Patrons d'administració de sistemes en entorns Devops

Alumne: Frederic Monpeat i Santo

Tutor: Josep Jorba Esteve

Resum

Amb l'aparició més generalitzada de plataformes grans, el rol de l'administrador de sistemes ha canviat de forma substancial. Amb infraestructures de milers d'equips, les tasques han de ser automatitzades, requerint un desenvolupament més especialitzat que el tradicional script. Ara bé, en un context d'operacions on les interrupcions són habituals el desenvolupament pot ser complexe. El següent article pretén analitzar les tasques d'un administrador, veure si es poden identificar patrons comuns que es puguin aplicar a la automatització en grans infraestructures i presentar la seva implementació en Python.

***Paraules clau**: devops, patrons, sistemes, automatització, python*

1. Introducció.

Un administrador de sistemes és la persona responsable de dissenyar , implementar i mantenir la infraestructura IT de l'organització. Tradicionalment aquests rols s'organitzen en departaments IT dins de grups d'operacions. En l'actualitat diverses empreses ofereixen solucions de *SaaS* i *PaaS* (el que s'anomena *cloud computing*) fet que està esdevenint en un canvi de paradigma sobretot en el camp de l'administració

En l' actual paradigma s'esperava que "el Cloud computing resoldria tots els problemes de l'administrador" (Wayner, 2013). S'ha facilitat la creació d'instàncies i serveis, sobretot amb els avenços en el camp de la virtualització. En aquest context és més fàcil revertir accions amb la creació d'un sistema que analitza fins a quin grau operacions poden ser revertides i alhora un sistema de revertiment que automàticament crea els processos de treball per poder retornar el sistema a l'estat inicial/usable (Weber et al. , 2013). S'ha avançat en escalabilitat (és més fàcil donar resposta a una major demanda incrementant la capacitat) i en eficiència (utilitzant només els recursos necessaris). Per contra s'ha produït una transformació dels problemes: actualment el desplegament d' equips i de programari es compten en centenars o milers de dispositius, instal·lacions que l'administrador ha de gestionar. A més, en un paradigma on es presenta l'infraestructura com a codi el rol de l'administrador i el desenvolupador es fusionen: les operacions s'han d'automatitzar.

En els següents punts s' investiguen les tasques en l'administració de sistemes en un entorn devops, on la infrastructura es presenta com a codi i on els nombre d'equips és de centenars o milers. Un context on els desenvolupadors realitzen tasques d'operacions, on l'entorn es caracteritza per les interrupcions constants esdevenint un medi no idoni pel desenvolupament.

A l'Estat de l'art es revisa la literatura existent, desgranant diferents aspectes de la l'administració de sistemes amb l'objectiu d'identificar quins àmbits poden ser automatitzats i quins models existents es poden aplicar en la implementació.

A la secció de desenvolupament, s'han identificat tasques subjectes a ser automatitzades en l' administració de sistemes en entorns Linux/Unix. S'han associat aquestes tasques a patrons de disseny re-usables per la implementació, alhora que s'han identificat àrees amb possibil·litat d'optimització.

A partir de les implementacions obtingudes, s'han realitzat tests per obtenir dades empíriques que han permés determinar la millor optimització. Aquest tests són explicats a la secció d'experiments i els resultats comentats a la secció següent.
Per establir el marc de treball, s'han utilitzat dades secundàries consultant la literatura existent en fonts acadèmiques (articles de recerca, llibres i publicacions), prioritzant en la documentació més recent. Mitjançant la realització d'experiments s'han obtingut dades primàries a les que s'ha aplicat un anàlisis quantitatiu que ha permés validar les optimizacions, així com la selecció dels possibles patrons de disseny trobats i tècniques proposadesa amb dades empíriques. Tot és recollit a la secció de Metodologia.

El present article detalla tot el procés de treball, els resultats obtinguts i l'anàlisi a partir de les dades obtingudes.En aquesta àrea s'ha detectat un buit en la literatura que l'article pretén adreçar.

2. Revisió de la Literatura

2.1 Infraestructura com a codi

En grans plataformes, el requeriment d'administrar centenars o milers d'equips esdevé en , sumat en molts casos a la virtualització dels recursos, una deriva cap a la "Infrastructura com a codi". Si la infraestructura es pot descriure mitjançant configuracions aquesta es pot administrar mitjançant processos automatitzats. L'automatització de la infraestructura és el procés de crear *scripts* de l'entorn , des de instal.lar un sistema operatiu, configurar serveis o instàncies, com el programari es comunica, etc. Scriptant l'entorn, una configuració pot ser aplicada a un o milers d'equips. L'objectiu és aconseguir escalabilitat a còpia de replicar l'entorn de forma ràpida i reduïnt la probabilitat d'error. (Duvall, 2012)

2.2 Devops

En aquest context l'administrador passa a realitzar tasques de desenvolupament. Segons la interpretació de Google, al desenvolupador se li encarreguen tasques d'operacions. Aquest rol és el que actualment es coneix com *Site Reliability Engineer* i que es pot englobar dins del moviment *Devops*, una cultura lligada

als valors de l'*opensource* que intenta trencar el mur entre el desenvolupament i les operacions (Haddad, 2014). Si la infraestructura es presenta com a codi i si aquesta s'ofereix com a servei, el desenvolupament de l'aplicació i el manteniment estan intrínsicament connectats.

