How to scale a one liner pipeline

Edouard Klein
Gendarmerie Nationale

Abstract

Parallelization, scaling, distribution, concurrency are hard to do right. Furthermore, we wish to use generic, portable and reusable software components as the atomic structure when building our data processing flowchart. We propose a dead simple yet very powerful solution that mimics the UNIX way of chaining filters with pipes and that leverages the file system abstraction to solve the distribution, fault-tolerance and synchronization issues. We show that this idea can be modeled as a Petri Net, and therefore allow the creation of arbitrary topologies. We provide design and visualization tools and examples.

1 Introduction

One of the many successes of UNIX is the ability to easily chain tools one after another using pipes [11], e.g.

```
ping example.com | sed -u 's/.*/ping/' | espeak
```

Once a useful pipeline is devised, it can easily be shared among coworkers, or with the whole world, by sharing the text of the command line[13, 1]. The flexibility and power of the pipe are well known and make it a landmark of computing history.

We would like to retain this flexibility and power, even when offering the functionality of our one-liner to multiple remote users using heterogeneous systems, scaling up and down the back-end according to the demand, and providing some safety guarantees.

With this new use-case we break multiple assumptions of the classic use case of a UNIX one-liner: The users may not

- be tech-savvy enough to use the command line,
- have the correct version of the software installed on their computer,
- have powerful enough hardware to run the software on,
- use a device that run a usable UNIX OS, etc.

In order to keep using and combining the generic reusable software components we usually find in one liners, we propose a system based on a daemon that watches one (or more) directory, processes any files dropped into it, and drops the result in another directory.

This system:

- relies on widespread UNIX tools and system calls (section 3), and very little (192 sloc of Go and 179 sloc of Python 3) additional code, and therefore
  - is very portable and can be used on virtually any UNIX platform (section 12),
  - benefits from a large number of off-the-shelf software components (section 6) and the large body of research on distributed and fault-tolerant file systems (section 8),
  - limits the ability of a given executable in the topology to wreak havoc on the system (section 3)
- can be mathematically modeled as a Petri Net (section 5), and therefore
  - allows the chaining (subsection 5.1), branching (subsection 5.2) and merging (subsection 5.3) of generic reusable software components, in an arbitrary topology,
  - provides insights about transition probabilities, transition time, waiting time, etc. (section 6)
  - lets users visualize the topology in an intuitive way (section 5),
  - lets users monitor the system (section 6),

- can be distributed among multiple computers (section 3, 8),
- scales up and down according to the load (section 3),
- handles partial failures gracefully (section 7),
- handles brutal total shutdown without data loss or data corruption (under certain assumptions) (section 7).

2 Scaling a pipeline

There is a huge variety of ways simple, generic software components such as `grep`, `sed`, `cut`, `cat` or `paste` can be combined using UNIX pipes to give rise to useful processing pipelines. Tales of days saved by a clever pipeline are routinely exchanged [6, 8] and have entered the UNIX lore.

Standard tools can be used to process text files in any imaginable way, but pipes are not limited to text files. Binary data (such as audio or video) can also be processed in a pipeline [2, 3], e.g. :

```
decode < in.mp4 | stabilize | watermark |
encode --format mp4 > out.mp4
```

Despite everyone carrying always-connected devices in their pockets whose processing power can be counted in Billions of FLOPS, we cannot easily have everyone install our software and process and serve their own data. Nowadays, we need to centralize and scale. People will send their videos to us, and we will process, host and serve them.

In order to *scale* this pipeline, we need to handle:

- Multiple users around the world using heterogeneous thin clients
- Potentially spiky usage
- $>$ .99 uptime

To achieve that, we need to wrap our one-liner in a system that:

- monitors resource usage and scale up and down horizontally[1] with the load,
- keeps track of jobs and give the output to the client who provided the input
- is redundant and gracefully handles failure

The UNIX shell was not created for this job, and it can not foot the bill without a lot of abuse. For example, when a pipe is broken, the whole pipeline fails. One has to start over as there is no checkpoint in the middle of the computation.

