CURRICULUM VITAE

LAST UPDATE: 20TH APRIL 2017

Name	Lucio Andrés Illanes Albornoz
Email address	<lucio@lucioillanes.de>
Mobile telephone number	07923 213 316
Address	975 Leeds Road, Huddersfield, West Yorkshire, HD2 1UP, United Kingdom
Date of birth	13 August 1987
Nationalities	Chilean, German
GitHub	<https://git.io/vS4CG>
Website	<http://www.lucioillanes.de/>

ADDENDUM: PROJECT SPECIFICATIONS

February 2016-ongoing — **midipix_build: build/cross-compilation infrastructure for midipix, a POSIX/Linux-compatible development environment for Windows NT-based OS**

Midipix[1] is a POSIX/Linux-compatible environment for Windows NT-based OS, facilitating cross-compilation and execution of applications written for POSIX/Linux without suffering substantial performance loss. Unlike **Cygwin**, **Midipix** does not require interaction with the Windows environment subsystem in order to implement its system calls. Unlike **Interix**, **Midipix** is not an environment subsystem itself and does not introduce its own subsystem server and client DLL(s). Neither virtualisation nor kernel-mode drivers are required in order to use **Midipix**; instead, a small set of "runtime components" mediate communication between **Musl**, the libc chosen for this project, and the **Windows NT executive** (NTOSKRNL.EXE.)

At the time the **midipix_build** project was started, [cross-]compiling the toolchain, the runtime components, and any number of 3rd party packages was handled by a number of Bourne shell scripts. This ultimately proved inscalable and often unreliable. Hence, a sufficiently comprehensive build/cross-compilation infrastructure was required.

As **Midipix** has been and still remains a moving target, an iterative development model was chosen: architectural considerations were thus of considerably low importance. Code refactoring has taken place only two times, and the present design has proven to be sufficient for the purposes of **Midipix**. The following goals and requirements emerged as part of this process:

1. **Portability**: **midipix_build**, similarly to NetBSD's build.sh sans make(1), had and to some extent still has to support cross-compiling a large number of packages on any number of platforms that proves to be sufficiently POSIX-conforming. Thus, Bourne shell was chosen as implementation language, albeit a small number of non-POSIX features are leveraged, such as function-local variables. This permits building **Midipix** on Linux, **Cygwin**, natively (**Midipix**) itself, and most varieties of **BSD**.
2. **Flexibility**: the architecture of **midipix_build** has to be sufficiently simple and mutable in order to allow for the addition and/or removal of features at a reasonably fast pace, e.g. flavours, signed distribution tarballs, automated package version checks, etc. pp. Furthermore, even the mere addition of 3rd party packages may often necessitate introducing new or revisiting current generalisations concerning particularly **GNU autotools**- and **Cmake**-based build systems.
3. **Convenience and reliability**: **midipix_build** is usually how people are introduced and become acquainted with **Midipix**, even if only via the distribution tarballs it produces. Both developers inasmuch as mere users will want to be able to build or rebuild a single package, a set of packages, the runtime components, or everything. Therefore, virtually all common tasks are accomplished by entering a short command line invoking "build.sh" and without any other required user interaction.
4. **Simplicity**: last but not least, low code complexity and size, especially in relation to flexibility, remain a priority: presently, the code sans package-specific subroutines/variables amounts to about 1200 SLOC, whereas package-specific code covers approximately 1800 SLOC. Reusable subroutines, in the form of "build steps" (e.g. fetch, extract, build, install, …) and pre/post-steps (e.g. setup_env, prereqs and sha256sums, tarballs, resp.,) reside in their own source modules.

References:

[1] midipix
<http://midipix.org/>

PROJECT SPECIFICATIONS (SEE 1st PAGE)

November 2016-December 2016 **Tcl TIP #458: design and implementation of epoll/kqueue support in the Tcl notifier on Linux/*BSD, resp. (FlightAware bounty programme)**

Tcl (pronounced "tickle" or *tee cee ell*, /ˈtiː siː ɛl/) is a high-level, general-purpose, interpreted, dynamic programming language[1]. Contributions to **Tcl** that alter public interfaces are submitted and processed through the "**Tcl Improvement Proposal (TIP)"** mechanism[2]. In early November of 2016, a **bounty programme** "for improvements to **Tcl** and certain **Tcl** packages" was uploaded to GitHub[3] on part of **FlightAware**[4]. I chose to implement "[s]upport for epoll()/kqueue() to replace select() in socket handling," which was finished by the end of December 2016[5].

Tcl implements an **event-based architecture** concerning I/O and particularly employs callback for I/O completion notification and handling, both of which are implemented in the platform-specific "**notifier**."

Originally, the only notifier available employed the **select(2)** system call for I/O multiplexing, which proved to be a major hindrance concerning scalability. The drawbacks of **select(2)** are well-known and elaborated on in my **TIP**[5] and shall not be discussed here; however, reliance on **select(2)** also generally imposes a limit on file descriptors of 1024. On the other hand, the new notifiers for **Linux**, employing **epoll(7)**, and ***BSD**, employing **kqueue(2)**, do not suffer either of those defects.

Furthermore, the original **select(2)**-based notifier implemented event notification and inter-thread IPC via the **notifier thread**. As multithreading, if at all justified, by itself introduces additional complexity and can often lead to equally complex issues such as race conditions, priority inversions, etc. pp., the new notifier code calls **epoll(7)/kqueue(2)** from within the thread context of the caller without itself introducing any new threads. The inter-thread IPC problem, after some amount of discussion on the mailing list and IRC, has been solved by introducing a per-thread trigger **pipe(2)** on ***BSD** and **eventfd(2)** on **Linux**; while this adds one file descriptor per thread that interacts with the notifier and two/one file descriptors for inter-thread IPC on ***BSD/Linux**, resp., this is considered to be acceptable considering the substantial loss of code complexity due to the removal of the notifier thread. Most importantly, threads, as far as the notifier is concerned, do not implicitly share file descriptors or state with each other.