Alguns detractors de la cultura *devops* argumenten que aquest canvi pot no ser positiu per a tots els models d'empresa. Si el rol de l'administrador es veu afectat, el del desenvolupador també és subjecte a canvi. Afegir tasques d'operacions a un desenvolupador implica treballar en un entorn ple d'interrupcions no idoni per tasques de desenvolupament, afectant al rendiment del treballador i en definitiva a la productivitat de l'empresa (Knupp, 2014). Per aquest tipus de rol pot ser beneficiós disposar de tècniques optimitzades per les tasques d'administració i identificar si existeixen patrons de disseny que es puguin aplicar a l'administració.

2.3 Patrons de programari

Diferents treballs publicats al voltant dels patrons de programari comenten alguns punts a tenir en compte, en el cas dels autors de "Past, present and future: Trends in Software patterns" (Buschmann et al. , 2007) recull l'evolució de patrons de programari, s'estableix una classificació basada en àrea de treball i es fa una previsió de les àrees on apareixeran nous patrons. Es referencien els sistemes distribuïts i els dispositius mòbils però no es menciona de forma directa patrons per l'administració de sistemes. Es recull en forma de llibre (Buschmann et al, 2000) un llistat de patrons de programari aplicables a sistemes concurrents i de xarxa, orientat a capes middle-ware i no directament a les tasques de sistemes. Per exemple, *The Wrapper Facade* permet encapsular funcions i dades utilitzant APIs no orientades a objectes de forma més concisa, robusta, portable i més fàcil de mantenir com una interfície de classes orientada a objectes, útil per la creació d'una API de tasques de sistemes. The *Client-Server* proposa un disseny orientat a objectes de la interacció client-servidor, útil per executar determinades tasques d'administració. L'objectiu d'aquest treball se centra en les tasques d'administració i no en els sistemes que permeten executar-les. En aquest sentit, el patrons de programari s'han revisat però no s'han aplicat als resultats proposats més endavant.

2.4 Administració de sistemes

En aquest sentit cal investigar les tasques a automatitzar per veure si es pot establir algun tipus de patró directament en aquesta àrea. Revisant la literatura trobem que hi ha situacions en les que no existeixen

“Best practices” perquè ningú s'hi ha trobat mai. (Merlin, 2013). Hi ha diferents factors que afegeixen complexitat a la multitud de tasques que un administrador pot realitzar. La varietat d'escenaris, els diferents nivells de competències de cada administrador, la complexitat dels entorns que administren, el risc amb el que es treballa (depenent de la indústria, una parada del sistema pot resultar en pèrdues econòmiques i de negoci), etc. En qualsevol cas es posa de manifest la necessitat de sistemes flexibles que es puguin adaptar fàcilment a noves demandes i la necessitat de solucions escalables. (Velasques and Weisband, 2008)

“The practice of System and Network Administration” (Limoncelli et al., 2007) inclou un extens recull de les tasques d'un administrador, des de la configuració d'equips passant per la contractació de serveis a proveïdors o la comunicació amb els usuaris. Abundant informació sobre les experiències acumulades durant anys en la professió. Per contra es descriuen tasques de la professió a nivell genèric (tracta amb els proveïdors, compra d’equips, etc), no centrades en la part tècnica de l’administració.

Des d’una perspectiva diferent, el llibre “Unix and Linux System Administration Handbook” (Nemeth et al. 2010) recull de forma exhaustiva les tasques a nivell de sistemes d'un administrador on els autors revisen abundant informació sobre àrees d’administració.

S'han trobat àrees d'administració que inclouen l' anàlisis, monitorització, instalació i configuració dels equips. Aplicat a grans infraestructures, en la literatura s'han trobat 3 grans grups que resoldrien aquestes àrees:

2.4.1 Sistemes Autonòmics

Quan el sistema entra en estat de fallada, l’administrador realitza un anàlisi per establir la causa i restaurar el servei. Aquest apartat recull eines o tasques que poden ser útils per analitzar fallades o possibles fallades del sistema. En un context d'automatització com el definit, un sistema amb capacitat d'analitzar pot pendre les decisions adequades per remediar problemes detectats.

Autonomic Computing és un concepte que busca millorar sistemes reduint la necessitat de intervenció humana amb l'objectiu de crear sistemes auto-gestionables. El concepte data del 1997 quan el projecte “Situational Awareness System1” (SAS) va ser desenvolupat per DARPA per aplicacions militars, tot i que no va ser fins el 2001 que IBM va introduir la referència a *Autonomic Computing*. Es va desenvolupar el model MAPE-K on un agent intel.ligent perceb el seu entorn mitjançant sensors i, en funció de la informació obtinguda, el sistema determina les accions a executar. L'element gestionat representa qualsevol recurs de programari o maquinari

amb capacitat de comportment autonòmic al qual se li és assignat un gestor. Els sensors recolliran la informació de l'element gestionat (ex: utilització de disc, ample de banda, CPU, etc)
Els efectors són el encarregats d'executar els canvis al sistema per exemple, afegint servidors a un clúster o canviant la configuració d'un servei. (Hubscher and McCann, 2008)