With the use of `tee` and named pipes, more complex workflow can be created, with e.g. branching, but merging is still an issue, and any mildly complex topology will quickly lead to not-so-readable glue code.

Utilities like `netcat`, `ssh` and `parallel` can handle some aspects of distribution, but one still have to handle bookkeeping in order to return the data to the appropriate user when a job is done.

We don't want to code a ad-hoc solution that would incorporate code from the components of the pipeline, and optimize the whole chain. We want to keep the components self-contained, and pass data around without having to modify the components. This allows us to keep the flexibility of the UNIX pipeline.

3 A file system and a daemon

There exists a very useful abstraction with many battle-tested implementations that are redundant and scalable, that every programmer understands, that all programming languages can use and that benefits from decades of research : the File System.

In order to scale the pipeline, we propose to replace each pipe with a daemon that watches a directory, launches the processing program, feeding it any file created in the directory, and writes the standard output in another directory. The choice of the file system of each directory will depend on the desired trade-off between performance, fault-tolerance, security, etc.

We coded an implementation of this daemon in Go, called `pmjq`, short for "Poor man's Job Queue". The implementation is quite small, and can easily be ported to other languages if need be. The basic prototype is:

```
pmjq <input-dir> <filter> <output-dir>
```

Chaining is implemented by making the daemon watch a directory where another daemon writes its output. Our original pipeline would become :

```
    #Original command was:
    #decode < in.mp4 | stabilize | watermark\
| encode --format mp4 > out.mp4
    #Pmjq's version is:
    pmjq input/ decode decoded/
    pmjq decoded/ stabilize stabilized/
    pmjq stabilized/ watermark watermarked/
    pmjq watermarked/ "encode --fomat mp4" output/
    cp in.mp4 input/‘date‘_example_job.mp4
    #Wait for the processing to finish
    cp output/*example_job* out.mp4
```

File names are used for job management. `pmjq` will simply use the input file's name when writing the output. An obvious scheme is to rely on a timestamp like in

the example above, but anything is possible (for example using a checksum of the data in the filename).

Horizontal scalability is achieved by having the daemon take the host's load into account. To scale up, the daemon call a specific command when its host's load exceeds a certain threshold, to scale down it calls another command when the load comes back down.

```
  pmjq [--max-load <max-load> <max-cmd>]
[--min-load <min-load> <min-cmd>]
<input-dir> <filter> <output-dir>
```

Typically, the command to call when the load exceeds the limit is a command that will launch another daemon on another machine (typically using `ssh`). This second daemon will be configured via the `--min-load` option to kill itself once the load comes back down.

Synchronization and mutual exclusion of multiple daemons watching the same directory (for parallel processing) rely on the underlying file system's features ; `pmjq` relies on the `flock` system call, but implementing a new scheme in case `flock` is not supported by the target file system can be as easy as defining a naming scheme for a lock-file.

Some file systems and operating systems support file system events, which could be used to avoid busy-waiting. We did not implement those yet in `pmjq` as they are quite platform-specific, and instead rely on delayed busy-waiting which we posit has a negligible impact on performance, though we should verify that assumption once we begin supporting file system events.

Security relies on the standard UNIX user and group permissions. For production environments, we suggest the following rules :

- each daemon runs as its own user, in its own group (e.g. the daemon that runs `stabilize` is run by user `pu_stabilize` in group `pg_stabilize`)
- this user and this group owns the input directory (e.g. `decoded/`), and the permissions are set on `r-w--w---`. That way:
  - all daemons that write into this input dir are run by users who belong to the processing daemon's group (e.g. user `pu_decode` belongs to the `pg_stabilize` group).
  - the processing daemon's user belongs to the group of the daemon whose input directory it writes into (e.g. user `pu_stabilize` is in the `pg_watermark` group).

This severely limits the ability for a daemon to create problems on the system, which is extremely useful if the inputs come from untrusted sources. Those rules are enforced by the helper code we wrote to help users create and visualize pipelines.

