a single tag/name combination.

Am Event ID uniquely identifies the exact binding, which can be later used to unregister the event handler.

Event handlers can be executed synchronously or asynchronously. In the former case they are executed **right after** the generated event, whereas in the latter, it is executed when the event loop becomes idle.

Blocking

An important requirement in MikroConf is that the user should be able to break from any executed command by pressing the break key sequence. This requirement affected MikroConf's design a lot, since we have to look at every place where blocking can happen, and figure out a way to interrupt it at user's request. All the possible places where blocking can occur are:

While executing an external program

Sometimes an external program can be long-running. Take for example a VI session. Those programs should be executed with the *ptyexec* command that can be interrupted at any time.

While evaluating a Tcl user script on Slave

As we have seen, the Slave interpreter is where user scripts are evaluated, and during evaluation the prompt is not available. We distinguish two cases of here:

1. A user script runs indefinitely, but without being blocked at any command (that is, it is in a runaway state.
2. A user script has blocked at some command waiting for some something (e.g. some input from a **blocking** channel).

None of these cases are addressed at the moment. This is an area that needs further investigation.

While waiting for user input from the terminal

We provide a non-blocking version of gets, called sgets, that does not block the event loop of the Session, when it is used to receive input from stdin. It works by emulating gets on a non-blocking channel, as it is the case for the input stream of a session. Therefore, when a command handler prompts the user for some input using the sgets procedure, the user can break out of it.

While executing the body of a command handler

There are times where some Tcl code can block, such as when waiting for a network event to occur. For these cases where blocking is a possibility, we provide a solution in the form of two commands: `timeout` and `blocks`. The former will interrupt execution of its body after a

predefined number of seconds (it uses the SIGALARM signal to wake up), whereas the latter will execute its body in a different Tcl thread. The two commands can be used in conjunction, in which case the `timeout` command should be in the body of the `break` command, and not the other way around.

URL Types

Certain MikroConf commands accept a special URL type argument, which enables access to local and remote locations. For example the `copy` command can be used to copy the running-config from a remote/local location and then apply it.

```
Wrap# copy ftp://ftp.example.com/pub/running.config running-config
```

To provide such functionality we make use of Tcl's Virtual File Systems mechanism. We use urltype Tcl VFS to create mappings for URL types to local directories, as well as Tcl VFSs which export specific network protocols.

The following URL types are currently supported:

URL Type	Location/Protocol
temp://	Maps to local directory: /tmp
flash://	Maps to local directory: /mnt/flash
system://	Maps to local directory: /
nvram://	Maps to local directory: /tmp/nvram
bootflash://	Maps to local directory: /mnt/boot
slot0://	Maps to local directory: /mnt/pcmcia0
slot1://	Maps to local directory: /mnt/pcmcia1
ftp://	Maps to the FTP protocol.
http://	Maps to the HTTP protocol.
webdav://	Maps to the WebDav protocol.
null://	Anything copied here is goes to the bit bucket.

Table 5: Tcl Virtual Filesystems.

More URL Types will become available in the future, such as nfs:// cfs:// shttp:// tftp:// scp://

Another example of MikroConf command that uses URL types is `dir`:

```
Router> en
Router# dir temp://
Directory listing for temp://*
Perm  Bytes       Access      Modified    Name
drwe  4096        2008-05-22  2008-05-22  1/
drwe  4096        2008-05-22  2008-05-22  2/
-rw   5563        2008-05-22  2008-05-22  3
Router#
```