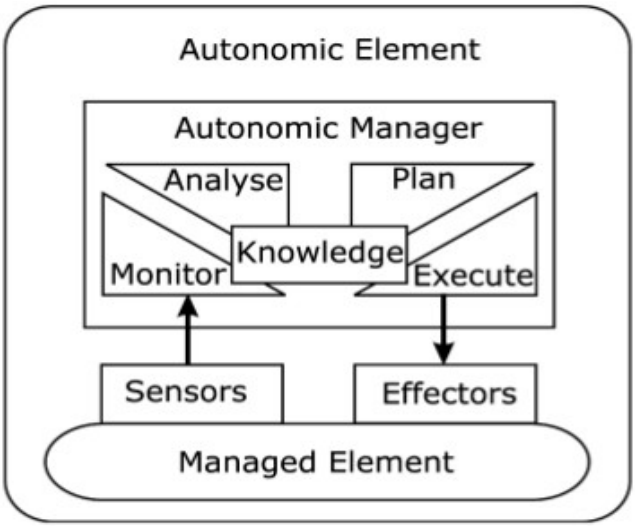

Diagrama de funcionament MAPE-K
Font: Hubscher and McCann, 2008

Una evolució del model inclou la creació d' una base ontològica que permeti recollir incidències passades i les accions que van tenir èxit amb l'objectiu de reaccionar a diversos canvis d'estat del sistema. En aquest cas es requereix un model semàntic que especifiqui per avançat el comportament que s'espera del sistema per assegurar que determinades accions s'executen, tot i que les accions no es defineixin. Com a consequència alguns canvis en serveis i vida del sistema succeixen: el comportament esperat ha de ser definit perquè la infraestructura porti a terme les accions necesàries. Per tant no es defineixen les accions tècniques si no el comportament desitjat. D'aquesta forma s'elimina el coneixement tècnic de l'administrador centrant-lo en els objectius de l'empresa. (González et al. 2009)
L'objectiu d'aquest tipus de model és reduir la pressió en l'administrador de sistemes quan es treballa en entorns grans, complexes i distribuïts a còpia d'automatitzar moltes de les tasques

habituals de gestió. Per contra, dotant del control tècnic al propi sistema pot crear un nivell d'abstracció que pot ser un obstacle entre l'administrador i els elements a gestionar, esdevenint en una pèrdua de control i manca de confiança en el sistema. En aquest sentit es proposa controlar les tasques una a una per l'administrador: aquest defineix una tasca de gestió de la mateixa manera que l'executaria en l'entorn i permet a la solució identificar els activadors que executarien la tasca. (Mclannon et al. 2013)

En aquest context pren rellevància el poder disposar de patrons per la implementació de cada tasca, per assegurar que aquesta es realitza de forma òptima utilitzant el menor nombre de recursos. Un fet rellevant que es posa de manifest en tots els casos és que el rol de l'administrador passa de gestionar equips a desenvolupar el codi que gestionarà els equips.En aquest context pren rellevància el poder disposar de patrons per la implementació de cada tasca, per assegurar que aquesta es realitza de forma òptima utilitzant el menor nombre de recursos. Un fet rellevant que es posa de manifest en tots els casos és que el rol de l'administrador passa de gestionar equips a desenvolupar el codi que gestionarà els equips.

2.4.2 Monitorització

Des del punt de vista d'un administrador, la monitorització és l'àrea que aporta major informació. La seva funció original era la d'establir si un servei funcionava, en l'actualitat la funcionalitat s'extén molt més. En la dècada dels 90 van aparèixer diverses solucions de monitorització sent Nagios una de les més populars. El seu funcionament és simple: mitjançant l'ús de plugins (scripts amb 4 possibles valors de sortida: *Critical, Warning, OK* i desconegut) es monitoritzen aspectes del sistema, tot ells visualizables des d'una interfície web. La solució estàndard incorpora plugins bàsics per la monitorització del sistema i amb l'ajut de la comunitat s'ha anat extenent fins a soportar centenars d'aplicacions i serveis. La característica que el va diferenciar de les solucions existents en aquell moment va ser la capacitat d'enviar alertes. (Dickson, 2013)

Actualment la monitorització s'extén a d'altres àreas. En una època on es poden emmagatzemar i accedir a quantitats enormes d'informació amb rapidesa, les mesures de la infrastructura prenen gran importància: per pendre decisions de capacitat (monitoritzant l'ample de banda utilitzat), de disseny (monitoritzant una interficíe pot identificar colls d'ampolla), d'escalabilitat (monitoritzant un clúster veiem que a l'afegir un node millora el rendiment) o entendre hàbits dels usuaris de la plataforma (monitoritzant el nombre de peticions veiem que hi ha més activitat al vespre).

En aquest sentit es presenten multitut de solucions en aquest camp (Nagios, Ganglia, Sensu, Graphite, mrtt, cacti, comandes de sistema, etc.) cadascuna d'elles especialitzada en diferents àrees (visualització de dades, generació d'alertes, gràfics, etc) i amb diferents tècniques per obtenir la informació:

- Plugins/Subprocés: scripts que utilitzaran la informació del sistema (llegint directament de /proc, executant crides al sistema (syscalls) o executant aplicacions (sar, top, dtrace, etc).
- Agents/SNMP/RPC: utilitzen agents que s'executen en l'equip a monitoritzar el qual comunicarà directament al servidor la informació rellevant.
- Logs: algunes eines obtenen informació llegint directament la informació dels logs del sistema/aplicació.

Totes les tècniques per obtenir informació poden ser automatitzades i per tant són subjectes a poder identificar patrons de desenvolupament.

