Treball de Fi de Grau

Grau en Enginyeria Informàtica

Disseny i implementació d'un sistema DevOps (CI / CD) per a projectes de codi obert multillenguatge

Autor: Sergi Martínez Rodríguez

Director: Rosa Maria Badia Sala

Co-Director: Jorge Ejarque Artigas

Convocatòria: Primavera 2020

Amb la col·laboració de Barcelona Supercomputing Center (BSC)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona

FIB

Índex

Resum

Avui en dia, amb la gran competència que hi ha d'empreses de software, és necessari poder garantir al client que el nostre producte funciona correctament. Per fer-ho, és important fer tants tests com sigui necessari per cobrir totes o quasi totes les línies del nostre codi font. Amb aquest projecte, s'ha integrat un sistema al software COMPSs que permet generar un informe sobre la cobertura de codi, amb la intenció que els desenvolupadors puguin saber quines línies del codi falten per cobrir.

Addicionalment, el projecte també ha consistit en integrar un sistema per desplegar tests de COMPSs automàticament a un Supercomputador. Abans d'aquest poojecte s'havien d'executar manualment un per un, però gràcies a aquest sistema ja es pot fer executant un sol script.

L'última part del projecte ha consistit en crear un sistema capaç d'autogenerar la documentació de desenvolupador de COMPSs automàticament, amb la intenció que aquesta estigui sempre actualitzada.

Agraïments

Vull agrair especialment al Jorge per haver estat pendent tot el semestre d'aquest projecte i no faltar mai a les reunions del dimarts, tot i els temps difícils que hem viscut aquests últims mesos.

Evidentment també agraeixo a la Rosa per haver confiat amb mi i donar-me l'oportunitat de realitzat aquest projecte. M'enduc un munt de coneixements que podré aplicar en el Màster que ara començaré i en el món laboral.

Finalment dono les gràcies a la meva mare per haver corregit aquest treball i per haver-me ajudat sempre amb els treballs de llengua de batxillerat, els quals d'una forma o una altra m'han portat fins aquí.

1. Introducció i contextualització

Quan es desenvolupa software es cometen errors. D'aquests errors alguns són pocs importants i poden causar pocs inconvenients als usuaris, però n'hi ha d'altres que poden ser perillosos i causar una pèrdua de diners i recursos importants. Es per això que és important provar tot el que desenvolupem, ja que el software sempre pot acabar fallant. Si assegurem la qualitat del producte, ens assegurarem la confiança dels clients i dels usuaris. Si per qualsevol cosa, el client no estigués satisfet amb la qualitat del producte marxaria sense dubtar-ho cap al software de la competència.

Context

Aquest és un Treball de Fi de Grau que pertany als estudis del Grau en Enginyeria Informàtica de la Facultat d'Informàtica de Barcelona. La especialitat d'aquest és l'Enginyeria del Software i és supervisat per el Barcelona Supercomputing Center (1). El software en què aquest projecte es basa és COMPSs (2). Desenvolupat per el BSC, COMPSs és un *framework* que facilita al usuari el desenvolupament d'aplicacions distribuïdes i que, a més a més, permet explotar el paral·lelisme de les aplicacions a executar. Està basat en el popular llenguatge de programació Java, però també disposa de *bindings* programats en Python i C/C++. En el cas del model Java no fa falta fer ús de cap *API*, ja que tot són crides a llibreries Java. En el cas de Python i C/C++, existeix una petita col·lecció de crides *API*.

Definició de termes

1.2.1. Cobertura de codi

La cobertura de codi (3) és una forma de mesurar quin percentatge del codi font d'un programa ha sigut provat. Podríem dir que es una forma de determinar la qualitat del codi que s'ha desenvolupat. Una cobertura de codi alta garanteix una experiència més positiva per l'usuari. A més a més, es pot fer servir com una tècnica d'optimització ja que permet localitzar codi mort que mai arriba a executar-se.

1.2.2. Deployment

El *deployment* (4) és el conjunt d'activitats normalment relacionades entre elles que fan el software accessible al usuari. Com que cada software és únic, el *deployment* serà definit i adaptat als requeriments d'aquest. Cada cop que s'actualitza el codi d'un software, aquest ha de ser sotmès al procés de *deploy* perquè pugui ser entregat als usuaris.

Problema a resoldre

COMPSs és un software que està preparat per treballar amb Supercomputadors, però no disposava de cap sistema per executar tests i recollir resultats automàticament. La primera part del projecte, doncs, va ser dissenyar i implementar un sistema que permetés a l'usuari desplegar tests cap al Supercomputador i recollir tots els resultats en un informe. Tot això havia de ser automàtic, doncs l'usuari només vol executar un script i rebre tots els resultats un cop les execucions dins del Supercomputador han acabat.

Un altre dels problemes que presentava COMPSs és que no es sabia a ciència certa quin era la cobertura del codi total. Hi havia tests fets, però no hi havia cap eina ni sistema integrat que especifiqués quines línies del codi font estaven cobertes i quines no. COMPSs disposa de codi escrit en Python i Java, així que en aquest projecte va caldre fer un estudi de totes les possibilitats que es tenien a l'abast i escollir aquelles que s'adequaven més als requisits d'aquest projecte.

Finalment, l'últim problema que es vol solucionar està relacionat amb la documentació. Quan s'actualitzava el *framework*, l'única forma d'actualitzat la documentació era fent-ho de forma manual. Per tant, l´últim objectiu d'aquest projecte va ser trobar alguna manera per fer que la documentació s'actualitzés automàticament quan hi hagués actualitzacions en el codi.

Actors implicats

La realització d'aquest projecte va implicar una sèrie d'actors que han estat directament implicats:

1.4.1. Barcelona Supercomputing Center

El projecte està basat en un software desenvolupat pel BSC, així que ells s'han vist directament beneficiats un cop el projecte es va desenvolupat amb èxit.

1.4.2. Desenvolupadors de COMPSs

Aquest projecte ha tret feina als desenvolupadors, doncs, tota la feina relacionada amb *code coverage,* documentació i integració de tests al *deploy* ja està feta.

1.4.3. Usuaris de COMPSs

Un dels objectius d'aquest projecte ha sigut augmentar la garantia del bon funcionament del software, per tant, els usuaris que facin servir COMPSs en un futur estaran directament implicats.

2. Justificació

Provar el software que desenvolupem i que anem a entregar als usuaris és necessari (5). En algunes ocasions aquesta etapa del desenvolupament es passa per alt, causant grans problemes al producte i a l'empresa. Si detectem els problemes de codi en les primeres etapes del desenvolupament ens estalviarem molta feina i malts de caps, ja que serà molt més fàcil arreglar-ho. En canvi, si no ho detectem fins al final, haurem de gastar molts més recursos per poder-ho solucionar. COMPSs és un *framework* encara en desenvolupament i no es coneixia del tot quin percentatge del codi estava cobert amb tests. Un cop s'han sabut les línies del codi font que no estan cobertes, els desenvolupadors es poden dedicar a realitzar els tests per aquestes línies en concret amb l'objectiu d'arribar a la major cobertura de codi possible, garantint així un correcte funcionament pels usuaris.

COMPSs està pensat per treballar amb Supercomputadors, així que es necessari tenir tests que comprovin el funcionament en aquests. És per això que ha sigut necessari implementar un sistema per a que els desenvolupadors puguin testejar les funcionalitat en Supercomputadors abans d'entregar el software als usuaris.

Solucions existents

2.1.1. Executar els tests en el Supercomputador manualment

Anteriorment a la realització del projecte existia una solució per provar que el *framework* funcionava correctament en Supercomputadors abans de ser entregat, però no del tot eficient. Aquesta solució consistia simplement en moure tots els fitxers dels tests cap al Supercomputador, i executar-los un per un manualment. Aquesta és una feina monòtona que es podia automatitzar perfectament.

2.1.2. Actualitzar la documentació manualment

Evidentment era una possibilitat, però també és una tasca monòtona i es podia automatitzar perfectament.

2.1.3. Llibreries de code coverage

Existeixen nombroses llibreries de *code coverage* tant per Python com per Java, per tant, s'ha hagut d'estudiar quins són els punts forts de cada una per decidir quines són les que més s'adapten als requisits d'aquest projecte.

Addicionalment existeixen nombrosos *frameworks* que fan el *code coverage* automàticament a partir de tests d'unitat i integritat, com ara *Spring, Django* o *Ruby on Rails*. Malauradament, el desplegament de COMPSs és propi i consta de diversos llenguatges, així que no han sigut d'utilitat.

3. Abast del projecte

És necessari definir quins han sigut els objectius d'aquest projecte per tenir clar quins són els punts que s'han encapsulat. A continuació definirem els objectius, subobjectius i altres requeriments no funcionals.

Objectius

Com s'ha explicat anteriorment, el projecte consta de tres objectius principals: integrar un sistema per desplegar i executar tests automàticament en un Supercomputador, descobrir quin percentatge del codi està cobert i, finalment, integrar un sistema que actualitzi la documentació automàticament cada cop que fem un canvi al software i l'entreguem al usuari.

3.1.1. Subobjectius

- Ser transparent als usuaris, doncs aquests no han de conèixer l'existència d'aquests tests.
- Aconseguir una forma de mesurar la cobertura de codi en aquest *framework*.
- Fer que els resultats del codi cobert apareguin en un informe.
- Fer que els resultats de les execucions en el Supercomputador apareguin en un informe.

Obstacles i riscs

A continuació es citen la sèrie de riscos i obstacles que s'han tingut en consideració:

3.2.1. Temps limitat

La durada del projectes ha sigut limitada i, per tant, s'ha agut de planificar correctament per poder-lo realitzar amb èxit i en el període establert.

3.2.2. Framework propi

COMPSs és un *framework* propi i per aquesta raó no ens hem pogut basar en altres projectes realitzats per altres desenvolupadors.

3.2.3. Desconeixement

Molts dels softwares i llibreries que ens han pogut ser d'utilitat per aquest projecte no les coneixíem en profunditat. Per tant, ha calgut dedicar el seu corresponent temps a aprendre com funcionen.

Metodologia i rigor

És necessari definir la metodologia que s'ha anat seguint durant la realització d'aquest projecte. Una de les metodologies que s'han fet servir més durant les diferents assignatures de la carrera ha sigut la metodologia *Agile* (6). Aquesta metodologia es basa en el desenvolupament iteratiu i incremental, on els requisits i les solucions evolucionen juntament amb el projecte. La metodologia consisteix en dividir la feina en iteracions, on cadascuna requereix planificació, anàlisis del requisit, disseny i codificació. Finalment, aquesta es revisada per el supervisor. D'aquesta manera, es pot adaptar el projecte en el cas que aparegués algun canvi necessari inesperat. La major part de la feina s'ha fet des de casa, i s'han aprofitat els dies establerts de reunió (un cop per setmana) per revisar tota la feina que s'ha realitzat i per resoldre dubtes que han aparegut.

Eines de seguiment

A continuació, es definiran el conjunt d'eines i infraestructures que s'han fet servir durant el desenvolupament del projecte.

3.4.1. Gitlab

Gitlab és l'eina de control de versions que s'ha utilitzat. Actualment, cada cop que algun desenvolupament vol fer un canvi en el software l'ha d'efectuar sobre una branca. Quan la branca es fusiona amb el tronc, s'invoca una *pipeline* que s'encarregarà de realitzar el *deploy*.

3.4.2. Correu electrònic

Ha sigut el mitjà de comunicació principal entre l'alumne i el supervisor. S'ha utilitzat per establir dies de reunió i resoldre dubtes esporàdics.

3.4.3. Skype

Skype ens ha servit per realitzar les reunions, té una eina per compartir la pantalla, així que no hi ha hagut necessitat de realitzar reunions presencials.

4. Planificació temporal

Duració del projecte

El projecte ha tingut una durada aproximada de 140 dies. Va començar el dia 10 de febrer i ha acabat el 30 de juny. S'han dedicat un total aproximat de 470 hores amb una mitjana de dos hores treballades per dia. La defensa d'aquest projecte serà el dia 30 de juliol.

Tasques del projecte

A continuació estan definides totes les tasques que s'han hagut de realitzar per desenvolupar el projecte correctament. Aquestes estan separades en grups tal com es pot veure a continuació. Al final de l'apartat s'inclou una taula amb el resum d'aquest apartat juntament amb les hores estimades per cada una de les tasques.

4.2.1. Gestió del projecte

Això és el conjunt de tasques que formen l'assignatura de GEP. L'objectiu d'aquesta assignatura és aprendre a realitzar una planificació d'un projecte de mitjana envergadura. Les tasques que formen GEP són:

4.2.2. Abast del projecte

Una part molt important de qualsevol projecte és definir quin és l'abast. Cal definir què formarà el projecte i què no.

4.2.3. Planificació temporal

Un cop definit l'abast, és necessari fer la planificació temporal del projecte. Per fer això, cal definir quines són les diferents fases del projecte, juntament amb els seus recursos i requeriments.

4.2.4. Pressupost

Cal fer un petit estudi per veure quina serà l'estimació de costos que portarà la realització d'aquest projecte.

4.2.5. Informe de sostenibilitat

És el document on es fa l'estudi sobre la sostenibilitat i l'impacte ambiental d'aquest si és que en té algun.

4.2.6. Realització de la memòria i presentació

Com a tot projecte, s'ha hagut d'elaborar un document on hi ha explicats tots els estudis realitzats, les conclusions i els sistemes desenvolupats juntament amb la seva documentació tècnica. Finalment, es farà una defensa del projecte en una presentació oral.

Implementació i disseny

Com s'ha dit al principi del document, aquest projecte té com objectiu la implementació de tres sistemes: una bateria de tests automàtics en el procediment de *deploy*, un sistema per poder calcular el

code coverage dels tests de COMPSs i, finalment, un sistema que actualitzi la documentació un cop s'hagi fet qualsevol canvi al codi font del software.

4.3.1. Implementació del code coverage

4.3.1.1. *Disseny*

Per a fer el sistema *code coverage* primer s'ha hagut d'estudiar de quines eren les opcions que disposàvem. Al mercat existeixen nombroses possibilitats, així que ha calgut fer un estudi d'aquestes per veure quina s'adapta millor a les particularitats de COMPSs.

4.3.1.2. *Implementació*

Un cop s'ha triat la solució o solucions que es volen fer servir, ja es pot començar amb la implementació. La idea és aplicar un *code coverage* tant al codi escrit en Java com al del Python, així que s'ha hagut de decidir per quin començar. Les eines en Java solen estar millor documentades així, per aquesta raó s'ha començat per aquí. Un cop es va implementat el *code coverage* per Java, es va poder procedir a implementar per Python.

