TSR 2019/20

TEMA 4 – DESPLEGAMENT DE SERVEIS. DOCKER

Guia de l'alumne

Objectius

- Descriure els aspectes a considerar durant el desplegament d'aplicacions distribuïdes.
- Mostrar els problemes derivats de les dependències i com resoldre'ls.
- Discutir exemple concrets de desplegament.

CONTINGUT

1 INTRODUCCIÓN

L'objectiu de tota aplicació distribuïda és proporcionar servei als seus usuaris. No obstant això, aqueix objectiu no pot aconseguir-se tan prompte com s'hagen desenvolupat els components de l'aplicació perquè aquests necessiten ser instal·lats en diversos ordinadors i ser iniciats abans que els seus clients puguen accedir a ells.

Citant a Carzaniga et al. [1]...

> *"Software applications are no longer stand-alone systems. They are increasingly the result of integrating heterogeneous collections of components, both executable and data, possibly dispersed over a computer network. Different components can be provided by different producers and they can be part of different systems at the same time. Moreover, components can change rapidly and independently, making it difficult to manage the whole system in a consistent way. Under these circumstances, a crucial step of the software life cycle is* deployment*—that is,* ***the activities related to the release, installation, activation, deactivation, update, and removal of components****, as well as whole systems."*

Per tant, el desplegament no consisteix únicament en la instal·lació i activació dels programes sinó també en la seua edició (o publicació), actualització, desactivació i desinstal·lació. De fet, el desplegament conté totes les tasques de la gestió del cicle de vida d'una aplicació informàtica que succeeixen després del seu desenvolupament.

Així, amb les fases d'anàlisi, disseny i desenvolupament s'hauran obtingut alguns components de l'aplicació, però aquestes peces han de desplegar-se per a arribar a ser un servei. Esquemàticament:

Programes + Desplegament --> Serveis

Revisem què ocorre quan volem desplegar una aplicació normal d'escriptori:

1. Aqueixa aplicació ha sigut editada per alguna empresa o organització. Pot ser descarregada una vegada s'haja comprat o pot haver sigut editada en algun suport extraïble (p. ex., en un DVD-ROM).
2. Ara, hem d'instal·lar-la. A primera vista, només necessitem copiar els seus fitxers executables en alguna carpeta (o conjunt de carpetes) en el nostre ordinador. No obstant això, aquest pas és més complex del que sembla. A més de copiar els fitxers executables i una altra informació de configuració, diverses dependències han de ser resoltes (p. ex., algunes biblioteques estàndard, o potser dependents del llenguatge de programació utilitzat, poden ser necessàries o algunes eines han de trobar-se en el sistema per a executar o instal·lar l'aplicació). Si alguna d'aquestes dependències no poguera resoldre's, les peces corresponents haurien de cercar-se i instal·lar-se correctament per a reprendre el desplegament.
3. Finalment, quan el pas anterior haja sigut completat satisfactòriament, la nova aplicació podrà ser iniciada i executada.
4. Depenent del tipus d'aplicació, en el seu primer inici (i en comprovacions periòdiques posteriors) aquests programes s'activaran comprovant enfront d'un servidor de llicències remot que tinga un identificador o tiquet d'activació vàlid.

Ja que el segon pas pot arribar a ser complex, diversos sistemes operatius faciliten una API i un conjunt d'eines per a desenvolupar instal·ladors d'aplicacions. Per exemple, en Windows existeix una API [2] per a Windows Installer [3] que ha de conèixer-se per a construir els paquets MSI (que són els utilitzats per a instal·lar aplicacions en els sistemes Windows). Entre les eines que acompanyen aquesta API trobem WiX (Windows Installer XML Toolset) [4] que serà l'eina a utilitzar per a construir els paquets MSI. El paquet MSI emmagatzema els components de l'aplicació i coneix quines altres dependències han de ser resoltes durant la instal·lació, reclamant la seua descàrrega i instal·lació en cas que no estigueren presents en l'equip on es faça el desplegament. Amb aquest suport, la instal·lació i actualització d'aplicacions pot automatitzar-se.

Existeixen eines i API similars en altres sistemes operatius. Per exemple, les distribucions Linux Fedora utilitzen paquets RPM (RPM Package Manager) [5] i les eines DNF [6] per a aquesta fi.

2 SERVEIS

En la secció anterior hem vist que per a facilitar serveis serà necessari desplegar aplicacions. En el cas dels sistemes distribuïts, les aplicacions estan formades per múltiples components, és a dir, per programes que implanten una funcionalitat determinada. Els components han de ser autònoms. Això significa que cadascun d'ells facilitarà, al seu torn, serveis als altres components. Amb un disseny adequat, els components seran reutilitzables i proporcionaran funcionalitats d'ús general que podran ser aprofitades en altres aplicacions futures.

Per tant, si es consideren les tasques de desplegament, un component...

- És la unitat de desplegament d'aplicacions distribuïdes. Cada component és un programa que pot dependre d'altres components per a construir serveis de major capacitat. A l'hora del desplegament, el codi de cada component es manté normalment en un paquet (o BLOB) independent.

- Es pot instanciar tantes vegades com es necessite (generant una rèplica del component en cadascuna de les seues instàncies) per a suportar la càrrega de treball prevista.

Quan es despleguen múltiples instàncies de cada component, llavors cada instància...

- Pot ser iniciada o parada independentment de les altres instàncies. Per regla general, s'iniciaran noves instàncies quan la càrrega del component s'incremente i es pararan instàncies existents quan no hi haja suficient càrrega a repartir entre elles. Així els components poden adaptar-se dinàmicament al nivell actual de càrrega de treball, minimitzant l'ús de recursos i el consum energètic quan es reduïsca la càrrega i usant una quantitat ajustada de recursos quan la càrrega cresca.
- Pot fallar independentment de les altres, ja que cada instància hauria d'instal·lar-se en un ordinador diferent. Les instàncies solen fallar quan hi ha algun problema en el seu entorn, com puga ser una fallada en l'ordinador que les mantinga.

La **Figura 1** mostra un exemple de desplegament d'un servei distribuït. En ell, el servei consta de tres components (C1, C2 i C3). Cadascun té un nombre diferent d'instàncies. C1 manté tres instàncies, C2 té dues i C3 només té una.

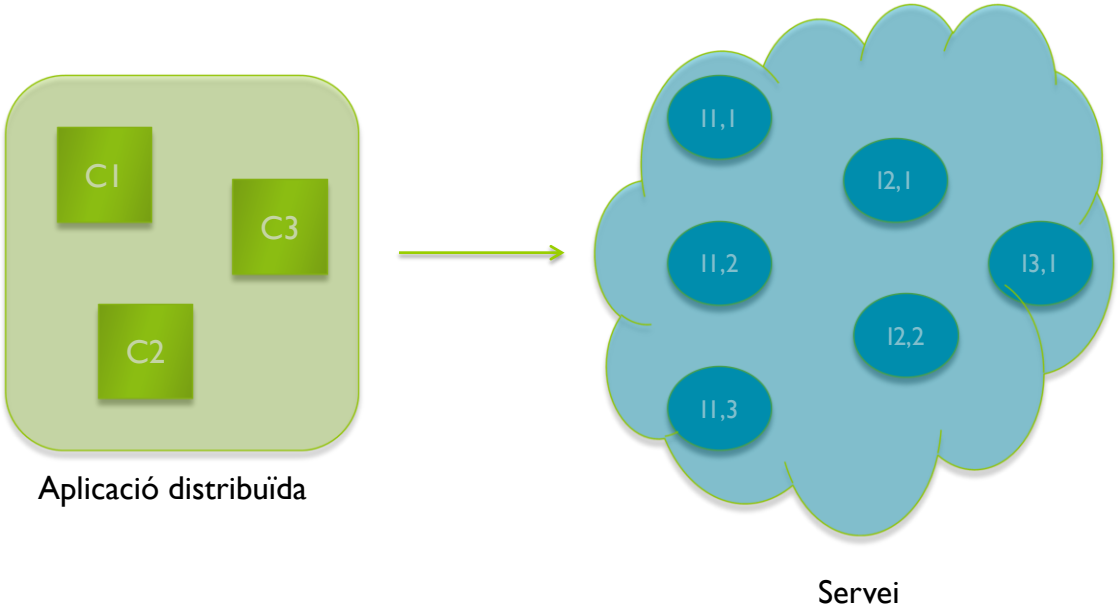


Figura 1. Desplegament d'un servei distribuït

Si el disseny de l'aplicació ha sigut acurat, els tres components estaran feblement acoblats. Això significa que quan una operació del component C_i siga invocada no es necessitarà invocar a altres components per a servir-la o, en cas que se'n necessitara algun, es farien molt poques invocacions i aquestes no implicarien l'intercanvi de molta informació. D'aquesta manera, l'administrador tindrà llibertat durant el desplegament per a triar en quin ordinador s'instal·larà cada instància.

Si en lloc d'això s'hagueren utilitzat components fortament acoblats tindríem un mal disseny. Per exemple, imaginem que els components C2 i C3 tingueren una interdependència forta (és

a dir, quan s'invoque una operació en un d'ells, el seu servei implicarà la invocació de diverses operacions de l'un altre component i en cada invocació es necessitarà transferir molta informació). En aquest cas, cada instància de C2 estaria fortament acoblada amb cada instància de C3. Per això, estaríem condemnats a desplegar aquests components a l'una; és a dir, si la primera instància de C2 s'instal·lara en un node N1 llavors la primera instància de C3 també hauria d'instal·lar-se en N1, si la segona instància de C2 s'instal·lara en N2, la segona instància de C3 també estaria en N2, i així successivament. Aleshores, C3 hauria de tindre el mateix nombre d'instàncies que C2 i qualsevol variació en el nombre d'instàncies en algun dels dos components obligaria a fer el mateix canvi en l'altre component.

2.1 Acords de nivell de servei (SLA)

Després de desplegar un servei, aquest pot ser utilitzat per múltiples usuaris. En cas de serveis distribuïts, el proveïdor de serveis sol ser una empresa especialitzada i els seus clients poden ser també empreses amb els seus respectius usuaris finals (que poden ser interns o externs a l'empresa client). En l'àrea dels serveis distribuïts se solen establir contractes o acords de nivell de servei (SLA, per les seues sigles en anglès) entre el proveïdor de serveis i els seus clients.

En un SLA es consideren múltiples aspectes, relacionats amb:

- La funcionalitat facilitada. S'ha d'assegurar que el servei facilitat pel proveïdor s'ajusta a les necessitats i requisits establits pels seus clients.
- Rendiment. Com el rendiment sol dependre de la càrrega, els clients sol·licitaran un nivell mínim de rendiment, però el proveïdor podrà també sol·licitar als clients que no excedisquen cert nivell màxim de càrrega.
- Disponibilitat. El proveïdor haurà de garantir que el servei estarà disponible quan els clients ho necessiten. Aleshores, un nivell mínim de disponibilitat (expressat com el percentatge de temps en què el servei estarà accessible, responent tota sol·licitud rebuda) haurà d'establir-se en el SLA.

Per a reflectir aquests tres aspectes (i qualssevol altres que es consideren rellevants) es defineixen algunes mètriques i els seus rangs de valors acceptables (coneguts com a objectius de nivell de servei, o SLO, per les seues sigles en anglès) han de ser acordats entre proveïdors i clients.

En el camp dels serveis distribuïts, una de les primeres especificacions de SLA va ser generada per IBM per als serveis web [7].

2.2 Tipus de servei

En l'àrea informàtica es poden distingir dos tipus principals de serveis:

- **Efímers**: Quan els serveis només es necessiten durant sessions breus, normalment interactives. Els serveis efímers són implantats per aplicacions d'escriptori a desplegar en ordinadors personals, utilitzats en sessions per part d'un únic usuari. Exemples: processadors de text, navegadors web, intèrprets d'ordres...
- **Persistents**: Aquests serveis necessiten estar contínuament disponibles i seran utilitzats concurrentment per multitud d'usuaris mitjançant connexions remotes. És el

cas habitual en els serveis distribuïts. Exemples: banca electrònica, administració electrònica, comerç electrònic, etc.

3 DESPLEGAMENT

En el desplegament inicial (és a dir, en l'edició, instal·lació i inici) d'una aplicació distribuïda es poden distingir els següents passos:

1. Decidir quins serveis dependents seran utilitzats quan l'aplicació s'estiga executant. En aquest pas necessitem establir les dependències que sorgiran quan els components de la nostra aplicació comencen a funcionar. El SLA per als altres serveis ha de ser considerat en aquest pas, establint els seus SLO i analitzant els seus costos.
2. Decidir quins components s'utilitzaran. Durant l'etapa de disseny s'haurà delimitat un conjunt de components. Alguns d'ells poden proporcionar funcionalitat opcional o implantaran solucions alternatives per a un mateix problema. Durant aquest pas, es triarà el conjunt de components a desplegar, depenent dels requisits dels seus usuaris. A més, l'administrador haurà de decidir quantes instàncies d'aquests components caldrà instal·lar inicialment.
3. Decidir en quin conjunt de nodes s'executarà cada component. Això dependrà de si existeixen alguns components fortament acoblats (aspecte a evitar, com ja hem explicat anteriorment), de la capacitat de cada node i dels requisits d'execució que impose cada tipus de component.
4. Portar el codi d'un component als nodes en els quals les instàncies del component hagen de ser executades. Hi ha eines (facilitades per la plataforma, el middleware o el sistema operatiu) que automatitzen la transmissió dels fitxers fins als nodes, així com la seua instal·lació parcial.
5. Decidir l'ordre en què els components seran iniciats. Ja que els components poden tindre algunes interdependències, les dependències han de ser considerades per a establir l'ordre d'inici.
6. Configurar cada component. Les configuracions depenen, de nou, de les dependències que existisquen. Per exemple, si el component A necessita utilitzar operacions dels components B i C, els punts d'accés (o "endpoints") de B i C han de ser comunicats a A, fixant valors per a un conjunt de paràmetres en el seu fitxer de configuració.
7. Utilitzar el sistema operatiu de cada node per a iniciar les instàncies dels components. Això ha de fer-se d'acord amb l'ordre d'inici decidit al pas 5.

Però el desplegament no solament consisteix a aquestos passos. Una vegada el servei estiga en execució, hi haurà altres tasques relacionades amb el desplegament que hauran de fer-se. Aquestes altres tasques són:

- **Gestió de les fallades** en les instàncies dels components. Mentre s'estiga executant un component, algunes de les seues instàncies podran fallar. Les fallades han de romandre ocultes per a la resta de components (i per als usuaris, en cas que el component proporcione la interfície d'accés al servei). Per a fer això s'utilitzaran tècniques de replicació i mecanismes de detecció de fallades. Això implica que les rèpliques defectuoses seran parades i reiniciades una vegada el defecte o error que

haja generat la fallada haja sigut corregit. Addicionalment, durant l'interval de recuperació, la resta de rèpliques hauran pogut modificar el seu estat i les modificacions han de ser propagades a la rèplica que s'estiga recuperant perquè aquesta les considere i aplique, utilitzant protocols de recuperació adequats.

- **Actualització del *programari*** (també coneguda com a "reconfiguració del *programari*"). De tant en tant els components necessiten ser actualitzats. Algunes raons per a fer això són: (1) la detecció d'errors en els programes, necessitant una nova edició d'aquests programes que corregisca els errors detectats, (2) els requisits dels usuaris poden canviar i els programes han de ser adaptats per a satisfer-los, (3) noves tecnologies (més eficients) han pogut sorgir, amb les quals seria possible millorar el rendiment o minimitzar el consum de recursos dels components, etc. Si això condueix a una actualització dels components, els seus SLA han de ser reconsiderats durant l'actualització. Aleshores, com els components estaran replicats, hauria de dissenyar-se algun procediment que permeta a ambdues versions del *programari* (l'antiga, a retirar, i la nova, a desplegar) coexistir mentre es transferisca l'estat des de la vella a la nova versió, sense interrompre el servei. Existeixen algunes solucions d'aquest tipus, descrites en [8].
- **Mantindre l'escalabilitat**, adaptant-la a la càrrega (és a dir, proporcionar *elasticitat* [9]). Quan s'incremente la càrrega, els components del servei han d'incrementar la seua capacitat de servei ("scale out"), iniciant més instàncies. Per contra, quan la càrrega decresca, algunes de les instàncies prèviament iniciades hauran de parar ("scale in"). En cas contrari, s'estarien utilitzant més recursos dels realment necessaris. Aquesta escalabilitat adaptativa (o elasticitat) necessita un subsistema de monitoratge adequat que vigile els valors actuals d'algunes mètriques (p. ex., el temps de resposta) i que reaccione quan els valors d'aquestes mètriques excedisquen certs llindars (p. ex., afegir una nova instància si el temps de resposta superara els 50 ms i eliminar alguna de les existents si el temps mitjà de resposta fóra inferior a 2 ms).

3.1 Entrades inicials

En la Secció 1 vam veure que per a desplegar una aplicació d'escriptori es necessitava construir cert paquet de *programari*. Aquest element es desplegarà utilitzant alguna eina dependent del sistema operatiu subjacent. Per a desplegar una aplicació distribuïda es necessitarà utilitzar més d'un ordinador. Per tant, l'escenari resultant serà bastant més complex i altres elements seran necessaris per a desplegar un servei. Revisem quines seran les entrades a considerar en aquest procés:

- **Aplicació distribuïda**. Bàsicament, una aplicació distribuïda consta d'un conjunt de programes (un per component) i un conjunt de dades de configuració. En una revisió detallada, podríem distingir aquests elements:
    - Cada component té el seu propi BLOB de codi; és a dir, un fitxer (o part d'un fitxer) que manté el programa a executar pel component.
    - Una plantilla de configuració per cada component. A més del seu BLOB, cada component necessita alguna informació de configuració. Aquesta informació s'utilitzarà per a emplenar una plantilla de configuració, especificant els valors per a múltiples paràmetres de configuració.

- Certa descripció de les dependències de cada component ha de ser proporcionada.
- Plantilla per a un pla d'interconnexió de components durant el desplegament. Aquesta plantilla contindrà els elements que es llisten a continuació i especifica la seqüència d'accions de resolució de dependències a fer durant el desplegament:
  - Llista de dependències que han de resoldre's durant el desplegament, especificant els punts d'accés (o “endpoints”) encara lliures i els seus clients potencials.
  - Llista de punts d'accés globals que es proporcionaran com a punts d'accés públics del servei (és a dir, aquells a utilitzar pels usuaris del servei i no per altres components interns).
- Configuració global de l'aplicació. Aquesta configuració pot ser gestionada mitjançant diferents polítiques. Depenent de la política triada, s'assignaran diferents valors a alguns dels paràmetres de les plantilles de configuració dels components, així com per a emplenar la plantilla del pla de desplegament.

