When Tcl meets Docker

Dr. Emmanuel Frécon
SSE - Software and Systems Engineering Laboratory
SICS Swedish ICT
emmanuel@sics.se

Abstract
Container technologies and especially Docker are quickly revolutionising the way we architect, develop and operate large applications. This paper presents four interrelated tools integrating the Docker sphere with the Tcl world. Three different containers aim at serving as the base for Tcl applications. An implementation of the Docker API can aid when glueing containers together or when supervising or introspecting containers. Concocter is a watchdog and dynamic generic controller process for using in containers as the first process. Finally, Machinery integrates all Docker tools (Engine, Swarm, Compose, Machine) to (re)create entire distributed architectures in a flexible and deterministic manner.

1. Introduction

Container technologies are enjoying a renewed interest in the IT industry and a lot of this interest crystallises into Docker and its set of tools. Containers encapsulate all the elements of a server in microcosm – such as code, runtime, system tools and system libraries, but Docker also provides related tools to privately network these containers together, bundle, cluster and scale the containers and manage the lifecycle of bare metal or virtual servers hosting these containers. Docker's containerisation technology seems to be entering a period of exponential growth, which can be measured through the number of "pulls" [1] from the Docker hub [2]. Each pull witnesses from a new container being started in a (remote) cloud service, thus serving as a good metrics for the growth of the technology and its increased acceptance and use in production systems.

This paper presents a number of interrelated tools seeking to ease the use of the Tcl language for the realisation of Docker-based micro-services architectures. These tools are not only about authoring the content of containers using Tcl, but also about managing the lifecycle of these containers as part of larger architectures, including the management of the architectures themselves. Most of these tools originates from the needs of realising flexible cloud architectures for the Internet of Things, but they have also been used in the context of an audio and video streaming solution. All tools are used in (semi-)production environments, from research prototypes to production servers through industrial pilots.

2. Background and Motivation

The containerised approach is not new: The `chroot` [3] system was first introduced in 1979, and refined as the `jail` [4] command in 2000. Control groups (`cgroups`) [5], the Linux kernel feature limiting and isolating resource usage was originally written in 2007. Containerised applications eliminate the need for each instance to run on its own separate operating system. Applications can be deployed within seconds and using fewer resources than when running hypervisors. However, the downside of traditional containerised applications is that they need to share the same exact OS version as the hosting kernel.

Docker brings portability to the equation. It relies on features such as `cgroups` [5] and `namespaces` [6] to ensure resource isolation but also packages an application with all its dependencies. This packaging allows the application to run across different operating systems so that developers can write in any language and easily move their containers from their laptop to production servers along the continuous integration chain. In doing so, Docker favours architecting applications using many components (containers), networked together using well-defined APIs. Each container can be written in the best language for the task at the time. With growth, necessary rewrites can occur later in the process, on a container basis, without any implications onto the remaining of the application architecture.

There is a lot of so-called “hype” around Docker and its ecosystem, and this means that the technology is often used in environments using modern computer languages of various sorts. This novelty is highly reflected by the language stacks [7] available in the list of curated official images on the Docker hub. While the main motivation for work described in this paper is to “get things done”, these tools also provide a way to easily integrate Tcl-based software as part of containerised architectures, thus making these architectures benefit from the stability, experience and flexibility that Tcl has had for years.

3. Tcl Containers

At the base of each running Docker container, there is an image. While organisations might install and use private repositories, many images originate from the central repository available at the Docker Hub [2]. Apart from a curated list of official Docker images [8], the hub also contains a large number of public images serving slightly different use cases or outside of the major trends directing microservice architectures. Typically, popular repositories with a maintainer who complies to the set of guidelines established for image creation [9] will migrate into the list of official images.

Docker images are often based on a linux distribution. However, these are not to be mistaken from running an entire linux system. Instead, the distribution is used to easily access the set of system libraries that are necessary to run the process(es) that form the core of the containers created out of the image. Packaging of libraries and their dependencies is an activity that is well maintained by a various number of linux distributions.