4.3.2. Implementació del sistema per desplegar tests al Supercomputador

Aquesta és la part on es dissenya i s'implementa el sistema que s'encarrega de desplegar i executar tests automàticament en un Supercomputador.

4.3.2.1. *Disseny*

La primera fase d'aquest apartat consisteix en fer el disseny del sistema. Per aconseguir-ho, va caldre fer un estudi de la infraestructura de la qual disposàvem (MareNostrum 4) juntament amb les solucions informàtiques compatibles. Com ja s'ha dit, el MareNostrum 4 disposa de bastantes restriccions i, per tant, no va ser una feina trivial.

4.3.2.2. *Implementació*

Un cop s'ha tingut fet el disseny, s'ha procedit a fer la implementació.

4.3.3. Implementació del sistema automatitzat de documentació

4.3.3.1. *Disseny*

Per al disseny d'aquest últim sistema a implementar s'ha estudiat quins softwares a l'actualitat compten amb un sistema que permeti generar documentació automàticament.

4.3.3.2. *Implementació*

Per la implementació va caldre mirar, dels sistemes ja implementats que existien, com funcionaven per poder-ho recrear. Al haver-hi bastants sistemes amb aquesta funcionalitat, va ser una tasca bastant senzilla.

Gestió del projecte: 100h

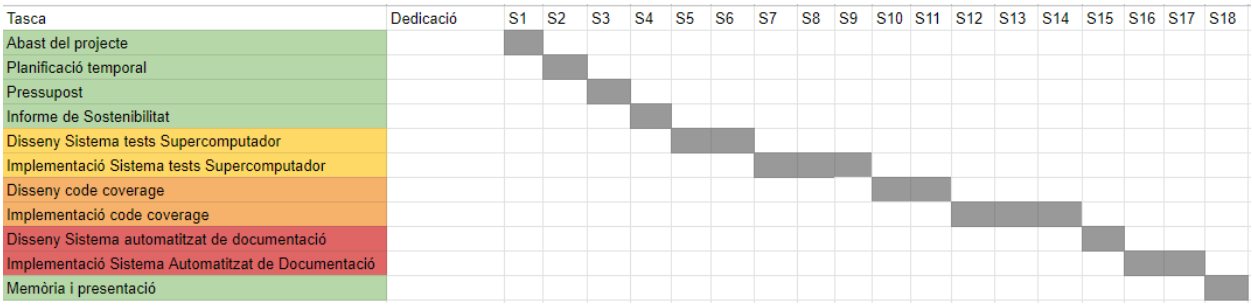
Codi	**Tasca**	**Temps**	**Dependències**
T1.1	Abast del projecte	25h	
T1.2	Planificació temporal	10h	T1.1
T1.3	Pressupost	10h	T1.1,T1.2
T1.4	Informe de sostenibilitat	10h	T1.1
T1.5	Realització de la memòria i presentació	35h	T1.1, T1.2, T1.3, T1.4

Disseny i implementació: 370h

Codi	**Tasca**	**Temps**	**Dependències**
T2.1	Disseny *code coverage*	50	
T2.2	Implementació *code coverage*	100	T2.1
T2.3	Disseny sistema tests Supercomputador	50	
T2.4	Implementació sistema tests Supercomputador	100	T2.3
T2.5	Disseny sistema automatitzat de documentació	25	
T2.6	Implementació sistema automatitzat de documentació	55	T2.5

Representació gràfica de la planificació

A continuació adjunto el diagrama de Gantt on es veuen representades les tasques citades anteriorment:



Gestió del risc

Quan es documenta un projecte, sobretot relacionat amb la informàtica, s'han de considerar tots els problemes i imprevistos que es poden manifestar. Aquests són els que s'han tingut en compte:

4.5.1. Problemes amb MareNostrum 4

El desenvolupament d'aquest projecte va involucrar l'ús del Supercomputador MareNostrum 4 (7). Aquest va ser pràcticament sempre accessible, però es va haver de tenir en compte que sempre podia haver alguna caiguda. A més a més és possible que es fes alguna actualització que podia afectar el projecte. Això no havia de passar, doncs no hi havia actualització programada per al temps que durava aquest projecte. En el cas que s'haguessin produït alguna d'aquestes circumstàncies, el temps aproximat que es perdria no havia de ser superior les 10 hores.

4.5.2. Implementació dels sistemes

Durant la implementació dels tres sistemes es possible que hi hagi problemes. Degut a que COMPSs és un *framework* propi, hi haurà moltes coses que s'hauran de fer de nou, per tant s'haurà de dedicar un temps a entendre el funcionament de les noves eines. En les assignatures de la carrera no s'ha fet res semblant, així que això és un risc que s'haurà de considerar. Les tasques, però, no s'haurien de complicar tampoc massa, doncs per això hi havia un tutor que em podia donar un cop de ma si s'arribava a aquesta situació. L'endarreriment total com a molt havia de ser de 20 hores.

4.5.3. Rendiment

Tot i tenir a l'abast el Supercomputador, hi havia vegades que s'havia de fer proves en local. Els tests de COMPSs requereixen bastants recursos, així que, si s'efectuen amb una màquina estàndard, l'execució d'aquests tests pot trigar bastant de temps. Tenint això en compte, vam estimar que el temps invertit en total en fer execucions amb una màquina local seria d'unes 20 hores. S'ha de tenir en compte, però, que la majoria d'execucions es van fer amb MareNostrum 4.

5. Pressupost

Al següent apartat està tot l'estudi previ que s'ha fet referent al pressupost, imprevists i control de gestió.

Costos

A continuació, es mostren diverses taules especificant els costos de cada un dels apartats que s'han considerat:

5.1.1. Costos de personal per activitat

Perfil	**Retribució**
Programador	15€/h
Cap de projecte	25€/h

5.1.2. Costos humans

Un cop tenim clar quins són els salaris de cada un dels perfils, es pot calcular quin és el preu total que ha costat cada tasca. Recordem que les tasques que mostren a continuació són les que s'han definit en l'apartat anterior.

Tasca	**Temps cap projecte**	**Temps programador**	**Cost total**
[T1.1] Abast del projecte	25h	0h	625€
[T1.2] Planificació temporal	10h	0h	250€
[T1.3] Pressupost	10h	0h	250€

[T1.4] Informe de sostenibilitat	10h	0h	250€
[T1.5] Realització de la memòria i presentació	35h	0h	875€
[T2.1] Disseny bateria de tests	40h	10h	1150€
[T2.2] Implementació bateria de tests	20h	80h	1700€
[T2.3] Disseny *code coverage*	40h	10h	1150€
[T2.4] Implementació *code coverage*	20h	80h	1700€
[T2.5] Disseny sistema automatitzat de documentació	20h	5h	575€
[T2.6] Implementació sistema automatitzat de documentació	10h	45h	925€
Total	**240h**	**230h**	**9450€**

5.1.3. Costos de hardware

Aquests són els costos que s'han tingut pel que fa al hardware.

Ordenador portàtil personal	720€
Ordenador d'escriptori personal	1000€
Total	**1720€**

5.1.4. Costos de software

Respecte el software, s'han considerat aquests:

Google Drive	0€
LlibreOffice	0€
Mozilla Thunderbird	0€
Ubuntu 18.04	0€
Pycharm	79.6€
Total	**79.6€**

5.1.5. Costos totals

Costos humans	9450€
Costos de hardware	1720€
Costos de software	79.6€
Total	**11249.6€**

5.1.6. Contingències i imprevists

Com a qualsevol projecte de software, s'ha de tenir en compte una sèrie d'imprevists amb el seu corresponent cost. Aquests són els imprevists que s'han considerat:

Imprevist	**Preu**	**Risc**	**Cost**
Nou PC portàtil	720€	10%	42€
Nou PC escriptori	1000€	5%	50€
Increment temps disseny (10h)	230€	15%	34.5€
Increment temps implementació (20h)	320€	15%	339.22€
Total			**465.72**

Respecte les contingències, s'ha considerat un percentatge del 5%.

Contingències = Total*0.05 = 572.46€.

Control de gestió de riscs

És possible que durant el transcurs del projecte hi hagi hagut desviacions en la planificació que puguin alterar els pressuposts. Teòricament no havia de passar, però sempre s'ha de tenir en compte aquesta possibilitat. Els costos humans són els únics que es podien veure alterats. Com que la planificació de les tasques va ser aproximada, es possible que algunes s'allarguessin o s'acabessin abans.

Cal tenir en compte que la majoria de tasques compten amb dependències, així que un retard en alguna de les tasques podia causar algunes desviacions en el temps. Respecte els pressuposts destinats al hardware i software no havien de patir variacions. Tot el software utilitzat és gratuït, i era poc probable que el hardware utilitzat patís variacions en el preu.

6. Informe de sostenibilitat

Autoavaluació

Un cop realitzada l'enquesta de *EDINSOST* m'he adonat que tinc alguns coneixements decents sobre la sostenibilitat de les TIC, però encara em queda molt per aprendre. Sóc conscient que els enginyers informàtics tenim una responsabilitat molt gran, ja que els nostres projectes poden tenir un impacte fonamental en la societat a part de tenir una vida útil molt llarga.

És per això que hem de ser conscients de l'impacte mediambiental produït i els seus efectes en el món. Per tant, a vegades és molt millor prioritzar la sostenibilitat al estalvi econòmic. Encara no he tingut l'oportunitat de treballar en cap projecte de caràcter sostenible, però estic convençut que si alguna vegada ho faig m'adonaré de la importància de tenir la sostenibilitat en ment en aquest tipus de projectes.

Dimensió econòmica

El desenvolupament d'aquest projecte ha comportat uns pressuposts bastant similars als demés projectes d'una envergadura semblant, doncs la majoria dels recursos que s'utilitzen ja han sigut adquirits o tenen preus assequibles. S'ha de tenir en compte el preu de l'energia elèctrica utilitzat durant la realització del projecte. Aquest inclou l'ús del meu ordenador personal, més l'ús que he fet del Supercomputador MareNostrum 4. El meu projecte ha tingut com a meta automatitzar algunes etapes dins del procediment d'entrega de software als clients, per tant, tot aquest temps que l'usuari s'estalvia el podrà fer servir per fer alguna altra tasca que hagi de realitzar.

Dimensió ambiental

L'impacte ambiental que ha tingut aquest projecte durant el seu desenvolupament ha sigut el consum d'energia que ha produït. Per tant, s'ha tingut en consideració l'impacte ambiental que ha causat l'ús del meu ordenador personal més el del MareNostrum 4. Tret de la memòria final, tots els demés documents s'han guardat en format digital, per tant s'ha reduït el consum de paper. A més a més, un cop acabada la vida útil del projecte, hi haurà algunes parts que es poden reutilitzar per a futurs projectes similars, disminuint així l'impacte ambiental d'aquests.

També cal afegir que tot el hardware que s'ha fet servir durant aquest projecte es seguirà fent servir per a més tasques en un futur. Si per qualsevol cas el hardware utilitzat s'espatllés i s'hagués de llançar, es llançaria als punts de recollida de residus electrònics més propers.

Dimensió social

El primer impacte social que m'agradaria comentar és el personal. Amb aquest projecte es donen per finalitzats els meus estudis del Grau en Enginyeria Informàtica de la UPC. Això comporta una satisfacció personal degut a tots els esforços i temps que he dedicat en aquesta carrera en tots els anys que ha durat.

Aquest projecte també té un impacte social per als usuaris de COMPSs. Com que el codi font del software es podrà testejar abans de ser entregat, aquest s'entregarà amb menys errors als usuaris, els quals podran gaudir d'una qualitat superior. No només això, si no que els propis desenvolupadors d'aquest software podran emprar el temps que hagessin estat fent aquest projecte en altres tasques, o simplement en oci. És per això que és necessari realitzar aquest projecte, doncs els usuaris i els desenvolupadors es veuran directament beneficiats.

7. Descripció tècnica del projecte

En aquest apartat es parlarà més detalladament de les solucions que s'han desenvolupat per els tres problemes presentats anteriorment.

COMPSs

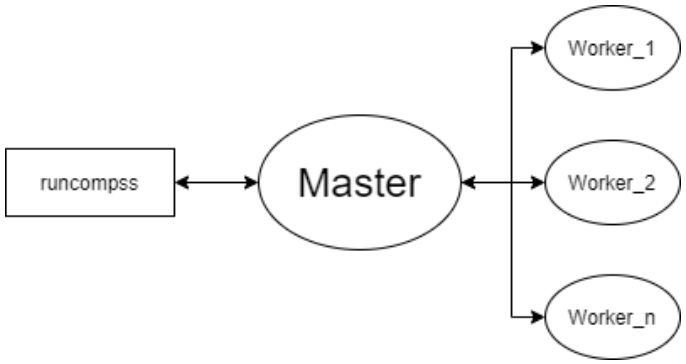
COMPSs és un software desenvolupat per Barcelona Supercomputing Center que pretén facilitar el desenvolupament d'aplicacions per a infraestructures distribuïdes. A més a més, també suporta un sistema d'execució per explotar el paral·lelisme de les aplicacions en temps d'execució. El model de COMPSs es basa en les següents característiques:

- **Programació seqüencial**: Els programadors no han de preocupar-se dels maldecaps que comporta la programació paral·lela, com ara la creació i sincronització de *threads*, distribució de dades, missatgeria o tolerància de falles. En canvi, el model es basa en una programació

seqüencial, cosa que facilita el treball dels programadors sense experiència en programació paral·lela.

- **Llenguatges de programació estàndards:** COMPSs està basat en Java, però també ofereix *bindings* per aplicacions de Python i C/C++. Això facilita l'aprenentatge del model, doncs els programadors poden aprofitar els seus coneixements adquirits prèviament.
- **Sense *APIs*:** En les aplicacions de COMPSs escrites en Java, el model no requereix l'ús de cap crida *API*, ja que tot està integrat en llibreries de Java. Respecte els *bindings* de Python i C/C++, COMPSs disposa d'un petit conjunt de crides *API*.
- **Transparència en la infraestructura:** COMPSs extreu l'aplicació de la infraestructura distribuïda subjacent. Per tant, els programes COMPSs no inclouen cap dada que els pugui relacionar amb alguna plataforma determinada.