2.4.3 Gestió de la configuració

En aquest context prenen relevància les eines de configuració de sistema, les quals presenten arquitectures molt similars. En general, cada eina de configuració de sistema proporciona una interfície a l'administrador de sistema on l'administrador defineix la configuració dels dispositius que gestionarà l'eina. Les especificacions introduïdes generaran la configuració de tots els dispotius que gestiona i el mateix sistema vetllarà perquè s'enforcin (Delaet et al. 2010)

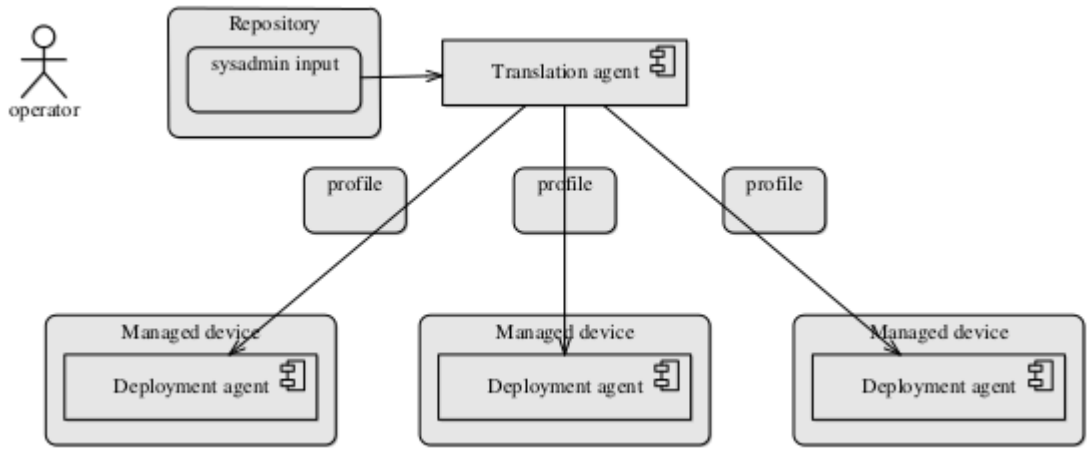

Diagrama de funcionament d'una eina de gestió de la configuració.

Font: Delaet et al. 2010

En aquests context diferents fabricant presenten les seves pròpies eines, per exemple:

- IBM Tivoli System Automation for Multiplatforms
- Microsoft Server Center Configuration Manager (SCCM)
- HP Server Automation System

En el camp opensource es presenten també opcions molt competitives que semblen liderar aquest sector. En destaquen 2 projectes: OpsCode-Chef i Puppet

Tots ells faciliten el desplegament de grans infrastructures en la gestió de la configuració, ajudant-se d'agents encarregats de forçar configuracions prèviament definides. La gestió de la configuració es presenta com a codi permetent a l'administrador desenvolupar les configuracions el que en determinades organitzacions pot ser un avantatge. Es pretén el que s'anomena "infrastructura com a codi" on l'administrador requereix cada cop més del desenvolupament de programari que li permeti gestionar els equips que administra. En el cas de Puppet, el funcionament es basa en definir procediments : "instal·la A abans que B" i B no s'instal·larà si A no està instal·lat.

Chef utilitza la modelització en el seu llenguatge, una recepte de com es defineix el sistema "Instal·lar B implica disposar de A". A part d'aquesta diferència substancial en l'enfoc, són solucions molt similars basades en agents de control.

3. Metodologia

L'objectiu ha estat identificar tasques comunes dins de l'administració de sistemes i veure quines millores es poden aplicar, a partir d' implementacions basades en patrons de programari coneguts i/o altres optimitzacions trobades en la literatura.

En els següents punts es detallaran primer patrons implementats en Python aplicables a les 3 accions principals trobades. S'ha utilitzat una estratègia de **Disseny i Creació**, realitzant una contribució al coneixement basada en la revisió de la literatura existent on, a partir dels següents punts, es defineix un

model aplicable a tasques d'administració. En alguns casos s'ha trobat més de una opció per la implementació de les tasques. Es realitzaran tests que generin dades quantitatives i permetin identificar els patrons més adients. En aquest sentit es pot incloure la recerca també dins la metodologia basada en **Experiments**.

Per l'estudi de casos d'ús s'opta per una recerca basada en **Documents**, principalment articles acadèmics, llibres científics i articles de publicacions/internet. Mitjançant taules i llistats, amb l'ús de dades nominals s'han establert:

- Àrees generals dins de l'administració que permeten trobar altres documents d'informació i classificar tasques subjectes a ser automatitzades. (Anàlisi, Monitorització i Gestió de la configuració)
- Tasques i accions subjectes a ser automatitzades.
- Requeriments de les accions.
- Optimitzacions específiques en *Python.*

4. Desenvolupament

La revisió de la literatura ha permés validar la manca de recerca en aquesta àrea específica. Com s'ha vist anteriorment, existeix abundant documentació de sistemes que realitzen tasques d'administració, però no se centren en la implementació específica de la tasca.

Per exemple, una tasca d'administració de sistemes és iniciar/aturar el servei de correu. A alt nivell la tasca es pot dividir en:

- Acció 1: Execució d'una comanda.
- Acció 2: Validar execució.

D'entrada ja veiem que aquest mètode es pot aplicar a qualsevol servei. A més, es pot aplicar a altres tasques (Matar un procés, eliminar un fitxer, copiar un arxiu…). Succeix de forma similar a la configuració d'un servei el qual pot implicar llegir/escriure un fitxer. D'aquesta manera es pot simplificar el nombre d'implementacions al treballar amb accions específiques.

4.1 Tasques

A partir de les tasques d'administració trobades al llibre "Unix and Linux System Administration Handbook" (Nemeth et al. 2010), s'ha creat un llistat de 68 tasques que serveixen de mostra per l'experiment.