- **Descriptor de desplegament**. Un descriptor de desplegament conté:
  - Plantilles de configuració farcides. Hi haurà tantes plantilles farcides com components es necessiten.
  - Plantilla del pla de desplegament farcida. Ha d'observar-se que aquesta plantilla no pot emplenar-se mentre no s'haja decidit on situar (és a dir, en quin node) cada instància de cada component. Una vegada es coneguen les direccions i ports en els quals situar les instàncies, s'emplenarà aquesta plantilla.
  - Resolució de dependències internes. Està implícita en el punt anterior, en establir la ubicació de cada instància.
  - Resolució de dependències externes. Alguns dels components a utilitzar en el desplegament hauran sigut desenvolupats per altres empreses i pot ser que ja estiguen desplegats en el sistema distribuït. En aquest cas, els punts d'accés dels components “externs” han de ser localitzats durant el desplegament i anotats en el descriptor tan prompte com siga possible. Un mecanisme per a aconseguir aquesta resolució dinàmica està basat en l'ús d'un servidor de noms. En aquest cas, el descriptor de desplegament manté els noms dels components externs i els seus corresponents punts d'accés s'esbrinaran resolent aquests noms.

3.2 Exemple de desplegament

La **Figura 2** mostra un exemple de desplegament d'una aplicació distribuïda.

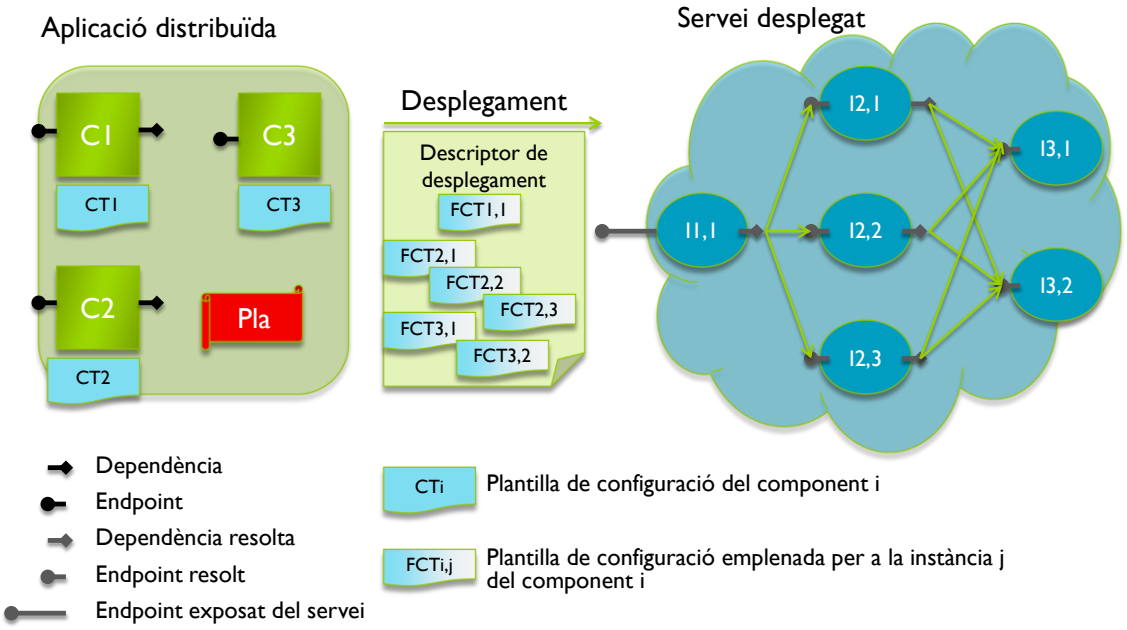


Figura 2. Exemple de desplegament.

En aquest exemple, l'aplicació a desplegar té tres components:

- C1 (equilibrador de càrrega). L'equilibrador de càrrega assigna cada sol·licitud entrant a una de les instàncies del component C2.
- C2 (lògica de negoci). Processa cada sol·licitud rebuda. Quan una sol·licitud modifique les dades de l'aplicació, un protocol de replicació del magatzem de dades serà utilitzat per C2 per a decidir a quines instàncies del component C3 es redirigirà la petició.
- C3 (magatzem de dades). Manté les dades persistents del servei. Cada instància de C3 no sap que està replicada. El protocol de replicació de C3 és gestionat per les instàncies de C2.

El pla de desplegament estableix que l'únic punt d'accés públic al servei és el facilitat per C1 i també indica que no hi ha dependències externes.

El descriptor de desplegament indica que hi ha: una única instància de C1, tres instàncies de C2 i dues instàncies de C3. També conté les plantilles de configuració farcides per a cada instància.

Les dependències han sigut resoltes en el descriptor de desplegament com segueix. L'única dependència de la instància C1 es correspon amb els punts d'accés de cada instància de C2. Cada dependència de cada instància de C2 es relaciona amb el punt d'accés de cada instància de C3.

Com els components d'aquesta aplicació estan feblement acoblats, cada instància de cada component pot ser situada en un ordinador diferent. Les direccions dels ordinadors s'utilitzen per a emplenar el descriptor de desplegament.

Utilitzant el descriptor de desplegament, l'aplicació s'instal·la finalment en els nodes especificats en el descriptor.

4 CONFIGURACIÓ: INJECCIÓ DE DEPENDÈNCIES

Un dels passos inicials del desplegament és la generació del descriptor de desplegament. Per a fer això, les plantilles de configuració de cada component han d'emplenar-se tantes vegades com instàncies vaja a tindre el component, en funció dels recursos existents en cadascun dels ordinadors que allotjaran els components.

Part de la informació a utilitzar per a emplenar les plantilles està relacionada amb la resolució de dependències. Cada component ha d'executar cert programa i aquest programa exporta un o més punts d'accés (els necessaris per a utilitzar les operacions que exporte) però també té alguns punts de dependència (és a dir, aquells fragments del seu programa des dels quals s'invoquen operacions facilitades per altres components). Els punts de dependència rares vegades podran resoldre's durant la compilació del programa. La seua resolució es farà durant el desplegament o durant l'execució, respectant uns principis similars als empleats a les biblioteques d'enllaç dinàmic [10]. En l'enllaç dinàmic les biblioteques són carregades en memòria sota demanda i la direcció a utilitzar per l'invocador per a accedir a una operació de la biblioteca es pren d'una taula d'indirecció que ha sigut emplenada en carregar-se la biblioteca per primera vegada. Aleshores, la resolució es retarda tant com siga possible (resolució dinàmica) i sense requerir cap canvi en la forma en què s'escriu el codi de les biblioteques o el codi dels programes que les utilitzen (transparència per al programador). Aquests són els principis a respectar a l'hora de gestionar l'enllaç entre les dependències i els punts d'accés dels components: resolució dinàmica i transparència.

Com pot obtenir-se aquesta transparència? Revisem l'exemple de les biblioteques d'enllaç dinàmic per a obtindre algunes pistes. Si assumim que una primera versió de la biblioteca a utilitzar va poder basar-se en enllaç estàtic, el codi d'aquella primera versió no haurà necessitat cap canvi per a usar enllaç dinàmic. Llavors... on s'han aplicat els canvis? Bàsicament, en l'eina utilitzada per a compondre els fitxers executables ("linker") que munta l'executable a partir de diversos fitxers objecte, resolent les seues interdependències durant el muntatge, i en el carregador de programes (un component del sistema operatiu que llig els fitxers executables i deixa algunes de les seues regions en memòria principal). Per tant, si pretenem aprofitar aquests principis en el desplegament d'aplicacions distribuïdes haurem d'aplicar alguns canvis en les eines que "carreguen" i mantenen el codi dels components i en les eines que gestionen l'enllaç entre punts d'accés i punts de dependència, però no serà necessari aplicar cap canvi en els programes dels components.

D'aquesta manera, necessitarem dos mecanismes complementaris: els contenidors i la injecció de dependències.

4.1 Contenidors

Els contenidors són eines que mantenen altres components de *programari*, proporcionant-los un entorn aïllat (és a dir, protegit) i la possibilitat d'establir una correspondència entre els seus punts d'accés i els punts d'accés de l'ordinador amfitrió. A més d'aquestes correspondències i la gestió de l'aïllament, els contenidors gestionen les etapes del cicle de vida dels components instal·lats en ells, sent capaços de generar (atenent les ordres dels seus usuaris) aquests esdeveniments relacionats amb el cicle de vida:

- **Creació**. Es necessitaran algunes ordres per a construir una imatge vàlida del component a instal·lar en el contenidor. Per a fer això, alguns elements dependents del contenidor han de combinar-se amb el codi del component per a generar la imatge. A més, la imatge resultant també tindrà certs paràmetres de configuració.
- **Registre** ("initiate"). Una vegada s'haja creat la imatge, haurà de registrar-se en el sistema de contenidors. Aquest és l'objectiu d'aquest esdeveniment.
- **Inici** ("start"). Aquest esdeveniment inicia l'execució del component en algun contenidor amfitrió.
- **Parada**. Aquest esdeveniment atura l'execució del component. Posteriorment, l'execució podrà ser represa o finalitzada.
- **Reconfiguració**. Partint d'alguna imatge prèvia, l'esdeveniment de reconfiguració modifica el contingut d'aquesta imatge (afegint o eliminant mòduls en o de la imatge) o la seua configuració.
- **Destrucció**. Aquest esdeveniment elimina la imatge del component del sistema de contenidors.

Un sistema de gestió de contenidors (com, per exemple, Docker) pot ser considerat una versió "lleugera" d'un hipervisor. Els sistemes hipervisors gestionen màquines virtuals sobre un ordinador. Una imatge de màquina virtual comprèn una pila completa de *programari* (incloent el nucli complet d'un sistema operatiu) que serà executada sobre un equip virtualitzat. Per la seua part, les imatges a utilitzar en contenidors només contenen el programa a executar al costat d'un reduït conjunt de biblioteques dependents. Aquestes imatges no necessiten incloure un nucli de sistema operatiu. El nucli del sistema operatiu amfitrió serà compartit per tots els contenidors en execució en aquest ordinador.

Per exemple, en Docker es necessita alguna distribució recent de Linux com a sistema operatiu amfitrió. Les imatges a executar en Docker han de mantindre algunes biblioteques específiques de les quals depenga el programa a executar (suposem que P1 és el nom del programa). Amb això es crea una imatge de contenidor (anomenada C1). El resultat podrà executar-se sobre diferents amfitrions Linux amb configuracions diverses. Les dependències específiques de P1 estaran ja resoltes en la imatge C1. Per exemple, si P1 fóra un programa escrit en node que necessitara la versió 4.4.0 de l'intèrpret de node, la imatge C1 contindria també aquest intèrpret i les biblioteques que necessite per a funcionar. La imatge C1 podrà executar-se sobre ordinadors amfitrió (amb el gestor Docker instal·lat) que no necessitaran tindre cap versió de node instal·lada o que podrien tindre instal·lada qualsevol altra versió de l'intèrpret (p. ex., la 0.12.12). L'aïllament proporcionat pels contenidors suporta un escenari com aquest.

Els contenidors també faciliten certa ajuda per a gestionar l'enllaç dinàmic de dependències. El desenvolupador d'un component pot escriure el codi del seu programa sense preocupar-se pels punts d'accés reals que s'utilitzaran. En lloc d'això, el programa assumirà algunes direccions estàtiques per als seus punts d'accés i els seus punts de dependència. Durant l'etapa de desplegament, el gestor de contenidors associarà els punts d'entrada estàtics de la imatge en punts d'entrada reals en l'ordinador amfitrió. Aquests últims seran els que hauran de conèixer i utilitzar els altres components per a interactuar amb el component que ara despleguem. Això seria suficient per a resoldre les dependències quan s'utilitzen *sockets* com a

mecanisme d'intercomunicació, en els quals els punts d'entrada consten d'una adreça IP i un número de port.

4.2 Injecció de dependències

Sense contenidors, cada component client C estaria obligat a llegir algun fitxer de configuració per a resoldre les seues dependències respecte a un altre component servidor S. Això implicaria una pèrdua de transparència, ja que el programa C seria conscient del mecanisme de resolució de dependències utilitzat (és a dir, la lectura d'un fitxer de configuració i la interpretació adequada del seu contingut).

En lloc de llegir un fitxer de configuració, quan s'utilitze un patró d'injecció *de dependències* [11], el component C només necessitarà conèixer la interfície de S per a interactuar amb ell. Aquesta interfície serà implantada per alguna classe *proxy*. L'objecte proxy P a utilitzar per a interactuar amb S és "injectat" en C quan es fa el desplegament (per exemple, prenent el codi de P com un dels elements a incloure per a construir la imatge del component C). Així, si el punt d'accés a S canviara, el programa C no necessitaria ser modificat. En el seu lloc, només es necessitaria una nova versió de P. Aquest patró manté la transparència pròpia dels mecanismes d'enllaç dinàmic. Els detalls de la resolució de dependències estan considerats en la implantació de P i tant els programes de C com de S no han de patir modificacions per a interactuar correctament.

5 COMPUTACIÓ EN EL NÚVOL

En les seccions anteriors s'ha explicat què és el desplegament, quines etapes podem distingir en ell i quins mètodes de configuració es necessiten per a desplegar una aplicació distribuïda. Encara queda una qüestió pendent... on es desplegaran els serveis?

Tradicionalment, els serveis de computació escalables han sigut proporcionats per grans empreses. Aquestes empreses mantenien grans centres de computació, però això requeria grans inversions. Aquests centres de dades implicaven un alt cost d'adquisició d'equips (ordinadors i xarxes) i també un alt cost operatiu per a mantindre el seu servei (considerant tant el seu consum energètic com els salaris dels administradors com les renovacions periòdiques dels equips).

Posteriorment, algunes empreses especialitzades en el desplegament han anat apareixent, llogant la seua infraestructura a aquestes empreses proveïdores de serveis. El seu negoci consisteix a mantindre grans centres de dades, mantenint les aplicacions desplegades pels proveïdors de serveis. Així, aquestes empreses "amfitriones" s'han especialitzat en l'administració de la infraestructura, reduint els costos operatius per als proveïdors de serveis. No obstant això, les primeres empreses d'aquest tipus no disposaven de grans infraestructures. Mantenien uns pocs centres de dades (en molts casos, un només), llogant-los a empreses proveïdores de serveis que ja estaven situades prop d'elles. Es necessitaven millors solucions: la gestió d'aquesta infraestructura encara havia de ser més barata, més fiable i amb major capacitat per a l'escalabilitat dels serveis desplegats en ella.

La solució a aquests requisits ha sigut la computació en el núvol. En concret, el model de servei IaaS ("Infrastructure as a Service"). En aquest model:

- El client només paga per allò que usa; és a dir, els costos dependran del nombre de màquines llogades i l'amplada de banda utilitzada.
- La unitat a utilitzar és la màquina virtual (MV). Cada proveïdor ofereix un conjunt de tipus de màquina virtual, segons la seua memòria disponible i la seua capacitat de còmput.
- El proveïdor manté múltiples centres de dades dispersos geogràficament. Això permet que els serveis es despleguen prop dels seus usuaris potencials.
- Es faciliten algunes eines per a gestionar les MV llogades i les dades persistents.

Però aquest model de servei encara és massa primitiu pel que fa a la gestió de les tasques de desplegament perquè...:

- La seua interfície permet seleccionar MV i instal·lar imatges en elles. Però no hi ha regles que automatitzen les decisions d'escalat dels components desplegats. Per a escalar cal donar ordres explícites d'assignació i alliberament de màquines virtuals i això no permet un escalat adaptatiu (és a dir, elàstic). Els serveis elàstics haurien d'adaptar-se per si mateixos a les variacions de càrrega.
- És responsabilitat del client la decisió sobre quin tipus de MV assignar en cada sol·licitud. Per tant, el client hauria de ser un expert sobre les capacitats de les MV i les necessitats dels components a desplegar per a prendre decisions correctes.
- En molts casos, el proveïdor IaaS no facilita cap eina per a monitorar el trànsit en la xarxa i seleccionar la ubicació de cada MV. Per tant, resulta difícil resoldre els problemes generats per una configuració de xarxa inadequada.
- Si la ubicació de les MV utilitzades no pot ser especificada, serà difícil garantir que no hi haja correlació entre les fallades que es donen en les diferents instàncies d'un component desplegat (per exemple, podria donar-se el cas que totes residiren en un mateix “rack” d'un centre de dades i que totes quedaren inaccessibles si la subxarxa que les interconnecta fallara).

Per tant, seria recomanable que hi haguera majors ajudes per a automatitzar el desplegament. Aquestes ajudes es proporcionen en el model de servei PaaS (“Platform as a Service”). Aquest model de servei té com a objectiu automatitzar les tasques del cicle de vida dels serveis, incloent tant el desplegament com la gestió de l'elasticitat.

La clau per a l'automatització és el SLA. En el SLA, client i proveïdor signen un contracte sobre les propietats no funcionals del servei (principalment, rendiment i disponibilitat). A partir del SLA, el proveïdor PaaS emplenarà el pla de desplegament i establirà les regles d'escalabilitat a utilitzar per a obtindre una adaptabilitat òptima. A més, hauria de gestionar les actualitzacions del *programari* de servei respectant també el SLA durant aquests intervals.

Desafortunadament, cap dels proveïdors PaaS actuals ha aconseguit per complet aquests objectius, encara. De moment, han aconseguit automatitzar parcialment el desplegament inicial dels serveis (per exemple, el suport per a injecció de dependències és encara primitiu en alguns casos) però encara no automatitzen les decisions d'escalat amb suficient precisió ni proporcionen procediments d'actualització prou fiables per a evolucionar serveis “stateful”. Això es deu al fet que la gestió dels SLA és encara insuficient en els sistemes PaaS.

Microsoft Azure (azure.microsoft.com) és un dels sistemes PaaS actuals més evolucionat. Pot obtenir-se més informació sobre ell en el seu lloc web oficial.

6 DOCKER

6.1 Introducció

En els apartats anteriors hem estudiat:

- Un model d'aplicació distribuïda integrada per components autònoms
- Múltiples aspectes a contemplar en el desplegament, destacant:
  - L'especificació i configuració dels components
  - El descriptor de desplegament
- Un exemple de desplegament (que convé repassar)
- La problemàtica de la destinació del desplegament, introduint conceptes i modalitats relacionats amb la Computació en el Núvol (CC)

Aquest apartat pretén afermar aquests conceptes mitjançant un cas de desplegament, amb suficients detalls per a fer-lo al laboratori. Les tecnologies concretes que intervenen són:

- Aplicacions (en NodeJS) que es comuniquen mitjançant missatgeria (ØMQ)
- Qualsevol altre programari addicional que puguen requerir les aplicacions
- Sistemes (LINUX) sobre els quals s'executen les aplicacions amb els seus requisits

Aquestes peces (aplicació + requisits + sistema) es reuneixen formant un component.

