Et si on arrêtait les cRonneries ?

Pierre Gambarotto
Institut de Mathématiques de Toulouse
Université Paul Sabatier
118, route de Narbonne
F-31062 Toulouse Cedex 9

Résumé

Le moyen historique de programmer l'exécution d'une tâche est de prendre une référence à une horloge : c'est ce que permet l'utilisation du vénérable Cron.

Les administrateurs système subissent tous l'épreuve initiatique de la création ou de la gestion d'une ligne dans une crontab, et la connaissance généralisée de cet outil explique en partie son utilisation intensive.

Cron déclenche un script à date fixe : un événement temporel. Or les systèmes actuels sont répartis : il faut être capable de coordonner des tâches sur plusieurs serveurs. Par exemple, la création de la représentation informatique d'une personne dans un logiciel de RH doit engendrer la création des identifiants de connexion dans l'annuaire d'entreprise de type LDAP ou Active Directory, et ensuite engendrer la création d'un espace de stockage et d'une boîte mail sur deux autres serveurs.

Les événements à considérer ne sont donc plus relatifs à une horloge, mais à des étapes d'un processus logiciel : la fin d'une tâche sur un serveur doit déclencher le début d'une autre tâche sur un autre serveur. De telles architectures sont appelées asynchrones.

Les outils présentés permettent aux administrateurs système de mettre en place et de gérer des successions de tâches sur des architectures asynchrones. Deux techniques sont exposées, chacune développée autour d'un exemple :

- *se rappeler que sous Unix, tout est fichier : inotify à partir du noyau Linux 2.6.13 permet de réagir aux événements rythmant le cycle de vie d'un fichier : création, modification du contenu, modification des métadonnées, suppression ;*
- *utiliser des hooks Git : la publication d'une nouvelle version de fichiers du dépôt Git entraîne l'exécution d'une tâche.*

Mots-clefs

events, architecture, tools

1 Introduction

Cron, abréviation de crontab (**ch**r**on**o **tab**le), permet de programmer l'exécution périodique d'une tâche. On spécifie une référence temporelle, et à chaque occurrence, la tâche est démarrée. Cet outil est largement répandu et utilisé dans notre communauté, malgré la syntaxe quelque peu ésotérique nécessaire.

La crontab suivante spécifie que launch_task sera exécutée à 2h00 chaque jour de la semaine de lundi à vendredi :

```
0 2 * * 1-5 launch_task
```

Il existe des outils en ligne qui permettent une traduction plus directe des différentes options, par exemple [1].

Essayons de comprendre les raisons historiques de ce succès.

Tout système informatique est ancré dans du matériel : le prix, la puissance et la connectivité du matériel vont conditionner fortement les architectures logicielles que l'on utilise.

Regardons 20 ans en arrière : la loi de Moore battait son plein, le stockage était cher, la RAM était limitée sur chaque système. Les architectures logicielles étaient très centralisées :

- stockage cher : on limitait la duplication des données ;
- RAM chère, nombre de CPU limité : on limitait le parallélisme.

D'autres facteurs que le matériel favorisaient également la centralisation : par exemple, les techniques d'administration système étant encore largement manuelles, dupliquer un système était coûteux, et on concentrait alors beaucoup d'exécutions sur un nombre réduit de machines.

Dans ce cadre, le rôle d'un administrateur est donc de distribuer le temps CPU sur différentes tâches.

Pour estimer la durée d'une tâche, on réalise des tests sur une machine non chargée. Dans le cas d'un traitement par lot, une estimation du coût revient à une simple multiplication. Exécuter des tâches simultanément rend l'estimation bien plus hasardeuse, considérant les limites du parallélisme.

Cron dans ce contexte permet de réserver une plage temporelle spécifique pour chaque tâche.

Dans l'exemple suivant, on arrête un serveur de base de données à 1h50, on réalise une sauvegarde locale à partir de 2h00, on copie cette sauvegarde sur une autre machine à partir de 3h00, et enfin on redémarre le serveur de base de données à 5h.

```
0 1 50 * * stop_database_server
0 2 *  * * backup_databases
0 3 *  * * copy_backup_external_site
0 5 *  * * start_database_server
```

La situation est bien différente de nos jours, la puissance CPU est décentralisée, le stockage peu cher donc on peut dupliquer. Le parallélisme est exploitable, que ce soit sur la même machine ou sur le réseau. L'outillage actuel permet d'automatiser la création et la configuration de nouvelles machines.

Si fondamentalement les tâches à automatiser n'ont pas changé, le traitement en est différent. Dans le modèle centralisé, la succession temporelle règle le séquencement des différentes tâches, et la crontab permet à l'administrateur d'exprimer et de visualiser en un seul endroit la programmation voulue.

Pouvoir distribuer sur plusieurs machines ces mêmes tâches implique de devoir expliciter le séquencement et les dépendances.

Dans ce cadre, les objectifs de cet article sont :

- décrire un modèle d'architecture basé sur des événements ;
- proposer un modèle de réflexion pour écrire des scripts dans une telle architecture ;
- lister des outils à l'administrateur système pour gérer la communication entre les scripts se déroulant sur une architecture répartie.

2 Exemple général de code

Nous allons prendre comme cas d'étude le traitement de la représentation informatique de personnes, intervenant typiquement dans le circuit d'arrivée d'une structure. À son arrivée, une personne est prise en charge par le service RH, qui se charge de créer son alter ego dans le système d'information, généralement à partir de son identité et de quelques données administratives. Une crontab est alors utilisée pour lancer périodiquement une série de scripts, selon le schéma suivant :

```
# extraction appli RH des nouveaux entrants
# création identifiant login
# création espace de stockage
# création boîte mail (stockage et adresse)
# mise à jour annuaire web
```

Examinons les différentes étapes de ce script, et les compétences et accès mis en jeu.