Historically, many Docker images, including the official ones, were based on various versions of Ubuntu. Matching these images, the image available on the Docker hub as `efrecon/tcl`[1] is also based on the latest version of Ubuntu. The intended usage for this image is to provide a "batteries-included" approach for developing (web) applications. It follows the versioning for Tcl related packages in the underlying Ubuntu distribution. The image is automatically rebuilt every time the main (official) Ubuntu image is modified, thus benefiting from all security updates that would affect underlying libraries and depending packages. Image creation is directed by the open source project `efrecon/docker-tcl`[2] at github. The project installs most Tcl-related packages part of the main Ubuntu distribution and provides by default a tclreadline capable prompt for interacting with the shell with `docker run -it --rm efrecon/tcl`. The exact list of available packages can be induced from the automatic image creation procedure defined by the `Dockerfile` contained in the project.

Images based on the Ubuntu distribution can be of non-negligible sizes. While it can be argued that storage costs have become negligible, there are still a number of reasons to minimise the size of images: large images imply longer download times, which might slow down cold container initialisation, large images might increase costs whenever using minimal virtual machines at hosting providers, etc. For this very reason, a number of official images provide variants based on Alpine Linux instead. A major difference between Alpine and Ubuntu is that the former is based on musl libc [10]. Alpine focuses on providing a minimal system and Alpine images are in general much more compact. `efrecon/mini-tcl`[3] is another automatically built image at the Docker hub, this time built on top of the packages available as part of Alpine. Special attention has been put into bringing in a modern version of the TLS package[4]. Different tags, following the naming conventions of the different versions of Alpine Linux are available and modern versions of TLS are only available starting from version 3.4. At the time of writing, the image is only 5.6MB in size, while still being almost on par with its Ubuntu-based sibling. The image uses a pure Tcl implementation of the readline capabilities to provide a similar experience at the prompt.

[1] The image `efrecon/tcl` is available on the hub at https://hub.docker.com/r/efrecon/tcl/. Build history, reflecting mainly changes in the main underlying Ubuntu image is detailed at https://hub.docker.com/r/efrecon/tcl/builds/.
[2] The `efrecon/docker-tcl` is available under the BSD license at https://github.com/efrecon/docker-tcl.
[3] The image `efrecon/mini-tcl` is available at https://hub.docker.com/r/efrecon/mini-tcl/.
[4] A patch was submitted by the author of this article so as to bring in version 1.6.7 of the TLS package, see http://patchwork.alpinelinux.org/patch/1604/.

In addition to these two well-proven images, recent efforts have been put into the creation of a Tcl Docker image for ARM architectures and the later versions of the Raspberry Pi in particular. `efrecon/armv7hf-debian` is an automatically built Debian-based image available with two different tags: `wheezy` and `stretch`. Build automation is courtesy of the resin.os project [11] and uses a statically built version of `qemu` for the Intel architecture. Using this binary, it is possible to create an image containing ARM binaries on the Docker hub, itself running all continuous delivery pipelines on Intel servers at the time of writing.

4. Docker API

The Docker remote API [12] is a REST-like API for communicating with the docker daemon either locally or from a distance. This API is not only used for communicating with the so-called engine, but serves also as the base for controlling swarm, Docker's clustering implementation. The API departs somewhat from the REST principles as it offers ways to follow the output of running containers as a stream of messages, building upon a mechanism similar to the connection upgrade that occurs when initialising websockets [13].

The Tcl implementation of the API is a work in progress. It provides an interface to introspecting containers, controlling them (start, stop, etc.) and attaching to them. There is also initial support to creating new containers. As the Docker daemon running on the host is both able to listen on a UNIX domain socket or TCP, with HTTP(S) layered on top, the current implementation of the API includes a minimal reimplementation of the HTTP-language necessary for the task, including chunk encoding and connection upgrade [14]. To connect to the UNIX domain socket, the API relays traffic using either `socat` or `nc`.

The implementation is provided as an open project on github[5]. In addition to the library implementing the necessary HTTP subset and a partial implementation of the Docker remote API, the project provides a semi-complex example of an application that is able to send the output of any container to a remote HTTP server. This can be used for capturing logs of containers and sending them to a cloud service, or to capture the output of a container and push it to a message queue for further processing, for example. To minimise HTTP overhead, the default behaviour for this application is to keep alive established connections.