La tecnologia que pretenem emprar ha aconseguit un punt en el qual existeix un consens sobre la seua utilitat i aplicabilitat, però les implementacions que trobem poden ser inestables donada la velocitat d'evolució actual. És especialment necessari assenyalar que la documentació, característiques i exemples poden estar molt lligats a la versió de la implementació.

En el cas de Docker, que és una implementació de la tecnologia de contenidors, hem intentat que tot el material i informacions arreplegades siguen compatibles amb la versió *comunitària* oficial en començar aquest curs 2019/20 (**docker-19.03-ce**).

És possible trobar documentació i exemples anteriors a aquesta versió. Haurem de fixar-nos que quan s'esmenta compose, s'especifique la **versió 3**[1] en el fitxer de configuració docker-compose.yml

- La versió actual entén especificacions de versions anteriors sempre que vengen anunciades en la clàusula version de l'arxiu YAML.

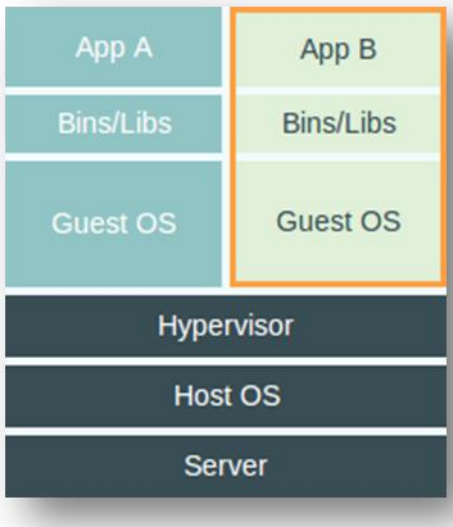
Aquestes precaucions poden no ser suficients, però almenys han de mantenir-vos alerta davant aquesta font d'errors.

[1] 3.7 aquest curs 2019/20

6.1.1 Aprovisionament

Diem **aprovisionament** (*provisioning*) a la tasca de reservar la infraestructura necessària perquè una aplicació distribuïda puga funcionar.

- Reservar recursos específics per a cada instància de component (processador + memòria + emmagatzematge).
- Reservar recursos per a la intercomunicació entre components.

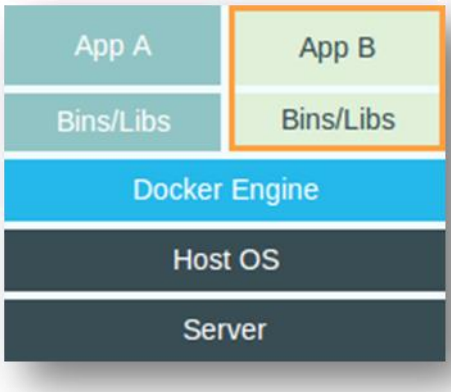


La infraestructura sol concretar-se en *un pool* de **màquines virtuals** interconnectades.

- El component i els seus requisits s'implementen i executen sobre una màquina virtual.

En el nostre cas optem per una versió *lleugera*[2] de màquina virtual, denominada **contenidor.**

- El sistema de l'hoste coincideix amb el de l'amfitrió, per la qual cosa no es requereix duplicar-lo ni emular-lo.
- L'amfitrió ha de disposar d'un programari de *contenerització*, que ofereix certs serveis d'aïllament i replicació.



Cada component s'implementa sobre un contenidor.

6.1.2 Configuració de components

Per a cada component hi ha una especificació de la seua configuració que inclou...

- El programari a executar.
- Les dependències que hauran de ser concretades.

La nostra decisió anterior sobre els contenidors suposa que el programari del component…

- Ha de ser compatible amb el S.O. de l'amfitrió.
- Pot completar-se amb algun programari addicional.
- Ha de configurar-se i inicialitzar-se.

Aquestes tasques seran responsabilitat de la pròpia operació de creació de la imatge, i del fitxer de propietats del contenidor.

6.1.3 Pla de desplegament

Tal com es va descriure en l'apartat 3, la llista d'accions a executar per a dur a terme el desplegament ve especificada com un algorisme o pla.

[2] Veure segona il·lustració

- En cas d'una eina automatitzada, existirà una aplicació que interpretarà (*orquestració*) l'especificació, i durà a terme les accions.
- En la seua absència, es desenvoluparà un programa a mesura de les indicacions del pla de desplegament.

El desplegament manual es reserva per a proves i casos senzills, perquè no és utilitzable per a una aplicació distribuïda de grandària mitjana.

6.2 De les màquines virtuals als contenidors

Actualment les tecnologies de virtualització han permès implementar serveis amb flexibilitat, però aquests han de rendibilitzar una instal·lació completa. P. ex., una instal·lació virtualitzada mínima pot consumir 500MB

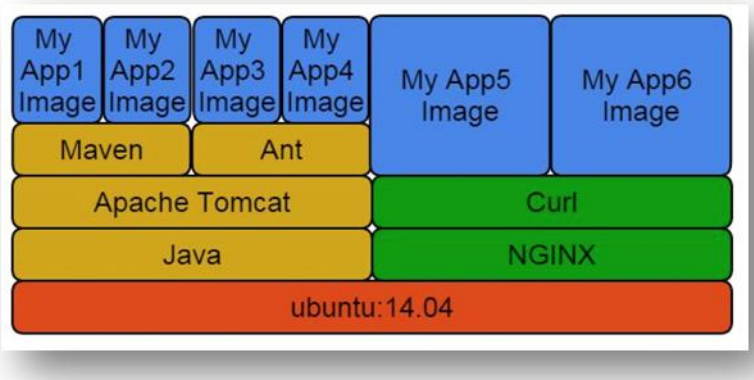
- No és pràctic prendre-la com a base per a un servei que retorne l'hora actual (*massa equipatge*)
- Sí que és rendible com a base per a un servei de correu electrònic amb una desena d'usuaris; per tant, és adequada per a serveis amb components relativament pesats

Les implementacions actuals en tecnologia de contenidors, com Docker, són prou madures per a plantejar-se com a alternativa a les màquines virtuals

- Avantatge: suposant que una imatge de 1GB usada per 100MVs consumisca 100GBs…
  - Si hi ha 900MBs immutables, 100 contenidors Docker consumirien 0,9+0,1*100=10,9 GBs -> redueix **espai**
    - Els contenidors consumeixen aproximadament entre 10 i 100 vegades menys recursos que els seus equivalents virtuals
  - Si la part immutable es troba "precarregada", ens estalviem aquest temps (90%) per a iniciar cada instància Docker -> redueix **temps**

- Inconvenients (com qualsevol sistema de contenidors)
  - Menys flexible que les MVs
  - L'aïllament imperfecte entre contenidors pot provocar interferències i problemes de seguretat

Amb algunes simplificacions, el sistema basat en contenidors il·lustrat a la dreta ens permet detallar l'estalvi respecte a un sistema basat en màquines virtuals: l'equivalent virtualitzat donaria lloc a **una MV completa per cada imatge d'aplicació** (6 en total), incloent (d'esquerra a dreta)…

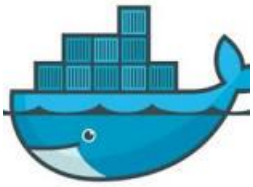


1. Ubuntu+Java+Tomcat+Maven+My App1 Image
2. Ubuntu+Java+Tomcat+Maven+My App2 Image
3. Ubuntu+Java+Tomcat+Ant+My App3 Image
4. Ubuntu+Java+Tomcat+Ant+My App4 Image
5. Ubuntu+NGINX+Curl+My App5 Image
6. Ubuntu+NGINX+Curl+My App6 Image

Sumant: 6Ubuntu+4Java+4Tomcat+2Maven+2Ant+2NGINX+2Curl+My App1 Image+...+My App6 Image … **La diferència és MOLT substancial!**

La tecnologia de contenidors és tan lleugera que possibilita la virtualització **d'una aplicació**

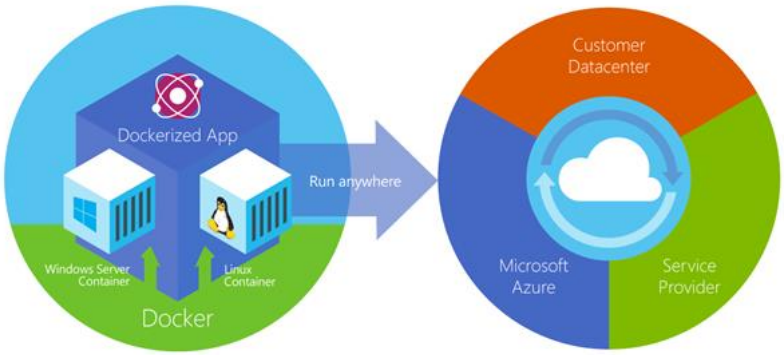
6.3 Introducció a Docker



Docker ofereix una API per a executar processos de forma aïllada, que pot prendre's com a base per a construir una PaaS.

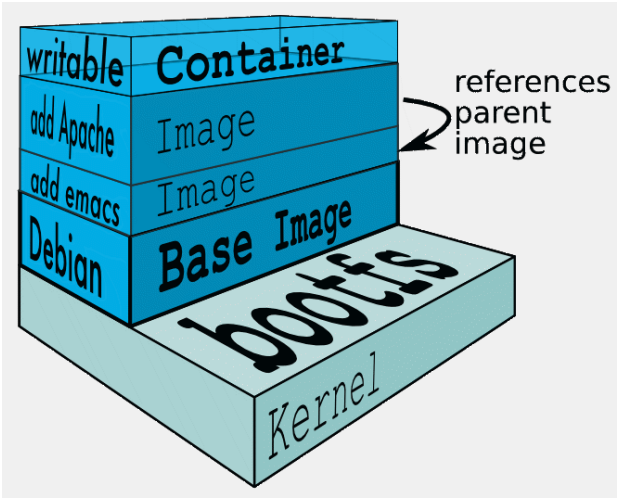
La implementació de Docker es basava originalment en LXC, però posteriorment aquesta dependència ha sigut suavitzada per a interactuar amb altres implementadors de contenidors. En una segona etapa es va utilitzar *libcontainer*

- Actualment *runc*
- Al costat de Microsoft s'ha aconseguit una implementació nativa per a Windows



Respecte a LXC, Docker afig:

- AuFS: sistema de fitxers amb una part de només lectura que pot compartir-se
  - Les modificacions formen capes que se superposen
  - Cada orden que cree, modifique o elimine un o més fitxers generarà una nova capa en la imatge.
  - A la pàgina 29 pots observar les capes de la imatge "misitio" (ordre: docker history misitio) construïda amb un Dockerfile per a un servidor web.

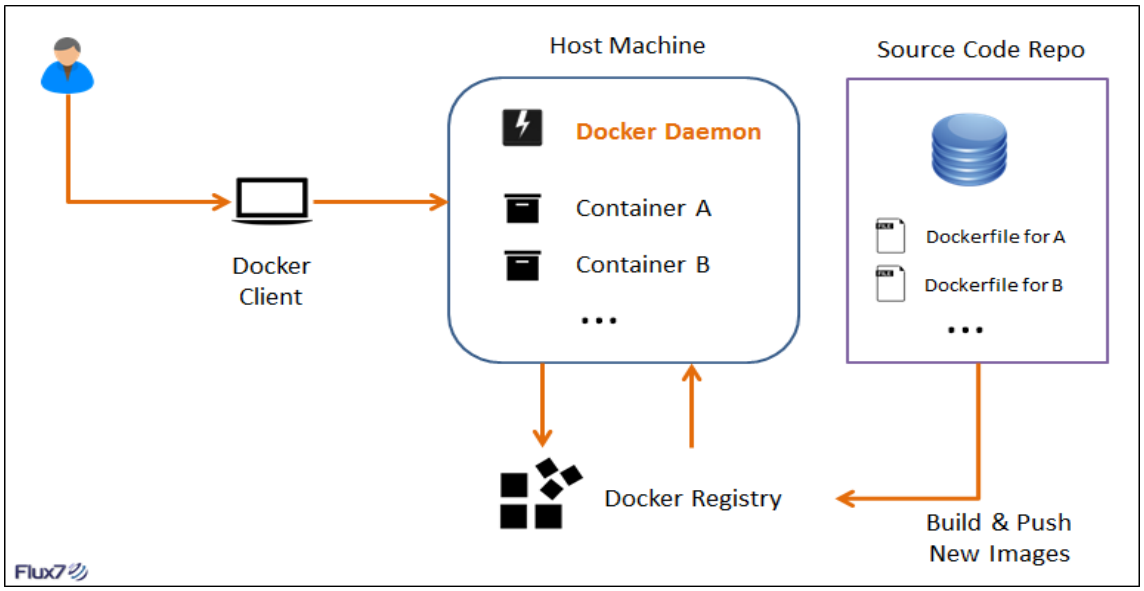


- Nova funcionalitat:
  - construcció automàtica
  - control de versions (Git)
  - compartició mitjançant depòsits públics

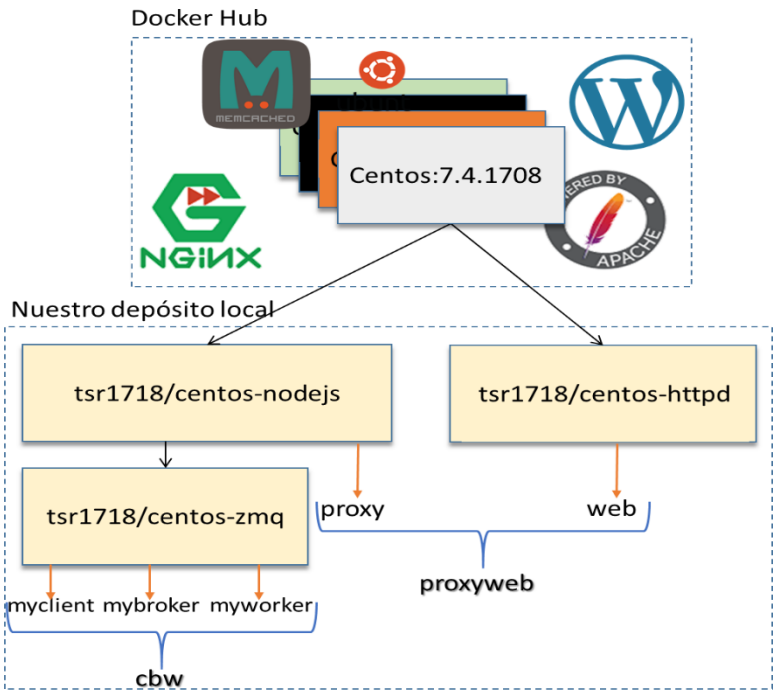
El sistema Docker disposa de 3 components:

1. **Imatges** (component *constructor*)
   - Les imatges docker són bàsicament plantilles de només lectura a partir de les quals s'instancien contenidors
2. **Depòsit** (component *distribuïdor*)
   - Hi ha un depòsit comú per a poder pujar i compartir imatges (hub.docker.com)
3. **Contenidors** (component *executor*)
   - Es creen a partir de les imatges, i contenen tot el que la nostra aplicació necessita per a executar-se
   - Es pot convertir un contenidor en imatge mitjançant docker commit

Internament Docker consta d'un *daemon*, un depòsit i una aplicació client (docker)



En aquest apartat emprarem únicament una imatge de depòsit central, i derivarem d'ella algunes especialitzacions per a, al final, desplegar un parell d'aplicacions multicomponent.



6.4 Funcionament

La secció 6.8 descriurà que Docker pot executar-se per a organitzar les activitats en un equip individual (*standalone*) o com a part d'un conjunt de nodes intercomunicats. En aquest darrer cas, ha d'existir una interacció on participen, com un conjunt, els sistemes docker de cada node, amb dues possibles alternatives:

- Una pròpia de Docker anomenada **swarm**. En aquesta opció, cada sistema docker individual s'executa en *mode swarm*.
- Una alternativa externa a Docker, destacant la veterana Kubernetes (amb un ús majoritari entre tots els orquestradors).

L'existència del *mode swarm* suposa que algunes ordres de Docker només tenen sentit en aquest mode de funcionament i que la semàntica d'altres ordres pot veure's influenciada. En la pràctica, en aquest document, la descripció de les ordres de Docker es basa en el *mode standalone*, evitant les funcions (grups *node, stack* i *swarm*) relacionades amb el *mode swarm*.

D'altra banda, malgrat la seua importància, les operacions relacionades amb la seguretat (autenticació amb certificats, per exemple) no són objecte d'estudi en aquesta assignatura. Això exclourà els grups d'ordres Docker relacionats amb aquests conceptes (*secrets* i *trust*).

La sintaxi de Docker ha anat evolucionant per a facilitar la incorporació de més de cent ordres. En el conjunt més actual, aquestes ordres s'agrupen atenent l'object sobre el que actuen (per exemple, contenidor, imatge, sistema...). Així, per a obtenir una relació de les imatges existents s'usaria "doker image list". No obstant això, és freqüent trobar aquestes ordres formulades sense atendre els nous grups, en un format on no s'indica expressament l'objecte sobre qui apliquem l'ordre, com "docker images".

Les agrupacions que sí considerem en aquesta assignatura són:

Grup	Descripció
config	Gestió de configuracions
container	Operacions sobre contenidors
context	Contexts per al desplegament distribuït (k8s, …)
image	Gestió d'imatges
network	Gestió de xarxes
service	Gestió de serveis (contenidors instanciats a partir de la mateixa imatge) com a part d'una aplicació distribuïda, normalment desplegada amb "docker-compose".
system	Gestió global de Docker
volume	Gestió de magatzematge secundari.