La première étape correspond soit à un accès à la base de données sous-jacente à l'application RH, soit à une API de cette application permettant l'accès aux données. Dans les 2 cas, cela suppose une connaissance approfondie de l'application, plutôt du domaine du développement ou des bases de données.

À partir de cette première extraction, les scénarios suivants vont graduellement créer la représentation informatique nécessaire à l'attribution de services informatiques et leur utilisation. A contrario de la première étape, les applications et services concernés sont du domaine de l'administration système.

Dans le cadre précis de l'attribution de services informatiques à des personnes, la bonne pratique est de fédérer les différents identifiants informatiques (login, adresses mail) grâce à un annuaire de type LDAP, comme OpenLDAP ou Active Directory pour les plus utilisés dans notre communauté.

Les flux d'informations qui se dégagent de l'étude de ce cas sont les suivants :

1. Création d'identifiant : base rh → annuaire LDAP ;
2. Création de l'espace de stockage : annuaire LDAP → serveur gérant les comptes fichiers, e.g. NFS/Samba ;
3. Création de compte mail et alias de messagerie : annuaire LDAP → infrastructure mail (gestion de comptes IMAP, gestion des alias de messagerie SMTP) → annuaire LDAP. Le deuxième flux sert à stocker dans l'annuaire LDAP les alias de messagerie générés pour un utilisateur ;
4. Mise à jour annuaire web : annuaire LDAP → serveur HTTP donnant accès à des pages web référençant les utilisateurs, comme des pages d'équipe dans une structure de recherche.

Chaque étape va mettre en jeu des serveurs et des services spécifiques.

Une particularité de la première étape est que son résultat peut déterminer la taille du lot de données à traiter. Pour l'exemple, le résultat sera un fichier contenant une liste d'identifiants, un par ligne :

```
achose
btruc
cmachin
```

Une autre particularité est que les identifiants générés sont des prérequis pour les étapes 2, 3 et 4.

Après cette première étape, plusieurs implémentations sont possibles.

La première option est de réutiliser systématiquement ce premier résultat.

```
# crontab
# 1) à 2:00, créer les logins pour les nouveaux arrivants
0 2 * * * rh_news2logins > /tmp/newcomers
# 2) à 3:00, création des comptes fichiers
0 3 * * * create_file_accounts /tmp/newcomers
# 3) à 4:00, création des comptes et alias de mail
0 4 * * * create_mail /tmp/newcomers
# 4) à 5:00, mise à jour de l'annuaire web
0 5 * * * update_web_directory /tmp/newcomers
```

La première étape stocke le résultat dans un fichier, les suivantes le réutilisent.

Pour chaque étape suivante, le schéma du script à écrire est très simple, et consiste à faire une boucle sur la liste des identifiants. Voici un exemple en shell :

```
for login in $(cat /tmp/newcomers); do
  do_something_with $login
done
```

Les étapes correspondant à des services informatiques sont alors relativement simples à coder. La crontab permet d'étaler l'exécution des tâches sur une plage temporelle définie. Chaque étape peut être réalisée séparément ; rajouter une étape consiste alors simplement à segmenter un peu plus la plage temporelle.

En revanche, les dépendances entre étapes N et N+1 ne seront respectées que si l'étape N est réalisée avant que ne commence l'étape N+1. Par exemple, le démarrage de l'étape 4 nécessite que l'étape 3 soit terminée.

Il n'y a pas de lien direct entre deux scripts exécutés par cron, un script ne peut pas directement savoir qu'un autre est terminé. Une solution est de démarrer le script de l'étape N+1 bien après celui de l'étape N, en espérant que le temps laissé soit suffisant.

Cette approche n'est guère satisfaisante pour du traitement par lot, le temps d'exécution d'une étape dépend linéairement de la taille du lot.

Cela conduit à la deuxième option : recalculer par étape le lot à traiter.

Par exemple pour l'étape 2, on peut trouver tous les utilisateurs ayant un identifiant LDAP, mais pas de compte fichiers et réaliser les opérations nécessaires pour ces

utilisateurs. Chaque étape est alors plus complexe à écrire et demande plus de compétences, mais les étapes deviennent indépendantes.

Une version simpliste de cette deuxième option est de non plus traiter un lot correspondant aux nouvelles données, mais de traiter systématiquement toutes les données.

Par exemple pour l'étape 4, les pages d'annuaire web peuvent être entièrement régénérées systématiquement, même en cas d'absence de changement.

Cette deuxième option amène des scripts plus complexes, mais dont le lancement dans le temps est plus souple : les tâches peuvent être exécutées de manière concomitante, en se recouvrant temporellement.

Enfin, mentionnons une troisième option, hybride : un seul script cron, lance les scripts de chaque étape pour chaque identifiant utilisateur. On perd alors l'indépendance de gestion entre les différentes étapes.

```
# crontab
# 1) à 2:00, créer les logins pour les nouveaux arrivants
0 2 * * * rh_news2logins > /tmp/newcomers
# 2) à 3:00, lancer la création de tous les services
0 3 * * *  create_services/tmp/newcomers
```

Le script create_services suit alors le schéma suivant :

```
for login in $(cat /tmp/newcomers); do
  create_file_account $login
  mail_alias=$(create_mail $login)
  update_web_directory($mail_alias,$login)
done
```

Ce script permet de gérer le séquencement et la coordination des différentes étapes pour un identifiant, on perd en revanche la séparation temporelle par lot de chaque étape.