The Specs File Structure

The *Specs file* is an XML file that adheres to the following DTD specification:

```
<?xml version="1.0" encoding="UTF-8"?>

<!ENTITY end SYSTEM "xml/end.xml">
<!ENTITY exit SYSTEM "xml/exit.xml">

<!ELEMENT module (confmode*, authmode*, mainconf*, tree+)>
    <!ATTLIST module name NMTOKEN #REQUIRED>

<!ELEMENT confmode EMPTY>
    <!ATTLIST confmode name NMTOKEN #REQUIRED
                                        string CDATA #REQUIRED
                                        parent NMTOKEN #REQUIRED
    >
<!ELEMENT authmode EMPTY>
    <!ATTLIST authmode name NMTOKEN #REQUIRED
                                        string CDATA #REQUIRED
                                        parent NMTOKEN #REQUIRED
                                        privilege NMTOKEN #REQUIRED
    >
<!ELEMENT mainconf EMPTY>
    <!ATTLIST mainconf name NMTOKEN #REQUIRED
                                        parent NMTOKEN #REQUIRED
                                        order NMTOKEN #REQUIRED
    >

<!ELEMENT tree (path | command)+>
    <!ATTLIST tree conf NMTOKENS #REQUIRED
                            priv NMTOKEN "0"
    >

<!ELEMENT path (desc?, no?, (path|command)*)>
    <!ATTLIST path name NMTOKEN #REQUIRED>

<!ELEMENT command (syntax?, desc, exec, print*, no?, argument*, grammar?)>
    <!ATTLIST command name NMTOKEN #REQUIRED>

<!ELEMENT syntax (#PCDATA)>
<!ELEMENT desc (#PCDATA)>
<!ELEMENT nodesc (#PCDATA)>
<!ELEMENT exec (#PCDATA)>
<!ELEMENT print (#PCDATA)>
    <!ATTLIST print section NMTOKEN #REQUIRED>
```

```
    <!ATTLIST print order NMTOKEN #REQUIRED>

<!ELEMENT no (syntax?, desc, argument*, grammar?)>

<!ELEMENT argument (desc, nodesc?, type)>
    <!ATTLIST argument name NMTOKEN #REQUIRED>
<!ELEMENT type (#PCDATA)>
    <!ATTLIST type name (exact|list|dlist|fixed|any) #REQUIRED>
```

Table 6: The DTD document that describes the structure of a Specs file.

In the following paragraphs we describe one-by-one the use of these XML entities.

Authentication Modes

An authentication mode (authmode) is declared with the XML element `authmode`. No two MikroConf modules should declare the same authentication mode. Although any module can use modes declared by other modules. Authentication modes form a tree hierarchy, with the user necessary to be in the parent authmode in order to switch to the child.

The `authmode` element accepts the following XML attributes:

- `name`: The string that uniquely identifies the new authentication mode.
- `string`: The string to be displayed on the command line when user is in this mode.
- `parent`: The name of the parent authentication mode.
- `privilege`: The Privilege Level the new mode will be associated with. It must be in the range of 0-15, where 15 is the maximum possible authority status.

For example, the two build-in authentication modes User and Privileged are defined with the following two XML elements:

```
<authmode name="user" string=">" parent="root" privilege="0"/>
<authmode name="priv" string="#" parent="user" privilege="15"/>
```

As you can see both User and Privileged authmodes have as parent the special "root" authmode. This is not really an authmode, but represents the root of the tree.

Configuration Modes

Configuration modes (confmode) are defined into two steps. In the first step, they must be

declared in a similar manner to authentication modes as we saw above. Like authentication modes, they have `name`, `parent`, and `string` XML attributes with same function. Just like authentication modes, a configuration mode is always declared by a single MikroConf module, which is called the *owner* of the confmode. Other modules use it freely as long as the owner module is loaded.

For instance, the declarations for the Entry and Global Configuration modes are:

```
<confmode name="entry" string="" parent="root"/>
<confmode name="global" string="(config)" parent="entry"/>
```

The second step is to create *Command Trees* for the previously declared configuration mode. A Command Tree appears in the Specs file with the `tree` XML element, and it can host any number of `path` and `command` elements. We will discuss these elements on the next section.