At-least-once delivery is the default mode, as `pmjq` writes the output file before unlinking the input file. If the system shuts down between these two operations, the worse penalty when the whole system is brought back up (provided the filters are without side-effects and the file system was persistent and stayed consistent) is that from that point on in the pipeline, the jobs will be ran twice on the same input, until the second instance of the data catches up to the first and one daemon overwrites the duplicate file.

Implementing at-most-once delivery is as easy as inverting the order, and unlinking the input file before writing the output file.

Branching and merging are done by having the daemon watch multiple input directories (merging) and launching a program that writes to multiple directories (branching).

4 Petri Nets

Petri Nets are a mathematical model of distributed systems. They provide an intuitive yet formal visual representation of distributed processes. We propose to use Petri Nets not only as a modeling tool, but also as a design tool. That way the user can enjoy both the intuitiveness of a visual representation of their pipeline and a myriad of theoretical results drawn from Petri Net analysis or more generally from Graph Theory, or even from neighboring models such as Markov Processes[2].

A gentle introduction, with examples of some design patterns can be found in [12]. Briefly, in Petri Nets, *tokens* move from *place* to *place* through *transitions*. A *transition* has a fixed set of input *places* and output *places*. Edges can be weighted by an integer which represent the number of tokens that will be *consumed* (edges from *places* to *transitions*) or *created* (edges from *transitions* to *places*) when the *transition fires*. A *transition* can only fire if its input *place(s)* have the necessary amount of *tokens*.

More than one *transition* can be ready to *fire* at any given time. Provided there exist in the implementation a way to deterministically choose which *transition* should *fire* first, Petri Nets have been shown to be Turing-Complete [7][3].

The fact that we can design and model `pmjq` pipelines with Petri Nets is a killer feature, as it allows the user to define an arbitrary topology with not only branching and merging, but also loops, counters etc. This may seem too expressive, and will certainly ruin anybody's day if abused, but it shows that there is no limitation (except uncomputability) on what kind of process can be implemented using `pmjq`.

5 Plumbing in an arbitrary topology

5.1 Chaining

We saw in section 3 that chaining filters with `pmjq` is straightforward. Our original video processing pipeline is transformed into 4 invocations of `pmjq`, using its intuitive syntax that recalls the left-to-right *input→filter→output* model.

In section 3, we also suggested a way to use UNIX's permissions to prevent a daemon from damaging the system. Setting up the users, directories and launching the daemon can be cumbersome, so we wrote some helper code to set everything up.

This interactive helper code generates three shell scripts. The first one (`setup.sh`) creates the directories, users and groups and sets the permissions. The second one (`launch.sh`) launches the daemons. The last one (`cleanup.sh`) removes the directories, users and groups. The user is asked, invocation after invocation, which are the input directories, which are the output directories, and which command `pmjq` is supposed to launch. A transcript of a user setting up our example pipeline using this tool would begin like this:

```
Command:decode
Input dir(s):input/
Output dir(s):decoded/
Command:stabilize
Input dir(s):decoded/
Output dir(s):stabilized/
```

The generated scripts may have to be tweaked afterwards to add mounting/unmounting of remote, RAM or virtual file systems, or to add options (e.g. `--max-load`) to the invocations of `pmjq`. They lend themselves quite well to merging, diffing and version control.

We also provide an utility that, by inspecting the `setup.sh` script, will draw the pipeline (Figure 1). The user can then visually check that everything is as expected. A very important feature of the `pmjq` tool suite is that the graphical model (expressed as a DOT file) is generated from the code. Thus, it is always an accurate representation of the actual settings.

This graphical model maps very easily to a Petri Net:

- *places* are directories
- A `pmjq` invocation using the filter syntax (`pmjq <input> <filter> <output>`) is a *transition* from one *place* to one other *place*.
- A file is a *token*.

In Petri Nets, transitions are atomic. It is obviously not the case with `pmjq`[4], which uses *at least once* delivery by default (see section 3).