Pour résumer :

- traiter en lots séparés rend plus difficile l’estimation temporelle de chaque étape. Une étape demande un accès et des connaissances distincts, cela permet de mieux adapter le traitement : on peut confier chaque tâche à une personne différente et utiliser une technologie spécifique ;
- à l'opposé, en complexifiant les scripts voire en les fusionnant, on peut mieux contrôler l'exécution.

Cron est bien adapté aux aspects suivants :

- assurer un traitement séquentiel, pour éviter toute surcharge ;
- assurer un déroulement hors période de charge ;
- faire des traitements par étapes successives sur des lots de données.

Cron est de plus un outil bien connu dans la communauté des administrateurs système.

En revanche, cron présente les problèmes suivants :

- la succession de tâches dépendantes est difficiles à gérer ;
- le traitement par lot entraîne une faible réactivité ;
- exécuter plusieurs tâches en parallèle complexifie le code ;
- morceler en étapes revient à morceler le temps.

Une solution par crontab est difficile à faire évoluer.

Pour un traitement donné, on peut vouloir améliorer la réactivité de la prise en compte de nouvelles données à traiter : une tâche cron est activée périodiquement et détermine les données à traiter à son lancement. Améliorer la réactivité implique donc de réduire la période, en d'autres termes d'augmenter la fréquence de lancement. Dans l'exemple des traitements intervenant dans le circuit d'arrivée des utilisateurs, au lieu de lancer les tâches une fois par jour, on peut les lancer plusieurs fois par jour, voire par heure. Cela peut être compliqué en fonction de la distribution temporelle. Dans l'exemple, on pourrait plus ou moins augmenter la fréquence si les nouveaux arrivants se présentent au fil de l'eau ou au contraire sur une plage horaire restreinte.

Si le volume des lots à traiter augmente et que le temps imparti à chaque tâche ne suffit plus, il va falloir augmenter la durée allouée à chaque tâche, voire augmenter la périodicité, ou alors migrer sur un serveur plus véloce. On ne peut pas répliquer le même script sur un autre serveur, les tâches seraient alors effectuées en double.

Dans la même veine, rajouter une nouvelle étape à un traitement existant va être aisé tant qu'il est possible d'allouer une période temporelle.

La principale difficulté, dans l'exemple envisagé, réside dans la gestion des dépendances des tâches successives. L'utilisation de cron revient à ignorer le problème, en linéarisant la succession des tâches, et en allongeant arbitrairement le temps disponible pour une tâche, en supposant que cela sera suffisant pour que la tâche se déroule avant la suivante.

3 Généralisation du problème

L'objectif de cette section est de généraliser l'exemple introduit précédemment.

Commençons par quelques définitions pour fixer le vocabulaire employé :

- ressource : entité du monde réel ou virtuel, que l'on va gérer dans notre système d'information grâce à des représentations numériques. Dans l'exemple, les ressources sont des personnes ;
- représentation numérique d'une ressource : un ensemble de paires clef/valeur proposant une vue d'une ressource. Pour l'exemple, une ressource « personne » peut être représentée par :

```
name: "Gambarotto"
forename: "Pierre"
```

Une même ressource peut être appréhendée par des représentations différentes. Une autre représentation pour une ressource « personne » peut être :

```
login: gamba
groups: imt, members, yoga
```

- cycle de vie : l'ensemble des étapes de gestion d'une ressource dans le système d'information. Chaque étape est constituée d'un état initial, d'une tâche à exécuter, et d'un état résultant. Chaque état correspond à une représentation ;
- tâche : un script/programme prenant la représentation d'une ressource en entrée, et produisant une autre représentation en sortie.

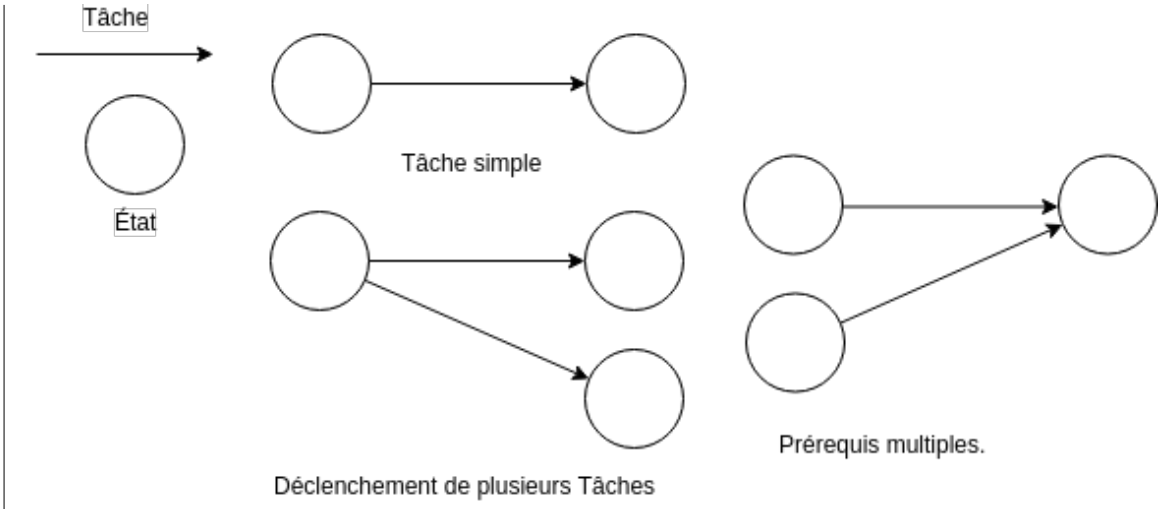

Figure 1 : Cycle de vie : État et Tâche

La Figure 1 représente également les cas de parallélisme.

Ce modèle est une version simplifiée d'une architecture basée sur des événements, voir par exemple le site EventModeling [2] pour un modèle plus général à même de traiter des applications comportant des interactions utilisateurs.