6.4.1 Docker des de la línia d'ordres

L'element més important és el client "docker" amb el que interactuem amb el servidor. Això necessita privilegis per al seu ús.

```
docker [OPTIONS] COMMAND
```

Tipus d'ordres:

1. Control del cicle de vida
   - docker **commit** (docker container commit)
   - docker **run** (docker container run)
   - docker **start** (docker container start)
   - docker **stop** (docker container stop)
2. Informatives
   - docker **logs** (docker container logs)
   - docker **ps** (docker service ps)
   - docker **info** (docker system info)
   - docker **images** (docker image ls)
   - docker **history** *imagen* (docker image history *imagen*)
3. Accés al depòsit
   - docker **pull** (docker image pull)
   - docker **push** (docker image push)
4. Altres
   - docker **cp** (docker container cp)
   - docker **export** (docker container export)

6.4.2 Exemple senzill (ordre: docker run)

```
docker run -i -t imatge programa
```

- Acció: run serveix per a construir i executar (-i -t per a interactiu)
  - P. ex. docker run -i –t ubuntu /bin/bash
- Imatge: p.ex. ubuntu
- Programa: p.ex. /bin/bash

Passos:

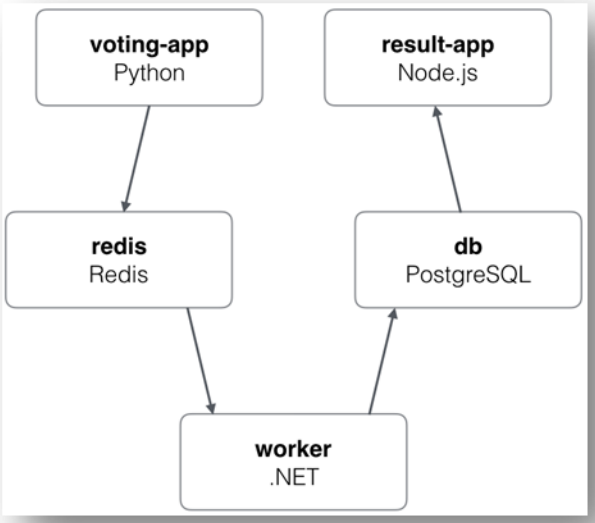
1. Descarregar la imatge (ubuntu) des del *Docker Hub*
2. Crear el contenidor.
3. Reservar un sistema de fitxers i afegir un nivell de lectura/escriptura.
4. Reservar una interfície de xarxa o un bridge per a comunicar amb l'amfitrió.
5. Reservar una adreça IP interna.
6. Executar el programa especificat (/bin/bash)
7. Capturar l'eixida de l'aplicació

6.4.3 Alguns escenaris i exemples d'ús

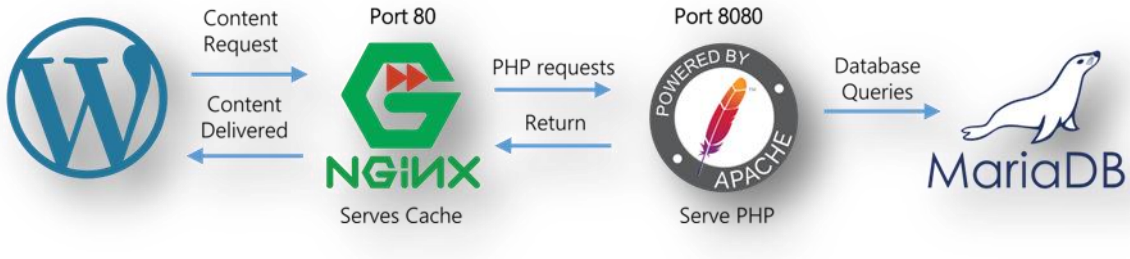
Hi ha una gran varietat d'escenaris en els quals es pot aplicar aquesta tecnologia, i seria molt complex establir criteris per a la seua classificació; no obstant això és molt interessant disposar d'exemples **interessants i il·lustratius** que ens permeten entendre l'abast d'aquesta tecnologia.

1. **Aplicacions monolítiques**: aplicacions d'usuari (p. ex. Firefox, LibreOffice), aplicacions servidor (p. ex. servidor de base de dades MySQL, servidor de web APACHE).
   - En aquests casos la inclusió de tot un sistema operatiu com a requisit de l'aplicació podria semblar desproporcionada, però ací resideix la *màgia* de la contenerització que aconsegueix que el sistema siga rendible.

2. **Serveis multicomponent**: la pròpia aplicació es compon de diverses peces, amb cert grau de llibertat, que s'han d'assemblar per a donar lloc al servei final. En el nostre cas és interessant destacar aquests tres exemples:
   - Un servei de votacions (*voting-app*[3]) que integra components desenvolupats en sistemes molt diferents (observa la il·lustració a la dreta), però que són capaces d'interactuar una vegada desplegats.



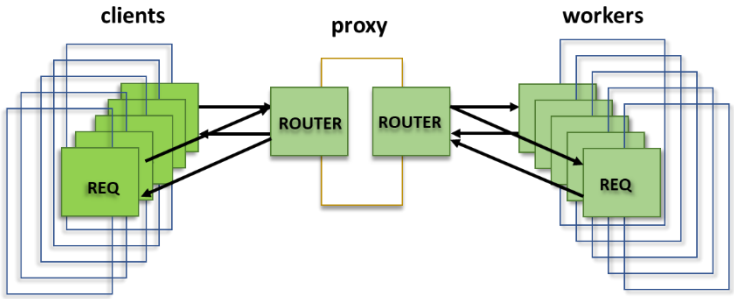
   - Un servei de blog constituït per un proxy invers Nginx, un servidor de web APATXE amb suport per a aplicacions PHP, el codi de WordPress executant-se en el servidor, un servidor de base de dades MariaDB (compatible amb MySQL).



     - El nostre interès no se centra en el codi font de les peces sinó en la seua configuració individual i les seues relacions amb les altres, que seran els únics aspectes en els quals puguem intervindre.

   - Un servei (p. ex. client+broker+worker) que nosaltres mateixos desenvolupem: no tindrem problemes a adaptar-lo si fóra necessari, encara que aquesta capacitat no evita la preocupació per interconnectar els diferents components que participen en el servei.

[3] https://github.com/dockersamples/example-voting-app



3. **Desplegament d'instal·lacions genèriques**. Encara que puga semblar el més habitual, en l'àmbit de la conteneització **no** sol abordar-se el cas d'equips d'escriptori complets, amb totes les aplicacions disponibles en una distribució tipus Ubuntu, Fedora o similars.
    - És molt més freqüent, i útil, trobar instal·lacions mínimes sobre les quals s'afigen les aplicacions necessàries per a cada cas. La conteneització aposta per imatges especialitzades per a cada cas. P. ex., una distribució mínima de CentOS sobre la qual s'ha afegit el suport per a executar aplicacions NodeJS.

D'altra banda, també s'ha de destacar que la **destinació del desplegament** condiciona tota l'operació, destacant dos extrems:

1. **Desplegament en un nombre discret d'equips**. Es disposa d'un conjunt de recursos limitat i poc flexible: com en el laboratori de l'assignatura. Els problemes a resoldre han de trobar solucions compatibles amb aquesta disponibilitat reduïda, i la falta de possibilitats redueix la complexitat del desplegament.
2. **Deplegament en el núvol**. Existeix un número elevat i indiferenciat d'amfitrions. La quantitat de recursos disponibles és molt major que en el cas anterior, i pot adaptar-se *elàsticament* a les exigències de cada moment. Infraestructures de núvol diferents poden imposar maneres de desplegament concretes, o limitar la nostra llibertat per a especificar com s'orquestren els components de l'aplicació.

Finalment, dins d'aquest apartat il·lustratiu és important esmentar que els contenidors són efímers, de manera que es crea per a ells un sistema de fitxers que no sobreviu al contenidor: de què ens serveix fer operacions el resultat de les quals es perd en finalitzar el contenidor?. La persistència i altres problemes es resolen mitjançant l'accés **a recursos de l'amfitrió**.

- Accés al **sistema de fitxers**. Establint una correspondència entre l'emmagatzematge real de l'amfitrió i una ruta del sistema de fitxers efímer del contenidor, la informació depositada no es destrueix amb aquest contenidor.
    - P. ex. usar l'opció -v en docker run

```
docker run ... -v /ruta/amfitrio:/ruta/contenidor ...
```

    - Un cas especial és la possibilitat d'interacció amb el propi servei Docker de l'amfitrió si el contenidor pot accedir al socket del servei.
        - P. ex. usar l'opció -v en docker run

```
docker run ... -v /var/run/docker.sock:/var/run/docker.sock ...
```

- Accés a **ports** de l'amfitrió. Establint una correspondència entre ports de l'amfitrió i del contenidor, aquest pot ser accedit des de l'exterior.
    - P. ex. usar l'opció –p en docker run

```
docker run ... -p 9999:9000 ...
```

    - On 9999 és el port de l'amfitrió que es fa correspondre amb el port 9000 del contenidor.
    - També ha de configurar-se correctament el tallafocs de l'amfitrió, que pot limitar la visibilitat externa dels ports.

6.4.4 Servidor web mínim sobre CentOS

Nota: Les activitats aplicades, com aquesta, únicament poden ser comprovades en els nostres equips virtuals de portal-ng

En aquest apartat pretenem prendre contacte amb un desplegament artesà, realitzat de manera progressiva. El servei a desplegar, en finalitzar, consistirà en un servidor de web connectat al port 80 del contenidor que retorna al navegador (http://localhost/) una pàgina de benvinguda:

Aquest primer cas s'ha ideat per a emprar-se en el laboratori; ens serveix per a posar a prova la nostra infraestructura: programari, configuració, ports, etc. Els passos[4] que realitzem per a aconseguir aquest objectiu requereixen prèviament que esbrinem què necessitem per a construir el servei.

Abreujant, la nostra llista de requisits és:

1. Una instal·lació base compatible amb un servidor de web (p. ex. CentOS i el servidor APACHE). És necessari preveure com s'aconsegueix instal·lar el servidor sobre la imatge CentOS.
2. Material per a *poblar* el directori amb documents del servidor (p. ex. el contingut de misitio.tar.gz, que es troba en el directori d'aquest tema en PoliformaT[5]). Inclou carpetes amb pàgines, estils, etc. És necessari conèixer on ha de col·locar-se cadascuna de les peces.
3. Via d'accés al servei resultant. Si posem en funcionament un navegador en la màquina virtual, amb quina URL accedim al servei del contenidor?. Hem d'indicar en l'amfitrió (la nostra virtual) que un dels seus ports (p.ex. el 8000) s'associe a un port (el 80) del contenidor.

4 basat en http://linoxide.com/linux-how-to/interactively-create-docker-container/

5

Anem per passos[6], però sense perdre la perspectiva global.

1. Connectem amb la nostra màquina virtual que disposa del programari Docker necessari, i d'una connexió a Internet.
2. Executem:

```
docker run -i -t centos:7.4.1708 bash
```

   - En aquesta instrucció seleccionem l'execució d'una imatge anomenada "centos[7]".
   - Com Docker no disposa localment de cap imatge amb tal identificador, connectarà amb el depòsit central (*docker hub*), trobarà la imatge[8] i la portarà (197MBs).
3. Quan aconseguisca posar-ho en marxa, executarà en el contenidor el programa bash (un shell) i quedarà a l'espera de les nostres ordres des del teclat.
4. Usem dins del contenidor l'ordre yum (ja estem en CentOS!) que és un gestor de paquets.
   - Primer caldria posar-lo al dia (yum update).
     - Però no ho farem perquè aquesta actualització demana baixar-se 96 paquets (66 MB).
   - ...i després afegim el servidor APACHE (paquet "httpd"). Aquest element necessita baixar 1+5 paquets (24 MB).

(dins del contenidor) yum install httpd

Un bon avantatge de Docker és que manté una còpia local d'allò que puga reutilitzar, de manera que una nova imatge basada en alguna anteriorment descarregada demanaria molt poc esforç.

5. De moment no necessitem res més dins del contenidor, així que eixim d'ell i tornem a la nostra virtual.

(dins del contenidor) exit

   - En línies generals, tot el que fem amb els contenidors és volàtil llevat que diguem el contrari.

6. Hem d'esbrinar el nom intern del contenidor. Com ja hem acabat amb ell, cercarem en el sistema un contenidor basat en centos que haja finalitzat recentment (uns minuts com a màxim)

```
docker ps -a
// triem el més recent basat en centos
// copiem els primers dígits de la columna CONTAINER_ID
// i fem un commit per a donar nom i congelar la imatge
docker commit CONTAINER_ID  tsr1718[9]/centos-httpd
```

7. Com a resultat, si executem docker images observarem dos elements: la imatge centos descarregada del *Docker Hub*, i nostra centos-httpd que acabem de crear localment.

```
docker images
```

[6] 12 en total

[7] Es tracta d'una distribució de LINUX emparentada amb RedHat

[8] De fet és una família en la qual cada membre centos té associat un número de versió. Si s'omet, es portarà l'últim (latest).

[9] Encara que no sempre aparega explícitament, col·loquem com a prefix tsr1718 a totes les imatges que creem per al seu ús local

- Amb això acabem amb el primer requisit i passem al segon (poblar directori).

8. Copiem misitio.tar.gz a la nostra virtual i ho descomprimim, donant lloc a una carpeta misitio. Ara hem d'instruir al contenidor perquè "agafe" aquests materials. Recorrerem a operacions automatitzades a través de l'arxiu de configuració Dockerfile[10] el contingut del qual serà:

```
FROM tsr1718/centos-httpd
ADD misitio/index.html /var/www/html/index.html
ADD misitio/css /var/www/html/css
ADD misitio/js /var/www/html/js
ADD misitio/fonts /var/www/html/fonts
EXPOSE 80
ENTRYPOINT [ "/usr/sbin/httpd" ]
CMD [ "-D", "FOREGROUND" ]
```

9. En aquest mateix directori executem:

```
docker build --rm -t misitio .
```

- No oblides el punt al final de l'ordre!

Amb això instruïm a Docker perquè a partir del Dockerfile del directori actual genere un contenidor anomenat misitio eliminant els nivells intermedis[11] que puga anar generant.

```
Sending build context to Docker daemon   1.36MB
Step 1/8 : FROM tsr1718/centos-httpd
 ---> ae038753361c
Step 2/8 : ADD misitio/index.html /var/www/html/index.html
 ---> ad34026e9310
Step 3/8 : ADD misitio/css /var/www/html/css
 ---> a3453a27b8ac
Step 4/8 : ADD misitio/js /var/www/html/js
 ---> 860d3be1c05b
Step 5/8 : ADD misitio/fonts /var/www/html/fonts
 ---> 9a320996e017
Step 6/8 : EXPOSE 80
 ---> Running in 26aac81023ab
Removing intermediate container 26aac81023ab
 ---> d091a2755e64
Step 7/8 : ENTRYPOINT [ "/usr/sbin/httpd" ]
 ---> Running in e7cc8a2c35d9
Removing intermediate container e7cc8a2c35d9
 ---> 04d8a28a2086
Step 8/8 : CMD [ "-D", "FOREGROUND" ]
 ---> Running in 540f5b5ce993
Removing intermediate container 540f5b5ce993
 ---> 404d8b9b7367
Successfully built 404d8b9b7367
Successfully tagged misitio:latest
```

- Com a informació, la imatge `misitio` conté les següents capes[12] (`docker history misitio`):

```
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
404d8b9b7367   4 minutes ago   /bin/sh -c #(nop)  CMD ["-D" "FOREGROUND"]      0B
04d8a28a2086   4 minutes ago   /bin/sh -c #(nop)  ENTRYPOINT ["/usr/sbin/ht…   0B
d091a2755e64   4 minutes ago   /bin/sh -c #(nop)  EXPOSE 80                    0B
9a320996e017   4 minutes ago   /bin/sh -c #(nop) ADD dir:79925ae94e3026d301…   216kB
860d3be1c05b   4 minutes ago   /bin/sh -c #(nop) ADD dir:0b8b79e43bfe2f09f6…   106kB
a3453a27b8ac   4 minutes ago   /bin/sh -c #(nop) ADD dir:6dbb89a19ef8f56f2e…   760kB
ad34026e9310   4 minutes ago   /bin/sh -c #(nop) ADD file:8c8e2b5d38125da50…   1.58kB
ae038753361c   10 minutes ago  bash                                            110MB
9f266d35e02c   7 months ago    /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      7 months ago    /bin/sh -c #(nop)  LABEL name=CentOS Base Im…   0B
<missing>      7 months ago    /bin/sh -c #(nop) ADD file:8e23335223edd895d…   197MB
```

[10] Dockerfile és objecte d'estudi en un dels pròxims apartats
[11] Cada nivell s'obté com a resultat d'executar una ordre del Dockerfile
[12] Citat des de la pàgina 18 en parlar de les capes d'una imatge.

- Però l'opció `--rm` provoca que les imatges intermèdies no es guarden.

10. Això finalitza el segon requisit, i ens quedarà el tercer: executar el contenidor permetent l'accés al servei que ofereix.

```
docker run -p 8000:80 -d -P misitio
```

- Aquesta ordre posa en funcionament el contenidor, associant el port 8000 de la virtual com a punt d'entrada per al 80 del contenidor.

11. Amb un navegador en la màquina virtual que accedisca a l'url http://localhost:8000 hauríem de veure la il·lustració que apareix al començament d'aquest subapartat.
    - Qüestió: Què necessitaries fer per a accedir al servidor des de la teua sessió d'escriptori?

12. Per a acabar amb el contenidor, esbrinem el seu identificador amb docker ps[13], i ho detenim amb docker stop.
    - La prova de foc consisteix a recarregar el navegador, que fallarà al no poder connectar.

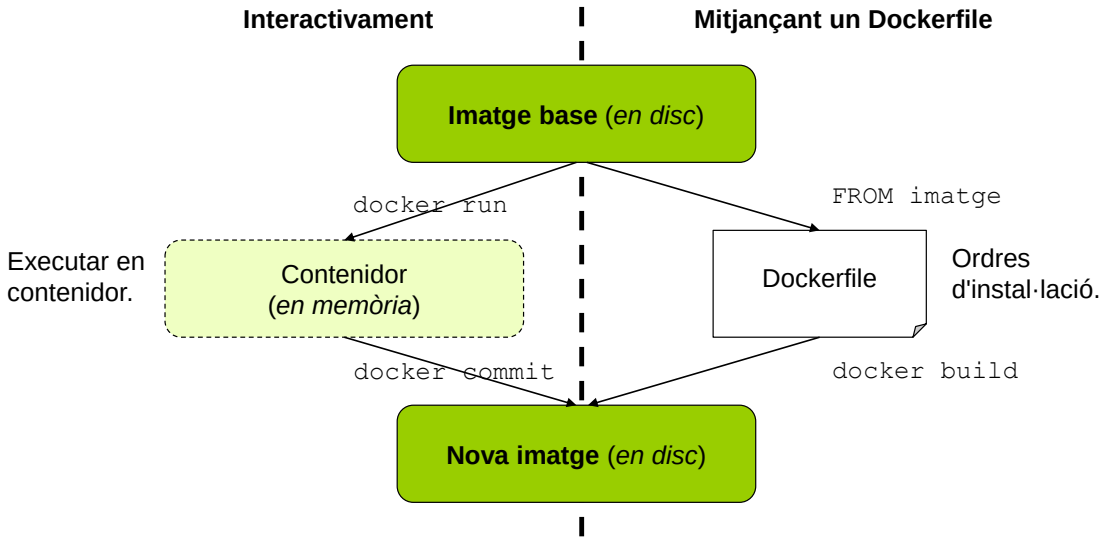
Conclusions destacables

- La creació d'una imatge nova pot consumir temps i recursos de forma apreciable. Si la imatge es basa en una altra que ja tenim, el sistema redueix els passos necessaris i el consum de recursos. Aquesta és una de les raons que aconsellen reaprofitar les imatges ja creades.
- Un Dockerfile complet permet reproduir les accions amb comoditat i reduint el risc d'errors, però és difícil escriure'l correctament a la primera. El mètode de prova i error mitjançant una sessió interactiva és una bona aproximació. Consultar la documentació és **imprescindible**.