COMPSs es desplega seguint un model *master-worker.* Un node *master* s'encarregarà d'invocar tants nodes *worker*s com l'aplicació necessiti i mantindrà comunicació amb ells, tal com es mostra a la figura:



El software funciona juntament amb un *runtime.* Aquest rep les tasques que envia el programa principal, i genera un graf de dependències amb la intenció que s'executin amb l'ordre correcte. L'usuari tant sols s'ha d'encarregar d'engegar el *runtime* dins el codi de l'aplicació a executar, d'especificar les parts del codi que vol executar com a tasques i d'apagar el *runtime.*

Un cop encesa l'aplicació, s'aixecarà el *master* amb un conjunt de *workers,* i a mesura que es vagin generant noves tasques, aquestes seran adjudicades pels diferents *workers* disponibles.

Quan l'usuari vulgui executar una aplicació amb COMPSs, ho haurà de fer mitjançant *runcompss,* quan estigui treballant en un entorn interactiu, i *enqueue_compss* quan estigui en un entorn amb cues. Tant *runcompss* com *enqueue_compss* es poden executar passant-hi diversos paràmetres de configuració,

com ara les opcions *Java Virtual Machine*, un *interpreter* de Python que l'usuari vulgui fer servir, o la carpeta on es vulgui guardar els resultats de l'execució. Un cop acabada l'execució, l'usuari pot comprovar tots els fitxers de sortida generats i els fitxers d'error en cas que l'execució hagi fallat. A més a més, també es possible recollir un graf de dependències per veure l'ordre d'execució de les tasques i les seves dependències.

Existeixen nombrosos tests que proven diferents aspectes del software organitzats en famílies: *performance, agents, autoparallel...* i aquests es poden executar i comprovar els seus resultats amb un script de Python. L'usuari pot executar diversos tests alhora individuals o per famílies, i podrà recollir els resultats de totes les execucions a la carpeta per defecte. Aquests tests avaluen els fitxers de sortida de cada execució per comprovar si aquesta ha sigut un èxit o no.

El *framework* consta de documentació tant d'usuari com de desenvolupador, tot i que aquesta última és molt susceptible a quedar-se desactualitzada. La documentació d'usuari és un document redactat a mà sobre com fer servir el *framework*, mentre que la documentació de desenvolupador consisteix en un document de text compartit amb tots els desenvolupadors, on s'ha d'anar actualitzant cada cop que es fa un canvi en el codi font.

Sistema per mesurar la cobertura de codi de COMPSs

En aquest capítol es parlarà sobre el sistema que s'ha dissenyat i implementat per mesurar la cobertura de codi del *framework* COMPSs.

7.2.1. Problemàtica

Actualment, COMPSs no disposa de cap sistema per poder mesurar la cobertura de codi dels seus tests. És a dir, no hi ha cap manera de saber quin percentatge de línies del codi cobreixen els tests que hi ha actualment fets. Això es un problema, doncs una de les claus per garantir una alta qualitat del software és tenir una bateria de tests amb una cobertura de codi lo més proper al 100% possible. Per tant, el primer objectiu d'aquest projecte és justament implementar un sistema que permeti al usuari saber quina és la cobertura de codi total que cobreix els tests que vol executar.

7.2.2. Eines disponibles

A continuació es citen les eines disponibles al mercat que s'han considerat en la realització d'aquest sistema:

7.2.2.1. *Jacoco*

Jacoco (8) és una eina de codi obert que, entre altres coses, ofereix un informe sobre la cobertura d'instruccions, línies i branques. Aquesta eina està integrada en nombrosos entorns integrats de desenvolupament, i també està ben documentada.

Jacoco funciona amb agent que, quan s'inclou dins les opcions JVM, genera un informe de cobertura de codi. Aquest suporta nombroses opcions, tot i que moltes no són del nostre interès. Al manual de *Jacoco* s'explica quina és la comanda per fer-ho funcionar:

```
-javaagent:[yourpath/]jacocoagent.jar=[option1]=[value1],[option2]=[value2]
```

7.2.2.2. *Coverage.py*

Coverage.py (9) és una llibreria bastant popular que també permet rebre un informe sobre les línies que ha executat un programa. Monitoritza el programa, apunta quines parts del codi s'han executat i analitza el codi font per veure quines línies es podrien haver executat, però no ho han fet.

Per executar una aplicació amb *coverage* és tant senzill com executar la següent comanda:

```
coverage run aplicacio.py
```

Això genera un fitxer *.coverage*, el qual pot ser visualitzat amb la comanda:

```
coverage report
```

Amb aquesta comanda apareix un informe de cobertura de les línies de Python executades. Si no especifiquem opcions addicionals, l'informe tindrà en compte l'aplicació executada juntament amb tots els mòduls de Python executats. Entre les nombroses opcions que Coverage.py suporta hi ha la possibilitat de generar un *.html* amb l'informe.

7.2.2.3. *Moduls de Python*

Per la realització d'aquest sistema també s'han fet servir diverses extensions de Python:

7.2.2.3.1. Subprocess

Subprocess (10) permet generar nous processos, connectar-se als seus canals de entrada/sortida/error i obtenir els seus codis de retorn. Entre altres coses, permet a un script executar comandes com en el *shell.*

7.2.2.3.2. Argparse

Argparse (11) és un mòdul que fa bastant fàcil la implementació de *flags* en els nostres programes. A més a més, genera automàticament missatges d'error i d'ajuda quan l'usuari introdueix arguments invàlids.

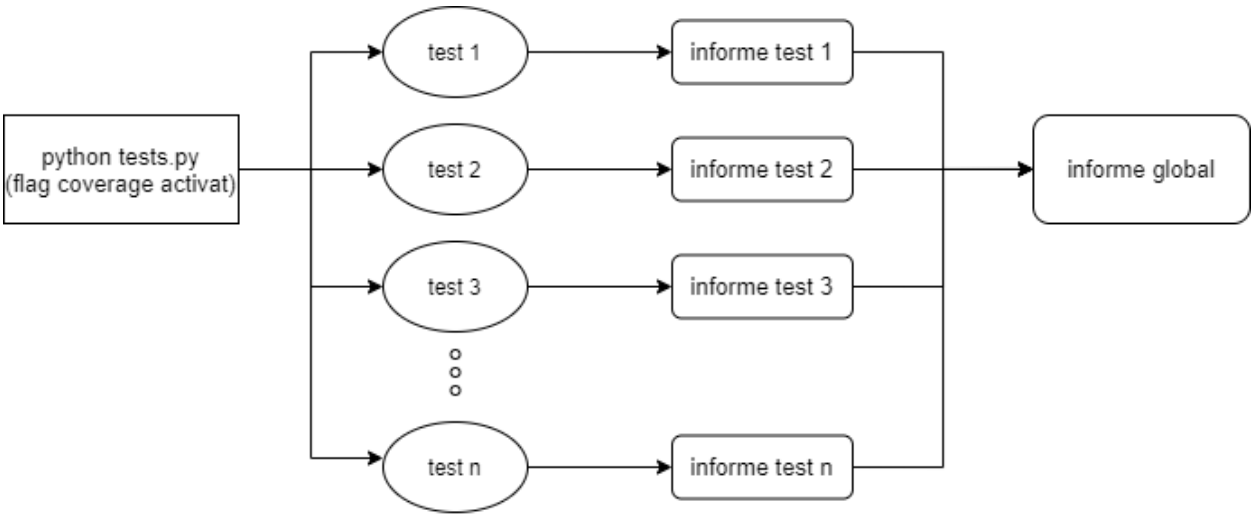
7.2.3. Solució

En aquesta secció es donarà una visió general de la solució proporcionada a aquest problema, juntament amb una explicació més detallada i tècnica del que s'ha fet.

7.2.3.1. *Visió general*

COMPSs disposa de nombrosos tests que avaluen el funcionament de diversos aspectes del *framework*. A més a més, l'usuari pot llençar els tests que desitgi mitjançant un script. Per tant, l'objectiu principal d'aquesta part del projecte ha sigut afegir una funcionalitat dins aquest script que, mitjançant un *flag,* permeti a l'usuari obtenir un informe sobre la cobertura del codi per on han passat els tests que ha executat. Aquest informe inclou informació rellevant per al desenvolupador, com ara el percentatge de línies totals per on han passat els tests o la llista de línies per on no han passat.

Aquest esquema mostra la visió general d'aquesta part del projecte:



7.2.3.2. *Implementació*

COMPSs disposa d'aquest script mencionat anteriorment que permet a l'usuari llançar diferents tests en local i rebre un informe dels resultats. Per tant, el que s'ha fet en aquesta primera part del projecte ha sigut implementar un *flag* que, quan s'activa, genera un informe sobre la cobertura de codi de tots els tests executats. Per poder implementar aquest *flag* ha sigut necessari fer alguns canvis no només al script, sinó també a l'executable *runcompss* de *COMPSs.*

7.2.3.2.1. Integració de la cobertura de codi en l'execució de COMPSs

Com s'ha mencionat en el capítol de COMPSs, per executar aplicacions de COMPSs és necessari fer-ho juntament amb *runcompss* (*enqueue_compss* si estiguéssim treballant en un entorn amb cues). Per tant, és un bon punt de partida afegir una funcionalitat a *runcompss* que generi un informe de cobertura de codi de l'aplicació que executa.

A continuació, s'explicarà quins són els canvis que s'han fet a *runcompss* per poder generar aquest informe de cobertura de codi tant per línies de Java com de Python.

7.2.3.2.1.1. Extracció Cobertura en Java

El *framework* està escrit en Java, així que té sentit que el primer objectiu d'aquest sistema sigui poder obtenir la cobertura del codi escrit en Java.

Entre les nombroses opcions que suporta *runcompss*, hi ha una que permet especificar les opcions *Java Virtual Machine* que vulguem. Aquest és el nostre punt d'entrada per poder generar un informe de cobertura de codi de les línies de Java executades.

Com es pot observar, la comanda requereix la ubicació del fitxer *.jar*, així que s'haurà de descarregar primer de la pàgina web.

Si volguéssim fer una execució d'una aplicació COMPSs i, addicionalment, generar un informe de cobertura hauríem d'introduir la següent comanda:

```
runcompss --jvm_master_opts="-javaagent:/home/usuari/jacocoagent.jar=destfile=/home/usuari/master.exec" --jvm_workers_opts="-javaagent:/home/usuari/jacocoagent.jar=destfile=/home/usuari/worker.exec" hello.Hello
```

L'anterior comanda generarà un fitxer *.exec* que, juntament amb l'eina *Jacococli.java*, ens permetrà ajuntar tots els fitxers *.exec* i generar un sol *.html* amb tot l'informe. La comanda per generar un *.html* a partir d'un *.exec* és:

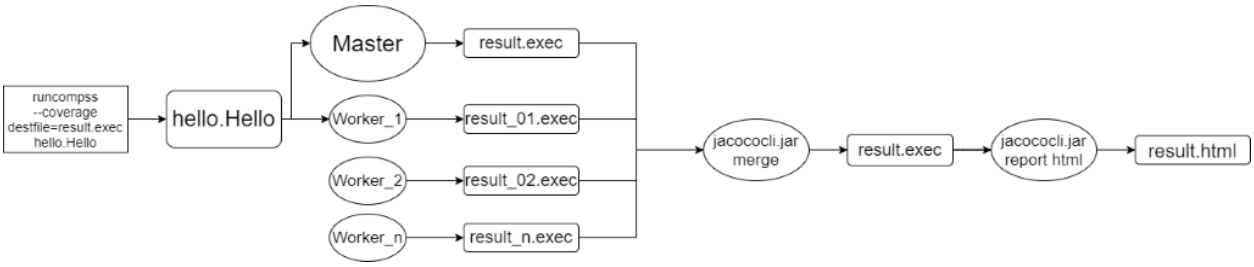
```
java -jar jacococli.jar report [<execfiles> ...] --classfiles <path> [--html <dir>]
```

Cal tenir en compte que això no soluciona del tot el nostre problema, doncs els fitxers generats pels *workers* s'aniran reemplaçant. Això és perquè la comanda només ens deixa especificar un sol nom per als fitxers *.exec,* així que tots els informes dels *workers* tindran el mateix nom i el sistema operatiu per defecte eliminarà l'antic per emmagatzemar el nou.

Es per això que ha calgut afegir aquest *flag* de mode *coverage.* El *flag* en qüestió s'activarà de la següent forma:

`runcompss –coverage=`**`/home/usuari/jacocoagent.jar`**`=destfile=`**`/home_usuari/resultat.exec`**, on el primer paràmetre en negreta és la ubicació de jacocoagent.jar, i el segon és el nom que tindrà el fitxer de sortida. En el cas que volguéssim afegir opcions de *Jacoco*, ho podríem posar separats per *#.*

A continuació es mostra un esquema que il·lustra el comportament d'aquest *flag:*



Aquest *flag –coverage* activa un condicional que s'encarrega de posar totes les opcions de COMPSs necessàries per poder executar l'aplicació juntament amb *Jacoco*. Per començar, afegeix l'agent de *Jacoco* dins les opcions de *Java Virtual Machine*, substituint els # per comes. També s'encarrega de cridar els script que generen els *workers* amb un nom diferent cada vegada, amb la intenció que els informes dels *workers* no es vagin substituint entre ells. Aquest nom serà de l'estil de *nom_uid.exec,* on *nom* és el nom del fitxer de sortida que s'ha especificat a la comanda, i *uid* és el *UID* del procés corresponent.

Un cop ha acabat l'execució, obtindrem un informe del *master*, juntament amb un informe per cada un dels *workers* que s'han executat. Aquests informes vindran en format *.exec*, així que s'haurà de fer servir l'eina *Jacococli.jar* per generar un *.html.*

7.2.3.2.1.2. Extracció de cobertura en Python

El codi base de COMPSs està escrit en Java, però una gran part dels *bindings* està escrit en Python. Per aquesta raó es convenient trobar una manera de generar un informe de cobertura de les línies de Python.