Un des avantages d'une telle architecture est de pouvoir expliquer facilement à une autre personne la vie d'une ressource quand elle traverse le système (*storytelling* ou « construction narrative » appliquée aux données gérées dans un système d'information et à la communication à des non informaticiens [3]).

On bâtit l'« histoire » d'une ressource en racontant la succession d'événements, chaque événement change l'état (la représentation sous forme de données) de la ressource. Les interlocuteurs ayant la connaissance métier peuvent être spécifiques à une étape, de même que l'administrateur/développeur qui va réaliser la traduction technique de l'étape.

Dans l'exemple du circuit d'arrivée d'une personne, l'étape de création de l'identifiant va consommer des informations provenant de l'application RH (nom, prénom) et insérer le nouvel identifiant dans un annuaire LDAP. Un gestionnaire utilisant l'application RH va pouvoir décrire les informations renseignées, l'administrateur de l'application va pouvoir expliciter le codage des informations et donner le moyen technique d'y accéder. Un administrateur système va combiner ces informations pour les insérer dans l'annuaire LDAP. Dans l'étape de création de l'espace de stockage, un administrateur système, potentiellement différent, va gérer l'étape en accédant à l'annuaire LDAP en lecture et au serveur de fichiers. Ces deux étapes impliquent des personnes, des connaissances et des compétences techniques différentes, et pourront être réalisées par des personnes différentes, éventuellement à des époques différentes, avec des choix techniques différents.

Une architecture basée sur des événements se prête bien à une réalisation modulaire, ce qui facilite l'adaptation à des contextes humains et techniques divers.

Deux étapes distinctes vont, au maximum, partager un état entre elles. Pour cette raison, il est facile de faire évoluer un système suivant ce modèle. L'ajout d'une nouvelle étape à un traitement existant est simple, il suffit de déterminer un état existant qui servira d'état de départ à la nouvelle étape. L'impact de la modification d'une étape existante est facile à circonscrire si on respecte le contenu initial et final des états.

Examinons maintenant une implémentation technique possible de ce modèle :

- une représentation d'une ressource est contenue dans un fichier ;
- une tâche va correspondre à l'exécution d'un programme hébergé sur un serveur. On désigne ces programmes sous le nom de *worker*. Plusieurs instances d'un worker peuvent être exécutées en parallèle pour traiter simultanément plusieurs (représentation de) ressources. L'objectif d'un worker est de modifier le système local pour prendre en compte une nouvelle ressource ;
- l'événement déclenchant un worker est l'apparition d'une nouvelle représentation à traiter ;
- un worker produit une nouvelle représentation pour une ressource. Enchaîner l'exécution de deux tâches dans le cycle de vie d'une ressource revient donc à transporter la représentation produite par la première tâche sur le serveur abritant le worker assurant la réalisation de la deuxième tâche.

Le principe directeur est ici de matérialiser la succession des différentes représentations d'une ressource sous forme de fichiers et d'effectuer la coordination entre tâches par une copie de fichiers. Nous sommes dans le domaine de confort d'un administrateur système.

Nous disposons donc d'un modèle simple à appréhender pour un être humain, d'une contrepartie technique adaptée. Il manque l'outillage nécessaire pour gérer la partie événement.

4 Architecture logicielle basée sur des événements : outils pour ASR

Commençons par définir la notion d'« outils pour ASR ». L'idée est de rester dans l'esprit UNIX d'origine :

- avoir une panoplie d'outils spécialisés dont on compose les applications successives ;
- KISS [4] : « Keep it simple, stupid ». Chaque outil doit être aussi simple que possible pour la fonctionnalité visée;ce principe vise à éviter les usines à gaz ;
- représenter les informations sous forme de fichier, dans un format lisible par un être humain.

La construction d'une fonctionnalité complexe se fait alors par la composition de briques simples, et non pas par l'ajout d'un produit magique ad hoc.

On ne veut pas imposer une uniformité sur les outils, les langages et les formats de données. Le but est de permettre la réalisation d'un worker avec des compétences simples en développement.

Par rapport à la section précédente :

- la représentation d'une ressource est un fichier texte ;
- un worker est implémenté avec un langage de script simple ; les entrées/sorties du script sont simples : un fichier en entrée, un fichier en sortie ;
- la création d'un fichier doit permettre de déclencher l'exécution d'un script, pour permettre d'enchaîner les traitements correspondant au cycle de vie d'une ressource.

Nous allons passer en revue les différents outils qui vont permettre de réaliser ces trois propriétés.

4.1 Langage pour coder les workers

Un script/programme servant à coder un worker a une signature simple :

```
worker resource_in
# generates resource_out
```

resource_in correspond au chemin d'un fichier qui sera consommé en lecture uniquement, resource_out est un fichier en écriture seule qui contiendra le résultat.

Il est possible de transformer le format de sortie si nécessaire, mais cette transformation se fera à l'extérieur du script.

```
# worker1 produces json format
worker1 resource_in # generates resource_out.json
# other format needed ? pipe to a transformer
cat resource_out.json | json2xml > resource_out.xml
```

Cette signature simple n'impose aucune restriction pour le choix du langage de programmation. Pour ne pas s'enfermer dans un langage spécifique, on privilégiera l'utilisation de programmes externes à l'utilisation d'une librairie propre au langage utilisé.

Un worker doit consommer un fichier en entrée, produire un fichier en sortie. Pendant ce processus, il est possible de manipuler le système. En revanche, pour limiter la complexité du code, un worker ne doit pas :

- gérer son propre déclenchement ;
- gérer le déclenchement d'un autre worker ;
- gérer plusieurs formats de fichier en sortie ;
- assurer la validation du contenu du fichier en entrée.