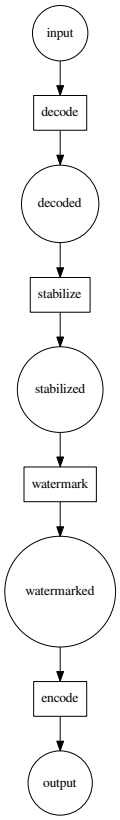

Figure 1: The basic example pipeline as drawn by the `pmjq` tool suite from the `setup.sh` script.

5.2 Branching

The `tee` utility gives the shell the ability to duplicate a stream. Named pipes or command substitution (for shells that support it) can be used to process data in parallel. The GNU Coreutils manual gives multiple examples, such as computing different hashes of a file while it downloads[5].

Branching is supported with `pmjq` as easily as providing a command that writes to more than one file (`tee` itself can be such a command, allowing the reuse of existing shell pipelines with minimal effort). The syntax for branching is: `pmjq [options] --inputs <pattern> <indir>... --cmd <cmd>`
The `<pattern>` and multiple `<indir>` arguments will be explained in subsection 5.3. There is no output argument, branching is simply done by letting `<cmd>` write to multiple directories.

Assume for example that we want to modify our pipeline to separate the audio and video streams of our user-submitted videos.

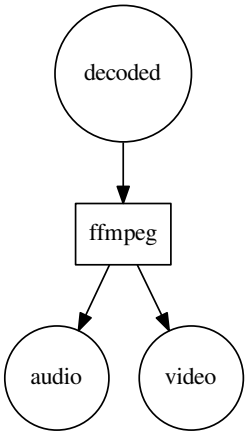

Figure 2: Branching, easy case: the `pmjq` and Petri Net graphs are the same.

The session with our interactive utility will be:

```
Command:ffmpeg -i $1 -map 0:v -vcodec copy \
video/‘basename $1‘.ogv -map 0:a -acodec copy\
 audio/‘basename $1‘.ogg
Input dir(s):decoded/
Output dir(s):audio/ video/
```

Figure 2 shows the resulting graph. The graphing utility knows that the `pmjq` invocation will write to `audio/` and `video/` because of the way users, groups and permissions are managed in `setup.sh`.

In this case, the invocation of `pmjq` one can see Figure 2 is, in Petri Net formalism, the same as a *transition*, because it will consume one *token* (one file) in the `decoded/` *place* (directory), and create two *tokens* (files), one in the `audio/` *place* (directory) and one in the `video/` *place* (directory). If for whatever reason one file is not created, it means there is a bug in the command, and pmjq will log it as such (provided the command returns with a non zero exit code).

We may want to do what some call a *XOR-branch*, that is to say, we may want a `pmjq` invocation to create a token in either one of two directories, but not in both. Imagine for example that instead of running the `stabilize` filter on all our videos, we only do it on videos that are actually shaky. This make sense, performance-wise. Checking that a video is shaky is quicker than running the stabilization process, so if enough videos are non-shaky, the overhead of checking for shakiness will be compensated by not having to run the stabilization process.

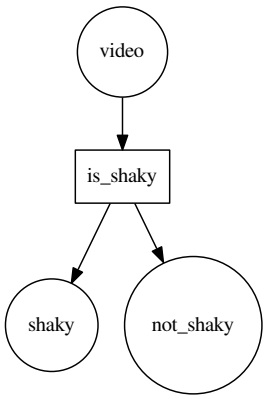

Figure 3: Branching, *xor* case: this is the `pmjq` graph.

To implement this, we need a `is_shaky` command, that will read its first argument, and write it in the directory given as the second argument if it is shaky, and instead write it in the directory given as the third argument if it is not shaky.