Preparar un contenidor per a executar aplicacions NodeJS amb ZeroMQ

Podem procedir a construir una imatge de forma interactiva i progressiva (*prova i error*) mitjançant algun *shell* sobre el contenidor: aniríem anotant els passos que ens semblen apropiats. Més tard, amb aquesta llista, podrem automatitzar la generació de la imatge en un Dockerfile.

[13] No necessita opcions perquè el contenidor es troba en execució



Encara que el número i varietat d'imatges disponibles, produïdes per una altra gent, és molt elevat, no sol trobar-se cap que s'ajuste exactament a les nostres necessitats. No obstant això, sí que és molt habitual que diverses d'elles puguen ser preses com a referència per a afegir posteriorment el nostre middleware i la nostra aplicació final.

En aquest apartat ens marquem com a objectiu construir un contenidor en el qual es puga executar una aplicació NodeJS amb la biblioteca ZeroMQ. Per familiaritat, prendrem com a base una distribució CentOS[14] de la qual ens interessa especialment la gestió de paquets mitjançant yum[15]: podem instal·lar aplicacions de manera que els seus prerequisits s'instal·len automàticament per la xarxa.

Detalladament, els passos a seguir són...

1. Esbrinar els **prerequisits** per a fer aquesta instal·lació sobre una distribució CentOS 7:
   - Necessitem donar d'alta el depòsit nodesource per a poder instal·lar NodeJS

Executar el script en https://rpm.nodesource.com/setup_10.x per a instal·lar l'últim NodeJS de la sèrie 10

```
curl --location https://rpm.nodesource.com/setup_10.x | bash -
```

   - Necessitem donar d'alta el depòsit EPEL per a poder instal·lar ZeroMQ

En un depòsit inclòs en la sèrie 7 de CentOS, hi ha un paquet epel-release que s'ha d'instal·lar per a poder accedir als paquets d'EPEL, incloent ZeroMQ.

```
yum install -y epel-release
yum install -y zeromq-devel
```

   - Per a algunes peces necessitarem les aplicacions make, python i C++ (ho unirem a l'ordre anterior)

```
yum install -y make python gcc-c++
```

[14] hi ha molta informació disponible en la xarxa

[15] https://wiki.centos.org/packagemanagement/yum/

2. Quan ja tinguem clars els prerequisits, caldrà aplicar aquestes accions per a generar una nova imatge. El primer pas serà iniciar un contenidor per a executar la imatge CentOS de manera interactiva:

```
docker run –i –t centos:7.4.1708 bash
```

3. Des de l'intèrpret d'ordres del contenidor, anirem llançant les següents ordres:

```
curl --silent --location https://rpm.nodesource.com/setup_10.x | bash -
yum install –y nodejs
yum install –y epel-release
yum install –y zeromq-devel make python gcc-c++
npm install zeromq@4
exit
```

4. Des de la línia d'ordres del nostre sistema (amfitrió), possiblement en un altre terminal, obtenim l'identificador o nom del contenidor utilitzat en els passos anteriors:

```
docker ps -a
CONTAINER ID   IMAGE    COMMAND   CREATED       ...  NAME
c7b87a37d012   centos   “bash”    9 minutes ago      condescending_einstein
```

5. Fer el commit del contingut actual del contenidor, generant així una nova imatge

```
docker commit condescending_einstein  tsr1718/centos-zmq-tmp
```

o, alternativament

```
docker commit c7b87   tsr1718/centos-zmq-tmp
```

- Ara ja tindrem una imatge Docker denominada “tsr1718/centos-zmq-tmp” des de la qual llançar programes node en contenidors.

6. Comprovar mitjançant docker images

Aquests passos que hem realitzat poden ser automatitzats si es col·loquen les instruccions i propietats en un arxiu de configuració (Dockerfile) que estudiem en el següent apartat.

- La imatge anterior només ha sigut produïda per a il·lustrar els passos necessaris en la seua construcció. No la usarem en la resta d'aquest tema **i pot eliminar-se**.

El més productiu és dividir en dues etapes la creació d'aqueixa imatge:

- una primera que instal·larà NodeJS (tsr1718/centos-nodejs),
- i una altra segona que afegirà ZeroMQ a l'anterior (tsr1718/centos-zmq).

6.5 Dockerfile: El fitxer de propietats de Docker

Docker pot **construir una imatge** a partir de les instruccions d'un fitxer de text anomenat Dockerfile, que ha de trobar-se en l'arrel del depòsit que desitgem construir

```
docker build <opcions> <ruta_depòsit>
```

Sintaxi general: INSTRUCCIÓ argument

- Per convenció, les instruccions s'escriuen en majúscules
- S'executen per ordre d'aparició en el Dockerfile

Tot Dockerfile ha de començar amb FROM, que especifica la imatge que es pren com a base per a construir la nova.

- Sintaxi: FROM <nom_imatge>

6.5.1 Ordres en l'arxiu Dockerfile

Disposes de material de referència en la "**Docker reference documentation**" d'aquest mateix tema.

1. **MAINTAINER**: Estableix l'autor de la imatge
   - Sintaxi: MAINTAINER <nom autor>
   - Actualment es prefereix l'ús de l'ordre LABEL maintainer=<nom autor>
2. **RUN**: Executa una ordre (*shell* o *exec*), afegint un nou nivell sobre la imatge resultant. El resultat es pren com a base per a la següent instrucció
   - Sintaxi: RUN <ordre>
3. **ADD**: Copia arxius d'un lloc a un altre
   - Sintaxi: ADD <origen> <destinació>
   - L'origen pot ser un URL, un directori (es copia tot el seu contingut) o un arxiu accessible en el context d'aquesta execució
   - La destinació és una ruta en el contenidor
4. **CMD**: Aquesta ordre proporciona els valors per defecte en l'execució del contenidor

Només pot usar-se una vegada (si n'hi haguera vàries, només s'executarà l'última)

   - Sintaxi: 3 alternatives
     - CMD ["executable","param1","param2"]
     - CMD ["param1","param2"]
     - CMD ordre param1 param2
5. **EXPOSE**: Indica el port en el qual el contenidor atendrà (*listen*) peticions
   - Sintaxi: EXPOSE <port>
6. **ENTRYPOINT**: Configura un contenidor com si fóra un executable
   - Especifica una aplicació que s'executarà automàticament cada vegada que s'instancie un contenidor a partir d'aquesta imatge
     - Implica que aquest serà l'únic propòsit de la imatge
   - Com en CMD, només s'executarà l'últim ENTRYPOINT especificat
   - Sintaxi: 2 alternatives
     - ENTRYPOINT ["executable", "param1","param2"]
     - ENTRYPOINT ordre param1 param2
7. **WORKDIR**: Estableix el directori de treball per a les instruccions RUN, CMD i ENTRYPOINT.
   - Sintaxi: WORKDIR /ruta/a/directori_de_treball
8. **ENV**: Assigna valors a les variables d'entorn que poden ser consultades pels programes dins del contenidor.
   - Sintaxi: ENV <variable> <valor>
9. **USER**: Estableix l'UID sota el qual s'executarà la imatge.
   - Sintaxi: USER <uid>
10. **VOLUME**: Permet l'accés del contenidor a un directori de l'amfitrió, la qual cosa permet que el seu contingut "*sobrevisca*" a l'execució del contenidor. S'usa en 2 moments complementaris: primer s'especifica la ruta desitjada dins del contenidor, i en l'ordre docker run s'estableix l'equivalent en l'amfitrió
    - Sintaxi: VOLUME ["/ruta/contenidor"] (en Dockerfile)
    - Sintaxi: docker run ... -v /ruta/amfitrio:/ruta/contenidor ... (en línia d'ordres)

Encara que siguen similars, hi ha algunes diferències entre RUN, CMD i ENTRYPOINT que són importants i cal estudiar. És fàcil distingir RUN: aquesta ordre fa referència a ordres

necessàries internament per a construir la imatge. Aquestes accions s'executen només durant la construcció i no seran accessibles una vegada iniciem els contenidors a partir de la imatge resultant.

- Cada ordre RUN genera una nova capa en la construcció de la imatge.

Imaginem[16] que es tracta d'una imatge (*example_container*) amb un Dockerfile que només instancia una imatge basada en Linux i després mostra un missatge en pantalla.

Si el Dockerfile d'*example_container* conté la línia CMD, aleshores els seus arguments seran l'ordre a executar per omissió.

```
FROM centos
CMD ["/bin/echo", "Hello"]
```

- `docker run -it example_container`, sense paràmetres addicionals, executaria `/bin/echo "Hello"`
- `docker run -it example_container /bin/sh`, executaria un *shell* al contenidor (és a dir, no s'executaria `/bin/echo "Hello"`)

D'altra banda, si el Dockerfile d'*example_container* conté una línia ENTRYPOINT, aleshores no s'atendrà cap argument extern.

```
FROM centos
ENTRYPOINT ["/bin/echo", "Hello"]
```

- `docker run -it example_container /bin/sh`, descartaria l'argument `(/bin/sh)` i executaria `/bin/echo "Hello"`

Finalment, quan es combinen tots dos (CMD i ENTRYPOINT), aleshores s'executa l'ordre referenciada en ENTRYPOINT, utilitzant per omissió els arguments que dóna CMD. Aqueixos arguments poden sustituir-se per aquells utilitzats en el "docker run" corresponent.

```
FROM centos
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["World"]
```

- `docker run -it example_container`, sense paràmetres addicionals, executaria `/bin/echo "Hello World"`
- `docker run -it example_container Moon`, amb aquest paràmetre addicional, executaria `/bin/echo "Hello Moon"`

6.5.2 Exemples de Dockerfile

Pots consultar altres exemples en la documentació addicional i en molts llocs de la web, com https://hub.docker.com/u/komljen/

Exemple 1: client/broker/worker amb ZeroMQ

Aquest exemple es va introduir en la pràctica 2: un servei implantat mitjançant components client, broker (tipus ROUTER-ROUTER) i worker, que podrà replicar-se tantes vegades com siga necessari.

Tant el client com el treballador necessitaran com primer argument l'URL del broker, però la resta d'informacions es deixaran en blanc.

[16] Exemple en https://goinbigdata.com/docker-run-vs-cmd-vs-entrypoint/

- Això suposa que no es negociaran els protocols ni ports de connexió.

Per comoditat s'afig el codi d'aquests 3 components (*myclient*, *mybroker*, *myworker*) en el primer apèndix d'aquest document.

Ens basem en una imatge Docker denominada "centos-zmq" amb...

- NodeJS instal·lat sobre una imatge d'alguna distribució[17] Linux.
- La biblioteca ZeroMQ instal·lada correctament.
- El mòdul zeromq instal·lat i disponible per al seu ús des de node.
- Sobre aquesta imatge es podrà executar qualsevol programa NodeJS que utilitze ØMQ.

El Dockerfile per a crear aquesta imatge "**centos-zmq**" és:

```
# Take the latest CentOS distribution as a base.
# Currently (October 31, 2017), it is CentOS 7.4
FROM centos:7.4.1708
# Install Node.js on that distribution.
# First of all, enable nodesource repo as specified in
#   https://nodejs.org/es/download/package-manager/
RUN curl --silent --location https://rpm.nodesource.com/setup_10.x | bash -
# We may use the yum RedHat/CentOS command to this end.
RUN yum install -y nodejs
# Next step: Install the zeromq library.
# Its package is called "zeromq-devel" in the EPEL
# repository that we'll install first.
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
RUN npm install zeromq@4
```

Passos a seguir per a utilitzar aquest servei...

1. Crear la imatge Docker per al component **broker**.

Necessitem un Dockerfile similar a aquest:

```
FROM tsr1718/centos-zmq
COPY ./mybroker.js mybroker.js
EXPOSE 9998 9999
CMD node mybroker.js
```

- L'ordre COPY suposa que el codi del broker es troba en aquest mateix directori, i ho copia a un directori del contenidor que s'està utilitzant per a construir la imatge.
- Es deixa públic el port 9998 per als clients...
- ... i el 9999 per als "*workers*".

I generem el seu contenidor amb aquesta ordre:

```
docker build –t broker .
```

- Llancem el broker (docker run –name mybroker broker[18]) i esbrinem la seua IP mitjançant...

```
docker inspect mybroker | grep IPAddress | cut -d '"' -f 4
```

- Anotem la IP que apareix (p. ex. A.B.C.D) i la usem per a construir el Dockerfile dels altres 2 components

2. Crear la imatge Docker per al component **client**.

Necessitem un Dockerfile similar a aquest:

[17] CentOS en aquest cas

[18] Amb –name donem un identificador a aquesta instància (contenidor mybroker) per a poder-lo usar com a argument de docker inspect

```
FROM tsr1718/centos-zmq
COPY ./myclient.js myclient.js
# We assume that each worker is linked to the broker
# container.
CMD node myclient tcp://A.B.C.D:9998
```

- I generem la seua imatge des del directori *client* amb aquesta ordre docker:

```
docker build –t client .
```

3. Crear la imatge Docker per al component **worker**.

Necessitem un Dockerfile similar a aquest:

```
FROM tsr1718/centos-zmq
COPY ./myworker.js myworker.js
# We assume that each worker is linked to the broker
# container.
CMD node myworker tcp://A.B.C.D:9999
```

- I generem la seua imatge amb aquesta ordre estant en el directori *worker*:

```
docker build –t worker .
```

Ara ja podem executar aquests processos comprovant que connecten amb el broker i que envien i reben missatges.

Exemple 2: TCPProxy per a un servidor Web

Aquest exemple gira entorn de l'addició d'un component (proxy TCP) a una instal·lació d'un servidor de web monolític extremadament senzill.

El codi per al proxy TCP de la pràctica 1 (ProxyConf.js) és un bon punt de partida per a experimentar un nou desplegament d'un servidor web, de manera que aquest proxy ha de mitjançar en les peticions de servei procedents dels clients, reexpedint-les al port i IP del servidor web.

Desitgem crear el Dockerfile necessari per a aquest component, i, més tard, un **docker-compose.yml** en el qual s'incorporen proxy i servidor web.

És important observar que l'aplicació distribuïda ha de mantindre les mateixes interfícies externes, de manera que el port 80, anteriorment associat al servidor web, ara haurà de vincular-se al proxy TCP. Això NO obliga a canviar el port del component web, sinó la seua visibilitat (p. ex. mitjançant expose) com a accés al servei en el fitxer de desplegament docker-compose.yml.

Dockerfile

Posseeix una part general sobre la qual haurem d'aplicar alguns ajustos:

```
FROM tsr1718/centos-nodejs[19]
```

a) El programa ***Proxy.js*** a executar, que cal copiar prèviament.

```
COPY ./ProxyConf.js Proxy.js
```

b) Fer accessible el port de servei al que atendrà aquest proxy TCP

```
EXPOSE 80
```

[19] Si no disposes d'aquesta imatge, empra centos-zmq

c) Col·locar els paràmetres adequats del component **web** (IP i port) perquè siguen transmesos al programa en la seua invocació.

```
CMD node Proxy IP? port?
```

Per tant, hauríem d'esbrinar abans els detalls necessaris, editar el Dockerfile i executar:

```
docker build -t proxy .
```

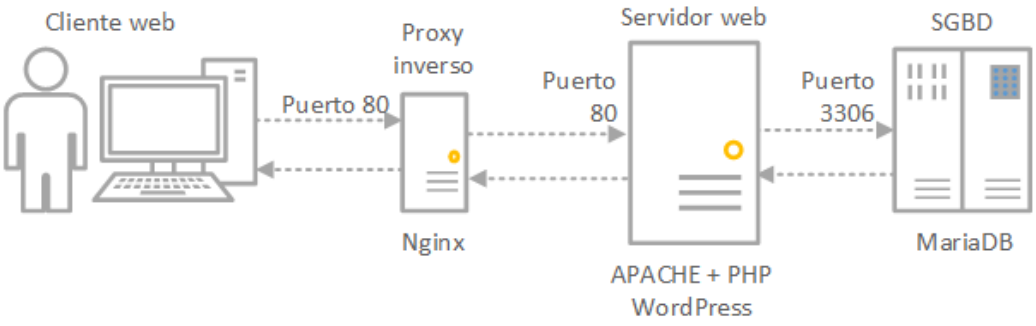
6.6 Múltiples components

En els exemples de serveis multicomponent de l'apartat anterior es facilita l'escalat i la disponibilitat col·locant cada component en un contenidor. Prenguem com a exemple el servei de blog basat en WordPress, construït com:

- El servidor de web APACHE és l'eix central de l'esquema.
  - Inclou un intèrpret de PHP que executa l'aplicació WordPress per a cada petició que li arriba.
  - Habitualment WordPress interactua amb el SGBD per a llegir i escriure informació.
- Per a alleugerir el treball del servidor de web, hi ha un proxy invers HTTP (Nginx) que filtra les peticions senzilles i les atén.

Podem situar el proxy invers, el servidor de web i el SGBD en **contenidors intercomunicats**.