Ces aspects seront gérés par le système hébergeant le worker. Le but est de permettre de se concentrer sur le cœur de métier du worker, à savoir la prise en compte d'une ressource pour agir sur le système local.

4.2 Représentation d'informations structurées dans un fichier.

L'état d'une ressource va être décrit sous la forme d'un tableau associatif, une liste de paires clef/valeur, contenu dans un fichier texte.

De nombreux formats de sérialisations existent, équivalents pour une représentation aussi simple : ldif, yaml, xml, json, csv, recutils, …

L'encodage au format texte apporte des propriétés communes indépendantes du format retenu :

- facile à traiter pour un être humain *et* un programme ;
- facile à convertir, que ce soit en entrée ou en sortie, il est simple de faire appel à un programme pour traiter un format particulier ;
- pérenne.

Le projet structured-text-tools [5] catalogue les outils en ligne de commande permettant de gérer des fichiers contenant du texte structuré.

À noter qu'au besoin, le fichier texte représentant une ressource peut contenir un chemin vers un fichier, ce qui permet de gérer du contenu binaire.

```
name: Gambarotto
forename: Pierre
picture: gamba.jpeg
```

4.3 inotify : déclencher un événement par des opérations fichiers

Venons-en au cœur du système : la gestion des événements, sur laquelle repose l'enchaînement des étapes.

Inotify [6] est un mécanisme intégré dans le noyau Linux à partir de la version 2.6.13, parue en 2005. Inotify permet de définir un *watch descriptor* sur un fichier simple ou un répertoire : le système va alors espionner les événements survenant au fichier, et déclencher des actions en conséquence.

Voici les principaux événements auxquels on peut réagir :

- **IN_CREATE** : un fichier est créé dans le répertoire surveillé ;
- **IN_DELETE_SELF** : le fichier observé est supprimé ;
- **IN_DELETE** : un fichier est supprimé dans le répertoire surveillé ;
- **IN_MOVED_FROM** / **IN_MOVED_TO** : le fichier est déplacé/renommé ;
- **IN_ACCESS** : le fichier est accédé en lecture ;
- **IN_MODIFY** : le fichier est modifié ;
- **IN_ATTRIB** : les attributs du fichiers sont modifiés ;
- **IN_OPEN** : le fichier est ouvert ;
- **IN_CLOSE_WRITE** : le fichier est fermé après avoir été ouvert en écriture ;
- **IN_CLOSE_NOWRITE** : le fichier est fermé après avoir été ouvert en lecture.

Inotify est à la base des clients Linux pour les solutions de synchronisation de fichiers tels que SeaFile, OwnCloud et NextCloud : ces logiciels vous permettent de choisir une arborescence de fichiers à espionner, et répliquent sur un serveur distant les opérations effectuées en local.

La suite inotify-tools, disponible sur toutes les distributions, apporte les commandes suivantes :

- inotifywait permet d'attendre la venue d'un événement inotify spécifique sur un fichier ;
- inotifywatch génère des statistiques sur les événements survenus sur les fichiers surveillés par inotify.

Voici un exemple général avec inotifywait :

```
# echo each time a file is created in /tmp/test
inotifywait -e create -m /tmp/test | while read event; do
  echo $event
done
# outputs
Setting up watches.
Watches established.
# from another terminal: touch /tmp/test/file1
/tmp/test/ CREATE file1
```

Inotify-tools est utile pour mettre au point une arborescence de fichiers à surveiller et réaliser les premiers tests. Une fois la liste des fichiers surveillés, les événements attendus et le script associé à lancer mis au point, il est plus simple d'utiliser incron.

Incron est la généralisation de cron aux événements sur les fichiers : au lieu d'utiliser une référence temporelle pour lancer un script, on se base sur un événement survenant sur un fichier.

Dans l'exemple suivant, la commande handle_new_file est appelée dès qu'un fichier est créé dans le répertoire /tmp/test. Le nom du fichier est donné en argument de la commande ($# est remplacé par le nom du fichier, $@ est remplacé par le nom du répertoire).

```
# incrontab
# path event command
/tmp/test IN_CLOSE_WRITE /usr/local/bin/handle_new_file $#
```

Attention : le format des fichiers incrontab est plus limité que celui des crontab. En particulier, les commentaires ne sont pas pris en charge, et il n'est pas possible de définir des variables d'environnement. Il n'est également pas possible de rediriger la sortie d'une commande dans un fichier ou un tube, ce qui impose de la gérer à l'intérieur de la commande elle-même.

Les outils basés sur inotify sont faciles à utiliser, mais comportent des limites :

- Déterminer l'événement à gérer est souvent contre-intuitif; par exemple, pour attendre la création d'un fichier dans un répertoire, on a le choix entre IN_CREATE et IN_CLOSE_WRITE, correspondant au début et à la fin de l'écriture du fichier ;
- Par défaut, inotify ne gère pas la récursivité, mais certains outils basés sur inotify apportent néanmoins l'option comme par exemple inotifywait ; ce n'est pas le cas d'incron ;
- Le nombre maximal de fichiers surveillés par inotify est par défaut relativement bas (8192 sur Debian/Ubuntu), il est néanmoins possible d'augmenter cette valeur ;
- Toute couche entre inotify et le système de fichiers à surveiller peut potentiellement perturber le fonctionnement si elle n'est pas prévue pour transférer les événements. Dans ce cas-là, on peut se tourner vers une solution de proxy telle que notify-forwarder. C'est en particulier le cas pour NFS, ou Docker/KVM dans le cas d'un système de fichiers partagé par l'hôte.

4.4 Extraction d'informations d'applications SI

Un cas particulier où l'on n'a pas le choix sur le format à utiliser se produit à l'entrée du cycle de vie : la première étape, où l'on va récupérer des informations dans une application existante dans le SI.