The session with our interactive tool will be:

```
Command:is_shaky $1 shaky/ not_shaky/
Input dir(s):video/
Output dir(s):shaky/ not_shaky/
```

The resulting `pmjq` graph (Figure 3) will be different from the underlying Petri Net (Figure 4) in which two transitions compete for the same tokens. This is because, short of solving the halting problem, `pmjq` can not guess which transition (if any) can and will fire. The only information the visualization tool knows is the name of the directories the command has write access to. The kind of branching one wishes to implement should be documented along with the command (in our example `is_shaky`), and best practice would be to somehow tie this documentation to the code and to rigorously test it.

It may be tempting to use multiple `pmjq` invocations competing for the same tokens from the same directory, but this is an anti-pattern. It is permitted by the Petri Net mathematical model, and it is possible in practice, but within the capabilities of `pmjq`, it introduces undeterminism or side effects.

When a file (*token*) is created in the directory (*place*) that both invocations (*transitions*) read from, in order to stay deterministic, there must exist a rule to say which invocation (*transition*) gets to consume the token[6]. The implementation of that rule necessarily rely on some form of communication between the two invocations. Therefore, either we have undeterminism or transitions have

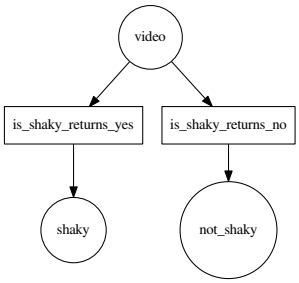

Figure 4: Branching, *xor* case: this is the intended Petri Net.

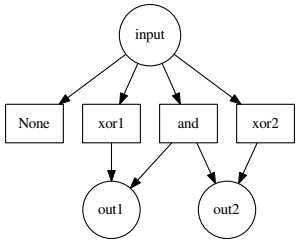

Figure 5: Branching, general case: this is the formal Petri Net equivalent of a two-way branch in the `pmjq` graph. In practice *and*- and *xor*-branching are the most useful, so not all transitions will ever fire, save for a bug in the branching program.

side effects.

To discourage users from using this antipattern, one of the assumption of our interactive helper code is that only one invocation is reading from an input directory. Users that choose to use the antipattern can still easily modify the generated shell scripts manually.

Instead, we encourage user to copy the mechanism illustrated in the `is_shaky` branching. More formally, a multiple output invocation (let n be the number of output directories) in the `pmjq` graph becomes in the Petri Net formalism the set of all 2^n transitions whose output(s) are a (potentially empty) subset of the n output directories. See Figure 5 for an illustration. This ensures that the logic of the branching is encapsulated within one process, and thus we avoid all the problems of synchronization between competing processes.

It is possible, but not yet implemented, to specify in some way which transitions are allowed among the 2^n possible transitions, and watch the log files to raise an alert if an unexpected transition fires.

5.3 Merging

Conceptually, merging is easier than branching because the `pmjq` graph and the Petri Net are the same. One just lists the transition's input places. The syntax, however deserves some explanations:

```
  pmjq [options] --inputs <pattern>
<indir>...  --cmd <cmd>
```

Most of the time, we don't want to merge any two (or three, four etc.) tokens together. We usually want tokens that have been branched from the same original token to be merged back together[7]. For example, we would like the audio stream and video stream of the same original file to be sewn back together, and avoid merging a video stream with the wrong audio stream just because one token went through the pipeline faster or slower than we thought it would.

This is what the `<pattern>` argument (a regular expression) is for. All files should follow the given regexp pattern, which should contain one group. If `pmjq` can find in the input directories a set of files so that:

- there is one and only one file per input directory,
- all the basenames in the set match the pattern,
- all the basenames have the same value for the group in the regexp,

then `pmjq` will launch the given command, replacing '`$1`' with the name of the file found in the first directory, '`$2`' with the name of the file found in the second directory, etc.