Processes that need to use the Docker API and talk to a (remote) engine do not need to be directly installed on the host. Instead, they can be containerised themselves. In this case, such containers should be given enough privileges for accessing the local UNIX socket on which the docker daemon listens, and the UNIX socket will be bind mounted into the container so that it can be accessed by any process that builds upon the library implementing the Docker API. The

[5] The git repository `efrecon/docker-client` is available at https://github.com/efrecon/docker-client under the premissive BSD license.

example application described in the previous paragraph is made available for this very use case on the Docker hub as `efrecon/htdocker`[6]. As a result, the container can be used as a building block for any micro-service based on Docker, whichever language is used for the implementation of the remaining containers. Another example of such an image is `efrecon/dockron`[7], an image that can be used to create containers that will operate on remote or local containers at regular intervals in the manner of `cron`.

5. Concocter

While there is no theoretical limit to the number of processes to encapsulate within a single Docker container, the recommended behaviour [15] is to run a single process per container and to bind containers together whenever cooperation is needed. A recurrent behaviour of containers is to acquire configuration variables from a number of resources (environment variables, key-value stores, remote servers, etc.) and then to modify or generate configuration files for the single process to run inside the container. To perform configuration, most containers use a sidekick shell script. This script is then replaced by the main process to be run as soon as configuration has ended.

The motivation for `concocter` is to formalise this process slightly while coping better with dynamic environments where the value for the (remote) resources affecting the configuration might change over time. The main goal of `concocter` is to generate all the necessary configuration files based on the content of remote locations and other resources before starting another program. In order to generate the configuration files, `concocter` supports a flexible templating system. The content of (remote) resources is represented as variables, and these variables can be used as part of the configuration files. In addition, `concocter` has direct support for Docker through the API described in the previous section. It will be able to generate configuration files using the current dynamic state of the docker daemon through exposing a wide number of properties for each running container, including their environment variables. These environments variables can be used to affect the content of the templated files.

In its most simple form, `concocter` will get the content of all specified variables, generate configuration files using the templates and the content of the variables and replace itself with the program under its control. But `concocter` is also able to regularly update the content of all variables, re-generate configuration files whenever content has changed and (re)start or more properly signal the program under its control as necessary. Whenever run in this detached mode, using `concocter` departs from the “one container, one process” principle. However, `concocter` will automatically forward all signals that it receives to its underlying process. In

[6] Container based on the image at https://hub.docker.com/r/efrecon/htdocker/ will be able to send the output of any running container on the host to remote web servers.
[7] `efrecon/dockron` is available at https://hub.docker.com/r/efrecon/dockron/ and is automatically built from a BSD-licensed project host at github: https://github.com/efrecon/dockron.

detached mode, `concocter` also implements a software watchdog facility. The watchdog automatically receives all lines output by the process under its control and is able to communicate back to `concocter` (restart) or to take other actions. The watchdog is placed in a separate Tcl interpreter, thus making it easy to forward lines to remote services using log-oriented protocols or to even restart the host. There are no restrictions as to what the separate interpreter shall be able to do in order to gain maximum flexibility; this is however at the expense of security.

A complete description and manual for `concocter` are available from its open source project location at github[8]. The implementation delegates variable acquisition to a set of plugins, making it possible to extend `concocter` for the support of more complex configuration sources such as key-values stores (consul, vault, zookeeper, etcd[9], etc.). `concocter` uses a templating system which has access to the values of all variables and can execute Tcl code in a safe interpreter [16]. The current set of variable plugins implements the following sources:

Variables which specification starts with a `@` are understood as the content of a (possibly) remote resource. All characters that follow the `@` sign should be an URL and `concocter` will get the content from that URL and assign it to the variable internally. `concoter` only recognises HTTP/S and a special docker construct at present, and considers any other URL as being a local file. The `docker://` construct specifies how to connect to a (remote) Docker daemon and will automatically instantiate and update variables based on the properties of all containers running at that daemon, or within that cluster when running against a swarm master.

Variables which specification starts with a = are understood as a proper Tcl mathematical expression. Within that expression, any string surrounded by `%` is considered the name of another variable and the whole string will be replaced by the content of that variable before the expression is evaluated.