L'anàlisi de les tasques ha revelat el següent:

- S'han identificat 3 accions comunes per realitzar tasques que poden resoldre totes les implementacions:
    - Executar una comanda: pot ser una crida de sistema, un script o un programa. Aquesta execució produirà un resultat que pot ser enviat als canals de sortida *stdout/stderr* i retornarà un *exitcode* quan finalitzi l'execució.
    - Llegir input: poden ser dels resultats d'un programa executat o un fitxer.
    - Escriure output: poden ser les sortides del programa executat o un fitxer.
- S'han detectat els següents tipus de fitxers amb els que les tasques d'administració interactuen:
    - Fitxers de configuració: són fitxers de text de desenes o centenars de línies habitualment petits (bytes o Kbytes). Poden requerir permisos de superusuari o d'un usuari concret. Durant el procés d'instal·lació/configuració poden requerir la creació d'un arxiu o afegir línies a un fitxer existent (per exemple, la configuració XML de màquines virtuals, la configuració de interfícies de xarxa, etc.)
    - /proc: en sistemes Linux trobem informació del kernel i paràmetres que es poden configurar. Aquests fitxers són petits i requereixen accés d'escriptura de superusuari (lectura és permesa en alguns casos).
    - Fitxers de sortida: algunes comandes mostren els resultats per pantalla (stdout). En un entorn automatitzat aquesta informació s'ha de capturar per ser analitzada. Si la comanda ho permet, es pot indicar un fitxer de sortida. En cas contrari la implementació haurà d'incloure un mecanisme per llegir de stdout
    - Fitxers de log: aquest tipus de fitxer de text inclou un nombre de línies més elevat que un fitxer de configuració i en conseqüència són de major mida (Mbytes, Gbytes). Per contra aquest tipus de fitxer acostumen només ha ser llegits i no escrits.

4.2 Automatització

Un cop identificades les tasques i les àrees a treballar, caldrà automatitzar les accions. Si la infrastructura és codi, aquest ha de ser clar, fàcilment modificable. Python es caracteritza per la seva simplicitat i elegància alhora de desenvolupar codi. Si bé no és el més eficient dels llenguatges, és capaç d'implementar tasques administratives amb rendiment adequat. Els exemples presentats són implementats i optimitzats per aquest llenguatge. En grans infraestructures és habitual reduir al mínim el nombre de paquets a mantenir, tant per auditories de seguretat com per eficiència del manteniment (Bellovin et al, 2009). En aquest sentit, es treballa amb els mòduls base de Python, en la versió 2.7. Tal i com s'ha observat en [articles], és recomenable utilitzar agents que executaran les tasques en el sistema de forma local. Per contra algunes tasques d'anàlisis poden requerir la connexió a centenars d'equips. En aquest sentit s'ha afegit una tasca de connexió als equips.

4.2.1 Executar Comanda

Python incorpora diferents llibreries per executar processos externs (os.system, os.spawn, os.popen, commands…). A partir de la versió 2.4 s'introdueix la llibreria *subprocess* amb l'objectiu de consolidar la funcionalitat de les anteriors llibreries i en conseqüència remplaçar-les. En la versió 2.7 aquest és el mètode recomanat a la documentació de Python. Entre altres coses permet executar processos, enviar i rebre informació de *stdin*, *stdout* i *stderr* i obtenir el *exitcode* del procés executat. Sileika (ProPython System Administration) destaca la classe *subprocess.Popen()*. Al constructor de la classe se li passen paràmetres, en forma de llista o de string. *Popen* interpreta el primer element de la llista com una comanda i els següents com paràmetres, millorant la seguretat ja que només s'ha d'auditar el primer element de la llista.

```
#Executar Process
cmd = 'comanda arg1 arg2'
p = subprocess.Popen(cmd.split())
```

En el cas de inicialitzar la opció *shell=True*, es pot passar directament un string amb les comandes a executar, amb el risc de ser vulnerables a injecció de codi. (ref: Python documentació). Sileika destaca la limitació a utilitzar només la *shell* pre-determinada en el sistema, el que es pot solventar passant les comandes de shell com a llista directament a *Popen*

(per exemple ['/bin/bash', ' -c', 'echo "exemple"']). També analitza l'argument *preexec_fn* el qual permet executar una funció abans que el nou procés comenci (el procés és cridat després del fork() però abans del exec()). Pot ser útil per canviar el UID d'un procés, per exemple quan es planifica l'arrencada d'un servei al cron de *root* però es vol executar amb permisos diferents de superusuari.

```
#Canviar UID
uid = 'UID num'
cmd = 'comanda arg1 arg2'
p = subprocess.Popen((cmd.split()), preexec_fn =
os.setuid(uid))
```

En el cas de processos que no finalitzen de forma immediata, Sileika proposa utilitzar la classe *poll* per establir si el procés és en execució ('None') o ha finalitzat retornant el *exitcode*. La classe *pid* permetrà establir l'identificador de procés en el sistema.

```
#Espera a que un proces finalitzi
p = subprocess.Popen(cmd.split())
while True:
    rc = p.poll()
    if rc is None:
      print "%s en execucio" % p.pid
    else:
        print "%s ha finalitzat amb exitcode %s" %
(p.pid, rc)
        break
```