The dialog between the user and our interactive tool is:

```
Command:ffmpeg -i $1 -i $2 -c copy \
output/‘basename $1‘.ogv
Input dir(s):audio video
Pattern (default "(.*)"):(.*).og[gv]$
Output dir(s):output
```

Because the user specified two input directories, the tool ask for the pattern argument. By answering `(.*).og[gv]$`, the user means that `pmjq` should wait for an audio (`.ogg` extension) or a video (`.ogv` extension) file to be present in each of the `audio/` and `video/` directories, and that both basenames should be the same before the extension (the parentheses `(.*)` define a group that will match anything before either '`.ogg`' or '`.ogv`').

This was *and*-merging. The other kind of merge, *xor*-merging, is even easier. There is no restrictions on the number of `pmjq` invocations (*transitions*) that can

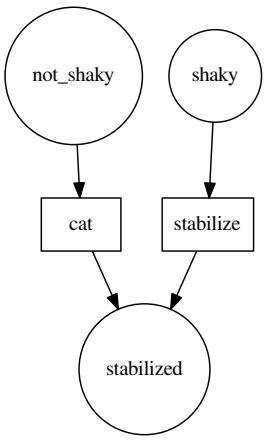

Figure 6: *Xor*-merging: the `pmjq` graph and the Petri-Net are the same.

write in the same output directory (*place*). Therefore, to do a *xor*-merge (such as when merging the shaky and not_shaky branches), we simply specify the same output directory twice:

```
Command:cat
Input dir(s):not_shaky
Output dir(s):stabilized
Command:stabilize
Input dir(s):shaky
Output dir(s):stabilized
```

The resulting `pmjq` graph and Petri Net are the same (Figure 6).

When stitching the whole thing together (Figure 7), we can easily remove unnecessary transitions and places (such as `cat` and `not shaky`, see Figure 6, 3). We also removed the `encode` and `watermark` filters, in order for the graph to take less room, and thus be more legible. Those space constraints don't apply when looking at the graph on screen.

Because our interactive tool reads the standard input, one can save the answers to the questions in a text file, that can be then modified with a text editor, put under version control, merged, diffed, patched etc. Comments can be included, and then removed with `grep`.

6 Tooling

Because `pmjq` relies on the file system and on users and groups, the standard UNIX tooling can easily manage the creation, installation, running and cleaning of advanced pipelines.

Files (*tokens*) in a directory (*place*) can be counted with `ls <dir> | wc -l`.

Because `pmjq` instances are launched in `launch.sh` with `sudo -u pu_something pmjq ...`, one can find the active processes for a given transition with `sudo -u pu_something ps -A`.

All user names are prefixed with `pu_`, and all groups with `pg_`, therefore one can kill all the `pmjq` processes by running :

```
cat /etc/passwd | cut -d: -f1 | grep -E '^pu_'\
 | xargs -I user sudo -u user ps -A | cut -d' '\
 -f1 | kill -9
```

The processes writing or reading from a directory can be listed with `lsof`, resource usage can be monitored using `top`, logs emitted by `pmjq` can be greped, logrotated, syslogded, etc.

Multiple simulation tools exists that can be leveraged to simulate, and find the statistical properties of a given Petri Net, without having to actually run the associated, potentially resource-intensive `pmjq` pipeline. Log analysis can provide data about the transition probabilities, the mean waiting time, etc. In our example pipeline, one could determine if the `is_shaky` branching is worth the trouble by searching for the probability of a token to take one branch or the other, and by measuring the delay induced by the `is_shaky` transition.

7 Fault-Tolerance

If the underlying file system is persistent, and provided transitions are idempotent and an integrity mechanism is in place[8], all active processes can be brutally killed, and then `launch.sh` can be run again to restart the processes, without any adverse effect. One can even change the topology before launching again.

Loud failure of one command will not crash the pipeline, but will simply cause files to accumulate in its input directory(ies). The rest of the pipeline will keep on running while the admins troubleshoot the problem. In our example (Figure 7), a bug in the `equalizer` program would probably be only a minor inconvenience. Because video processing is so resource intensive, the computer(s) will stay busy running `is_shaky` and `stabilize` while the programmers remove the bug in `equalizer`. When it is launched back up, the files that have accumulated in `audio` will be processed.