Variables which specification starts with a `^` are understood as the gathering of file statistics for the path formed by the remaining of the specification. The variable will be a Tcl array reflecting the regular calling of `file stat` on the path. Whenever the path is a directory, the array will also contain an index called `files` that will contain the list of files directly in the directory (no-recursion).

[8] `efrecon/concocter` is the open source repository available at https://github.com/efrecon/concocter under the permissive BSD license hosting the main implementation for `concocter`.
[9] `efrecon/etcd-tcl` is an implementation of the v2 of the etcd protocol in Tcl. It is available under the BSD license at https://github.com/efrecon/etcd-tcl.

Variables which specification starts with a `!` are understood as an external process to execute. The result of the process will be set to the content of the variable. At present, there is no protection whatsoever against malicious usage, so you should use this facility with caution.

Otherwise, the specification will be the content of the variable. Within that specification, any string surrounded by `%` is considered the name of another variable and it will be replaced by the content of that variable before the expression is evaluated. In addition to internal variables, `concocter` is also able to pick up the content of environment variables and to default to a value whenever a variable does not exist. The default value is then separated from the name of the variable using a `|` sign.

6. Machinery

`machinery`[10] tries to be the missing piece at the top of the Docker pyramid. `machinery` is (mostly) a command-line tool that integrates Machine, Swarm, Compose and the Docker Engine itself to manage the lifecycle of entire clusters. `machinery` combines a specifically crafted YAML file format with compose-compatible files to provide an at-a-glance view of whole clusters and all of their containers. In addition to its command-line interface, `machinery` also provides a REST-like API to ease integration and automation with external projects and tools.

Through the provision of an integrated view of entire clusters, `machinery` eases tasks such as creating or removing virtual machines hosted at any of the providers supported by Machine, but also managing the creation or removal of containers onto those machines. Containers can either be pinpointed to specific machines, either be placed onto the cluster using any of the controlling facilities provided by Swarm. To quicken container cold starting in dynamic scenarios, `machinery` is able to initialise (virtual) machines with a number of docker images ready to be instantiated whenever needed.

`machinery` formalises an entire Docker-based infrastructure via a single YAML description file. In this file, each machine (both bare metal servers and virtual machines are supported) is described through a number of properties that completely describes its behaviour, purpose and content. Machines part of the project and cluster can contain the following properties:

- Properties for the creation of the machine, such as all necessary credentials to access the cloud service (Azure, AWS, etc.) or to access a generic machine (including bare metal servers).
- Properties for dimensioning the machine: amount of CPUs, memory, storage. The exact effect of these properties will depend on the target machine: physical machines cannot change their amount of available memory through changing a YAML file!

[10] `machinery` is available as an open project at github. The project is released under the BSD license and documentation, code and issues are managed from its home page at: https://github.com/efrecon/machinery.

- Properties for swarming will permit to elect different machines that will act as master in the cluster, or opt out of the cluster. In addition, free-form labels can be associated to machines so they can be pinpointed in various ways whenever scheduling containers into the cluster.
- A prelude and an addendum can contain the specification of any number of scripts or processes to execute as soon as the machine has been created or once all initialisation steps have been performed. Both can either be executed locally on the host running `machinery` or on the machine host, in which case they can be transferred from any location in the project's directory context. The prelude and addendum can be used to implement anything that would not be formalised and implemented as part of `machinery`. Examples would be the opening of specific ports through the Azure CLI[11] or on a generic remote host's firewall, or the installation and configuration of architecture specific packages for common storage.
- A list of shares formalises the synchronisation of directories on the host running `machinery` and the machine. Whenever the machines are virtual machines on the host, these will be proper mounts. In other cases, synchronisation will automatically install and use `rsync`. Synchronisation will automatically be performed when machines are started and taken down, and can also be scheduled to happen at regular intervals. For more complex scenarios such as distributed filesystems, Docker's own volume can also be used.
- A list of files and directory specifies content that will be transferred from the project directory to the (virtual) machine. These can contain, for example, architecture and project-specific configuration files, secrets, etc.
- A list of registries and credentials to access those registries can be used when the Docker daemon located on the machines needs to pull images for the creation of containers, apart from the regular Docker hub.
- A list of images formalises the exact containers that can be run on a specific machine, but also implements a security mechanism. The default behaviour is to download publicly available images directly on the remote machine, but to download private images on the host and securely transfer those images onto the remote machine[12]. This behaviour obsoletes the list of repositories described in the previous point and minimises the attack vector by providing a single point of possible information leakage (the host machine).
- YAML files in the Docker compose format [17] and relative to the project directory can be scheduled to run onto specific machines.