For the two aforementioned confmodes there are three Command Trees specified in *base* module:

```
<tree conf="entry" priv="0">
  ...
</tree>

<tree conf="entry" priv="15">
  ...
</tree>

<tree conf="global" priv="15">
  ...
</tree>
```

As you can see, the `conf` attribute associates a command tree, with a previously declared configuration mode. The `priv` attribute, on the other hand, says that all the commands (and paths) contained within this tree are accessible to uses that have the specified privilege level or higher.

You might have also noticed that a configuration mode can be spitted in many Command Trees which can exist in the same Spec file. This is handy to categorize commands based on the required privilege level. It is also possible for Command Trees to exist in different Specs files (belonging to different modules) and collectively define a single configuration mode. This design

allows for instance the *rip* module to register commands that appear in the *interface* configuration mode (which is owned by the *interface* module).

Beware that before a configuration mode is defined by a module, no other module can register Command Trees for it.

Running Configuration Sections

The internal representation of the running-config is a tree structure, that comprises of *Sections* and *Entries*. A running-config section is declared with the XML element:

```
<mainconf name="..." parent="..." order="..."/>
```

For example, *base* module defines the followings:

```
<mainconf name="register" parent="root" order="0"/>
<mainconf name="auth" parent="root" order="10"/>
<mainconf name="terminal" parent="root" order="20"/>
<mainconf name="line" parent="root" order="40"/>
    <mainconf name="console" parent="line" order="10"/>
```

The name attribute is the name of the new section; parent is the name of the parent section; and order is the *Order Number* of the new section. Obviously, no two MikroConf modules should declare the same running-config section. Although any module can use sections owned by other modules.

Paths, Commands and Arguments

Commands are declared in the module's *Specs file*, which is an XML file. Commands can be added or removed at any time during a module's lifetime.

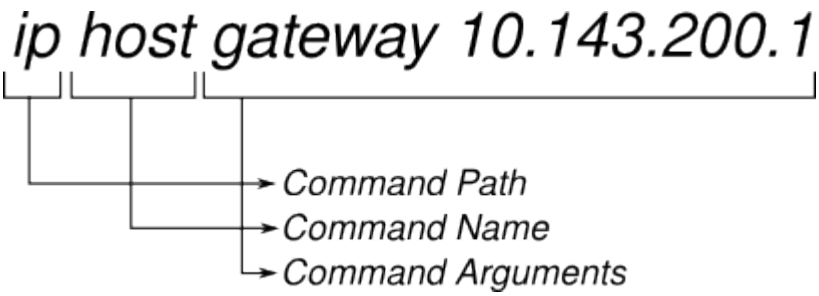
Any valid MikroConf command can be broken down to the following parts:

- The command path
- The command name
- Command arguments

Take for example the following three commands:

```
ip host gateway 10.143.200.1
ip domain-name aspisos.org
ip name-server 10.143.200.1
```

They all share the same path, which is `ip`. They have different command names: `host`, `domain-name`, and `name-server`. And, of course, they support different arguments, which is everything that follows after the command name.



Drawing 9: Break-down of an example command, in its constituents.

The definition of the ip path in the above example could be:

```
<path name="ip">
   <desc>IP Configuration</desc>

   <command name=”host”> ... </command>
   <command name=”domain-name”> ... </command>
   <command name=”name-server”> ... </command>

</path>
```

A `command` XML element defines the entry-point for a MikroConf command. It accepts a `name` attribute which is the word you need to enter on the command-line to access the command. It also accepts a number of child-elements, which we discuss hereafter:

<syntax>

This element is optional. If present, its body describes how the command arguments can appear on the command line. In other words, it defines the *Command Syntax*. Notice, that the declaration of the arguments themselves is done by another element, the argument element, which we discuss later. If this element is absent, then the command will not accept any arguments at all; which is of course a valid option.