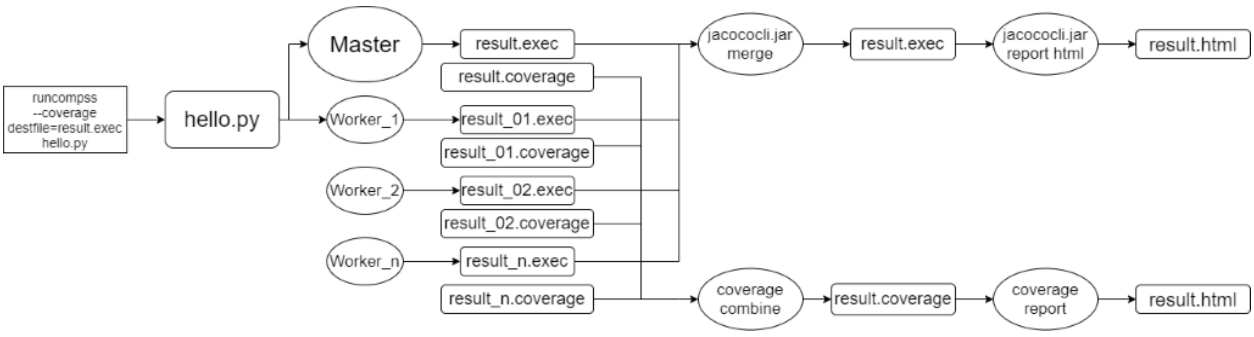
A l'igual que *runcompss* permet triar les opcions *Java Virtual Machine*, també permet triar el *interpreter* de Python que vulguem. Per tant, hi ha la possibilitat d'executar *runcompss* juntament amb *Coverage.py* La forma d'executar *runcompss* junt amb *Coverage.py* seria:

```
runcompss –python_interpreter=coverage run hello.py
```

El fitxer de sortida és d'extensió *.coverage* i es pot convertir en un informe llegible pels humans amb la comanda `coverage report.` Aquest espai que s'ha de posar a la comanda genera un conflicte amb l'execució , així que s'ha hagut d'arreglar també utilitzant el nou *flag* de *–coverage .*

Amb la comanda `coverage html` s'obté l'informe en format *.html.*

Amb aquest esquema es mostra com el sistema genera els informes de les línies de Python:



Com es pot observar l'execució generarà dos informes, un per línies de Python i un altre per línies de Java. Això es degut a que els tests de COMPSs sempre executaran línies de Java. El condicional que activa el *flag* s'encarrega d'activar l'ús de *coverage.py* en l'*interpreter* de Python. L'execució, doncs, hauria de generar dos informes per el *master,* i dos informes per cada un dels *workers* executats.

7.2.3.2.2. Llançament de tests automàticament

Tests.py és el script mencionat anteriorment que permet a l'usuari llençar diversos tests a la vegada. Degut a que utilitza *runcompss* per executar aquests tests, es pot aprofitar el mode de cobertura de codi per mesurar la cobertura dels tests que s'executaran. Per fer-ho, s'ha afegit el següent *flag*:

```
1.  parser.add_argument("-c", "--coverage",
2.                          action="store_true",
3.                          dest="coverage",
4.                          default=False,
5.                          help="Executes in Coverage mode")
```

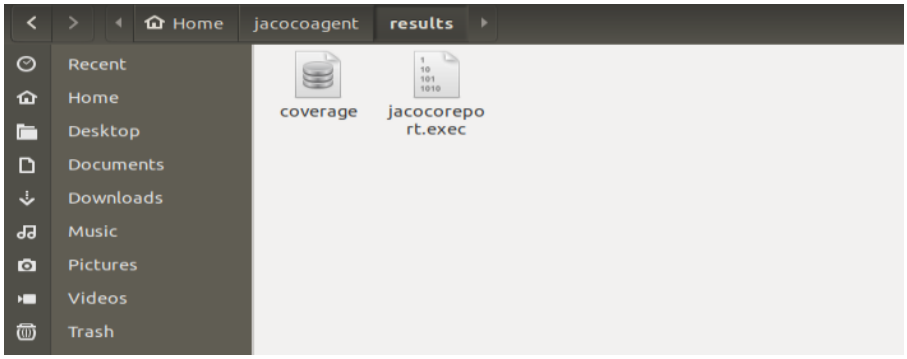
Amb aquest *flag*, l'usuari només s'ha de preocupar d'afegir les opcions que ell desitja al fitxer de configuració de *runcompss.* Aquestes opcions són la ubicació de *Jacocoagent.jar*, la carpeta on guardar els resultats, i opcions addicionals de *Jacoco.* Un cop tots els informes s'han generat, es fa un *merge* deixant només un informe de línies de Java, i un de Python.

Gràcies a aquesta nova funcionalitat, l'usuari pot executar una sèrie de tests i comprovar quin és el percentatge de codi total que cobreixen aquest conjunt de tests.

7.2.4. Resultats

En aquesta secció es mostraran els resultats aconseguits. La comanda `python tests.py -t 101 -t 102 -c` executarà els tests num. 101 i 102 en mode cobertura de codi. En la carpeta que hàgim

especificat per guardar els informes dins el fitxer de configuració de *runcompss* hauria d'aparèixer el següent:



Com s'ha comentat abans, aquests fitxers correspondran als informes de cobertura de codi de les línies de Java i Python. Si generem un .html de l'informe de Java *jacocoreport.exec* amb la comanda: `java -jar /home/usuari/jacococli.jar report jacocoreport.exec --classfiles /opt/COMPSs/Runtime/adaptors/nio/master/compss-adaptors-nio-master.jar --html html` ens genera el següent *.html*:

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
es.bsc.compss.nio.master		53%		38%	228	326	514	1,095	69	145	1	8
es.bsc.compss.nio.master.utils		42%		25%	21	24	36	65	3	6	0	1
es.bsc.compss.nio.master.configuration		47%		33%	8	16	25	47	2	10	0	1
es.bsc.compss.nio.master.handlers		82%		n/a	2	9	3	21	2	9	0	2
es.bsc.compss.nio.requests		100%		n/a	0	2	0	4	0	2	0	1
Total	2,609 of 5,534	52%	249 of 394	36%	259	377	578	1,232	76	172	1	13

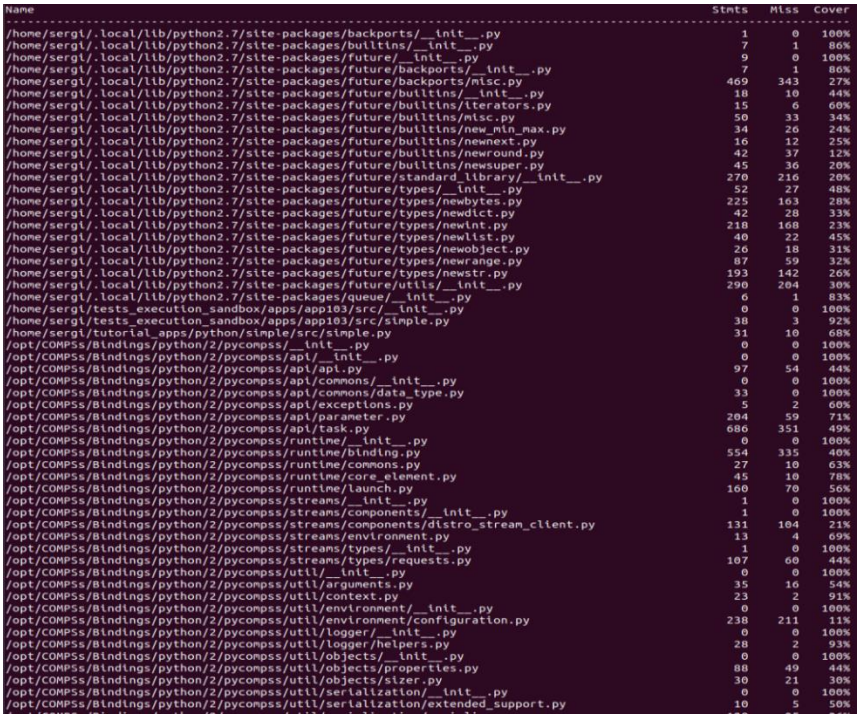
Aquest *.html* ens permet navegar per les diferents classes i mètodes del *.jar* que hem especificat:

WorkerStarter

Source file "es/bsc/compss/nio/master/WorkerStarter.java" was not found during generation of report.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
startWorker()		53%		30%	5	6	17	40	0	1
startWorker(String, String, int, String)		43%		40%	5	6	11	30	0	1
ender(NIOWorkerNode, int)		35%		40%	5	6	7	15	0	1
killPreviousWorker(String, String, int)		8%		16%	3	4	8	10	0	1
getStopCommand(int)		0%		0%	2	2	5	5	1	1
executeCommand(String, String, String[])		93%		87%	1	5	4	30	0	1
generateStartCommand(int, String, String)		91%		n/a	0	1	2	15	0	1
checkWorker(NIONode, String)		96%		80%	2	6	2	19	0	1
static {...}		95%		50%	6	7	0	14	0	1
setWorkerIsReady()		100%		n/a	0	1	0	3	0	1
getCleanWorkerWorkingDir(String)		100%		n/a	0	1	0	5	0	1
WorkerStarter(NIOWorkerNode)		100%		n/a	0	1	0	5	0	1
getWorkerStarter(String)		100%		n/a	0	1	0	1	0	1
setToStop()		100%		n/a	0	1	0	2	0	1
Total	311 of 908	65%	35 of 68	48%	29	48	56	194	1	14

Si volguéssim obtenir un informe llegible de les línies de Python hem d'executar la comanda `coverage report` i obtindrem:



Name	Stmts	Miss	Cover
/home/sergi/.local/lib/python2.7/site-packages/backports/__init__.py	1	0	100%
/home/sergi/.local/lib/python2.7/site-packages/builtins/__init__.py	7	1	86%
/home/sergi/.local/lib/python2.7/site-packages/future/__init__.py	9	0	100%
/home/sergi/.local/lib/python2.7/site-packages/future/backports/__init__.py	7	1	86%
/home/sergi/.local/lib/python2.7/site-packages/future/backports/misc.py	469	343	27%
/home/sergi/.local/lib/python2.7/site-packages/future/builtins/__init__.py	18	10	44%
/home/sergi/.local/lib/python2.7/site-packages/future/builtins/iterators.py	15	6	60%
/home/sergi/.local/lib/python2.7/site-packages/future/builtins/misc.py	50	33	34%
/home/sergi/.local/lib/python2.7/site-packages/future/builtins/new_min_max.py	34	26	24%
/home/sergi/.local/lib/python2.7/site-packages/future/builtins/newnext.py	16	12	25%
/home/sergi/.local/lib/python2.7/site-packages/future/builtins/newround.py	42	37	12%
/home/sergi/.local/lib/python2.7/site-packages/future/builtins/newsuper.py	45	36	20%
/home/sergi/.local/lib/python2.7/site-packages/future/standard_library/__init__.py	270	216	20%
/home/sergi/.local/lib/python2.7/site-packages/future/types/__init__.py	52	27	48%
/home/sergi/.local/lib/python2.7/site-packages/future/types/newbytes.py	225	163	28%
/home/sergi/.local/lib/python2.7/site-packages/future/types/newdict.py	42	28	33%
/home/sergi/.local/lib/python2.7/site-packages/future/types/newint.py	218	168	23%
/home/sergi/.local/lib/python2.7/site-packages/future/types/newlist.py	40	22	45%
/home/sergi/.local/lib/python2.7/site-packages/future/types/newobject.py	26	18	31%
/home/sergi/.local/lib/python2.7/site-packages/future/types/newrange.py	87	59	32%
/home/sergi/.local/lib/python2.7/site-packages/future/types/newstr.py	193	142	26%
/home/sergi/.local/lib/python2.7/site-packages/future/utils/__init__.py	290	204	30%
/home/sergi/.local/lib/python2.7/site-packages/queue/__init__.py	6	1	83%
/home/sergi/tests_execution_sandbox/apps/app103/src/__init__.py	0	0	100%
/home/sergi/tests_execution_sandbox/apps/app103/src/simple.py	38	3	92%
/home/sergi/tutorial_apps/python/simple/src/simple.py	31	10	68%
/opt/COMPSs/Bindings/python/2/pycompss/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/api/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/api/api.py	97	54	44%
/opt/COMPSs/Bindings/python/2/pycompss/api/commons/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/api/commons/data_type.py	33	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/api/exceptions.py	5	2	60%
/opt/COMPSs/Bindings/python/2/pycompss/api/parameter.py	204	59	71%
/opt/COMPSs/Bindings/python/2/pycompss/api/task.py	686	351	49%
/opt/COMPSs/Bindings/python/2/pycompss/runtime/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/runtime/binding.py	554	335	40%
/opt/COMPSs/Bindings/python/2/pycompss/runtime/commons.py	27	10	63%
/opt/COMPSs/Bindings/python/2/pycompss/runtime/core_element.py	45	10	78%
/opt/COMPSs/Bindings/python/2/pycompss/runtime/launch.py	160	70	56%
/opt/COMPSs/Bindings/python/2/pycompss/streams/__init__.py	1	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/streams/components/__init__.py	1	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/streams/components/dlstro_stream_client.py	131	104	21%
/opt/COMPSs/Bindings/python/2/pycompss/streams/environment.py	13	4	69%
/opt/COMPSs/Bindings/python/2/pycompss/streams/types/__init__.py	1	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/streams/types/requests.py	107	60	44%
/opt/COMPSs/Bindings/python/2/pycompss/util/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/util/arguments.py	35	16	54%
/opt/COMPSs/Bindings/python/2/pycompss/util/context.py	23	2	91%
/opt/COMPSs/Bindings/python/2/pycompss/util/environment/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/util/environment/configuration.py	238	211	11%
/opt/COMPSs/Bindings/python/2/pycompss/util/logger/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/util/logger/helpers.py	28	2	93%
/opt/COMPSs/Bindings/python/2/pycompss/util/objects/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/util/objects/properties.py	88	49	44%
/opt/COMPSs/Bindings/python/2/pycompss/util/objects/sizer.py	30	21	30%
/opt/COMPSs/Bindings/python/2/pycompss/util/serialization/__init__.py	0	0	100%
/opt/COMPSs/Bindings/python/2/pycompss/util/serialization/extended_support.py	10	5	50%

Sistema per desplegar tests automàticament al Supercomputador

En aquest apartat es descriurà amb profunditat la problemàtica i la solució que s'ha dissenyat i implementat per poder desplegar automàticament tests cap a un Supercomputador.

7.3.1. Problemàtica

Anteriorment, l'única manera d'executar tests en un Supercomputador era copiant tots els fitxers dels tests de la nostra màquina local cap al Supercomputador, per així poder-los executar en remot a continuació. Evidentment aquesta és una possibilitat, però un sistema per automatitzar aquesta acció podria estalviar molt de temps al desenvolupador.

Per qüestions de seguretat, MareNostrum 4 està configurat de tal manera que permet la connexió des de la nostra màquina local al Supercomputador, però mai a l'inrevés. A més a més, els nodes de *login*

estan limitats en memòria, així que les execucions s'hauran de realitzar sempre mitjançant el sistema de cues.

7.3.2. Eines disponibles

A continuació es citen les eines disponibles al mercat que s'han considerat per realitzar aquest sistema:

7.3.2.1. *Mòduls de Python*

7.3.2.1.1. Polling

Polling (12) és un mòdul que s'utilitza per esperar a que una funció retorni una determinada condició esperada. Permet nombroses possibilitats, entre elles el poder comunicar-se amb una màquina remota. És una eina molt útil ja que permet a un script poder preguntar al Supercomputador si ja ha acabat l'execució de tots els tests.