[11] There are two major CLI for Azure on github: the initial one written in node, available at https://github.com/Azure/azure-xplat-cli and the new python based one, available at https://github.com/Azure/azure-cli.
[12] On Windows and Mac, the local virtual machine installed as part of the Docker Toolbox or the Docker for Windows are used as the source for the private images.

Based on this YAML file syntax, `machinery` provides a number of commands to (re)create entire clusters and manage the lifecycle of clusters and their containers, including ways to search and operate on containers in islands of the cluster. `machinery` is capable of long-running sessions through the implementation of an HTTP server. This server provides a REST-like layer on top of most of the commands that are otherwise available at the command-line.

7. Future Work

This paper has presented a number of interrelated tools aiming at bringing together the Docker ecosystem and the flexibility and maturity of the Tcl programming language. These tools are at varying levels of readiness, but they are all used in production, albeit sometimes under circumstances that do not require complete implementations of the tasks at hand. Consequently, planned future work is in varying conditions, depending on the tool.

A major undertaking would consist in taking the necessary steps to output a community supported Tcl Docker image that would comply to the rules and requirements driving the adoption of projects into official Docker images [9]. Past experience with the Docker maintainers have shown that they are opened to proposals and input. Such an endeavour would raise the awareness of Tcl as a possible player in the field of complex and scalable cloud systems.

The Docker documentation tree does not list the Tcl implementation of the API [18]. Reaching such an official status would need (near) completeness of the API and to separate the example application from the library implementation itself. The implementation should also be amended so as to be able to prefer the use of the various libraries offering support for UNIX sockets over having to run underlying `socat` or `nc` processes to implement the necessary subset of the HTTP protocol.

`concocter` is the youngest of all tools described in this paper and still needs to mature to reach a greater target. `concocter` should provide a larger set of source plugins for the variables and be able to dynamically get these plugins from (trusted) remote locations. In addition, plugins and watchdogs should prepare for larger complexity and the implementation would benefit from being able to load these from directories and/or mounted archive formats (such as ZIP or TAR). In general, `concocter` needs more testing in a variety of environments to squeeze out bugs.

Adversely, `machinery` is probably the most mature of all tools. The Docker ecosystem is a fast moving target and Docker benefits from a high momentum to bring in new features at a rapid pace. `machinery` is lagging a little behind and still does not benefit from the new networking and swarm features that were introduced in the two latest versions of the engine. `machinery` implements an example of “infrastructure as code” through the provision of a single YAML file containing or pointing at all the necessary information to (re)create an entire cluster. However, container scheduling onto `machinery`-driven clusters still is a manual step and the file format should be amended to integrate this feature into the core set of commands and features. Finally, a rudimentary UI could bring more visibility to the project. This interface would benefit from the REST-like API that is already implemented.

8. Conclusion

Docker has recently emerged as a major player for the realisation of large and scalable cloud applications. This paper started by the description of a number of images ready for the creation of Tcl containers in the era of microservice architectures. These images aim at easily writing containers based on the Tcl language, from server farms to smaller gateways typically used in IoT applications. The paper also presented an implementation of the Docker API in Tcl as a library. This library makes it possible to use the glueing capabilities of the language to quickly implement applications that orchestrates or introspects in heterogeneous environments. In addition, the paper introduced `concocter`, an application that systemises the starting and controlling of processes that are placed in Docker containers, together with how they dynamically interact and integrate with the remaining of the cluster. Finally, `machinery` attempts to fill a missing spot at the top of the Docker pyramid through the implementation of a flexible cluster controller that offers command-line and web interfaces to manage the whole lifecycle of Docker-based architectures.