Je vais évoquer 2 possibilités :

- l'application est dotée d'une API de type Rest fournissant du json ou du xml ; on utilise alors un client HTTP comme curl ou httpie pour récupérer la représentation initiale de la ressource ;

```
#API Rest, curl/httpie => json/xml
# retreive json representation of resource keyed 42:
curl -H 'Accept: application/json' -H 'AuthCookie: secretpass'
https://application_si/path/to/resource/42

{"id":42,
  "name":"Stan",
  "email":"stan@example.org",
  "created_at":"2019-03-26 21:13:12",
  "updated_at":"2019-03-26 21:13:12"
}
```

- l'application repose sur un moteur de base de données, et un accès réseau est possible. Il faut alors utiliser conjointement une requête dans le langage idoine et un client en ligne de commande.

```
# Request postgreSQL database and output csv file
# authentication in .pgpass
psql -d dbname -h dbserver.example.org -t -A -F"," \
  -c "select name, forename from people" > output.csv
```

4.5 Aspects sécurités

Les privilèges nécessaires pour exécuter le code d'un worker dépendent des actions locales à réaliser. Quand c'est possible, on créera un utilisateur spécifique pour cela. De manière plus efficace, on peut également faire tourner un worker en isolation dans un conteneur ou une VM.

La transmission d'un événement d'un système à l'autre revient à transférer un fichier.

Pour cela, on utilise ssh/scp et l'authentification par clefs ssh.

En pratique, il faut générer une paire de clefs pour l'utilisateur correspondant au worker, sans chiffrer la clef privée pour permettre une utilisation non-interactive.

```
worker@server1:/home/worker ssh-keygen -t ed25519 -q -N ""
# output keys in ~/.ssh/id_ed25519{,.pub}
```

Pour chaque worker devant être prévenu, il faut copier la clef publique générée dans le compte sur le serveur approprié.

```
worker@server1:/home/worker ssh-copy-id  id@other.example.org
# copy local pub key to remote .ssh/authorized_keys
```

5 Combinaison des outils dans la modélisation d'une architecture logicielle basée sur des événements.

Utilisons les outils introduits dans la section 4 pour modéliser l'architecture basée sur des événements décrite dans la section 3.

Dans un premier temps, considérons chaque unité de traitement (worker) en isolation du reste du système. Pour chaque worker, il va falloir fixer les paramètres suivants :

- WATCHDIR : un répertoire à surveiller ; tout fichier créé dans ce répertoire déclenchera le worker ;
- le format des fichiers créé dans WATCHDIR ;
- OUTPUTDIR : un répertoire pour stocker le résultat produit ;
- le format des fichiers créés dans OUTPUTDIR ;
- WORKER : le chemin du fichier contenant le script à exécuter ;
- USER utilisateur servant à exécuter le script.

Pour permettre à un utilisateur spécifique d'utiliser incron, il faut que le login soit spécifié dans /etc/incron.allow. Cette contrainte s'applique également au super-utilisateur root.

Il est alors possible de gérer les événements de début d'exécution d'un worker, avec l'incrontab suivante, en remplaçant les paramètres en majuscule par leurs valeurs respectives:

```
WATCHDIR IN_CLOSE_WRITE WORKER $@ $#
```

Le script WORKER recevra à chaque appel le répertoire WATCHDIR ($@) et le nom du fichier dans le répertoire ($#).

Pour déclencher l'exécution, il suffit maintenant de créer un fichier dans le répertoire WATCHDIR.

D'après le modèle, le script doit créer un fichier résultat dans le répertoire OUTPUTDIR à chaque appel.

```
#!/usr/bin/env bash
# general worker scheme
WATCHDIR=$1
RES_IN=$2
# environment
OUTPUTDIR=/path/to/store/results
# consume RES_IN, extract resource_id
RES_ID=$(extract_id_from_file $RES_IN)
# have some effect on the system

# generate result
echo "stuff=something" > OUTPUTDIR/${RES_IN}
# delete input file
rm WATCHDIR/${RES_IN}
```

Le déroulement d’une tâche consiste à consommer le fichier en entrée, faire des opérations sur le système local. Pour finir la tâche, on va détruire le fichier initial, créer un fichier résultat dans un autre répertoire.

Il est maintenant temps de considérer l'enchaînement des tâches. Le cas le plus simple est l'enchaînement direct de deux workers.

Prévenir un worker revient à déposer un fichier sur un serveur dans un répertoire, ce qui peut se représenter par un URL ssh, suivant le schéma user@server:/path/to/dir. La commande à exécuter sera :

```
#!/usr/bin/env bash
# trigger_worker sshpath file
# sshpath ~ user@server:/path/to/dir/
sshpath=$1
file=$2
scp $file $sshpath
rm $file
```

La gestion de l'événement associé correspond à l'incrontab suivante :

```
OUTPUTDIR IN_CLOSE_WRITE trigger_worker sshpath $@/$#
```

Pour gérer le déclenchement de plusieurs workers, il faut rajouter une copie de fichier supplémentaire. Chaque worker distant est associé à un répertoire local.

```
OUTPUTDIR IN_CLOSE_WRITE multimv $@/$# WORKER1DIR WORKER2DIR
WORKER1DIR IN_CLOSE_WRITE trigger_worker sshpath1 $@/$#
WORKER2DIR IN_CLOSE_WRITE trigger_worker sshpath2 $@/$#
```

multimv est le script suivant :

```
#!/bin/bash
file=$1
for target in "$@"; do
  cp "$file" "$target"/
done
rm "$file"
```