Silent failure and data corruption can not be mitigated by `pmjq` alone, of course. Therefore it is important to fail loudly and as soon as possible.

8 File System choice matters

Compromises can be made between security, fault-tolerance and speed. RAM file systems are great for speed, but a computer crash will cause data loss. Encrypted disks are very secure, but performance may become an issue, etc.

The performance of `pmjq` will be strongly tied to the performance of the underlying file system.

There has been decades of research on File Systems, and once performance becomes an issue one has to carefully weigh a lot of criteria (safety, security, performance, cost, maintainability, etc.). Here is an incomplete list of File Systems, OS or tools that can be useful in conjunction with `pmjq` :

- Distributed File Systems
  - GlusterFS `https://www.gluster.org/`
  - SeaweedFS `https://github.com/chrislusf/seaweedfs/`
  - HDFS `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`
- Versionned File Systems
  - GitFS `https://www.presslabs.com/gitfs/`
  - Fossil [10]
- Others
  - Plan 9 from Bell Labs `http://9p.io/plan9/index.html`
  - FUSE `https://github.com/libfuse/libfuse`
  - SSHFS `https://github.com/libfuse/sshfs`
  - IPFS [5]

9 Related works

Easy and flexible parallel computing can be done with GNU Parallel (`https://www.gnu.org/software/parallel/`).

Upgrading to full-grade concurrency necessitate the use of frameworks like 0MQ (whose manual `http://zguide.zeromq.org/page:all` is a great introduction to concurrency in general), RabbitMQ (`https://www.rabbitmq.com/`), Kafka (`https://kafka.apache.org/`), etc. All these systems are great and big projects are running thanks to them. They provide lots of features, and are well documented and battle-tested.

The most elegant alternative to `pmjq` is Plan 9's plumber [9], which, used within a Plan 9 cluster, will provide seamless concurrency, strong authentication (with `factotum`), and handle distribution without having to think about it. Showing that some carefully-constructed plumbing rules map to Petri Nets should not be too hard.

10 Conclusion and further work

We provide a proof-of-concept real-life implementation of a mathematically sound model of concurrency (the Petri Nets) by having a daemon watch directories for files to process and write the output(s) in other directories. The implementation is very simple yet extremely powerful. Accompanying tools let the user intuitively design, visualize and manage complex systems.

The reliance on file systems and standard UNIX tools leverage the huge number of off-the-shelf sysadmin tools and the decades of research in distributed, fault-tolerant, secure file systems.

`pmjq` is already used as a prototyping tool, and has yet to reach a point where the performance issues of the underlying file system negatively impact its usefulness. More formal tests are going to be run.

It is also a great tool for ungodly hacks, such as self-modifying topologies (which of course break the Petri Net model).

More work is expected to be done on the GUI front, most notably for real-time monitoring tools.

11 Acknowledgments

The author would like to thank his family for the support, his current employer (Gendarmerie Nationale) and its former employer (Sekoia) for letting him develop parts of `pmjq` on duty/company time, and all the developers of free and open source software around the world for the many tools used to produce this paper.

12 Availability

`pmjq` and the accompanying tool's source code is available at:

`https://github.com/edouardklein/pmjq`

Except where explicitly specified in the paper, all the mentioned features are implemented and tested. There will be a small delay between the deadline for the submission and the uploading of the version of `pmjq` that we used to write this paper.

References

[1] Command line magic (@climagic). `https://twitter.com/climagic`.

[2] Pipe section of the ffmpeg documentation. `https://ffmpeg.org/ffmpeg-all.html#pipe`.

[3] Use and abuse of pipes with audio data. `https://www.debian-administration.org/article/145/use_and_abuse_of_pipes_with_audio_data`.

[4] AGERWALA, T. A complete model for representing the coordination of asynchronous processes. *Proceedings of the First Annual Symposium on Computer Architecture* (1974).

[5] BENET, J. Ipfs - content addressed, versioned, p2p file system. `https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf`.