Modern (web) applications are often so-called “one-page applications”, with the majority of the front-end code written in a dialect of Javascript and communicating to the backend via REST and JSON APIs. Their backend is often orchestrated through various types of queues and messaging systems, Apache Kafka [19], RabbitMQ [20], etc. In combination with protocol implementations [21][22][23] compatible with these messaging platforms, the tools described in this paper can help leverage the flexibility and maturity of the Tcl language to the realisation of complete infrastructures where scale and the continuous evolution of the applications are key requirements.

9. Acknowledgements

Work on the various software components described in this paper has been sponsored by the Swedish Energy Agency and Vinnova, Sweden’s innovation agency. A smaller slice of the work was also performed within the framework of JoiceCare AB (now a part of Phoniro Systems AB).

10. References

[1] “*Docker Hub Hits 5 Billion Pulls*”, M. Marks, available at https://blog.docker.com/2016/08/docker-hub-hits-5-billion-pulls/, 2016-08-11.
[2] “*Docker Hub*”, available at https://hub.docker.com/, last checked 2016-10-18.
[3] “*chroot*”, https://en.wikipedia.org/wiki/Chroot, last checked 2016-09-10

[4] "*FreeBSD jail*", https://en.wikipedia.org/wiki/FreeBSD_jail, last checked 2016-09-10
[5] "*cgroups*", https://en.wikipedia.org/wiki/Cgroups, last checked 2016-09-11
[6] "*Linux namespaces*", https://en.wikipedia.org/wiki/Linux_namespaces, last checked 2016-09-11
[7] "*Docker Hub Official Repos: Announcing Language Stacks*", S. Johnston, available at https://blog.docker.com/2014/09/docker-hub-official-repos-announcing-language-stacks/, 2014-09-24
[8] "*Understanding Official Repos On Docker Hub*", M. Ponticello, available at https://blog.docker.com/2015/06/understanding-official-repos-docker-hub/, 2015-06-01.
[9] "*Docker Official Images*" (section "Contributing to the Standard Library"), available at "https://github.com/docker-library/official-images#contributing-to-the-standard-library", last checked 2016-10-18.
[10] "*musl libc*", Home Page, available at https://www.musl-libc.org/, last checked 2016-10-18.
[11] "*Building ARM containers on any x86 machine, even DockerHub*", P. Angelatos, available at https://resin.io/blog/building-arm-containers-on-any-x86-machine-even-dockerhub/, 2015-12-25.
[12] "*Docker Remote API v1.24*", available at https://docs.docker.com/engine/reference/api/docker_remote_api_v1.24/, last checked 2016-10-18.
[13] "*The WebSocket Protocol*", I. Fette, A. Melnikov, IETF RFC 6455, 2011-12.
[14] "*Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*", R. Fielding, J. Reschke, IETF RFC 7230, 2014-06.
[15] "*Best practices for writing Dockerfiles*", available at https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/, last checked 2016-10-18.
[16] "*The Safe-Tcl Security Model*", J. Levy, L. Demailly, J. Ousterhout, B. Welch, Proceedings of the USENIX Annual Technical Conference (NO 98), New Orleans, Louisiana, 1998-06.
[17] "*Compose file reference*", available at https://docs.docker.com/compose/compose-file/, last checked 2016-10-18.
[18] "*Docker Remote API client libraries*", available at https://docker.github.io/engine/reference/api/remote_api_client_libraries/, last checked 2016-10-18.
[19] "*Kafka: a distributed messaging system for log processing*", J. Kreps, N. Narkhede, J. Rao, ACM SIGMOD Workshop on Networking Meets Databases, 2011-06-12.
[20] "*RabbitMQ*", available at https://www.rabbitmq.com/, last checked 2016-10-20.
[21] "*KafkaTcl, a Tcl interface to the Apache Kafka distributed messaging system*", available at https://github.com/flightaware/kafkatcl, last checked 2016-10-20.
[22] "*RabbitMQ TCL*", available at https://github.com/dereckson/rabbitmq-tcl, last checked 2016-10-20.

[23] “*biot - Information Pipelines in IoT-Clouds*”, E. Frécon, Proceedings of the 22nd Tcl Conference, Manassas, Virginia, 2015-10-22.