- WordPress requereix la presència de l'intèrpret de PHP i no pot separar-se d'APACHE[20]



El desplegament ara es complica perquè es troben dependències que han de ser resoltes:

- com sap el primer component (proxy invers) els detalls necessaris per a connectar amb el segon (servidor web)?
  - Almenys la seua IP (resta de valors per defecte)
- Pot ser que no es resolguen fins que el segon component inicie la seua execució
- La mateixa pregunta es pot fer per al servidor de web i el SGBD

Simplificadament, es requereix …

1. Crear els Dockerfile dels 3 components
   - Seran un subconjunt del mostrat anteriorment
2. Iniciar el tercer (SGBD), obtenint la seua IP

[20] almenys amb facilitat

3. Iniciar el segon (servidor web), transmetent-li la IP del tercer, obtenint la seua IP
4. Iniciar el primer (proxy invers), transmetent-li la IP del segon

No obstant això, aquesta aproximació artesana és inaplicable per a casos d'envergadura mitjana i gran. Es necessita:

- Llenguatge per a generalitzar la descripció dels desplegaments
  - Que puga expressar components, propietats i relacions
  - P. ex. YAML (per a Compose, Kubernetes), OASI-TOSCA
- Automatitzar (***orquestrar***) l'execució dels desplegaments
  - Mitjançant un motor que executa el desplegament segons la descripció
  - P. ex. Docker-Compose[21], Kubernetes[22], APACHE-Brooklyn

És convenient disposar d'eines que faciliten la creació i simulació d'aquests desplegaments, com https://lorry.io/

- Els imprevistos no són benvinguts mentre es fa un desplegament de gran magnitud!!

Docker admet "enllaços" entre contenidors l'establiment dels quals pot **automatitzar-se mitjançant "docker-compose**".

6.7 Desplegament en Docker Compose

Docker Compose és una aplicació per a definir i executar aplicacions situades en diversos contenidors Docker.

Tres passos:

1. Definir l'entorn de l'aplicació amb un **Dockerfile**
2. Definir els serveis que constitueixen l'aplicació en un arxiu docker-compose.yml perquè puguen executar-se conjuntament
3. Executar docker-compose up, amb el que Compose iniciarà i executarà l'aplicació completa

Limitat a contenidors en el mateix equip, però pot completar-se amb Docker Swarm (o un altre programari d'orquestració) per a controlar un *clúster*.

Aquesta aplicació interactua amb *el daemon* docker i no ve inclosa en Docker.

6.7.1 Característiques destacables

- Es pot disposar de diversos entorns aïllats en un mateix equip
- Es poden situar les dades fora dels contenidors (volums)
- Només cal crear de nou els contenidors que s'hagen modificat
- En docker-compose.yml es poden usar capacitats per a adaptar-se a l'entorn

L'element crucial és el programa docker-compose

```
docker-compose [OPTIONS] [COMMAND] [ARGS...]
```

Cicle típic d'ús:

[21] Només si no afecta a més d'un equip

[22] https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes, https://kubernetes.io/

```
$ docker-compose up -d
… activitats …
$ docker-compose stop
$ docker-compose rm –f
```

6.7.2 Docker Compose des de la CLI

Ordres més significatives:

- **build**: (re-)construeix un servei
- **kill**: deté un contenidor
- **logs**: mostra l'eixida dels contenidors
- **port**: mostra el port associat al servei
- **ps**: llista els contenidors
- **pull**: puja una imatge
- **rm**: elimina un contenidor detingut
- **run**: executa una ordre en un servei
- **scale**: nombre de contenidors a executar per a un servei
- **start** | **stop** | **restart**: inicia | deté | reinicia un servei
- **up**: build + start (*aproximadament*)

Exemple d'ordre **run**

```
$ docker-compose run web python manage.py shell
```

1. Inicia el servei **web**
2. Envia al servei l'ordre **python manage.py shell** perquè l'execute
3. L'execució d'aquesta ordre en el servei es desvincula del nostre shell (des d'on hem llançat docker-compose)

Hi ha dues diferències entre **run** i **start**

- L'ordre que passem a run té preferència sobre la qual s'haja especificat en el contenidor
- Si hi ha ports en col·lisió amb uns altres ja oberts, no es crearan els ports nous

6.7.3 El fitxer de descripció del desplegament docker-compose.yml

https://docs.docker.com/compose/compose-file/

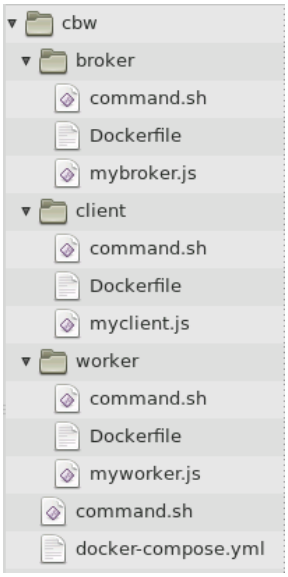
És un arxiu que segueix la sintaxi YAML

A més de les ordres anàlogues als paràmetres de "docker run", les principals són:

- **image**: referència local o remota a una imatge, per nom o tag
- **build**: ruta a un directori que conté un Dockerfile
- **command**: canvia l'ordre a executar en l'inici
- **links**: enllaç a contenidors d'un altre servei.
- **external_links**: enllaços a contenidors externs a compose
- **ports**: ports exposats (millor expressar-los entre cometes)
- **expose**: Ídem, però accessible només a serveis enllaçats (amb links)
- **volumes**: munta rutes com a volums

6.7.4 Completant l'exemple client/broker/worker (aplicació cbw)

En l'exemple amb client, broker i worker desplegat anteriorment, s'ha resolt *artesanalment* la dependència de la resta de components respecte al broker: necessitaven conèixer la seua IP.



Aquest mètode és incompatible amb l'automatització del desplegament, fins i tot amb el moment en què cada informació es requereix: necessitem ***fixar*** el broker abans de poder desplegar la resta!

- Si es canviara la IP del broker, caldria desplegar de nou client i worker?

Ha d'haver-hi una solució millor, i aquesta és una de les aportacions de docker-compose i el seu fitxer de descripció del desplegament.

Suposem que el broker puga anunciar detalls sobre si mateix (com la seua IP), i que existeix una forma d'injectar aquesta informació en els components adequats. Això ho podem especificar en un arxiu de configuració del desplegament docker-compose.yml:

```
version: '2'
services:
  cli:
    image: client
    build: ./client/
    links:
     - bro
    environment:
     - BROKER_URL=tcp://bro:9998
  bro:
    image: broker
    build: ./broker/
    expose:
      - "9998"
      - "9999"
  wor:
    image: worker
    build: ./worker/
    links:
     - bro
    environment:
     - BROKER_URL=tcp://bro:9999
```

També necessitem que els Dockerfile de client i worker puguen consultar aquesta informació. Observareu que hem col·locat l'URL complet per a accedir al socket ZMQ del broker, per la qual cosa l'última línia del Dockerfile dels components hauria de canviar-se per la següent:

- En el Dockerfile del client:

```
CMD node myclient $BROKER_URL
```

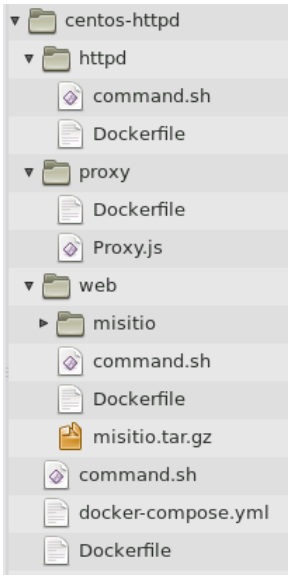
- En el Dockerfile del treballador:

```
CMD node myworker $BROKER_URL
```

6.7.5 Unint TCPProxy i el servidor de web (aplicació proxyweb)

La introducció d'un element com el proxy TCP entre un altre procés (el servidor de web) i l'exterior (els clients) ha de considerar alguns aspectes d'interès:

Interessa que els clients no puguen percebre cap diferència entre l'accés anterior al servidor i l'actual, que travessa un proxy. Això suposa que l'*endpoint* de servei ha de ser mantingut, i nosaltres haurem de preocupar-nos dels detalls:



- L'URL d'invocació serà capturat pel proxy
- El punt d'entrada del servidor serà usat com a destinació pel proxy
- El punt d'entrada del servidor NO podrà ser accedit des de l'exterior

Com ja s'ha avançat en l'apartat anterior, desplegar un component que depèn d'un altre obliga a algun ajust en l'arxiu de configuració de desplegament del component (Dockerfile) i del desplegament del servei (docker-compose.yml). Nomenarem al primer paràmetre (port del servidor) WEB_PORT, i al segon (IP del servidor) WEB_ADDR.

En resum, el Dockerfile per al Proxy serà:

```
FROM tsr1718/centos-nodejs
COPY ./ProxyConf.js Proxy.js
EXPOSE 80
CMD node Proxy $WEB_ADDR $WEB_PORT
```

En el directori en què es troba tant el Dockerfile com l'arxiu ProxyConf.js, executem:

```
docker build -t proxy .
```

docker-compose.yml

Necessitem una secció per al component proxy, destacant:

a) Associació amb el component que hem creat (ordre image)
b) Accés a la informació creada per al component web en el seu desplegament (valor web per a l'ordre links)
c) Registre per a usar un port (80) de l'amfitrió com si es tractara del 8000 (variable LOCAL_PORT) del contenidor (valor “8000:80” per a l'ordre ports)

I també necessitem “ocultar” el port 80 del component web per a impedir que puga ser accedit directament sense passar pel proxy.

a) Retirar ordre ports de la secció web
b) Col·locar valor “80” per a l'ordre expose en la secció web (permet que siga utilitzat internament)

```
proxy:
  image: proxy
  links:
    - web
  ports:
    - "8000:80"
  environment:
    - WEB_PORT=80
    - WEB_ADDR=web
web:
  image: misitio
  command: /usr/sbin/httpd -D FOREGROUND
```

```
  expose:
    - "80"
```

Si s'han seguit els passos anteriors, i ens col·loquem en el directori que conté aquest fitxer de configuració, per a fer funcionar la màgia i arrancar una instància de cadascun dels components executarem:

```
docker-compose up
```

6.8 Múltiples nodes

L'objectiu de disseny de Compose es limita a components que has d'executar en un únic node; no obstant això, l'escalabilitat no pot procedir del repartiment dels recursos d'un node entre els components de l'aplicació, sinó de l'agregació d'altres nodes amb els seus recursos a la nostra aplicació.

- La tolerància a fallades, amb un sol node, queda completament desnaturalitzada.
- La mera concepció d'una aplicació distribuïda limitada a un node és contradictòria.

Què es podria necessitar perquè els sistemes Docker de múltiples nodes puguen interactuar i integrar-se com si d'un sistema únic es tractara? Un director d'orquestra que els coordine.

- El programari d'orquestració posarà en contacte a tots els nodes entre si, oferint propietats al sistema i funcionalitat a les aplicacions.

En general, les aplicacions més veteranes dissenyades amb aquest propòsit van nàixer a l'entorn de la virtualització i del núvol. Alguns casos destacables són Kubernetes (de Google) i Apache Mesos. Amb l'arribada de les tecnologies de contenerització, aquests sistemes també s'han adaptat per a interactuar amb Docker, però han de rivalitzar amb la proposta nativa denominada "Docker en **mode Swarm**", incorporada a partir de la versió 1.12[23] de Docker.

- El vencedor és clarament Kubernetes, de tal forma que l'equip de Docker va decidir facilitar la integració amb aquest sistema, reduint les possibilitats d'èxit de Swarm en aquest àmbit.

Malgrat que el nostre temps de laboratori és limitat i exclou l'ús d'aquesta opció multinode, creiem imprescindible conèixer els elements més rellevants de k8s.

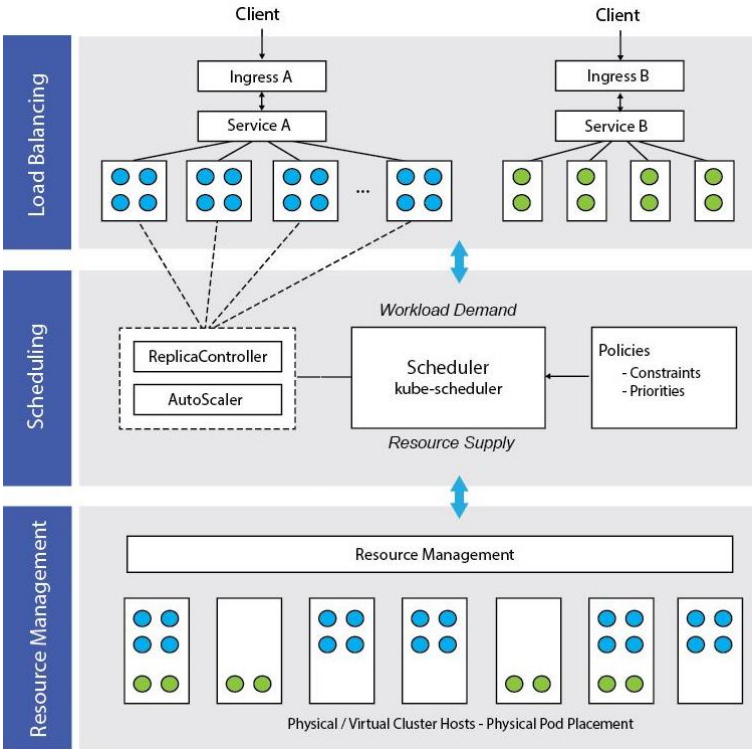
7 KUBERNETES

Kubernetes és un programari de codi lliure que va començar en 2014 a partir d'un projecte (**Borg**) de Google dissenyat per a automatitzar el desplegament i escalat d'aplicacions conteneritzades. K8s, que és la seua abreviatura, és el orquestrador de contenidors més recolzat en la comunitat, destacant organitzacions i empreses de primer ordre com Linux Foundation, Cisco, Docker, Google, RedHat, IBM, Microsoft, Oracle, Suse, VMware, Ebay, SAP i Yahoo!.

[23] Anteriorment existia un Swarm-kit limitat i incompatible amb el sistema actual

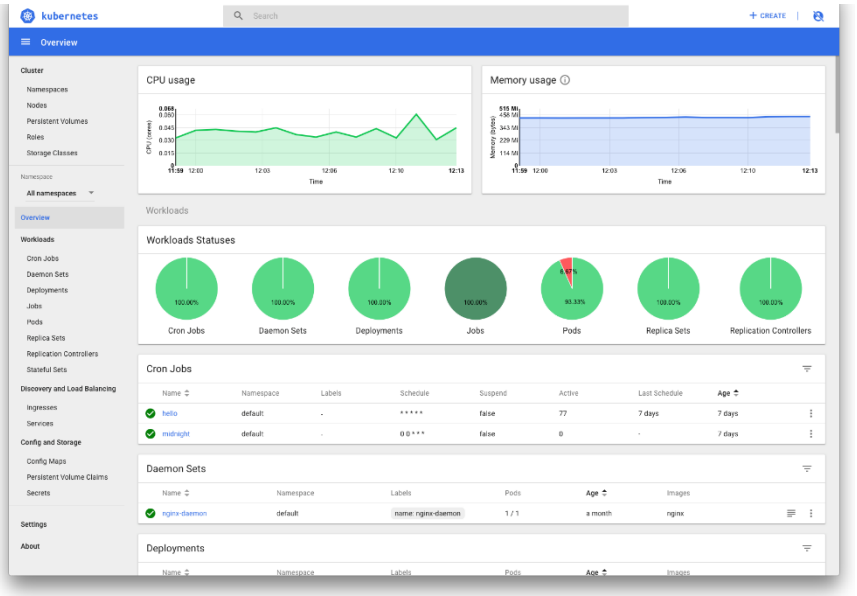
K8s proporciona tot el necessari per a mantindre en producció la nostra aplicació distribuïda, de forma portable, extensible i automatitzable. S'encarrega de …

- Muntatge de volums per a la seua persistència (múltiples possibilitats).
- Distribució de secrets i gestor de la configuració.
- Gestió del cicle de vida dels contenidors.
- Replicació de contenidors.
- Autoescalat horitzontal.
- Descobriment i equilibrat de serveis.
- Monitoratge.
- Accés a anotacions de seguiment i depuració.



Existeixen múltiples eines construïdes al voltant de k8s, destacant…

- Kubectl (interfície de línia d'ordres per a controlar el clúster)
- Kubeadm (interfície per a aprovisionar un clúster sobre servidors virtuals o reals)
- Kubefed (interfície per a administrar una federació de clústers)
- Minikube (eina que facilita la construcció d'un clúster k8s mononode local per a desenvolupament)
- Dashboard (interfície web per a desplegar i gestionar aplicacions conteneritzades d'un clúster k8s)



- Helm

(https://helm.sh/, gestió de paquets amb recursos k8s preconfigurats, anomenats charts)

- Kompose (facilita el pas de Docker Compose a k8s)

7.1 Elements clau de k8s

A manera introductòria, en k8s trobem:

- **Clúster**: Conjunt de màquines físiques o virtuals i altres recursos utilitzats per k8s.
- **Node**: Una màquina física o virtual executant-se en k8s en la qual es poden programar *pods*.
- **Pod**: És la unitat més xicoteta desplegable que pot ser creada, programada i manejada per k8s. És un grup d'un o més contenidors amb emmagatzematge compartit entre ells (mateixa màquina) i les opcions específiques de cadascun per a executar-los.
    - Els contenidors del mateix pod són visibles entre si mitjançant *localhost*.
    - En termes de Docker, un pod és un conjunt de contenidors amb *namespace* i volums compartits.
- **Controlador de replicació** (*replication controller*, que abreugem com *rc*): S'encarrega del cicle de vida d'un grup de pods, i defineix les seues polítiques de restauració. Assegura que estiga executant-se la quantitat especificada de rèpliques del pod. Permet escalar de forma fàcil els sistemes i maneja la recuperació d'un pod quan ocorre una fallada.
- **Controlador de desplegament** (*deployment controller*): S'encarrega de l'actualització d'una aplicació distribuïda.
- **Servei**: És una abstracció que defineix un conjunt de pods i la lògica per a accedir a ells.
- **Espais de noms** (*namespaces*): Estableix un nivell addicional de separació entre contenidors que comparteixen recursos d'un clúster.
- ***Configmap***: Servei per a gestionar la configuració de les aplicacions.
- **Secrets**: Servei per a gestionar informació privada (p. ex. credencials) de les nostres aplicacions.
- **Volums**: Servei per a gestionar la persistència dels contenidors.

7.2 Treballant amb k8s

En aquest apartat esmentarem les operacions bàsiques sobre pods, controladors de replicació, serveis i accés web.

7.2.1 Pods

Els pods són **entitats efímeres** amb un cicle de vida que, en la seua creació, els assigna un UID vàlid fins que els pods acaben o s'eliminen. Un pod pot ser reemplaçat en un altre node, rebent un nou UID.

Els pods poden utilitzar-se per a l'escalat horitzontal, però és preferible disposar de microserveis en contenidors diferents per a aconseguir un sistema distribuït més robust.

Els pods es poden crear mitjançant kubectl, directament per línia d'ordres o a través d'un fitxer de tipus YAML. Veiem ambdues alternatives a continuació.