[6] BLEDSOE, G. Back from the dead: Simple bash for complex ddos. *Linux Journal* (March 2011).

[7] JAVAN, A., AKHAVAN, M., AND MOEINI, A. Simulating turing machines using colored petri nets with priority transitions. *International Journal on Recent Trends in Engineering and Technology* (July 2013).

[8] MUUSS, M. The story of the ping program. `http://ftp.arl.army.mil/~mike/`.

[9] PIKE, R. Plumbing and other utilities. `http://doc.cat-v.org/plan_9/4th_edition/papers/plumb`.

[10] QUINLAN, S., MCKIEN, J., AND COX, R. Fossil, an archival file server. `http://doc.cat-v.org/plan_9/4th_edition/papers/fossil/`.

[11] RITCHIE, D. M. The evolution of the unix time-sharing system. *AT&T Bell Laboratories Technical Journal* (October 1984), 1577–93.

[12] VAN DER AALST, W. Classical petri nets: The basic model. `http://cpntools.org/_media/book/classicalpn.pdf`.

[13] WINTERBOTTOM, D. commandlinefu.com. `http://www.commandlinefu.com/commands/browse`.

Notes

[1] I did not expect to associate up and down with horizontally and still make sense.

[2] Mappings exist from certain extensions of Petri Nets to Markov Processes.

[3] There exist many more extensions that make Petri Nets Turing-Complete. *Inhibitor arcs* are usually used to that end [4]. We choose the Prioritised Petri Net model as it is easier to map to the actual behavior of `pmjq`.

[4] Because atomic transitions are impossible in real life.

[5] `https://www.gnu.org/software/coreutils/manual/html_node/tee-invocation.html`

[6] Note that if the implementation of that rule is based on an exclusive pattern in the filename (*color*) of the file (*token*), then both invocations are not competing with one another. Still, it may be cleaner to change the topology to avoid the situation.

[7] In the Petri Net formalism, tokens are fungible. A backward-compatible extension called Coloured Petri Nets allows the dinstinction between tokens by attaching a value to each token; `pmjq` uses filenames to discriminate between tokens.

[8] Such as appending the hash of the data to the filenames.

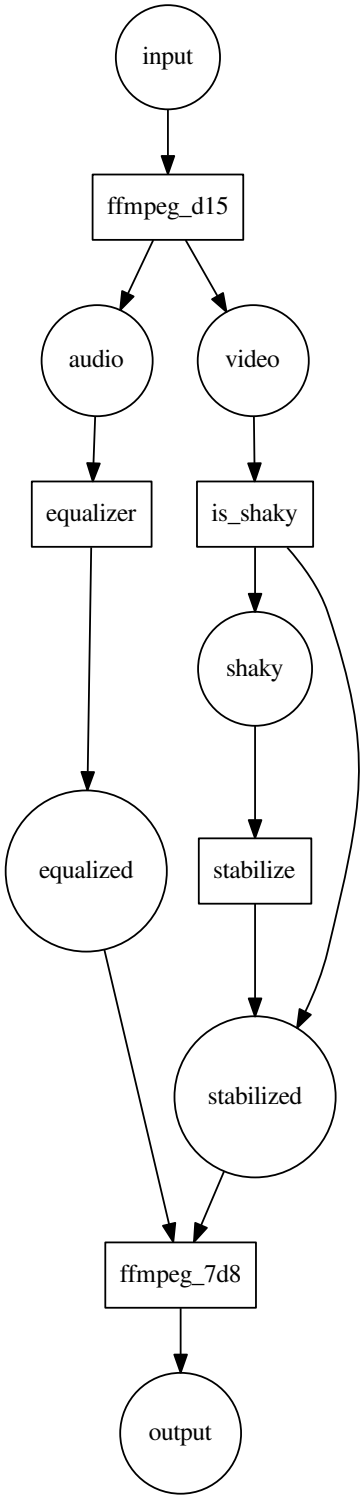

Figure 7: The new pipeline, with branching and merging.