Scaling a systemd-managed industrial software system leveraging software containerization and message queues, a case study

Morandini, Daniel
daniel.morandini@keepinmind.info

May 2020

1 Scaling without software containerization is cumbersome

We have a server which is taking user requests through an HTTP web interface. Users have the ability to execute some resource-consuming processes through it, such as video transcoding among them, which is an high CPU/GPU usage activity by its nature. The different nature of the web server with the other processors made us naturally build the infrastructure as a set of specific-purpose servers, each with a clear and defined purpose: the transcoding server, the web server, etc. Each service is deployed separately in a different physical server with the most convenient hardware specs for the task. The services communicate using an asynchronous message queue system. The technology of choice is *RabbitMQ* [1]. An overview of the setup is shown in figure 1. Even

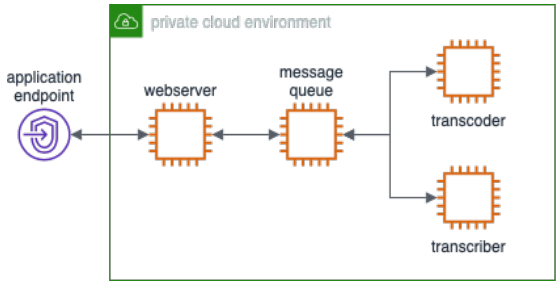

Figure 1: Servers composing the current architecture

though each service runs in a separate server, it does not mean that it cannot be overwhelmed by the number of task requests, leading to a resource overload.

[1] `https://www.rabbitmq.com`

This is why each service has a configurable number of parallel instances that it is allowed to run to complete the requested tasks. Those numbers were obtained by overloading the servers on purpose.

The services run as *systemd* [2] supervised processes. Each service is wrapped around a library that we reuse for establishing the queue connection, bounding the process concurrency and using a homogeneous policy for message acknowledgements (we do not want to loose messages when failures occur).

We deploy the services using custom *Ansible* [3] scripts. Each time we release a new version of a service, it is tagged, built, its artifacts uploaded in an *AWS S3* bucket and eventually downloaded and run in the targeted servers.

The above infrastructure performs quite well in terms of maintainability and reliability when the number of customers accessing the platform is stable, which was the case in the early stages of the product. Now we are facing a usage increment, and we are aware that, if not addressed, this configuration will arise the following non exhaustive list of issues

1. we do speech to subtitles transcription of live video transmissions. The number of concurrent transcribers that can run is upper bounded, which means that **if a user requests a transcription in the wrong (or right, one might say) moment, the service will not be available**. We want to address this issue by increasing the number of service processes on demand.

2. Transcoding requires high performance hardware do deliver the results to the users in a meaningful time. The *AWS g3s.xlarge* server instance types suite our needs, with a cost of *$0.796/hour*, or roughly *€500/month*. We handle up to four concurrent tasks on each of these instances, but most of the time they sit there idle. We want to be capable of shutting those kind of servers down when we do not need them, to **save energy and money**.

This configuration does not address very gracefully another issue too not related to the usage increment stated above. Services run as always on servers. When a code update is performed, servers need to be taken down and up again with the new version. Many approaches exist to address this problem (cit?), but none seem straightforward to implement when the old service cannot be stopped and restarted when it is performing a task. In our case, we need the live transcribers to terminate their sub-processes (some live-streams last up to six hours) before shutting them down. It would be a better approach to start the services always with a dynamically selected version of the software (which might be the latest). In this case though we would not have a daemon process consuming the tasks, but rather a process spawned for each tasks. When the latter is complete, the process is terminated.

[2] `https://en.wikipedia.org/w/index.php?title=Systemd&oldid=955047185`
[3] `https://www.ansible.com`

2 Introducing a task dispatcher dropping always-on services

The new infrastructure introduces an always-running component, a *task dispatcher* which responsibility will be to collect tasks and spawn a responsible service for each one of them, dynamically. The services are then terminated and released when the task is completed (either successfully or not). A schematic overview of the system is shown in figure 2. This setup allows the dispatcher

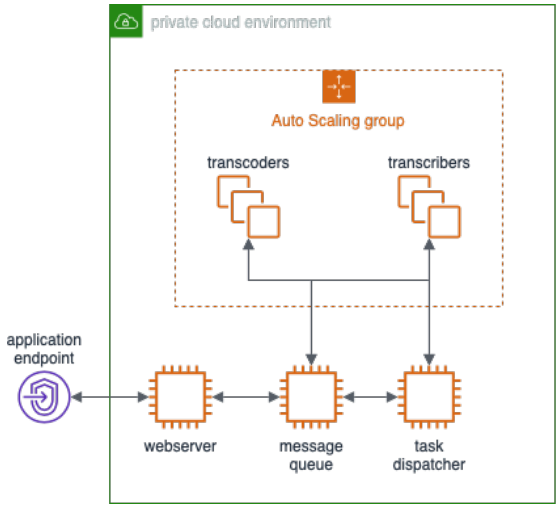

Figure 2: New system architecture composed by a web server, message queue, task dispatcher and a group of dynamically allocated containers.

to make some useful decisions at run-time, like picking the preferred version of the software to use for solving the task. This way we avoid the problems of upgrading the software by bounding its life-cycle to its task. A simplified view of this sequence is shown in figure 3. Readers should interpret *spawn* as a batch of actions involving the discovery of the tool suitable for solving the task, selecting the preferred tool version, selecting a host and executing the tool there.