El mètode *wait()* permet bloquejar el control de l'aplicació fins que el procés finalitzi, estalviant la iteració de control però té el risc de *deadlock* si el procés produeix molta informació de sortida. Per aquest motiu es prefereix el mecanisme de control proposat prèviament.
Per interaccionar amb l'entrada i la sortida des de l'aplicació, Sileika planteja assignar *stdin*, *stderr* i *stdout* a *subprocess.PIPE*, proposant dues opcions per obtenir les dades:

- Utilitzant *communicate()*, es retorna una tuple amb dos strings el primer conté el canal de sortida i el segon el de error. Per passar el input al canal d'entrada, es proposa *communicate(input="string")*. Es destaca que aquest mètode cparrega tota les dades en memòria de manera que grans quantitats d'informació podrien causar resultats inesperats (depenent dels limits del sistema).

```
#Mètode 1 - communicate():
p = subprocess.Popen(cmd.split(),
stdout=subprocess.PIPE)
out_dades, err_dades = p.communicate()
```

- Com alternativa es proposa llegir i escriure directament al *File Object* disponible mitjançant la classe *Popen*: *stdin*, *stdout* i *stderr*. Aquest mètode no carrega totes les dades en memòria de cop.

```
#Mètode 2 - Llegir i escriure directament al file
object
p =  subprocess.Popen(cmd.split(),
stdout=subprocess.PIPE)
for line in p.stdout:
    print line
```

4.2.2 Llegir/Escriure Fitxer

Python disposa del mecanismes de *File Objects* per tractar amb fitxers. Amb el mètode open() es crea l'objecte que servirà de interfície per interactuar amb el fitxer. A "Python for Unix and Linux System Administration", Gift descriu 3 mecanismes per llegir un fitxer: *Read()* permet llegir el contingut del fins el final, *Readline()* llegeix una línia mentre que *Readlines()* llegeix també una línia amb la diferència de com es gestiona la memòria. *Read()* i *Readlines()* carreguen el fitxer en memòria abans de llegir.

```
#Mètode 1 - Carrega en memòria
fitxer = open('arxiu.txt', 'r')
fitxer.readlines()
```

Especificant *size* (en bytes) es llegirà només la informació continguda en el nombre de bytes demanat, encara que impliqui llegir una línia de forma parcial, un problema en cerques de *strings* a fitxers. En un context devops l'analisis automàtitzat de logs podria produir resultats incorrectes degut al truncament de les línies. Per aquest motiu no es considera aquesta opció.

A la documentació de Python es proposa iterar sobre el objecte, el qual és un iterador. Segons la documentació, és ràpid, eficient en memòria i presenta un codi clar.

```
#Mètode 2 - Eficient en memòria
with open("arxiu.txt") as fitxer:
    for linia in fitxer:
        if "string a trobar" in linia:
            print linia
        else:
            pass
```

El mateix *File Object* proporciona el mètode w*rite()* per escriure a fitxer. És important que l'accés al *file object* s'hagi creat afegint permisos d'escriptura.

```
#Mètode 1 - File Object per tractar amb arxius
fitxer = open('arxiu.txt', 'w')
fitxer.write('string')
fitxer.close()
```

Els materials consultats semblen coincidir amb l'ús de *File objects* per accedir a fitxers. Sileika proposa a més accedir a l'entrada i sortida a baix nivell mitjançant les classes de fitxers al mòdul os. No especifica cap avantatge concret, només es mostra com a opció.

```
#Mètode 2 - E/S de baix nivell per tractar amb arxius
fitxer = os.open('arxiu.txt', os.O_CREAT|os.O_WRONLY)
subprocess.Popen('date', stdout=fitxer)
fitxer.close()
```

5. Experiments

S'han identificat accions que es poden realitzar amb implementacions diferents. Amb els experiments es generararn dades empíriques que permetin realitzar anàlisis de rendiment determínistic, el qual es basa en prendre mesures precises de intérvals de temps entre crides de funció, retorns i excepcions que són monitoritzades durant l'execució del codi. En *Python* l'intèrpret és actiu durant l'execució i permet obtenir aquestes dades afegint poca càrrega de procés adicional al sistema.

5.1 Velocitat d'execució

Per la realització dels tests, un primer factor que es té en compte és el temps d'execució total acumulat. Estadístiques de temps acumulades permeten identificar errors d'alt nivell a la selecció d'algoritmes i permetran establir quin codi és més ràpid. Una limitació és la precisió d'aquesta informació, determinada pel rellotge intern que és de 0,001 segons en el sistema.
Per l'anàlisi s'utilitza el mòdul de tercers *cProfile*, el qual utilitza la mateixa interfície que el mòdul de la llibreria estàndard Profile però sense afegir una elevada sobre-càrrega al sistema.

5.2 Ús de memòria

Un segon factor que és té en compte és l'ús de memòria. El mòdul *memory_profiler* permet analitzar la quantitat de memòria utilitzada per l'algoritme durant l'execució. Aquest mètode detalla la memòria màxima utilitzada per línia de codi, permetent així identificar la funció/mètode que consumeix més memòria i en quin punt es produeix la càrrega/alliberament. Per agilitzar l'execució i minimitzar la sobre-càrrega al sistema, s'instal·la el mòdul *psutil*. En aquest cas les mesures es prenen des de l'entorn *iPython* per minimitzar les modificacions de codi.

5.3 Disseny de les proves

S'estudiaran 3 aspectes analitzats durant el desenvolupament dels patrons, presentant dues implementacions diferents cada un d'ells. L'objectiu és triar un mètode per cada aspecte a partir de les dades obtingudes. A partir de 100 linies extretes d'un fitxer syslog, s'han generat arxius de 100, 1000, 100.000, 1.000.000 i 10.000.000 de línies. S'han generat també arxius de 100MiB i 1GiB. Els aspectes que es volen observar són:

1 - Ús de *subprocess.communicate()* vs redirecció directe a *stdout*.