6 Application : gérer le circuit d'arrivée d'une personne dans un laboratoire

Reprenons l'exemple initial, et instancions-le avec les outils et le formalisme présentés précédemment.

```
# extraction appli RH des nouveaux entrants
# création identifiant login
# création compte fichier
# création mail (compte et adresse)
# mise à jour annuaire web
```

Chacune ligne va correspondre à une étape dans le cycle de vie, et donc à l'exécution d'un worker spécifique. Le cycle de vie d'une personne dans notre système d'information peut être représenté par le graphe suivant :

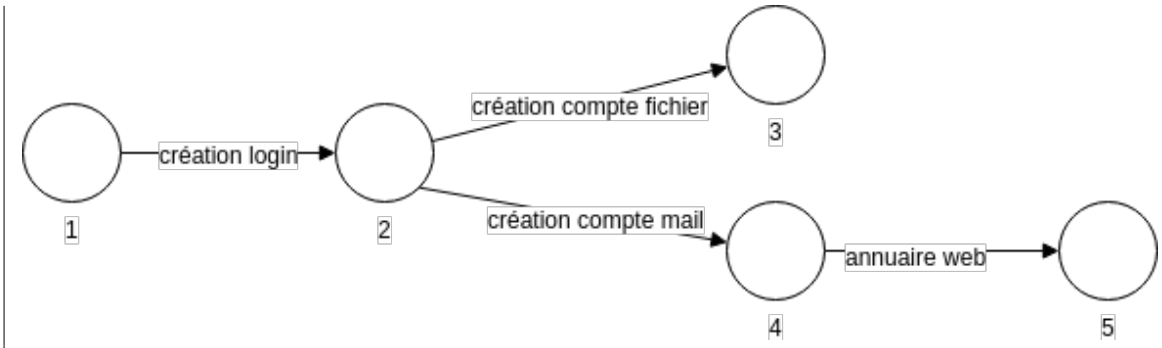

Figure 2: Cycle de vie « Circuit d'arrivée d'une personne »

Cette première étape fait apparaître les différents workers, ainsi que les états initiaux et résultants. Un worker correspond à une flèche, les états 1 à 5 vont être matérialisés par des fichiers fournissant une représentation partielle d'une personne.

En json, on trouvera par exemple :

```
{ "name": "Gambarotto", "forename": "Pierre",
  "team": ["sysadmin"], "comment": "state 1" }

{ "name": "Gambarotto", "forename": "Pierre",
  "team": ["sysadmin"], "login": "gamba",
  "comment": "state 2" }


{ "name": "Gambarotto", "forename": "Pierre",
  "team": ["sysadmin"],"login": "gamba",
  "comment": "state 3, identical to state 2" }


{ "name": "Gambarotto", "forename": "Pierre",
  "team": ["sysadmin"],  "login": "gamba",
  "mail": "pierre.gambarotto@example.org",
  "comment": "state 4" }


{ "name": "Gambarotto", "forename": "Pierre",
  "team": ["sysadmin"], "login": "gamba",
  "mail": "pierre.gambarotto@example.org",
  "comment": "state 5, identical to state 4" }
```

L’attribut « comment » est uniquement donné pour référencer les différents états.

Chaque worker va résider sur un serveur séparé.
Voici un squelette commun, ici écrit en bash :

```
#!/usr/bin/env bash
worker=$(basename -- "$0")
# sets OUTPUTDIR
source /etc/workers/${worker}.conf
# general worker scheme
WATCHDIR=$1
RES_IN=$2
#--------- specific to worker
# extract information from RES_IN, e.g. "name" attribute
name=$(jq .name ${RES_IN})
# have some effect on the system

# generate result, by appending new value to RES_IN
jq '. |= . + {"new_attr": "new_value"}' > OUTPUTDIR/${RES_IN}
#--------- specific to worker
# delete input file
rm WATCHDIR/${RES_IN}
```

Voici le code spécifique pour le premier worker, qui génère un identifiant à partir du nom et prénom d'un utilisateur :

```
# extract name and forename
name=$(jq .name ${RES_IN})
forename=$(jq .name ${RES_IN})
# generate login in ldap directory
login=$(generate_login $forename $name)
# generate result, by appending new value to RES_IN
jq '. |= . + {"login": '${login}"'"new_value"}' \
  > OUTPUTDIR/${RES_IN}
```

On voit que la partie spécifique à chaque worker est simple, et permet d'enrober facilement un script existant, generate_login dans l'exemple ci-dessus.

Pour installer et configurer un worker, on peut par exemple choisir :

- /srv/worker : le code du worker ;
- /etc/workers/worker.conf, la configuration spécifique, sert à positionner les variables OUTPUTDIR, WATCHDIR et WORKER_USER.

```
# /etc/workers/worker.conf
WORKER_USER=worker
OUTPUTDIR=/home/worker/out
WATCHDIR=/home/worker/in
NEXTDIR=/home/worker/next.d
```

Les étapes à suivre pour l'installation sont les suivantes :

- création de l'utilisateur système WORKER_USER ;
- génération d'une paire de clefs ssh pour WORKER_USER ;
- création des répertoires NEXTDIR, WATCHDIR et OUTPUTDIR, WORKER_USER doit en être propriétaire et pouvoir lire/modifier le contenu ;
- pour chaque worker devant être déclenché en suivant, créer un répertoire dans NEXTDIR puis un fichier sshpath dans ce répertoire contenant les coordonnées où copier les résultats du worker courant ;
- génération de l'icrontab dans /var/spool/incron/${WORKER_USER}.