In this specific case, we deliver the tools as native binary executables (cross-compiled using *Go* [4]), which (only in the transcoders case) have a single run-time dependency, *ffmpeg* [5]. The packaging and distribution complexity is hence reduced, but we don't want to develop a solution that is language dependent, nor one that does not gracefully manage a greater number of run-time dependencies. *OS-level virtualization* [6] is the state-of-the-art approach for coping with

[4] `https://golang.org`
[5] `https://ffmpeg.org`
[6] `https://en.wikipedia.org/w/index.php?title=OS-level_virtualization&oldid=948517170`

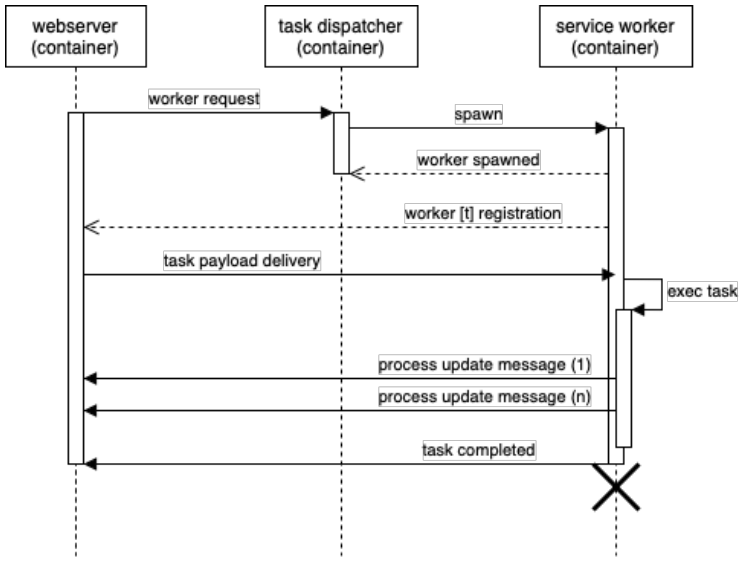

Figure 3:

this kind requirements. *Docker* [7] is our technology of choice: the tools are distributed as images, which can be executed on each machine that is running a docker daemon. An image is a self-contained environment for our service to run, ships without any external dependency with the exception of the docker daemon. With this configuration, we obtained a well defined differentiation between the services themselves and the actual environment hosting the *containers* (images at execution-time), leading to a greater division of concerns: on one side the code with its dependencies, on the other the management of the container distribution, i.e. hardware resource management.

The task dispatcher only has to ask for a container to a docker daemon, which in turn could have at its disposal a set of machines that might run the selected image. In our case, we selected as starting point *AWS Fargate* [8] as container management infrastructure. Based on the task, the container is run on the most appropriate host environment, in terms of performance capabilities. This vendor-locking dependency may be dropped at will, as the solution is flexible enough to allow us to migrate to any kind of container management environment, e.g. a self-hosted *Kubernetes* [9] or *Docker Swarm* [10] cluster.

The flexibility of this setup is obtained with the direct service-to-service communication carried by the message queue. The services are always coded as *Unix* inspired executables (increasing their usefulness in day-to-day tasks), later wrapped around the communication library, a thin glue layer between the

[7] `https://www.docker.com`
[8] `https://aws.amazon.com/fargate/`
[9] `https://kubernetes.io`
[10] `https://docs.docker.com/engine/swarm/`

services themselves and the web server.

2.1 Lambda functions do not fit well

Readers might think about leveraging *AWS Lambda* [11] server-less solution to solve this kind of scalability issues. While bringing advantages on one side, it makes it more difficult to cope with other critical aspects of the software system we're building: some services require to expose endpoints over the Internet, such as the live transcoders and transcribers. While running, they read a stream of bytes from a TCP port **previously selected in coordination with the web server**, which must now this information before starting the live-streaming pipeline. This task is not easily accomplished within lambda's context (cit?). Another drawback is hardware capabilities selection. For transcoding we need to be able to choose the GPU that will be available within the container, even though we do not care **which machine is eventually picked**, as long as these requirements a met. At the time of writing, this is another feature not available if using lambdas (ref?). Finally, the live transcription tasks may last more than 6 hours long, which make lambda's pricing model less convenient for our use-case (cit?).

[11] `https://aws.amazon.com/lambda/`