Creació d'un pod per línia d'ordres

Usem kubectl per a interactuar per línia d'ordres amb la API de k8s:

```
kubectl run my-nginx --image=nginx --port=80
```

- El primer paràmetre indica l'acció (arrancar un pod)
- Després el nom que volem assignar-li (en aquest cas *my-nginx*)
- El tercer és la imatge a partir de la qual es va a instanciar el pod (es diu *nginx*)
- Finalment, el port en el qual escoltarà

Nota: la creació d'un pod per línia d'ordres inclou implícitament un rc que s'encarregue de restaurar el pod quan siga eliminat, perquè la política de restauració per defecte és *Always*

Per a consultar informació del pod i del rc executem…

```
kubectl get pods
kubectl get rc
```

```
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# kubectl get pods
NAME             READY     REASON    RESTARTS   AGE
my-nginx-48onk   1/1       Running   0          2m
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR       REPLICAS
my-nginx     my-nginx       nginx      run=my-nginx   1
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
```

Pot comprovar-se que el nom del pod coincideix amb el del rc esmentat en l'ordre (*my-nginx*) seguit d'un identificador únic per a cada pod.

Per a eliminar el controlador de replicació juntament amb el pod executem…

```
kubectl delete rc my-nginx
```

Creació d'un pod mitjançant un fitxer YAML

- **/opt/kubernetes/examples/nginx/nginx-pod.yaml**

```
# Número de versió de l'API a utilitzar
apiVersion: v1
# Tipus de fitxer que es crearà.
kind: Pod
# Dades pròpies del pod com el nom i les etiquetes que té associats per a seleccionar-ho
metadata:
    name: my-nginx
    # Especifiquem que el pod tinga una etiqueta amb clau "app" i valor "nginx"
    labels:
        app: nginx
# Conté l'especificació del pod
spec:
    # Ací es nomenen els contenidors que formen part d'aquest pod, tots visibles per localhost
    containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
    # Política de recuperació si el pod es deté o acaba per a una fallada interna.
    restartPolicy: Always
```

Procedim a crear el mateix pod però des del fitxer que acabem de crear:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-pod.yaml
```

En aquesta modalitat podem observar que no es crea cap controlador de replicació.

```
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get pod
NAME       READY     REASON    RESTARTS   AGE
my-nginx   1/1       Running   0          1m
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx#
```

Eliminem aquest pod mitjançant…

```
kubectl delete pod my-nginx
```

7.2.2 Controlador de replicació

El **controlador de replicació** (rc = *replication controller*) s'usa per a assegurar que sempre hi haja algun pod disponible. Si hi ha molts pods, n'eliminarà alguns; si n'hi ha pocs, en crearà de nous.

Creació d'un rc

En aquest exemple crearem un rc anomenat **nginx-rc.yaml** encarregat d'executar un servidor nginx:

- **/opt/kubernetes/examples/nginx/nginx-rc.yaml**

```
# Número de versió de l'API que es vol utilitzar
apiVersion: v1
# Tipus de fitxer que es crearà.
kind: ReplicationController
# Dades pròpies del controlador de replicació
metadata:
    # Nom del controlador de replicació
    name: my-nginx
# L'especificació de l'estat desitjat que volem que tinga el pod.
spec:
    # Nombre de rèpliques que volem que el rc mantinga (#això crearà un pod)
    replicas: 1
    # En aquesta propietat s'indiquen tots els pods que aquest rc gestionarà. en aquest cas,
    # tots els que tinguen el valor "nginx" en l'etiqueta "app"
    selector:
        app: nginx
    # Aquesta propietat té el mateix esquema intern que un pod, però no necessita
    # "apiVersion" ni "kind" perquè ja ha sigut especificat
    template:
        metadata:
            name: nginx
            labels:
                app: nginx
        spec:
            containers:
                - name: nginx
                  image: nginx
                  ports:
                    - containerPort: 80
```

Ara crearem el rc amb l'ordre:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-rc.yaml
```

Veiem com ara se'ns ha creat un pod amb un UID.

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get pods
NAME            READY     REASON    RESTARTS   AGE
my-nginx-l15ux   1/1       Running   0          1m
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx#
```

També podem comprovar l'estat del nostre controlador de replicació (nom, número de pods i els seus respectius estats, ...) a través de l'ordre kubectl:

```
kubectl describe rc my-nginx
```

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl describe rc my-nginx
W0318 10:00:38.855526   10502 request.go:302] field selector: v1 - events - involvedObject.namespace - default: need to check if this is vers
W0318 10:00:38.856097   10502 request.go:302] field selector: v1 - events - involvedObject.kind - ReplicationController: need to check if thi
W0318 10:00:38.856432   10502 request.go:302] field selector: v1 - events - involvedObject.uid - d18350fd-ecef-11e5-a835-0800273a0b5b: need t
W0318 10:00:38.856710   10502 request.go:302] field selector: v1 - events - involvedObject.name - my-nginx: need to check if this is versione
Name:           my-nginx
Image(s):       nginx
Selector:       app=nginx
Labels:         app=nginx
Replicas:       1 current / 1 desired
Pods Status:    1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Events:
  FirstSeen                             LastSeen                        Count   From                            SubobjectPath   Reason
  Fri, 18 Mar 2016 09:57:26 +0000       Fri, 18 Mar 2016 09:57:26 +0000 1       {replication-controller }                       successfulCre

root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx#
```

Treballant amb controladors de replicació

Una vegada creat, un rc et permet:

- **Escalar-ho:** pots triar el nombre de rèpliques que té un pod de forma dinàmica.
- **Eliminar el rc:** pots esborrar només el controlador de replicació o esborrar-lo amb tots els pods dels quals s'encarrega
- **Aïllar al pod del rc:** Els pods poden no pertànyer a un controlador de replicació canviant les etiquetes. El pod eliminat serà substituït per un altre nou que crearà el rc.

Per a afegir un pod controlat pel rc executem…

```
kubectl scale rc my-nginx --replicas=2
```

Per a eliminar un rc executem…

```
kubectl delete rc my-nginx
```

7.2.3 Serveis

Com ja sabem, els pods són elements volàtils amb una adreça IP que pot canviar. Els serveis permeten que un pod puga comunicar-se amb un altre.

Un servei és una **abstracció que defineix un grup lògic de pods i una política d'accés a aquests**. Els pods apunten a un servei mitjançant la propietat label. En el cas anterior[24], si alguna cosa causara la destrucció d'aquest pod, el rc en crearia un de nou amb una nova IP, de manera que la resta de la infraestructura que depenguera d'aquest pod per aquesta IP fixa deixaria de funcionar. **El servei aconsegueix que el pod sempre siga accessible de la mateixa manera**. Per a il·lustrar-ho, crearem un servei i el farem accessible fora del clúster.

Creació d'un servei

- **/opt/kubernetes/examples/nginx/nginx-svc.yaml**

```
# Número de versió de l'API a utilitzar
apiVersion: v1
# Tipus de fitxer que es crearà.
kind: Service
# Dades pròpies del pod com el nom i les etiquetes associats per a seleccionar-ho
metadata:
    name: my-nginx-service
# Conté l'especificació del pod
spec:
    # En aquesta propietat s'indiquen tots els pods que apunten a aquest servei. En aquest cas,
    # s'encarregarà de tots els que tinguen el valor "nginx" en l'etiqueta "app"
    selector:
        app: nginx
    ports:
      # Indica el port en el qual aquest servei atén peticions
      - port: 80
```

[24] amb un rc encarregat d'executar un pod amb un contenidor nginx

Una vegada creat el fitxer comencem el servei amb l'ordre:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-svc.yaml
```

```
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl create -f /opt/kubernetes/examples/nginx/ng
 services/my-nginx-service
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get pods
 NAME       READY     REASON    RESTARTS   AGE
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get rc
 CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS
[root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl get svc
 NAME               LABELS                                    SELECTOR    IP(S)           PORT(S)
 kubernetes         component=apiserver,provider=kubernetes   <none>      192.168.3.1     443/TCP
 my-nginx-service   <none>                                    app=nginx   192.168.3.110   80/TCP
 root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx#
```

Ara habilitem el rc d'aquest servei, que és el que hem creat anteriorment:

```
kubectl create -f /opt/kubernetes/examples/nginx/nginx-rc.yaml
```

Per a accedir a l'aplicació del pod des de dins del clúster necessitem saber la IP que té actualment el pod. Usem l'ordre kubectl:

```
kubectl get -o template pod my-nginx-m5mni --template={{.estatus.podIP}}
```

A continuació accedim al servei mitjançant aquesta IP usant un client HTTP (p. ex. el programa curl).

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# kubectl get -o template pod my-nginx-m5mni --t
172.17.0.1root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster# curl 172.17.0.1:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
```

De moment aquest pod només és accessible dins del clúster. El servei ens serveix per a abstraure'ns del pod en qüestió que estem utilitzant, permetent disposar de múltiples aplicacions de *backend* darrere d'un únic *frontend*.

Accés a un servei des de fora del clúster

Necessitarem afegir la propietat **type: NodePort** al servei, exposant el servei en cada node del clúster, i aconseguint contactar amb el servei des de qualsevol IP dels nodes. El nostre servei quedaria de la següent manera:

- **/opt/kubernetes/examples/nginx/nginx-svc.yaml**

```
# Número de versió de l'API a utilitzar
apiVersion: v1
# Tipus de fitxer que es crearà.
kind: Service
# Ací van les dades pròpies del pod com el nom i les seues etiquetes
metadata:
    name: my-nginx-service
# Conté l'especificació del pod
spec:
    type: NodePort
    # En aquesta propietat s'indiquen tots els pods que apunten a aquest servei. En aquest cas,
    # s'encarregarà de tots els que tinguen el valor "nginx" en el label "app"
    selector:
        app: nginx
    ports:
      # Indica el port en el qual s'hauria de servir aquest servei
      - port: 80
```

Eliminem l'anterior servei i afegim el nou amb les ordres que ja sabem, i comprovem què port extern s'ha associat amb la connexió:

```
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl delete svc my-nginx-service
services/my-nginx-service
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# vim nginx-svc.yaml
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx# kubectl create -f nginx-svc.yaml

                            You have exposed your service on an external port on all nodes in your cluster.
                            If you want to expose this service to the external internet, you may need to set
                            firewall rules for the service port(s) (tcp:80) to serve traffic.

                            See https://github.com/GoogleCloudPlatform/kubernetes/tree/master/docs/services-
                            services/my-nginx-service
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/examples/nginx#
```

```
Name:                   my-nginx-service
Labels:                 <none>
Selector:               app=nginx
Type:                   NodePort
IP:                     192.168.3.158
Port:                   <unnamed>       80/TCP
NodePort:               <unnamed>       32138/TCP
Endpoints:              172.17.0.1:80
Session Affinity:       None
No events.
root@vagrant-ubuntu-trusty-64:/opt/kubernetes/cluster#
```

Veiem que la connexió s'associa al port 32138 (automàtic). Ara hauríem de canviar la configuració de l'amfitrió per a permetre l'accés a aquest port des de localhost o una altra IP (els detalls depenen de cada cas).

Finalment accedim des del navegador a http://localhost:32138 (pot tardar una mica a mostrar-se)

7.2.4 Accés a la interfície gràfica

K8s compta amb una sèrie d'afegits (*add-ons*) accessibles a través del seu API. Ens interessa usar un, la interfície gràfica, per al que en primer lloc habilitem el *namespace* kube-system, (en el fitxer **/opt/kubernetes/cluster/ubuntu/namespace.yaml**). Executem…

```
kubectl create -f /opt/kubernetes/cluster/ubuntu/namespace.yaml
```

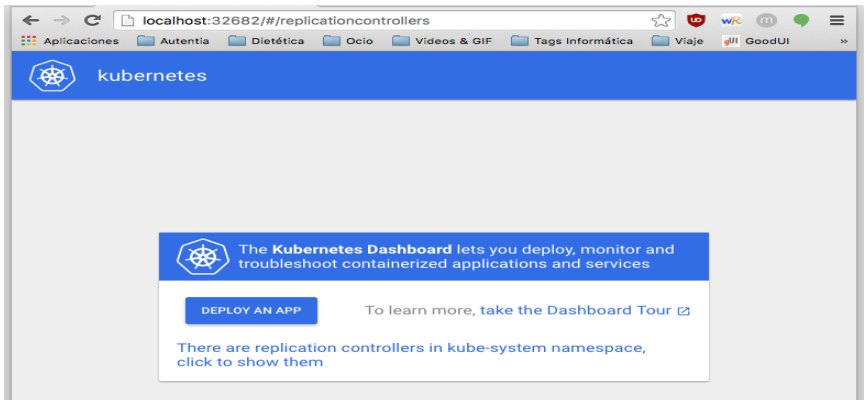
Ara habilitem la interfície gràfica, creant el rc i el servei que vénen en el directori **/opt/kubernetes/cluster/addons/dashboard**, però prèviament hem d'exposar el servei fora del clúster per a poder accedir des del navegador. Per a fer això simplement afegim **type: NodePort** just després del spec.

- **/opt/kubernetes/cluster/addons/dashboard/dashboard-service.yaml**

```
apiVersion: v1
kind: service
metadata:
  name: kubernetes-dashboard
  namespace: kube-system
  labels:
    k8s-app: kubernetes-dashboard
    kubernetes.io/cluster-servicio: "true"
spec:
  type: NodePort
  selector:
    k8s-app: kubernetes-dashboard
  ports:
  - port: 80
    targetPort: 9090
```

Ara ja estem llestos per a iniciar la interfície gràfica amb les següents ordres…

```
kubectl create -f dashboard-controller.yaml
kubectl create -f dashboard-service.yaml
```



7.3 Referències per a Kubernetes

- **Tutorial Online oficial:** En https://kubernetes.io/docs/tutorials/ podeu fer un curs interactiu per a usar un minikube en el vostre equip i aplicacions per a les vostres proves i aprendre els conceptes bàsics.
  - Especialment la **part bàsica** (https://kubernetes.io/docs/tutorials/kubernetes-basics/) composta per 5 passos:
    - Crear un clúster k8s (/cluster-intro/)
    - Desplegar una aplicació (/deploy-intro/)
    - Explorar l'aplicació (/explore-intro/)
    - Exposar l'aplicació (/expose-intro/)
    - Escalar l'aplicació (/scale-intro/)
    - Actualitzar l'aplicació (/update-intro/)
- Curs Katakoda: https://www.katacoda.com/courses/kubernetes

També poden trobar-se diverses opcions per a experimentar amb k8s:

- **Minikube** és el mètode recomanat per a crear un clúster de k8s local d'un sol node per a desenvolupament i proves. La configuració és completament automàtica i no requereix un compte de proveïdor del núvol.
- **Kubeadm-dind** és un clúster k8s multinod que només requereix un procés docker.
- **PWK** El lloc PWK (http://play-with-k8s.com/) permet muntar clústers de k8s i llançar serveis replicats de manera ràpida i senzilla durant un màxim de 4 hores. Es tracta d'un entorn on disposem de diverses instàncies de Docker sobre les quals podem usar kubeadm per a instal·lar i configurar k8s, creant un clúster en menys d'un minut.

8 APÈNDIXS

8.1 Codi de client/broker/worker

Amb paràmetres des de la línia d'ordres, en les condicions comentades anteriorment en aquest document.

myclient.js

```
01: // myclient in NodeJS with URL & id arguments
02: const zmq = require('zeromq')
03: let req = zmq.socket('req')
04:
05: let args = process.argv.slice(2)
```

```
06: let brokerURL = args[0] || 'tcp://localhost:9998'
07: let myID = args[1] || 'CID'+parseInt(Math.random()*10000)
08: let myMsg = args[2] || 'Hola'
09:
10: req.identity = myID
11: req.connect(brokerURL)
12: req.on('message', (msg)=> {
13:   console.log('resp: '+msg)
14:   process.exit(0);
15: })
16: req.send(myMsg)
```

mybroker.js

```
01: // ROUTER-ROUTER request-reply broker in NodeJS
02: const zmq = require('zeromq')
03: let cli=[], req=[], workers=[]
04:
05: let args = process.argv.slice(2)
06: let fePortNbr = args[0] || 9998
07: let bePortNbr = args[1] || 9999
08:
09: let sc = zmq.socket('router') // frontend
10: let sw = zmq.socket('router') // backend
11: sc.bind('tcp://*:'+fePortNbr)
12: sw.bind('tcp://*:'+bePortNbr)
13: sc.on('message',(c,sep,m)=> {
14:   if (workers.length==0) {
15:     cli.push(c); req.push(m)
16:   } else {
17:     sw.send([workers.shift(),'',c,'',m])
18:   }
19: })
20: sw.on('message',(w,sep,c,sep2,r)=> {
21:   if (c=='') {workers.push(w); return}
22:   if (cli.length>0) {
23:     sw.send([w,'',
24:       cli.shift(),'',req.shift()])
25:   } else {
26:     workers.push(w)
27:   }
28:   sc.send([c,'',r])
29: })
```

myworker.js

```
01: // myworker server in NodeJS with URL & id arguments
02: const zmq = require('zeromq')
03: let req = zmq.socket('req')
04:
05: let args = process.argv.slice(2)
06: let backendURL = args[0] || 'tcp://localhost:9999'
07: let myID = args[1] || 'WID'+parseInt(Math.random()*5000)
08: let replyText = args[2] || 'resp'
09:
10: req.identity = myID
11: req.connect(backendURL)
12: req.on('message', (c,sep,msg)=> {
13:   setTimeout(()=> {
14:     rep.send([c,'',replyText])
15:   }, 1000)
16: })
17: req.send(['','',''])
```

8.2 Xicotets trucs per a Docker

- **Identificadors més recents**

```
alias dl='docker ps -l -q'
docker run ubuntu echo hello world
docker commit $(dl) helloworld
```

- **Consultar IP**

```
docker inspect $(dl) | grep IPAddress | cut -d '"' -f 4
```

- **Assignar port**

```
docker inspect -f '{{range $p, $conf := .NetworkSettings.Ports}} {{$p}} -> {{(index $conf
0).HostPort}} {{end}}' <id_contenidor>
```

- **Localitzar contenidors mitjançant expressió regular**

```
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d" "); do echo $i; done
```

- **Consultar entorn**

```
docker run --rm ubuntu env
```

- **Finalitzar tots els contenidors en execució**

```
docker kill $(docker ps -q)
```

- **Eliminar contenidors vells**

```
docker ps -a | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```

- **Eliminar contenidors detinguts**

```
docker rm -v $(docker ps -a -q -f status=exited)
```

- **Eliminar imatges *penjades***

```
docker rmi $(docker images -q -f dangling=true)
```

- **Eliminar totes les imatges**

```
docker rmi $(docker images -q)
```

- **Eliminar volums *penjats***

```
docker volume rm $(docker volume ls -q -f dangling=true)
```

(En 1.9.0, dangling=false no funciona i mostra tots els volums)

- **Mostrar dependències entre imatges**

```
docker images -viz | dot -Tpng -o docker.png
```

- **Monitorizar el consum de recursos en executar contenidors**

Per a esbrinar el consum de CPU, memòria i xarxa d'un únic contenidor, pots usar:

```
docker stats <container>
```

Per a tots els contenidors, ordenats per id:

```
docker stats $(docker ps -q)
```

Ídem ordenats per nom:

```
docker stats $(docker ps --format '{{.Names}}')
```

Ídem seleccionant els que procedeixen d'una imatge:

```
docker ps -a -f ancestor=ubuntu
```

8.3 Docker-compose.yml

Aquest annex **no** és exhaustiu, i docker-compose es troba activament **en evolució**, de manera que la documentació original és una font irrenunciable d'informació i referència.

- Disposes de material de referència en la "**Docker reference documentation**" d'aquest mateix tema.

L'arxiu docker-compose.yml especifica les característiques per al desplegament mitjançant docker-compose d'una aplicació distribuïda. El seu contingut són línies de text que s'ajusten a una sintaxi especificada a partir de YAML (https://docs.docker.com/compose/yml/). A destacar:

- No poden usar-se tabuladors, tan sols espais en blanc.
- Les propietats i llistes es deuen indentar amb un espai o més.
- Les majúscules i minúscules són significatives punt en els identificadors de propietats com en els de claus.

Cadascun dels serveis que es definisquen en docker-compose.yml ha d'especificar exactament una imatge. La resta de claus són opcionals, i són anàlogues als seus corresponents per a l'ordre run de Docker. Les ordres incloses en el Dockerfile no necessiten repetir-se en docker-compose.yml.

image

Identificador de la imatge, local o remota. Compose intentarà obtenir-la (*pull*) si no es disposa d'ella localment.