References:

[1] Tcl – Wikipedia
<https://en.wikipedia.org/wiki/Tcl>
[2] Tcl Improvement Proposal
<http://wiki.tcl.tk/983>
[3] GitHub - flightaware/Tcl-bounties: Bounty program for improvements to Tcl and certain Tcl packages
<https://github.com/flightaware/Tcl-bounties>
[4] FlightAware - Flug-Tracker / Flugstatus / Flugverfolgung
<https://www.flightaware.com/>
[5] [TIP] Intent to implement epoll()/kqueue() support for sockets on Linux/FreeBSD · Issue #14 · flightaware/Tcl-bounties · GitHub
<https://github.com/flightaware/Tcl-bounties/issues/14>

PROJECT SPECIFICATIONS (SEE 1st PAGE)

April 2017-ongoing

Arable: a configuration management, backup, cloud and dotfile synchronisation, host provisioning, monitoring, software deployment, and ad-hoc task execution tool that emphasizes architectural and code simplicity, a sound DSL, flexibility, and speed, intended to compete with Ansible, Chef, Puppet, and Salt (WORK IN PROGRESS)

Arable is, primarily, a file synchronisation and command/script execution tool that employs CFEngine's architecture whilst leveraging Rsync, SSH, Tcl, and the comparative simplicity and learning curve of Ansible. On the other hand, Ansible (as well as Chef and Puppet) suffers considerable drawbacks that were considered, along with source code inspection, to be categorically unacceptable:

1. **Inevitable performance, resource usage, and scalability issues stemming from questionable architectural choices**
   Ansible's task execution engine implements a laborious process of separately connecting to each node managed via SSH, uploading the set of Python scripts required along with data specific to the task(s) at hand, and lastly executing them on the nodes managed. "Batching" is in principle supported, most modules, particularly file transfer modules, however do not, at present, support this. While the task engine leverages OpenSSH's "ControlMaster" feature, this is still either insufficient or simply unnecessarily convoluted, as the most obvious and clearly superior alternative of establishing persistent, and as such, reusable, SSH connections is trivially implemented.
   This is exacerbated by Ansible's architecture which, whilst characterised by comparative simplicity, places emphasis on the client, as opposed to particularly CFEngine, which places emphasis on the nodes managed. Ansible's architecture, arguably "ad-hoc," on the surface appears to facilitate very fast adoption along with comparatively small to minimal maintenance costs. In reality, this architecture imposes insurmountable restrictions concerning both resource usage, scalability, and, depending on use cases, security.
   Nodes managed by CFEngine have all the node-specific information required at their disposal. On the other hand, Ansible has to obtain this information on a node-per-node basis prior to performing node management. Furthermore, Ansible's "one-to-many" architecture imposes, past a trivial amount of hosts and/or tasks, high to very high hardware requirements on the host running Ansible, none of which are reasonably justified as CFEngine's "many-to-one" (or "many-to-many") architecture introduces none of those problems: pragmatically speaking, Ansible, despite its apparent simplicity, involves an unreasonably large number of steps in executing a single task and thus categorically fails to attain to scalability.
   Lastly, nodes managed by Ansible cannot impose any restrictions or policies on what Ansible is permitted to do and prohibited from doing on the nodes in question. Thus, Ansible's architecture can be considered to be less secure than CFEngine's architecture or indeed, any "many-to-one/many" architecture.
2. **Ill-conceived Domain-Specific Languages lacking first-order constructs for fundamental language aspects**
   Ansible's choice of DSL appears to be predicated on the perceived notion of simplicity and as such, as mentioned above, intended to facilitate very fast adoption along with comparatively small to minimal maintenance costs. Unfortunately, this proves, upon closer inspection, to have the exact opposite effect: loop constructs as well as constructs operating on non-scalar input are not of the first order and appear to have been designed and implemented entirely in an iterative "ad-hoc" fashion. Whilst this may, at first, be entirely sufficient and present the appearance of simplicity, languages emerging, moreso than explicitly conceived of and designed, from such a process very quickly turn into inconsistent minefields facilitating with great ease the production of unmaintainable and unreliable code. This places Ansible's DSL into the same category as fundamentally flawed languages such as Perl or PHP, which in the context of configuration management, directly relating to downtime and related costs, is absolutely unacceptable.
   Lastly, the very assumption that a custom DSL is indeed needed is questionable. There is a sufficiently large amount of high-level languages, interpreted or not, that lend themselves to the derivation of a DSL and that, most importantly, have attained to maturity and are founded on solid architectures and design choices.

Thus, Arable employs CFEngine's "many-to-one" (supporting "many-to-many" as well for increased redundancy) architecture combined with persistent SSH connections, rsync as file synchronisation/transfer protocol, and Tcl as DSL. This affords Arable the ability to scale down- and upwards without the high resource usage restriction of Ansible, whilst retaining CFEngine's superior and mature architecture, yet without the arguably steep learning curve of CFEngine.
Lastly, Arable's simplicity, both in terms of design as well as implementation, allows it to assume a wide range of roles and cover a large amount of territory: backups, cloud synchronisation, configuration management, host provisioning, monitoring, software deployment, and, as Ansible, simple "ad-hoc" task execution, but without suffering any of Ansible's drawbacks.