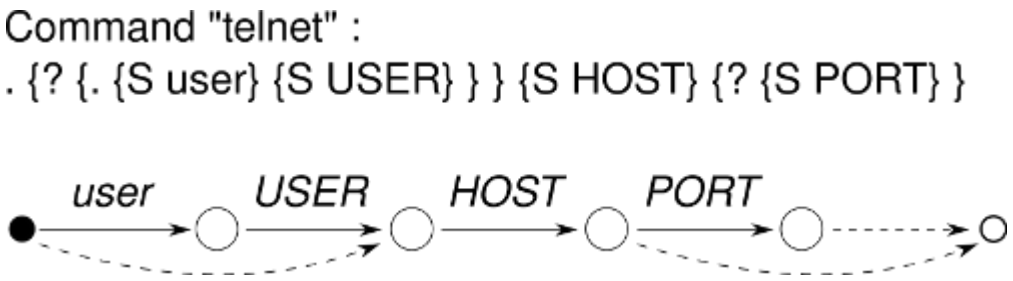
The syntax is described by a regular expression, which internally is processed as a deterministic finite automaton. The states of the automaton correspond to the argument positions on the command line. The arcs from each state are the arguments that are valid for the corresponding position. The command's arguments are the elementary blocks of the regular expression (the symbols accepted by the automaton). This design gives maximum flexibility and control over how arguments can appear on the command line.

The regular expression in the body of this element is represented by a nested Tcl list, which forms a syntax tree. The following structures are legal to appear in the regular expression:

Expression	Description
{S x}	Atomic regular expression. Everything else is constructed from these. "x" is the name of an argument of the command, as declared with the `argument` element.
{. A1 A2 ...}	Concatenation operator. Accepts the concatenation of the regular expressions A1, A2, etc. Note that this operator accepts zero or more arguments. With zero arguments the represented language is epsilon, the empty word.
A1 A2 ...}	Choice operator, also called "Alternative". Accepts all input accepted by at least one of the regular expressions A1, A2, etc. In other words, the union of A1, A2. Note that this operator accepts zero or more arguments. With zero arguments the represented language is the empty language, the language without words.
{& A1 A2 ...}	Intersection operator, logical and. Accepts all input accepted which is accepted by all of the regular expressions A1, A2, etc. In other words, the intersection of A1, A2.
{? A}	Optionality operator. Accepts the empty word and anything from the regular expression A.
{* A}	Kleene closure. Accepts the empty word and any finite concatenation of words accepted by the regular expression A.
{+ A}	Positive Kleene closure. Accepts any finite concatenation of words accepted by the regular expression A, but not the empty word.
{! A}	Complement operator. Accepts any word not accepted by the regular expression A. Note that the complement depends on the set of symbol the result should run over. See the discussion of the argument over before.

Table 7: Regular expression syntax rules.

The following picture depicts the deterministic automaton for the syntax of the `telnet` command.



Drawing 10: Break-down of an example command, in its constituents.

<desc>

This element is mandatory. Its body is the help text that appears when the user presses the question mark. It is advised to be less than 80 characters long.

<exec>

This element is also mandatory. Its body is the name of the command handler. If it is not given with an absolute namespace path, then the handler is looked up in the private namespace of the module.

Moreover, if additional arguments are provided in excess of the handler name, then these arguments are passed verbatim as the first arguments of the command handler, (before the standard API arguments).

<print>

```
<print section="..." order="...">
```

This element is optional, but it can also appear multiple times. With every instance it registers a callback procedure to be executed when generating the running-config. In other words it registers a running-config entry.

The `section` attribute specifies the running-config section this entry should attach to, and the `order` attribute specifies the *Order Number* for the entry. If an entry with the same order number already exists, then the order of execution of the two entries is unpredictable. The section a `print` element refers to must be previously defined with a `mainconf` element.

<no>

This element is optional. If present, it enables the negative form of the command. It can have its own `syntax`, `desc` and `argument` child elements, which have the same function as those for the command element.