```
image: ubuntu
image: orchardup/postgresql
image: a4bc65fd
```

build

Ruta a un directori amb el Dockerfile. Si es tracta d'una ruta relativa, ho serà respecte a la de l'arxiu yml. Compose ho construirà i designarà amb un nom.

```
build: ./dir

build:
  context: ./dir
  dockerfile: Dockerfile-alternate
  args:
    buildno: 1
```

dockerfile: Ruta de l'arxiu dockerfile per a construir la imatge.

command

Substitueix a l'ordre per defecte.

```
command: bundle exec thin -p 3000
```

links

Enllaça amb contenidors d'un altre servei. Especifica el nom del servei i l'àlies per al link (SERVEI:ALIAS), o només el nom si es mantindrà el mateix en l'àlies.

```
web:
 links:
  - db
  - db:database
  - redis
```

El nom del servei actua com a nom DNS del servidor llevat que s'haja definit un àlies.

Els enllaços expressen dependències entre serveis, i influeixen en l'ordre d'arrencada del servei.

external_links

Enllaços a contenidors externs a aquest compose.yml, o fins i tot externs a Compose, especialment per als quals ofereixen serveis compartits. La semàntica d'aquesta ordre és similar a links quan s'usa per a especificar el nom del contenidor i l'àlies de l'enllaç (CONTENIDOR:ALIES).

```
external_links:
 - redis_1
 - project_db_1:mysql
 - project_db_1:postgresql
```

ports

Ports exposats, bé indicant tots dos (EQUIP:CONTENIDOR), o bé indicant únicament el de l'equip contenidor (es prendrà un port de l'amfitrió a l'atzar).

Nota: Es recomana especificar els ports i la seua correspondència com a cadenes amb cometes.

```
ports:
 - "3000"
 - "8000:8000"
 - "49100:22"
 - "127.0.0.1:8001:8001"
```

expose

Exposa ports sense publicar-los a l'equip amfitrió, de manera que únicament tindran accés els serveis enllaçats dins de Docker. Només es pot indicar el port intern.

```
expose:
 - "3000"
 - "8000"
```

volumes

Munta rutes com a volums, especificant opcionalment una ruta en l'amfitrió (HOST:CONTENIDOR), o una manera d'accés (HOST:CONTENIDOR:ro).

```
volumes:
 # Just specify a path and let the Engine create a volume
 - /var/lib/mysql

 # Specify an absolute path mapping
 - /opt/data:/var/lib/mysql

 # Path on the host, relative to the Compose file
 - ./cache:/tmp/cache

 # User-relative path
 - ~/configs:/etc/configs/:ro
```

```
# Named volume
- datavolume:/var/lib/mysql
```

Es pot muntar una ruta relativa a l'amfitrió, que serà expandida de forma relativa al directori de l'arxiu de configuració de Compose que s'estiga emprant. Les rutes relatives comencen sempre per . o ..

environment

Afig variables d'entorn

Els valors de les variables d'entorn que només disposen d'una clau es calculen en la màquina en la qual s'executa Compose, sent especialment útil per a valors secrets o a mesura de l'amfitrió.

```
environment:
  RACK_ENV: development
  SESSION_SECRET:

environment:
  - RACK_ENV=development
  - SESSION_SECRET
```

labels

Afig metadades a contenidors mitjançant labels de Docker. Pot triar-se entre array i diccionari.

```
labels:
  com.example.description: "Accounting webapp"
  com.example.department: "Finance"
  com.example.label-with-empty-value: ""

labels:
  - "com.example.description=Accounting webapp"
  - "com.example.department=Finance"
  - "com.example.label-with-empty-value"
```

8.4 Alguns exemples de casos d'ús de Docker

Un lloc de referència inevitable per a trobar exemples i aplicacions conteneritzades és https://hub.docker.com/explore/

En aquest apartat donem cabuda a alguns exemples que pretenen il·lustrar des de l'ús de Docker per a recobrir el desplegament d'una aplicació gràfica d'escriptori (OpenOffice), i un contenidor que actua com GUI de Docker (portainer).

8.4.1 LibreOffice en un contenidor

Contingut del Dockerfile

```
FROM debian:stretch
MAINTAINER Jessie Frazelle <jess@linux.com>

RUN apt-get update && apt-get install -y \
    libreoffice \
    --no-install-recommends \
    && rm -rf /var/lib/apt/lists/

ENTRYPOINT ["libreoffice"]
```

Invocació en línia d'ordres

```
docker run -d \
  -v /etc/localtime:/etc/localtime:ro \
```

```
  -v /tmp/.X11-unix:/tmp/.X11-unix \
  -e DISPLAY=unix$DISPLAY \
  -v $HOME/slides:/root/slides \
  -e GDK_SCALE \
  -e GDK_DPI_SCALE \
  --name libreoffice \
  jess/libreoffice
```

Una altra **alternativa**, sense necessitat de Dockerfile (d'http://linoxide.com/how-tos/20-docker-containers-desktop-user/)

```
xhost +local:docker
docker run \
-v $HOME/Documents:/home/libreoffice/Documents:rw \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-e uid=$(id -o) -e gid=$(id -g) \
-e DISPLAY=unix$DISPLAY --name libreoffice \
chrisdaish/libreoffice
```

8.4.2 Portainer

Exemple d'aplicació que accedeix al socket de Docker, mitjançant el qual pot conèixer l'estat dels altres contenidors allotjats en aquest amfitrió.

- Es troba instal·lat en els nostres servidors de portal

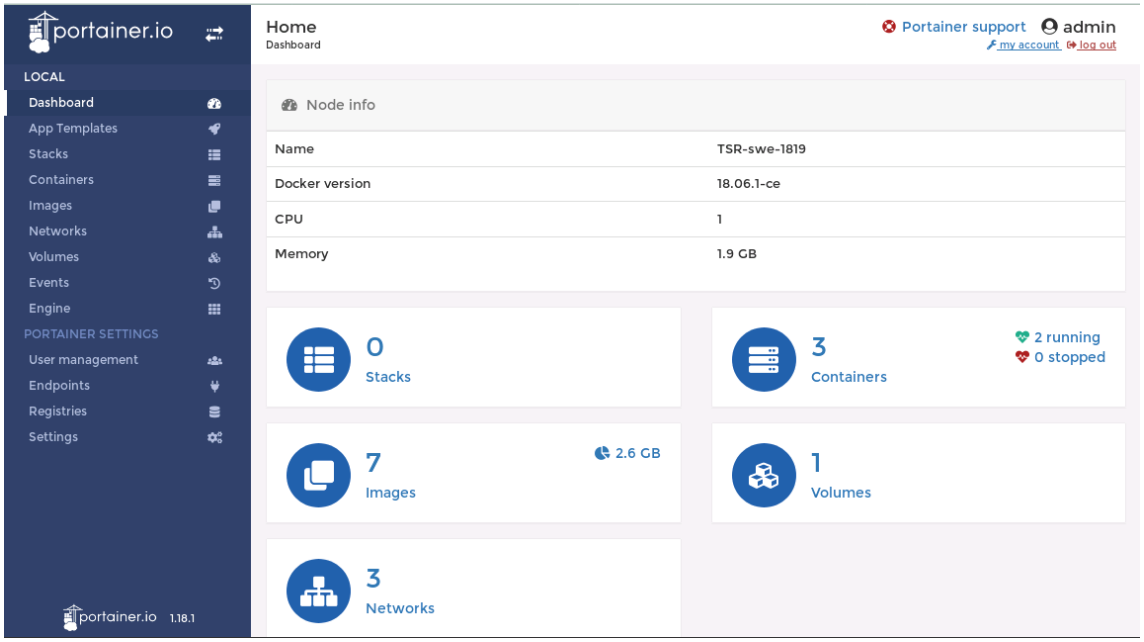
```
#!/bin/bash
# URL: https://portainer.io

#docker volume create portainer_data
docker run -d -p 9999:9000 --name portainer --restart always -v /var/run/docker.sock:/var/run/docker.sock -v
/root/Downloads/.portainer_data:/data portainer/portainer
```

Accedim amb el navegador a l'URL http://localhost:9999/

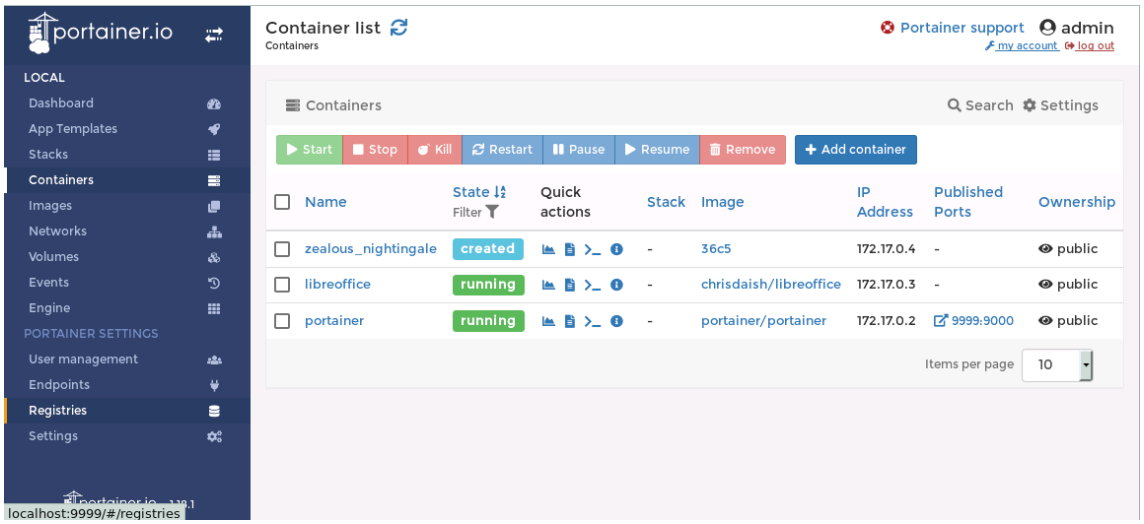
El primer accés permet crear una contrasenya per a l'usuari admin. En el nostre cas és la mateixa que la inicial de root[25] en els servidors virtuals de portal.

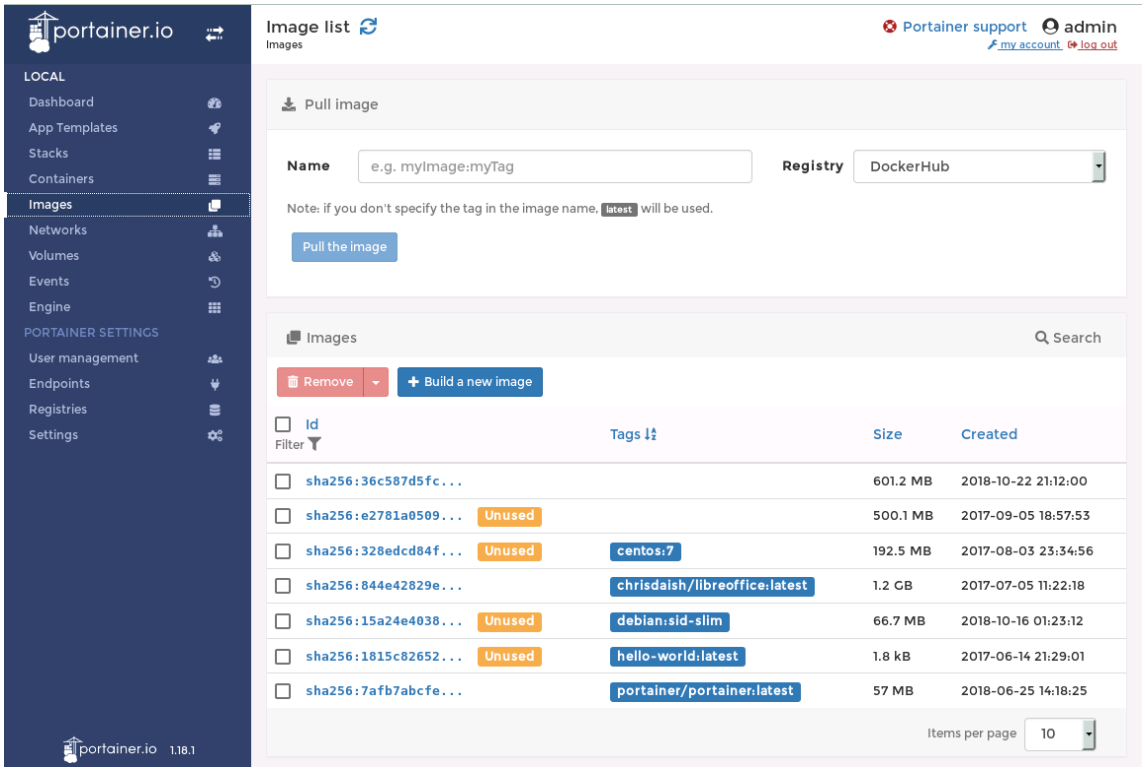
Queda a la nostra disposició el tauler de control (dashboard) de l'aplicació. Pot observar-se un menú a l'esquerra per a seleccionar apartats dins de l'aplicació.

[25] GalYMatias



A títol il·lustratiu es mostra l'aspecte dels apartats *Containers* i *Images* d'un cas real:





- Més informació en portainer.io

9 REFERÈNCIES

9.1 En la web

- **www.docker.com** (lloc oficial de Docker) Atenció a les diferències entre les versions 1, 2 i 3 de Compose, i a la possible confusió entre SwarmKit (obsolet) i el mode Swarm
  - docs.docker.com/userguide/ (**documentació oficial**)
  - docs.docker.com/compose/ (Compose)
    - docker-compose.yml: https://docs.docker.com/compose/compose-file/
  - docs.docker.com/samples/
    - Exemples il·lustratius i variats
  - docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/
    - Consells sobre Dockerfile
- **github.com/wsargent/docker-cheat-sheet**
  - Un resum molt encertat sobre ordres i fitxers de Docker (sense Compose)
- **github.com/veggiemonk/awesome-docker**
  - Llista exhaustiva d'enllaços sobre Docker
- comp.photo777.org/wp-content/uploads/2014/09/docker-ecosystem-8.6.1.pdf
  - Pòster sobre l'ecosistema Docker
- www.mindmeister.com/389671722/docker-ecosystem
  - Mapa conceptual sobre Docker/Open Contenidor
- flux7.com/blogs/docker/ (blog sobre Docker)
- 12factor.net/ (metodologia de disseny d'aplicacions "*The twelve-factor app*")
- **kubernetes.io**

9.2 Bibliografia

[1] A. Carzaniga, A. Fuggetta, R. S. Hall, Sr. Heimbigner, A. van der Hoek and A. L. Wolf, "A Characterization Framework for Software Deployment Technologies," Technical Report ADA452086, Defense Technical Information Center , Dept. of Defense, USA, 1998.

[2] Microsoft Developer Network, "Windows Installer Guide," Microsoft Corp., [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa372845%28v=vs.85%29.aspx. [Accessed 9 March 2016].

[3] Microsoft Developer Network, "Windows Installer Reference," Microsoft Corp., [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa372860%28v=vs.85%29.aspx. [Accessed 9 March 2016].

[4] FireGiant, "WiX Tutorial," Outercurve Foundation, [Online]. Available: https://www.firegiant.com/wix/tutorial/. [Accessed 9 March 2016].

[5] E. Foster-Johnson, "RPM Guide," 2005. [Online]. Available: http://rpm5.org/docs/rpm-guide.html. [Accessed 9 March 2016].

[6] "DNF, the next-generation replacement for Yum," Xarxa Hat, Inc., 2012. [Online]. Available: http://dnf.readthedocs.org/en/latest/. [Accessed 9 March 2016].

[7] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," Research report RC22456, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA, 2002.

[8] W. Li, "Evaluating the impacts of dynamic reconfiguration on the QoS of running systems," *Journal of Systems and Software,* vol. 84, pàg. 2123-2138, 2011.

[9] S. Dustdar, I. Guo, B. Satzger and H. L. Truong, "Principles of Elastic Processes," *IEEE Internet Computing,* vol. 15, no. 5, pàg. 66-71, 2011.

[10] R. A. Gingell, M. Llig, X. T. Dang and M. S. Weeks, "Shared Libraries in SunOS," in *USENIX Association Conference*, Atlanta, Georgia, USA, 1987.

[11] D. Nene, "A beginner's guide to Dependency Injection," THESERVERSIDE.COM, 1 July 2005. [Online]. Available: http://www.theserverside.com/news/1321158/a-beginners-guide-to-dependency-injection. [Accessed 10 March 2016].