7.3.2.1.2. Pandas

Pandas (13) és una eina que permet la manipulació i l'accés de fitxers amb gran quantitats de dades. Aquesta eina és útil per processar llistes, com ara l'informe amb els resultats de les execucions dels tests que es mostra a l'usuari. És una eina molt popular i està correctament documentada a la seva pàgina web.

7.3.3. Solució

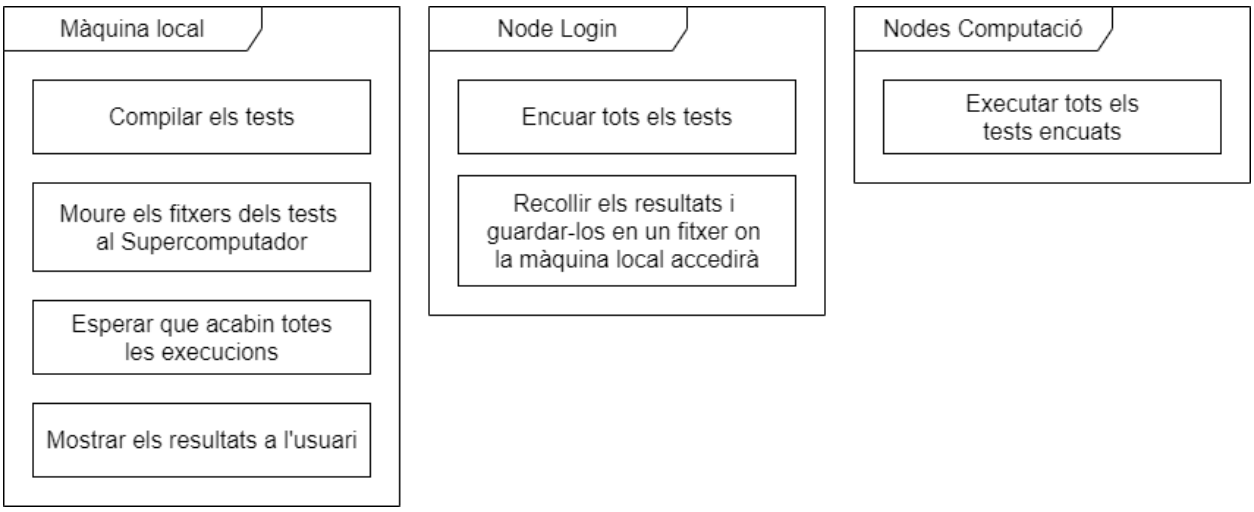
En aquesta secció es donarà una visió general de la solució proporcionada a aquest problema, juntament amb una explicació més detallada i tècnica del que s'ha fet.

7.3.3.1. *Visió general*

Per resoldre aquest problema s'han dissenyat i implementat un total de quatre scripts en Python, els quals es detallaran en profunditat en el capítol d'implementació. Bàsicament, aquests scripts s'encarreguen de compilar i desplegar els tests en local, moure'ls al Supercomputador que l'usuari hagi especificat, executar aquests tests i, finalment, recollir els resultats. Aquests resultats es mostren en un llistat a l'usuari, indicant els tests que s'han executat amb èxits i quins no. Per a aquells tests que no s'hagin executat amb èxit, l'usuari pot entrar dins el Supercomputador i comprovar els resultats.

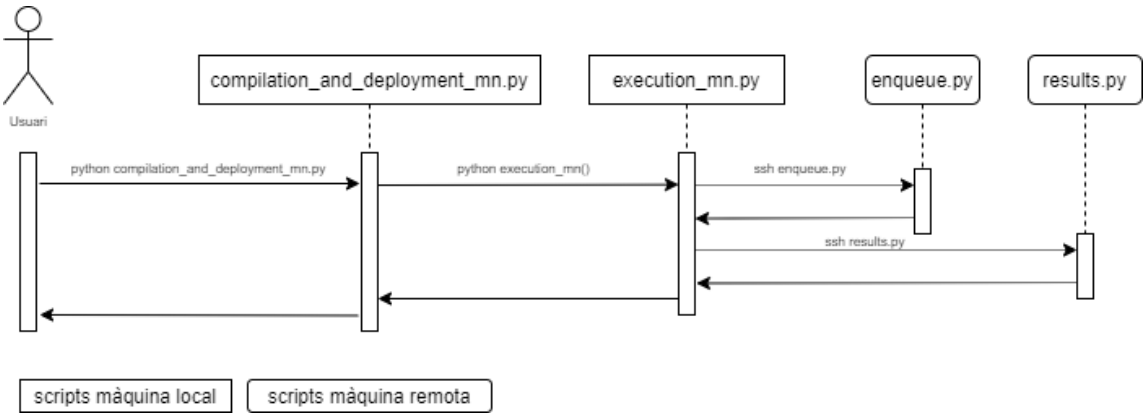
Degut a que la configuració de MareNostrum 4 no permet la connexió entre el Supercomputador i la nostra màquina local, un dels scripts s'encarregarà d'anar preguntat periòdicament al Supercomputador si tots els tests que ha llançat han finalitzat.

Aquest esquema mostra com estan repartides les tasques entre la màquina local i el Supercomputador:



7.3.3.2. *Implementació*

Com s'ha comentat abans, s'han dissenyat i implementat quatre scripts de Python. Aquests scripts s'executaran seqüencialment en l'ordre que es mostra a l'esquema següent:



Cal tenir en compte que addicionalment s'ha hagut de crear un fitxer *MN.cfg* on es guardaran els paràmetres de configuració de l'usuari. Al llarg d'aquest capítol s'explicarà el funcionament dels més essencials.

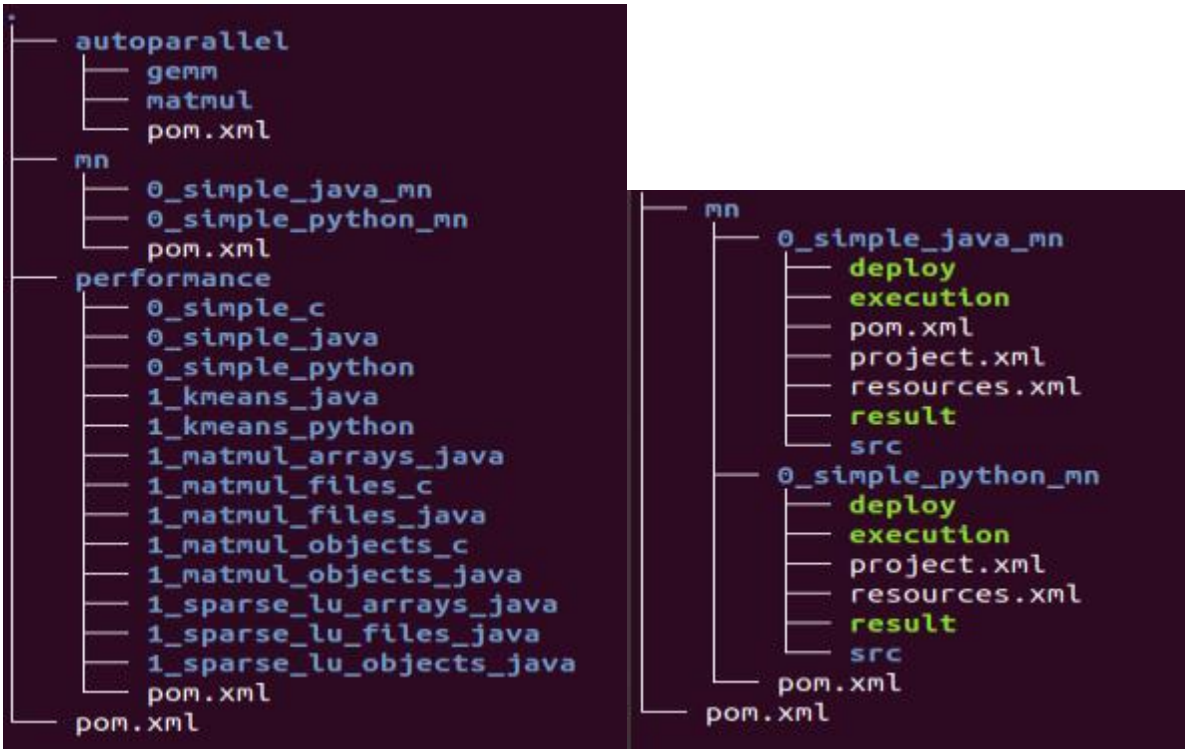
7.3.3.2.1. Script per compilar i desplegar els tests al Supercomputador (compilation_and_deployment_mn.py)

Aquest és el primer script que s'executa i l'únic que haurà d'executar manualment l'usuari per engegar tot el procés. Bàsicament, s'encarrega de compilar i desplegar al Supercomputador els tests que hem

especificat al programa. Per fer això, primer compila i desplega els tests en la màquina local, i a continuació copia tots els fitxers a la carpeta dins del Supercomputador que hàgim especificat al paràmetre *deploy_path* dins el nostre fitxer de configuració.

7.3.3.2.1.1. Estructura dels tests en local

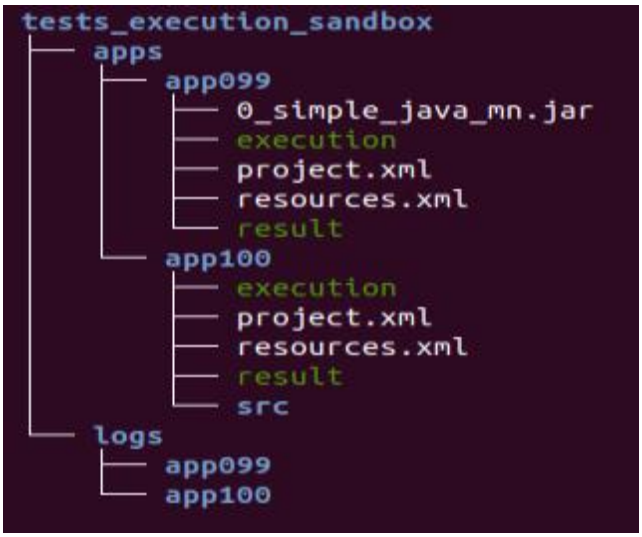
Per tal que els tests es puguin compilar i desplegar correctament, aquests han d'estar emmagatzemats dins la nostra màquina local de la forma correcta. La següent imatge mostra quina hauria de ser l'estructura del nostre directori de tests per assegurar que es compilin i es despleguin correctament:



La figura de l'esquerra mostra com estan les famílies dels tests estructurades dins la carpeta de tests, mentre que la figura de la dreta mostra com estan els tests estructurats dins les famílies. És a dir, en aquest directori, el test *0_simple_java_mn* forma part de la família *mn.*

7.3.3.2.1.2. Estructura dels tests en remot

Un cop els tests han sigut compilats i desplegats en local, el programa s'encarregarà de moure tots els fitxers al Supercomputador. La següent figura mostra quina serà l'estructura dels tests desplegats en el Supercomputador.



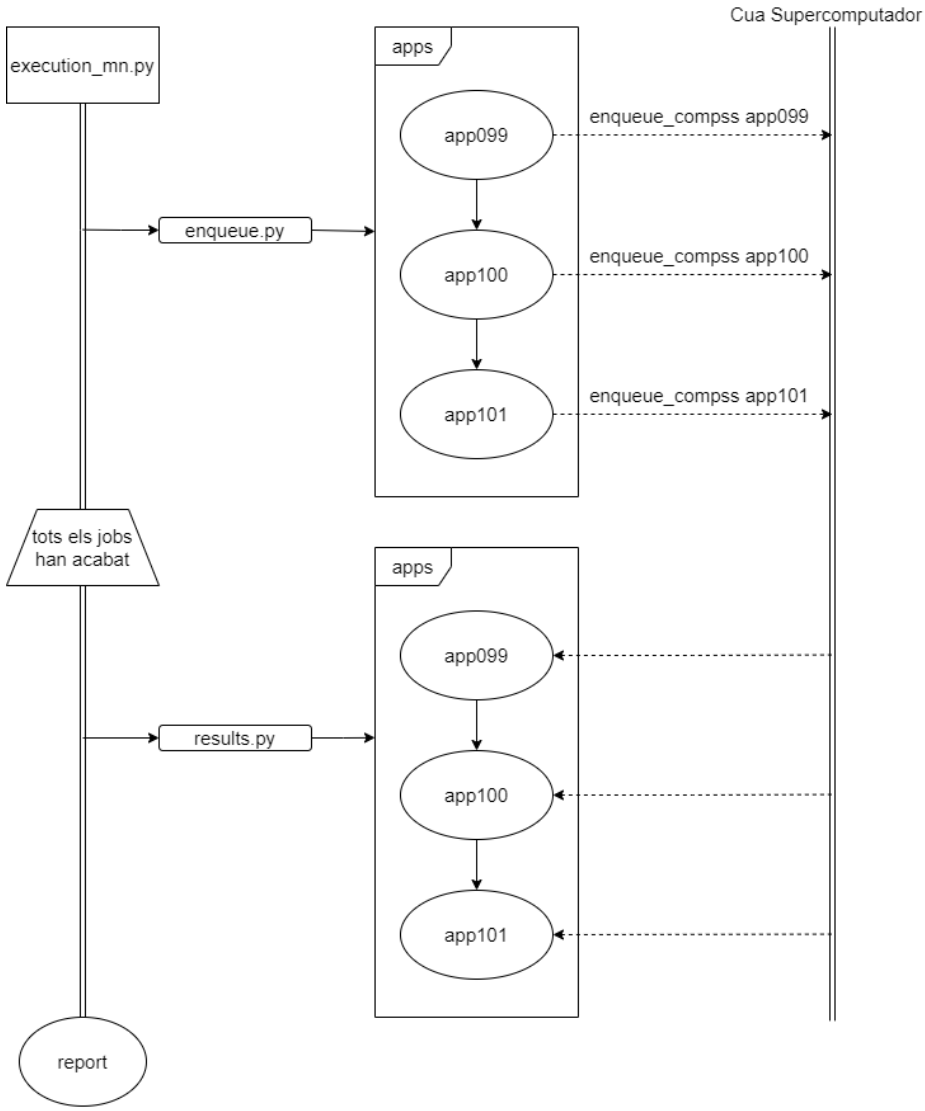
Com es pot observar, l'estructura dels tests dins del Supercomputador ha canviat lleugerament. Els tests que s'han desplegat en aquest exemple han sigut els dos dins de la família *mn*. El programa s'encarrega d'associar un identificador a cada un dels tests, el qual s'utilitzarà com a nom dins del Supercomputador. És a dir, en aquest cas el test anomenat *0_simple_java_mn* té com identificador el 99, i per tant, al Supercomputador apareix com a *app099.*

En la carpeta de *logs* es guardaran els resultats de l'execució que s'utilitzaran per avaluar si el test ha passat o no. S'explicarà a l'apartat del *results.py.*

7.3.3.2.2. Script per comunicar-se amb el Supercomputador (execution_mn.py)

Aquest és el script que s'executarà un cop els tests ja han sigut desplegats dins del Supercomputador. És el script que s'encarrega de comunicar-se amb el Supercomputador i d'enviar-li la majoria de paràmetres del nostre fitxer de configuració *MN.cfg*.