Es simula l'execució de *grep* sobre un fitxer de *syslog* de 100MiB i 1GiB. Els resultats trobats s'enllacen amb *communicate()* o redireccionant directament a *stdout*.

2 - Readlines() vs Iterador

Es llegeix un fitxer *syslog* de 100, 1000, 100.000, 1.000.000 i 10.000.000 línies amb aquest dos mètodes.

3 - Ús de File Objects vs E/S de baix nivell

S'escriu un fitxer de text de 100MiB i 1GiB amb aquest dos mètodes.

En tots els casos es mesura el temps acumulat d'execució i l'ús de memòria, dades empíriques que serviran per valorar l'ús de recursos de cadascun dels mètodes.

6. Resultats

S'han generat taules de resultats amb els valors obtinguts, calculats de la següent manera:

- Temps d'execució: per minimitzar l'error en la mesura del temps d'execució es prenen 10 mostres i es calcula:
    - Moda, el valor que es repeteix més vegades (arxius de 100 i 1000 Linies)
    - Mitja aritmètica (resta de casos)
- Memòria utilitzada: es pren el valor més alt de *Mem Usage* ja que serà el màxim de memòria requerit per l'algoritme per completar l'execució.

Subprocess.Communicate() vs stdout

Communicate()	1	2	3	4	5	6	7	8	9	10	Temps	Memòria
100M	0.193	0.187	0.185	0.207	0.191	0.234	0.186	0.189	0.191	0.184	0.1947	16.9MiB
1G	1.792	1.709	1.696	1.771	1.698	1.967	1.738	1.73	1.71	1.694	1.7505	55.9MiB
stdout	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Temps**	**Memòria**
100M	0.203	0.206	0.248	0.225	0.227	0.2	0.214	0.197	0.201	0.208	0.2129	12.5MiB
1G	1.901	1.896	2.046	1.887	1.876	1.894	1.894	1.894	1.903	1.89	1.9081	13.2MiB

S'observa que en temps d'execució comunicar les dues sortides directament és més ràpid la redirecció a *stdout*. Per contra el segons mecanisme és més eficient en la gestió de memòria: en el primer cas els resultats són carregats en memòria per complet, mentre el segon cas carrega en memòria només la part

requerida. Durant un incident, en un context d'anàlisis, durant un incident l'automatització podria no executar-se perquè els resultats generats són més grans que la memòria disponible al sistema. Amb la opció més eficient, la memòria s'ajustaria de forma automàtica permetent l'execució. En aquest sentit l'autor es decanta per la opció més eficient en memòria, ja que l'altre opció està limitada a la memòria total del sistema i als resultats

Readlines() vs Iterador

Readlines()	1	2	3	4	5	6	7	8	9	10	Temps	Memòria
100L	0.001	0.001	0.001	0.002	0.001	0.001	0.001	0.001	0.001	0.001	0.001	12.4MiB
1000L	0.005	0.006	0.004	0.009	0.004	0.005	0.004	0.003	0.009	0.005	0.005	12.8MiB
100000L	0.718	0.11	0.047	0.05	0.056	0.06	0.044	0.058	0.044	0.041	0.1228	24.2MiB
1000000L	0.632	0.406	0.423	0.425	0.408	0.405	0.417	0.415	0.42	0.414	0.4365	125.5MiB
10000000L	-	-	-	-	-	-	-	-	-	-	-	1119MiB
Iterador	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Temps**	**Memòria**
100L	0.001	0	0	0	0	0	0	0	0	0	0	12.4MiB
1000L	0	0	0	0	0.001	0	0	0	0	0	0	12.4MiB
100000L	0.031	0.034	0.039	0.032	0.036	0.035	0.036	0.036	0.034	0.032	0.0345	12.4MiB
1000000L	0.329	0.307	0.353	0.313	0.322	0.311	0.315	0.329	0.326	0.328	0.3233	12.4MiB
10000000L	7.636	5.957	5.931	5.601	5.742	5.736	5.787	5.611	5.67	5.642	5.9313	19.2MiB

En aquest cas l'ús de l'iterador és superior en tots els casos: millor velocitat d'execució i millor gestió de la memòria (només carrega en memòria la part que requereix). Es pot comprovar que l'últim test no es va poder completar per la opció de *readlines()*, degut a que el procés requeria més memòria del sistema de la disponible. L'autor, coincidint amb la literatura, es decanta per l'ús de l'iterador.

File Object vs E/S de baix nivell

File Object	1	2	3	4	5	6	7	8	9	10	Temps	Memòria
100M	0.004	0.005	0.006	0.005	0.011	0.009	0.005	0.007	0.008	0.009	0.0069	12.4MiB
1G	0.005	0.004	0.005	0.466	0.008	0.779	0.025	0.396	0.132	0.415	0.2235	12.9MiB
E/S baix nivell	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Temps**	**Memòria**
100M	0.004	0.004	0.007	0.005	0.006	0.004	0.005	0.006	0.004	0.006	0.0051	12.4MiB
1G	0.004	0.007	0.005	0.004	0.009	0.185	0.217	0.004	0.077	0.109	0.0621	13.1MiB

Per temps d'execució veiem que E/S de baix nivell accedeix més ràpid al fitxer a còpia d'incrementar lleugerament l'ús de memòria quan treballem amb fitxers de 1G. Amb fitxers de 100M els temps d'execució i ús de memòria són pràcticament idèntics. En aquest sentit la tria del patró adequat recaurà en la prioritat de la tasca a realitzar. Si es disposa d'un accés a disc ràpid, pot ser interessant treballar a baix nivell per aprofitar el màxim rendiment de E/S. S'ha de tenir en compte que aquest mètode genera un *file descriptor* enlloc d'un objecte iterable el que pot fer més complexe treballar amb les dades. En aquest sentit l'autor es decanta per l'ús de *File Objects* pel seu rendiment moderat i la seva flexibilitat per treballar amb les dades en àmbits generals.