When syntax is present, it defines the syntax for the negative form of the command, which can be different from that of the positive form. The element `syntax` can involve arguments declared either under the `command` or `no` elements. If an argument appears with the same name under both `command` and `no`, then the one under `no` takes precedence.

Finally, the element `desc` specifies the help text for the negative form.

<argument>

```
<argument name="...">
```

Every instance of this element registers an argument for the command. There should be exactly one instance for every distinct argument name that appears as symbol inside the `syntax` element. The symbol name used in `syntax` and the `name` XML argument of this element must match.

The `argument` element must have as child elements one `desc` and one `type`. Optionally it can also have one `nodesc` element. The `desc` has exactly the same function as we have seen so far, only that this time is for the argument. The `nodesc`, if present, specifies the help text for the argument when the negative form of the command is used. If not present then help text for the positive version is printed in both positive and negative forms.

Finally, the `type` element specifies the type of the argument. For instance, it can specify if the argument accepts one of a finite number of fixed strings, or an arbitrary string. It has the form:

```
<type name="..."> ... </type>
```

The `name` XML attribute can be set to one of the literals: `exact`, `list`, `dlist`, `fixed`, and `any`. Depending on what it is set to, the body of the element takes different meaning. We

will see each one of the cases separately:

```
<type name="exact"/>
```

In this form, the argument accepts only one word, which is the string specified in the `name` XML argument of the `argument` element. Command Completion and Abbreviation will consider this word as an option, and In-line Help will display it as well.

An example use of this argument type is the command:

```
show running-config [all]
```

`show` is a path, and `running-config` is the command name, and `all` is a keyword that can be implemented as an exact type argument.

```
<type name="list">item1 item2 item3 ...</type>
```

The argument accepts a finite number of words that are specified as a Tcl list on the body of this element. Command Completion and Abbreviation will consider these words as alternatives, whereas In-line Help will only list the name of the `argument` element and its associated help text.

An example use of this argument type is the command:

```
log {debug | info | ... | critical | alert} message
```

The alternatives of the second argument are known in advanced, therefore we can implement them with an argument type of `list`.

```
<type name="dlist">tcl_procedure_name</type>
```

This is similar to the `list` type above, but with the difference that the list of strings is actually generated dynamically at run-time. It is done so by calling the `tcl_procedure_name` procedure, which is expected to return a Tcl list. If `tcl_procedure_name` is not in an absolute namespace path, then it is looked for in the private namespace of the module which owns the command.

As with the `list` type, Command Completion and Abbreviation will consider all the dynamically generated words as alternatives, whereas In-line Help will only list the name of the

`argument` element and its associated help text.

An example use of this argument type is the command:

```
interface {name0 | name1 | ...}
```

We don't know in advance all the available interface names. We can compute them, however, at run-time when the command is being edited or executed. The `dlist` argument type is the solution to this.

```
<type name="fixed">
   word1 {help text for word1}
   word2 {help text for word2}
   ...
</type>
```

This argument type explicitly lists all the words that the argument accepts, along with their help message texts. The body of the element must be a valid Tcl serialized array. Command Completion will consider the words as valid alternatives, whereas In-line Help will display them along with the corresponding help messages.

This argument type exists solely to simplify the command syntax. It is equivalent to using multiple `exact` argument types, set as alternatives in the command syntax. However, this will enlarge the automaton that corresponds to the command syntax (one state per `exact` argument), whereas with the `fixed` argument type we have a single automaton state for all keywords.

```
<type name="any"/>
```

This argument type accepts any string as valid word. It is used for user supplied values that cannot be confined within a finite number of alternatives, such as for the case of an email address.

The syntax of the command should not allow two different arguments of type `any` to compete for the same argument position, because in this case it is indecisive to which argument the word should map to. Nevertheless, it is ok to combine argument types with finite number of alternatives, with a single argument type of `any`.