A continuació, es mostra un esquema on s'hi veu il·lustrada aquesta comunicació:



Com es pot observar, aquest script té el paper de cridar els dos scripts que hi ha al Supercomputador en el moment precís. Un cop s'ha cridat *enqueue.py,* tots els tests que estaven dins la carpeta de desplegament dins del Supercomputador seran encuats.

7.3.3.2.2.1. Sistema de polling

Un cop tots els *jobs* s'han enviat a la cua, caldrà esperar a que acabin tots per poder recollir els resultats. Una possible solució podria ser que el Supercomputador comuniqués a l'usuari quan els tests hagin acabat, però la naturalesa del protocol que s'utilitza per la comunicació (SSH) no ho permet.

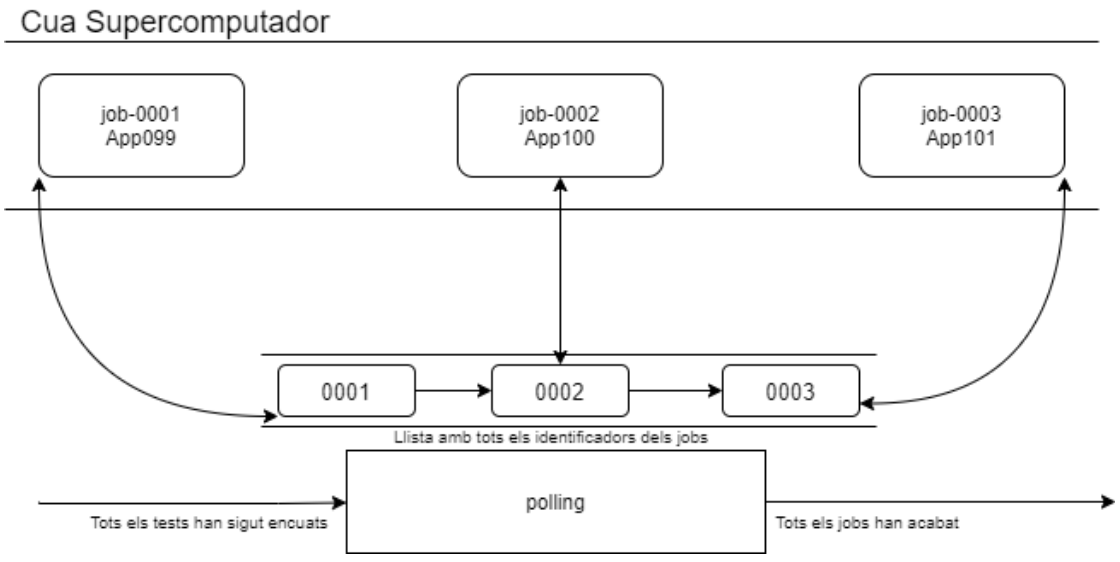
Per aquesta raó s'ha hagut de pensar una manera per fer que el script *execution_mn.py* sàpiga quan han acabat tots els *jobs* de la cua per així procedir a recollir els resultats.

La solució a aquest problema ha sigut implementar un sistema de *polling* dins del script. És a dir, el script *execution_mn.py* preguntarà periòdicament al Supercomputador si tots els *jobs* que s'han enviat a la cua prèviament han acabat.

Per aconseguir-ho s'ha fet el següent codi:

```
1. for job in jobs:
2.     polling.poll(
3.         lambda: not subprocess.check_output("ssh user@host.es 'squeue -h -j {}'".format(job), shell=True),
4.         step=10,
5.         poll_forever=True
6.     )
```

Bàsicament, per cada un dels *jobs* que s'han enviat a la cua, el script pregunta al Supercomputador si aquest ha acabat. En el cas contrari, s'espera 10 segons i ho torna a preguntar. En el següent esquema es mostra la lògica:



7.3.3.2.3. Script per encuar els tests a Supercomputador (enqueue.py)

Aquest és el script dins del Supercomputador que s'encarrega de recórrer tota la carpeta dels tests desplegats per encuar-los. Com que aquest és cridat per *execution_mn.py*, rebrà per paràmetre tots els paràmetres de configuració del fitxer *MN.cfg*.

7.3.3.2.3.1. Llençament d'un test

Per llençar un test s'ha d'executar un fitxer *bash* que està inclòs en tots els tests anomenat *execution*. Aquest fitxer *bash* requereix diversos paràmetres i sempre farà una crida al executable *runcompss* si es tracta d'un test per executar en local, o a *enqueue_compss* si s'executarà el test en un Supercomputador. És a dir, *runcompss* s'utilitza per entorns interactius i *enqueue_compss* en entorns amb cues. Els dos scripts requereixen paràmetres diferents, per tant, aquests fitxers *bash* seran diferents.

A continuació, es mostren quins són els paràmetres que requereix *enqueue_compss.* Cal tenir en compte que aquests paràmetres es podran definir al fitxer de configuració *MN.cfg.*

- **Exec_time:** Temps límit en minuts que tindrà el *job* per acabar.
- **Num_nodes:** Número de nodes màxim que farà servir el *job.*
- **Comm:** Classe que implementa l'adaptador per les comunicacions.
- **Master_working_dir**: Directori on es guardaran els fitxers de sortida del *Master.*
- **Worker_working_dir**: Directori on es guardaran els fitxers de sortida dels *Worker.*
- **Base_log_dir**: Directori on es guardaran els fitxers de sortida de l'execució.
- **Runcompss_opts**: Paràmetres opcionals de *runcompss.*

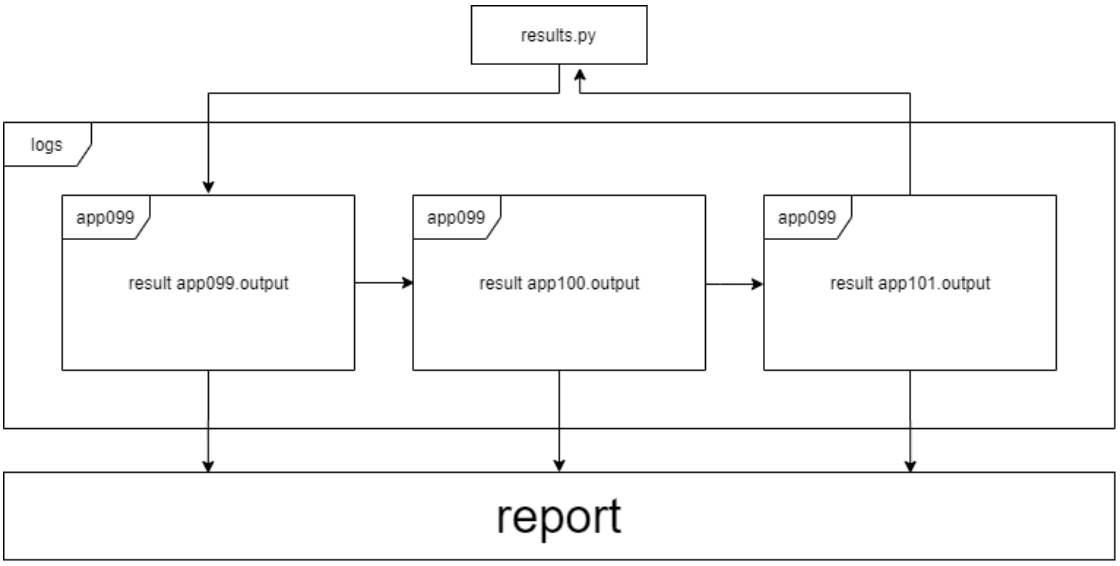
7.3.3.2.4. Script per recollir els resultats de les execucions (results.py)

Un cop tots els *jobs* de la cua han acabat, és el torn del script *results.py* de recollir els resultats. Com s'ha mencionat anteriorment, la carpeta de desplegament dels tests conté una directori anomenat *logs*, el qual contindrà una carpeta per cada un dels tests que s'hagin executat en el Supercomputador. Aquesta carpeta contindrà els fitxers de sortida que hagi generat l'execució.

Seguidament, el script *results.py* s'encarregarà de recórrer la carpeta de *logs* avaluant cada una de les execucions i guardant en un fitxer els resultats. Aquests resultats són recollits per el script *execution_mn.py* i mostrats a l'usuari en un *report*.

Result és un fitxer *bash* que està inclòs en tots els tests. Aquest s'encarrega d'avaluar el test mitjançant els fitxers de sortida que hagi generat l'execució. Per tant, si volem saber si el test s'ha executat correctament, hem d'executar aquest fitxer juntament amb tots els fitxers de sortida que l'execució ha generat.

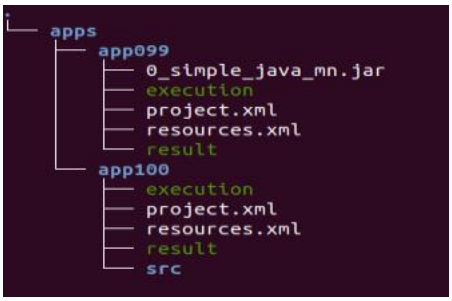
El següent esquema mostra la lògica de *results.py:*



7.3.4. Resultats

En aquesta secció es mostraran els resultats que validaran el correcte funcionament d'aquest sistema.

Si executem el script compilation_and_deployment_mn.py i escollim els tests 099 i 100, primer hem de comprovar que efectivament els dos tests s'han desplegat a la carpeta corresponent del Supercomputador:



Com podem observar, els dos tests que s'han triat per executar estan correctament guardats a la carpeta. La comanda *tree* s'ha llançat des de el directori on es despleguen els tests i que nosaltres hem especificat en el fitxer de configuració.

Un cop els tests han estat desplegats, es posen tots a la cua. La comanda *squeue* ens permet visualitzar l'estat de la cua:

```
bsc19073@login1:~> squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          10597224      main   COMPSs bsc19073 PD       0:00      5 (QOSMaxJobsPerUserLimit)
          10597225      main   COMPSs bsc19073 PD       0:00      5 (QOSMaxJobsPerUserLimit)
          10597223      main   COMPSs bsc19073  R       0:23      5 s05r2b57,s17r2b[25-28]
```

A mesura que va transcorrent el temps, podem consultar l'estat de la cua:

```
bsc19073@login1:~> squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          10597224      main   COMPSs bsc19073 PD       0:00      5 (QOSMaxJobsPerUserLimit)
          10597225      main   COMPSs bsc19073 PD       0:00      5 (QOSMaxJobsPerUserLimit)
```

```
bsc19073@login1:~> squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          10597225      main   COMPSs bsc19073  R       0:27      5 s17r2b[25-28],s22r2b21
```

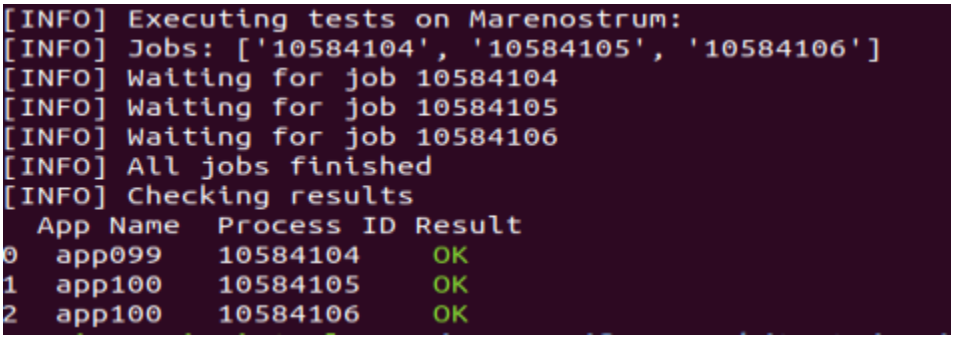
Com es pot observar, s'han executat dos tests, però s'han creat tres processos. Això és degut a que l'execució del test número 100 requereix de dos processos.

Un cop tots els processos han acabat, podem comprovar que les execucions han guardat els resultats a la carpeta que haguem especificat al fitxer de configuració:

```
.
└── logs
    ├── app099_1
    │   ├── compss-10597223.err
    │   ├── compss-10597223.out
    │   ├── counter
    │   ├── simple_01.errorlog
    │   └── simple_01.outputlog
    └── app100_1
        ├── compss-10597224.err
        ├── compss-10597224.out
        ├── compss-10597225.err
        ├── compss-10597225.out
        ├── counter
        ├── simple.py_python2_01
        ├── simple.py_python2_01.errorlog
        ├── simple.py_python2_01.outputlog
        ├── simple.py_python3_01
        ├── simple.py_python3_01.errorlog
        └── simple.py_python3_01.outputlog
```

Aquests fitxers seran necessaris per comprovar si els tests s'han executat amb èxit o no.

Finalment, des de la consola on haguem executat el script compilation_and_deployment_mn.py hauríem de poder visualitzar els resultats:



En el cas que algun dels tests fallés, podríem accedir al Supercomputador per comprovar els missatges d'error.

Sistema per actualitzar la documentació automàticament

En aquest capítol s'explicarà el sistema que s'ha desenvolupat per actualitzar automàticament la documentació de COMPSs.

7.4.1. Problemàtica

COMPSs disposa de dos tipus de documentació: documentació pels usuaris i pels desenvolupadors. La documentació per usuaris està escrita manualment, doncs ha d'estar el més ben explicada possible per a què usuaris nous del *framework* la puguin entendre. En canvi, la documentació per desenvolupadors és un document accessible només pels desenvolupadors on hi ha tota la informació rellevant per a ells, com ara les classes i mètodes, i la seva funcionalitat.

Aquest document està escrit a mà, així que sempre que hi hagi un canvi en el codi font el document pot quedar-se obsolet.

7.4.2. Eines disponibles

A continuació es citen les eines disponibles al mercat que permeten generar documentació automàticament del codi font i que s'han fet servir per desenvolupar aquest sistema.