Le script suivant, à lancer en tant que root, automatise les étapes ci-dessus.

```
# configure worker /srv/$1
source /etc/worker/"$1".conf       # read conf
adduser ${WORKER_USER}             # creates user
ssh-keygen -t ed25519 -q -N ""     # creates ssh keys
mkdir -p ${WATCHDIR} ${OUTPUTDIR} ${NEXTDIR}
chown ${WORKER_USER}:${WORKER_USER} ${WATCHDIR} ${OUTPUTDIR}
incron="/var/spool/incron/${WORKER_USER}"
cat << EOF > $incron
${WATCHDIR} IN_CLOSE_WRITE /srv/$1 ${WATCHDIR} \$#
${OUTPUTDIR} IN_CLOSE_WRITE multimv \$@/\$# \
"$(for dir in $NEXTDIR/*;do printf '%s ' $dir; done)"
EOF
for next in $NEXTDIR/*; do
  echo "${NEXTIDR}/${next} IN_CLOSE_WRITE trigger_worker $(cat $
{next}/sshpath) \$#" >> $incron
done
```

La dernière opération à faire est de copier la clef publique du worker dans les serveurs hébergeant les workers suivants, voir 4.5.

7 Gérer des collections de ressources : Git

Une fois que l'on a compris le principe de la gestion par événement, on se surprend à chercher la fonctionnalité dans tous les outils. Une des limites de l'approche précédente est que l'on considère uniquement les ressources qu'une par une.

Git permet de gérer un ensemble de fichiers.

Si on reste sur le paradigme « une représentation de ressource = un fichier », nous obtenons facilement la représentation d'une collection de ressources.

On pourrait gérer une collection de fichiers à la main, avec les workers tels que vus précédemment. L'avantage de Git par rapport à un simple répertoire tient dans les opérations spécifiques disponibles, et à la possibilité de réagir dans le cycle de vie de ces opérations.

Git permet de positionner des scripts, appelés « git hooks » (crochets) [7], qui permettent de réagir aux opérations de Git sur un dépôt. Le crochet que nous allons utiliser est déclenché quand on réalise un « git commit », i.e. quand on enregistre une nouvelle version des fichiers dans l'historique local.

```
$ git clone git@gitlab.example.org:/gamba/annuaire.git
# creates clone in annuaire
# sometimes later
$ cd annuaire
# modify one of the resource, e.g. gamba.json, then
$ git add gamba.json
$ git commit -m "new gamba representation"
# this launches .git/hooks/post-commit
```

Le script lui-même est à créer sous le chemin .git/hooks/post-commit, en suivant le schéma suivant :

```
#!/bin/bash
# working directory = root of git repository
# OLDPWD
unset GIT_INDEX_FILE
# copy last version of all files to another directory
TARGET=/var/www/annuaire
git --work-tree=$TARGET --git-dir=$(pwd)/.git checkout -f
cd $TARGET
# launch other command
```

Revenons à notre exemple de gestion du cycle de vie d'une personne. Dans ce cadre, la commande lancée par le hook post-commit sera la mise à jour des pages web décrivant chaque équipe du laboratoire.

8 Cron

Cron garde quand même quelques cas d'utilisations spécifiques.

En premier, cron est à favoriser quand l'élément qui doit déclencher une modification est de nature calendaire. Par exemple, pour publier des statistiques mensuelles sur les volumes des impressions réalisées dans une structure, un cron tous les premiers du mois est parfaitement adapté.

Un autre usage fréquent, mais moins évident va être de limiter la prise en compte d'événements, par exemple quand un service nécessite un redémarrage pour prendre en compte de nouvelles données, et que l'on ne veut pas que trop de redémarrages successifs limitent la disponibilité du service. Dans ce cas là, on peut substituer un déclenchement par cron à une gestion événementielle.

9 Conclusion : limites et perspectives

Décrire l'évolution d'un système comme une succession d'événements et les réactions associées est naturel pour un esprit humain. Les outils présentés permettent d'exprimer cette représentation duale événement/action au niveau administration système.

Chaque action est un simple script, avec un fichier en entrée et un en sortie, dans un format texte facilement exploitable. La partie événementielle repose sur les événements survenant aux fichiers, transmettre un événement d'une machine à une autre revient donc ainsi à transmettre un fichier.

Le cadre technique est restreint, mais suffisant pour automatiser des scénarios comme le circuit d'arrivée d'une personne sous une forme modulaire. Il existe plusieurs limites à l'approche présentée. Plusieurs aspects techniques n'ont pas été évoqués, comme le monitoring de l'exécution des workers et la gestion des erreurs. Ces aspects peuvent être gérés au niveau de chaque worker. Une autre limite est au niveau de l'architecture envisagée : un worker doit connaître le worker suivant à contacter, et la contrepartie technique (distribution de clefs ssh) est lourde à gérer. Une solution possible pour diminuer ce couplage serait de centraliser la connaissance des différents workers en un seul point, qui serait un point relais de toutes les communications entre workers. Ce rôle pourrait être endossé par un agent de messages tels que RabbitMQ ou Redis, mais cela demande des connaissances plus spécifiques en développement.

Bibliographie

[1] Cronitor, Éditeur de crontab, https://crontab.guru/

[2] What is Event Modeling ?, EventModeling, https://eventmodeling.org/posts/what-is-event-modeling/

[3] Killian Bazin,Storytelling: la prise de décision devient émotionnelle, https://toucantoco.com/blog/data-storytelling-prise-decision-emotionnelle/

[4] Wikipedia, Principe Kiss, https://fr.wikipedia.org/wiki/Principe_KISS

[5] D. Bohdan, Structured Text Tools, https://github.com/dbohdan/structured-text-tools

[6] Denis Dordoigne, Exploiter Inotify, https://linuxfr.org/news/exploiter-inotify-c-est-simple

[7] Scott Chacon, Git book, Customizing Git - Git Hooks, https://git-scm.com/book/uz/v2/Customizing-Git-Git-Hooks