7. Limitacions

L'àmbit de la recerca s'ha centrat en entorns *devops* amb plataformes grans basades en Unix/Linux. Per tant, les tasques proposades parteixen de la base de l'execució en entorns amb aquests sistemes operatius. A més, es planteja l'execució d'aquestes de forma local (no remota) mitjançant l'ús d'agents ja que aquesta és la forma habitual trobada en la literatura.

8. Conclusions

Amb el canvi de paradigma que suposa el Cloud Computing, el rol de l'administrador de sistemes està evolucionant cap a tasques més orientades al desenvolupament. Si es poden definir receptes per administrar un equip, llavors la mateixa recepta es pot aplicar de forma iterativa a més equips. Però l'aplicació de la recepta de forma massiva suposa nous reptes per l'adminitrador: requereix de noves eines per orquestrar els canvis. I aquestes eines s'han d'integrar amb les tasques tradicionals d'administració. En un era on la infraestructura es presenta com a codi, l'automatització és capital. L'administrador continua implementant les mateixes tasques a gran escala el que afegeix complexitat al rol. Es requereix de eines més sofisticades que permetin gestionar tota la infraestructura, desenvolupaments més complexes: l'administrador es converteix en programador. Ja no gestiona sistemes, sino que gestiona el codi que gestiona els sistemes, tot i que les tasques són les mateixes.

En aquest sentit la recerca ha permés identificar les tasques comunes que poden ser automatitzades, establint patrons per la seva implementació que es puguin adaptar en entorns de desenvolupament amb interrupcions constants.

9. Referències

[1] Markus C. Huebscher, Julie A. McCann , “A survey of Autonomic Computing—Degrees, Models and Applications ”, ACM Computing Surveys (CSUR), Volume 40 Issue 3, Article No. 7 , August 2008

[2] Barry McLarnon, Philip Robinson, Peter Milligan, Paul Sage , “An Interative Approach to Trustable Systems Management Automation and Fault Handling ”, Journal of Network and Systems Management. Springer Science+Business , November 2013

[3] Steven M. Bellovin , Randy Bush “Configuration Management and Security” IEEE Journal on selected areas in comunnications. Vol. 27, No.3, April 2009

[4] Thomas Delaet, Wouter Joosen, Bart Vanbrabant , “A Survey of System Configuration Tools ”, Proceedings 24th Large Installation System Administration Conference (LISA’10), San Jose, USA , November 2010

[5] Juan M. González, José A. Lozano, A. Castro , “Autonomic System Administration. A Testbed on Autonomics “, Fifth International Conference on Autonomic and Autonomous Systems, Valencia, Spain, April 2009

[6] Marc Merlin , “Live Upgrading Thousands of Servers from an Ancient Red Hat Distribution to 10 Years Newer Debian Based One”, Proceedings of the 27th Large Installation System Administration Conference (LISA’13), Washington DC, USA , November 2013

[7] Nicole F. Velasquez, Suzanne P. Weisband “Work practices of system administrators: implications for tool design“ Proceedings of the 2nd ACM Symposium on Computer Human Interaction for Management of Information Technology , November 2008

[8] Ingo Weber, Hiroshi Wada, Alan Fekete, Anna Liu, Len Bass , “Supporting Undoability in Systems Operations ”, Proceedings of the 27th Large Installation System Administration Conference (LISA’13). Washington DC, USA , November 2013

[9] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt , “Past, Present and Future Trends in Software Patterns ”, IEEE Software, 24(4), 31 , July 2007

[10] Mark Summerfield , “Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns ”, AddisonWesley Professional, Portland, USA , February 2014

[11] Evi Nemeth, Garth Snyder, Trent R. Hein, Ben Whaley , “Unix® and Linux® System Administration Handbook (Fourth Edition) ”, Prentice Hall , July 2010

[12] Thomas A. Limoncelli, Christina J. Hogan, Strata R. Chalup, “The Practice of System and Network Administration”, Addison-Wesley Professional, March 2007

[13] Rytis Sileika “Pro Python System Administration, Second Edition”, Apress, November 2014

[14] Gabriele Lanaro “Python High Performance Programming”, Packt Publishing, December 2013

[15] “Python 2.7.9 Documentation”, https://docs.python.org/2.7/, December 2015

[16] Chris Haddad , “DevOps amplifies your open source credentials ”, Opensource.com , April 2014

[17] Jeff Knupp , “How *DevOps* is killing the developer ”, http://jeffknupp.com , April 2014

[18] Paul Duvall, “Agile DevOps: Infrastructure Automation”, http://ibm.com, September 2012

[19] Peter Wayner , “Puppet or Chef: The configuration management dilemma ”, InfoWorld.com , March 2013

[20] Caskey L. Dickson, “A working Theory of Monitoring”, LISA 2013, November 2013