7.4.2.1. *Javadocs*

Javadocs (14) és una utilitat d'*Oracle* per a la generació de documentació de *APIs* en format *html* a partir del codi font. El desenvolupador pot documentar directament el codi font mitjançant comentaris i

etiquetes , i *Javadocs* s'encarregarà d'extreure aquesta informació per generar la documentació. Aquests comentaris no afecten el rendiment, doncs els comentaris es treuen en temps de compilació.

A continuació es mostra un exemple de com documentar un mètode:

```
1.  /**
2.   * @param a Primer parametre d'exemple
3.   * @param b Segon parametre d'exemple
4.   * @throws IllegalArgumentException Si algun dels String te longitud 0
5.   * @return La suma de caracters dels dos Strings d'entrada
6.   */
7.  public static int exemple(String a, String b){
8.
9.      if (a.length() == 0 || b.length() == 0){
10.         throw new IllegalArgumentException("Els String d'entrada no poden estar buits");
11.     }
12.
13.     return a.length()+b.length();
14. }
```

Aquest codi generarà la següent entrada dins la documentació generada:

exemple

```
public static int exemple(java.lang.String a,
                          java.lang.String b)
```

Parameters:

a - Primer parametre d'exemple

b - Segon parametre d'exemple

Returns:

La suma de caracters dels dos Strings d'entrada

Throws:

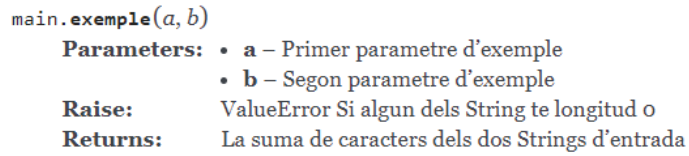
`java.lang.IllegalArgumentException` - Si algun dels String te longitud o

7.4.2.2. *Docstrings*

Els *strings de documentació* de Python o *docstrings* (15) permeten associar documentació amb mòduls, funcions i classes de Python. Al igual que *Javadocs*, la documentació va inclosa amb el codi font. A continuació, es mostra com documentar un mètode de Python amb *docstrings*:

```
15. def exemple(a,b):
16.     """
17.     :param a: Primer parametre d'exemple
18.     :param b: Segon parametre d'exemple
19.     :raise: ValueError Si algun dels String te longitud 0
20.     :return: La suma de caracters dels dos Strings d'entrada
21.     """
22.     if len(a) == 0 or len(b) == 0:
23.         raise ValueError("Els String d'entrada no poden estar buits")
24.     return len(a)+len(b)
```

Això generarà la documentació següent:



main.**exemple**(*a*, *b*)

Parameters: • **a** – Primer parametre d'exemple
• **b** – Segon parametre d'exemple

Raise: ValueError Si algun dels String te longitud 0

Returns: La suma de caracters dels dos Strings d'entrada

7.4.2.3. *PlantUML i plantuml-generator*

PlantUML (16) és un software de codi obert que permet als usuaris crear diagrames UML a partir d'un llenguatge de text normal. L'usuari pot definir un diagrama de forma simple utilitzant un llenguatge intuïtiu, i la seva pàgina web compta amb bastant documentació.

Existeix un *plugin* de *Maven* anomenat *plantuml-generator* (17) que permet extreure un diagrama de classe en format *PlantUML* a partir del pom.xml d'un projecte:

```
<dependency>
        <groupId>de.elnarion.util</groupId>
        <artifactId>plantuml-generator-util</artifactId>
        <version>1.1.2</version>
</dependency>
```

Gràcies a aquest *plugin*, l'usuari pot generar un diagrama de classes del corresponent projecte al compilar-lo.

7.4.2.4. *Sphinx*

Sphinx (18) és una eina que permet fàcilment generar documentació tant de Python com de Java amb el suport de *docstrings* i *Javadocs.* És una eina molt còmoda, ja que li pots especificar la ubicació del codi font que vols documentar, i aquesta s'encarregarà de generar un *index.html* que indexarà tots els mòduls i paquets del codi font documentats.

La documentació feta amb *Sphinx* està formada per fitxers d'extensió *.rst.* Dins d'aquests fitxers s'hi pot redactar la documentació del nostre codi font i indexar altres fitxers *.rst.*

Amb la crida *sphinx-apidoc* és possible obtenir els *.rst* del nostre codi font automàticament.

7.4.2.5. *Extensions Sphinx*

Sphinx és un software bastant extens, i com a tal compta amb nombroses extensions. A continuació es citen les que s'han fet servir:

7.4.2.5.1. Javasphinx

Javasphinx (19) proporciona a *Sphinx* un domini per documentar projectes fets amb Java, juntament amb una utilitat anomenada *javasphinx-apidoc* capaç de generar documentació d'una API extreta directament del codi font. Per a que l'extensió funcioni correctament el codi font ha de seguir el model de *Javadocs.*

7.4.2.5.2. Sphinxcontrib-plantuml

Sphinxcontrib-plantuml (20) permet renderitzar un fitxer *PlantUML* en format imatge per a que es pugui mostrar a la documentació. Tant sols cal especificar la ubicació del fitxer *PlantUML* dins el fitxer *.rst* on el vulguem col·locar.

7.4.2.5.3. Sphinx RTD Theme

Aquesta extensió (21) renderitza els nostres fitxers *.rst* en un projecte *.html* bastant intuïtiu i agradable per la vista. A més a més disposa d'alguns botons de navegació bastant útils que el tema que s'utilitza per defecte no inclou.

7.4.3. Solució

En aquesta secció es donarà una visió general de la solució proporcionada a aquest problema, juntament amb una explicació més detallada i tècnica del que s'ha fet.

7.4.3.1. *Visió general*

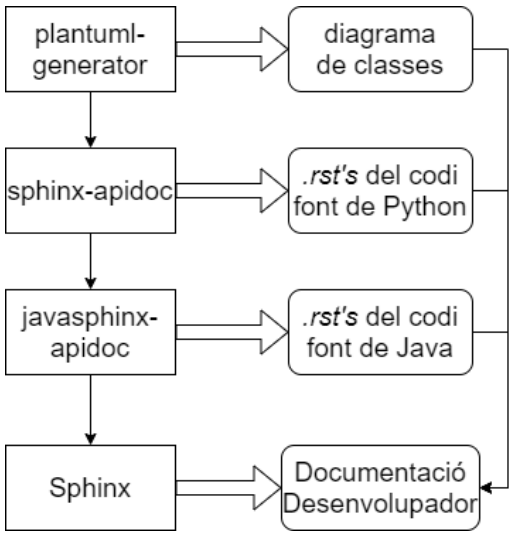
Per a poder prescindir d'aquest document tant susceptible de quedar-se obsolet, s'ha decidit dissenyar i implementar un sistema que permeti generar un *.html* amb tota la documentació de desenvolupador automàticament. Per aconseguir-ho, s'ha hagut d'integrar l'execució de les diferents eines mencionades anteriorment per aconseguir el següent:

- Un diagrama de classes *PlantUML,* obtingut amb *plantuml-generator.*
- Tots els fitxers *.rst* extrets del codi font de Python i Java del projecte a documentar. Les crides *sphinx-apidocs* i *javasphinx-apidoc* s'encarregaran d'obtenir aquests fitxers.
- Un fitxer *.rst* principal que indexi tots els *.rst* del codi font.

L'extensió *Sphinx* s'encarrega de ajuntar tots aquests elements en un conjunt de pàgines *.html*.

D'aquesta manera, només cal executar aquest script per generar tota la documentació i aquesta ja està llesta per penjar i compartir amb els demés desenvolupadors.

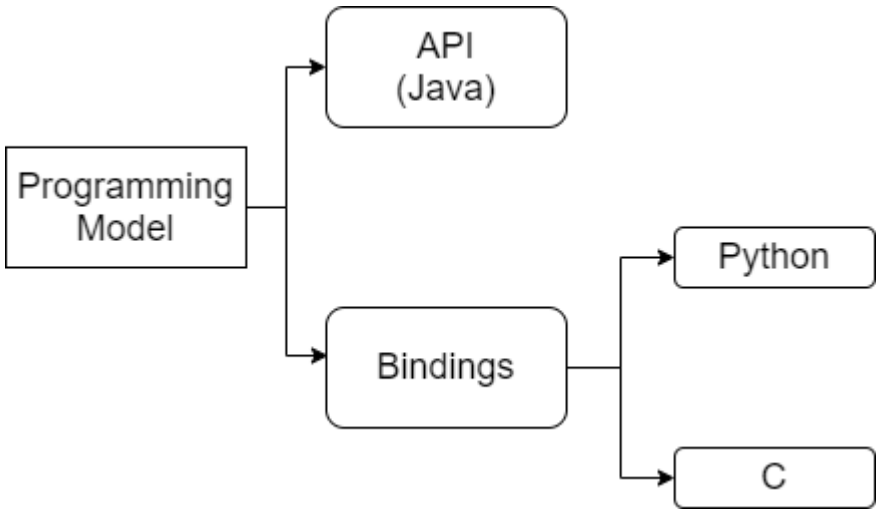
El següent esquema mostra quin és l'ordre de l'execució de les eines:



7.4.3.2. *Implementació*

Degut a que el codi font de *COMPSs* és bastant extens, i la durada d'aquest projecte és limitada, s'ha decidit integrar aquest sistema de documentació en només una part del codi, el Model de Programació. D'aquesta manera, s'ha pogut validar que el sistema es pot integrar en qualsevol altra part del codi que es desitgi.

El Model de Programació està organitzat de la següent forma:



Aquest sistema està integrat en el Model de Programació, però està pensat per a que funcioni en altres projectes sense haver de fer gaires canvis en el codi.

7.4.3.2.1. Script per generar la documentació automàticament

Com s'ha dit anteriorment, per solucionar aquest problema s'ha hagut de implementar un script de Python que faci les següents coses seqüencialment:

7.4.3.2.1.1. Compilar el projecte a documentar

La primera cosa que s'ha de fer es compilar el projecte. La compilació extraurà el diagrama de classes del *pom.xml* en format *PlantUML*, i així es podrà adjuntar a la nostra documentació.

Aquesta simple línia permet compilar un projecte de *Maven* en el directori que vulguem:

```
p = subprocess.Popen(["mvn", "clean", "install"], cwd="..")
```

7.4.3.2.1.2. Generar la documentació de Java

Com s'ha comentat anteriorment, existeix una extensió de *Javasphinx* que proporciona una utilitat per extreure tota la documentació del codi font fet en Java. Per tant, el script de Python ha d'executar la corresponent comanda i guardar tota aquesta documentació a la carpeta on vulguem.

Aquesta és la comanda que el script de Python crida per generar la documentació del codi Java del Model de Programació:

```
p = subprocess.Popen(["javasphinx-apidoc", "-o", "packages", "-f",
"../api/src/main/java/es/bsc/compss"])
```

7.4.3.2.1.3. Generar la documentació de Python

En cas que el projecte a documentar tingui parts fetes en Python, aquestes també s'hauran de tenir en compte. La següent crida de *Sphinx* extreu la documentació del codi fet en Python que vulguem:

```
p = subprocess.Popen(["sphinx-apidoc", "-o", "modules", "-f", "../bindings/python/src/"])
```

7.4.3.2.1.4. Compilar el .html

Un cop ja tenim el diagrama de classes, la documentació del codi Java, i la documentació del codi en Python, ja és possible compilar un .html amb tota la documentació per a que es pugui entregar al desenvolupador. Amb aquesta línia, el script crida realitzat el *make* de tota la documentació que hem generat.

```
p = subprocess.Popen(["make", "html"])
```

Això generarà una carpeta amb tot el *build* de la documentació.

7.4.3.2.2. Organització

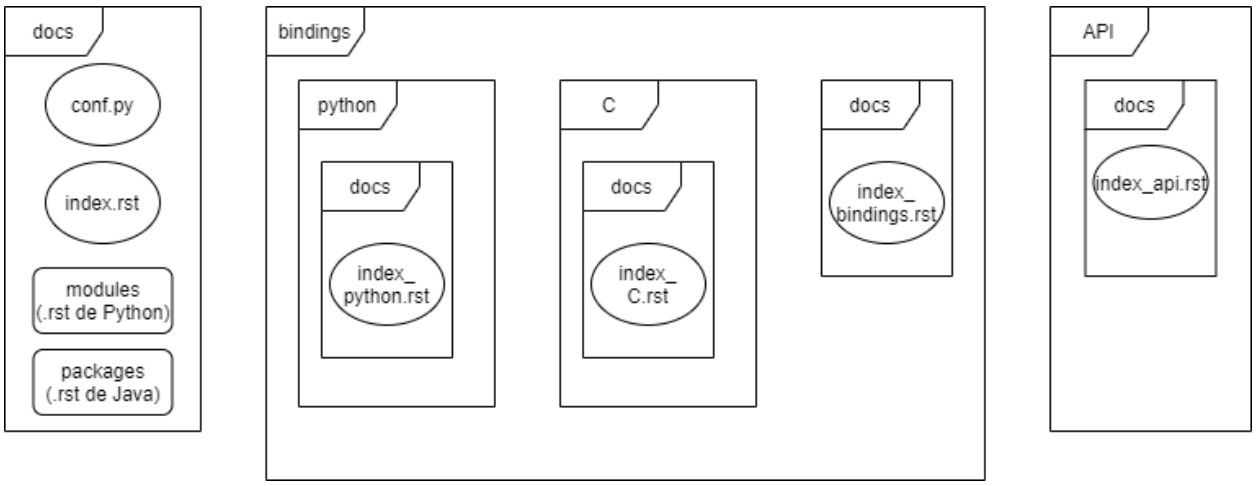
Com s'ha mencionat anteriorment, *Sphinx* proporciona crides *API* per generar documentació. Amb això no hi ha prou, doncs l'únic que s'obté és un munt de fitxers *.rst* separats en diferents carpetes dins la màquina local amb la documentació de cada mòdul o paquet individual. El lector hauria de navegar carpeta per carpeta buscant els *.rst* amb la documentació que trobi rellevant, i això és poc pràctic.

Els fitxers *.rst* tenen la capacitat d'indexar altres fitxers *.rst.* Això es molt útil, doncs permet establir una jerarquia de *.rst* que correspongui a l'estructura dels directoris del projecte que vulguem documentar. Per tant, s'ha decidit tenir fitxers *.rst* que indexin altres fitxers *.rst*, i fitxers *.rst* amb la corresponent documentació del projecte. Llavors, els fitxers *.rst* es poden entendre com pàgines *.html* que estan enllaçades amb altres pàgines i que permeten posar-hi tot el text que necessitem. De fet, parlant en baix nivell, el compilador s'encarrega de convertir aquests *.rst* en pàgines *.html*.