Beware that multiple types for the same argument are not supported. You will need to define

different arguments of different types to accomplish this.

Conclusions

In this paper we presented MikroConf, a framework for building Command-Line Interfaces (CLI), and the application of it in the administration of Linux-based network devices. We covered MikroConf's design and implementation aspects the pertain to Tcl.

References

1. MikroConf Router Configuration Interface, http://mikroconf.sourceforge.net/
2. The Tcl Programming Language, http://www.tcl.tk/
3. Small Tcl, http://wiki.tcl.tk/1363
4. Tcl Modules, http://www.tcl.tk/cgi-bin/tct/tip/189
5. Tcl Script Cancelation, http://www.tcl.tk/cgi-bin/tct/tip/285.html
6. tcldoc Documentation Generator, http://tcl.jtang.org/tcldoc/tcldoc/tcldoc.html
7. taccle Parser, http://wiki.tcl.tk/11425
8. fickle Scanner, http://wiki.tcl.tk/3555
9. Tclx, http://tclx.sourceforge.net/
10. Memchan, http://memchan.sourceforge.net/
11. tcllib, http://tcllib.sourceforge.net/
12. TclTest, http://www.tcl.tk/man/tcl/TclCmd/tcltest.htm
13. TCL Modules, http://www.tcl.tk/cgi-bin/tct/tip/189
14. Tcl Binding to Tcl Virtual File System and urltype,
15. TclVfs, http://sourceforge.net/projects/tclvfs
16. Tcl Virtual File System, http://wiki.tcl.tk/2138

17. Tcl SNMP Tools, https://sourceforge.net/projects/tcl-snmptools

18. Tcl Inotify, https://sourceforge.net/projects/tcl-inotify

19. Tcl POSIX Message Queues, https://sourceforge.net/projects/tcl-mq

20. Tcl Syslog, https://sourceforge.net/projects/tcl-syslog

21. Tcl PAM Authentication, https://sourceforge.net/projects/tcl-pam

22. Tcl mmap Interface, https://sourceforge.net/projects/tcl-mmap

23. The uClibc Project, http://www.uclibc.org/

24. The uClibc++ Project, http://cxx.uclibc.org/

25. The busybox Project, http://busybox.net/

26. The linux-pam Project, http://www.kernel.org/pub/linux/libs/pam/

27. The tDOM Project, http://www.tdom.org/

28. Tcl's Build-in Memory Introspection, http://wiki.tcl.tk/3248, http://wiki.tcl.tk/3248

29. The Net-SNMP Project, http://www.net-snmp.org/

30. The GNU Project, http://www.gnu.org/

31. The Linux Kernel, http://www.linux.org/

32. The Expect Project, http://expect.nist.gov/

33. Elliotte Rusty Harold, W. Scott Means, “XML in a Nutshell, 3 Edition”, O'Reilly

34. Simon StLaurent, Michael Fitzgerald, “XML Pocket Reference”, O'Reilly

35. Bouglas R. Mauro, Kevin J. Scmidt, “Essential SNMP, 2 Edition”, O'Reilly

36. Tony Stubblebine, “Regular Expression Pocket Reference”, O'Reilly

37. John E. Hopcroft, Rajeev Motwani, Jefrey D. Ullman, “Introduction to Automata

38. Theory, Languages, and Computation, 3rd Edition”, Addison-Wesley

39. Flemming Nielson, Hanne Riis Nielson, Chris Hankin, “Principles of Program Analysis”,

Springer

40. W. Richard Stevens, “TCP/IP Illustrated, Volume 1: The Protocols”, Addison-Wesley
41. Ethan Cerami, “Web Services Essentials”, O'Reilly
42. The Etch Protocol, http://en.wikipedia.org/wiki/Etch_(protocol)
43. The SOAP Protocol, http://en.wikipedia.org/wiki/SOAP
44. The CORBA Protocol, http://en.wikipedia.org/wiki/CORBA