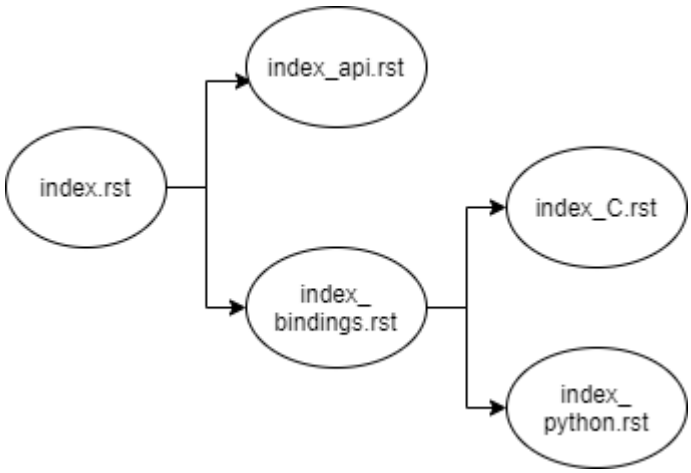
Per integrar aquest sistema dins del Model de Programació, s'ha decidit organitzar la documentació de la següent forma:

1. Una carpeta *docs* dins del directori de Model de Programació per guardar tots els fitxers de configuració de *Sphinx*, un *index.rst* que indexi tota la documentació del codi font, una carpeta *modules* amb la documentació relacionada als *bindings* de Python, i una carpeta *Packages* amb tota la documentació de l'*API.*

2. Una jerarquia de *index.rst* que respecti l'estructura del Model de Programació, tal com es mostra a la figura:



Aquests fitxers *.rst* s'han indexat de la següent manera:



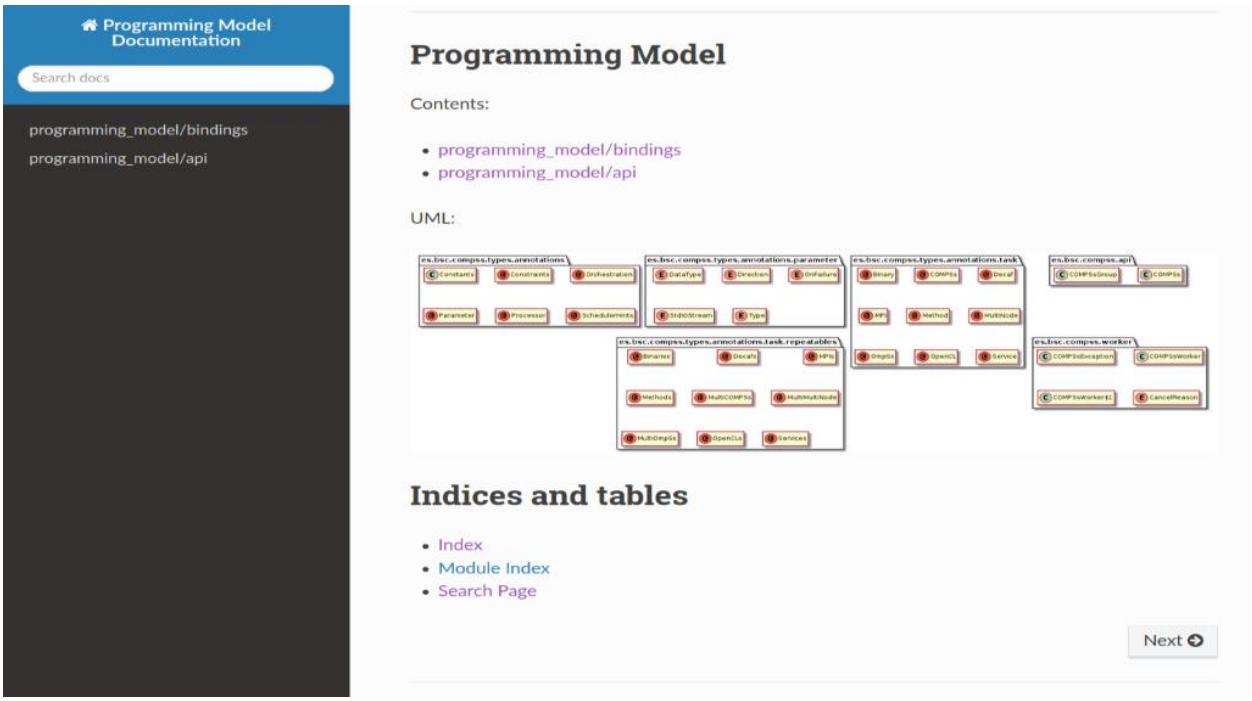
Com s'ha comentat, aquesta organització correspon al Model de Programació, tot i que es podria aplicar a qualsevol altre projecte fàcilment.

7.4.4. Resultats

A continuació es mostren un conjunt d'imatges que validen que el sistema funciona correctament.

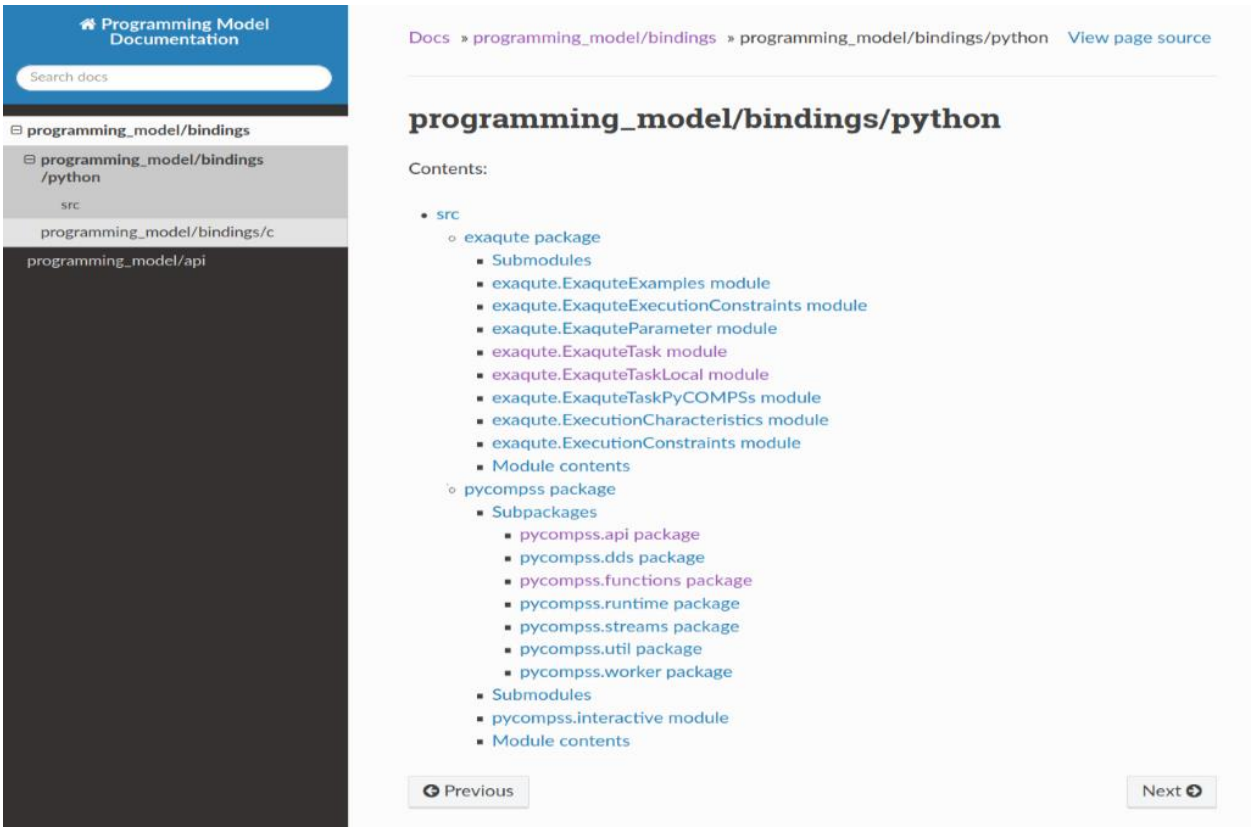
Executant el script *docs.py* obtenim les següents pàgines *.html.*

La pàgina principal:

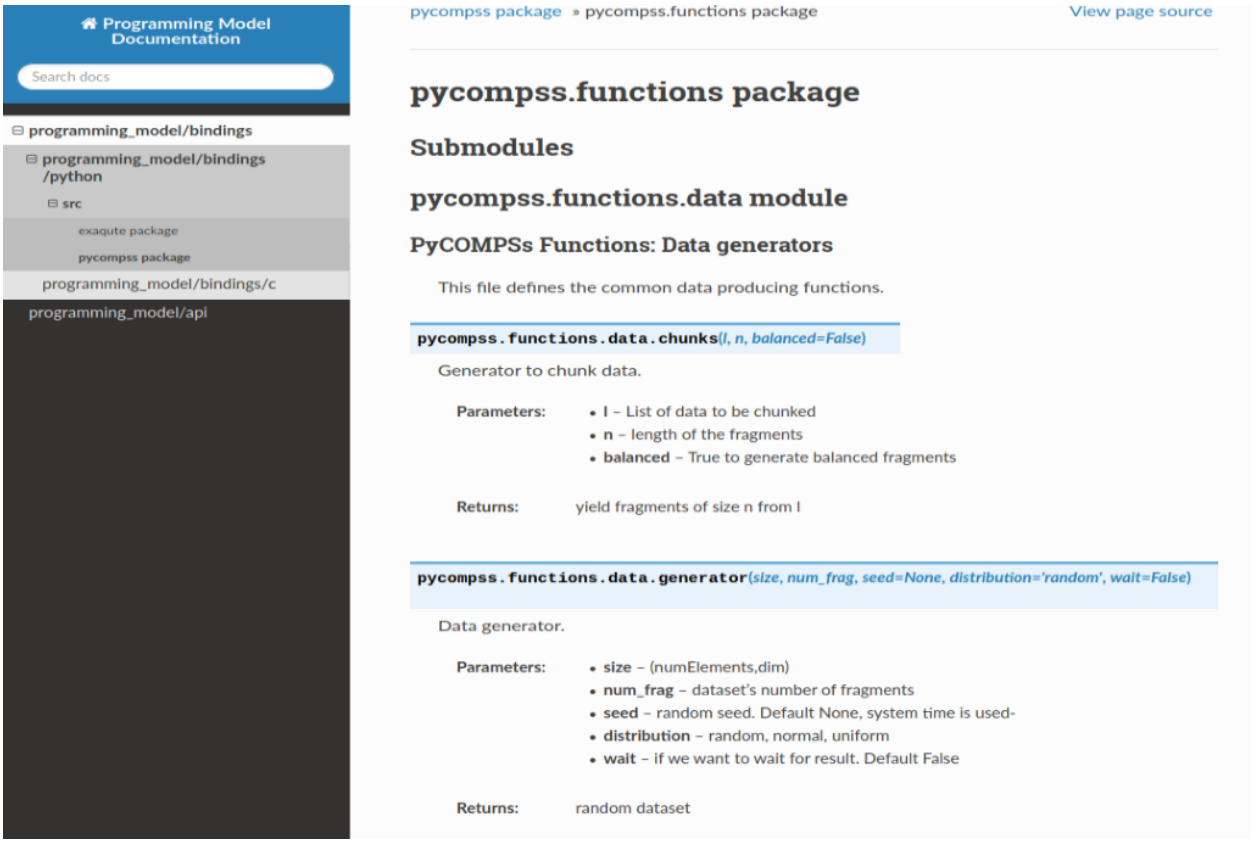


La pàgina de navegació dels *bindings:*

La pàgina del *binding* de Python:



1. I, finalment, la pàgina amb tota la documentació d'un dels mòduls del codi fet en Python:


Programming Model Documentation
Search docs
programming_model/bindings
programming_model/bindings /python
src
exaqute package
pycompss package
programming_model/bindings/c
programming_model/api
pycompss package » pycompss.functions package
View page source
pycompss.functions package
Submodules
pycompss.functions.data module
PyCOMPSs Functions: Data generators
This file defines the common data producing functions.
pycompss.functions.data.chunks(l, n, balanced=False)
Generator to chunk data.
Parameters:
l – List of data to be chunked
n – length of the fragments
balanced – True to generate balanced fragments
Returns:
yield fragments of size n from l
pycompss.functions.data.generator(size, num_frag, seed=None, distribution='random', wait=False)
Data generator.
Parameters:
size – (numElements,dim)
num_frag – dataset's number of fragments
seed – random seed. Default None, system time is used-
distribution – random, normal, uniform
wait – if we want to wait for result. Default False
Returns:
random dataset

8. Conclusions

La realització d'aquest projecte m'ha servir per ser conscient de la importància de comptar amb un sistema per saber la cobertura de codi total del software que s'està desenvolupant. Jo ja era conscient de la importància de tenir tests unitaris i d'integració, però no sabia que també era important saber el percentatge de línies totals que els nostres tests cobreixen. M'he adonat que el fet d' integrar un sistema per comprovar la cobertura de codi no comporta grans complicacions, així que en els futurs projectes que hagi de realitzar estic segur que integraré un d'aquests sistemes. És possible que no tots els projectes que realitzi siguin fets en Python o Java, però també s'ha pogut demostrar que al mercat hi ha nombroses eines, la majoria prou documentades com per poder-les integrar en els nostres projectes.

Realitzant el sistema per desplegar tests automàtics a un Supercomputador m'ha servir per adonar-me que, si tenim una tasca monòtona que hem de realitzar nombroses vegades, és molt millor estudiar si es factible desenvolupar un sistema per automatitzar-la. Executar manualment els tests a un Supercomputador era una tasca monòtona i molt poc creativa, però ara ja se'n encarrega de tot el sistema realitzat.

Finalment, el sistema per actualitzar la documentació de desenvolupador automàticament també m'ha servit per adonar-me de la importància de tenir un sistema d'aquest tipus en projectes grans o que es vagin actualitzant contínuament. Amb un simple script podem extreure tota la documentació del codi font per compartir-la o pujar-la al nostre lloc web.

Espero que aquests tres sistemes facilitin la feina als desenvolupadors, i que els puguin anar actualitzant a mesura que COMPSs vagi creixent.

9. Referencies

1. **Barcelona Supercomputing Center. [En línea] [Citado el: 20 de 2 de 2020.] https://www.bsc.es/.**

2. *COMPSs.* [En línea] [Citado el: 20 de 2 de 2020.] https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar.

3. Code coverage. *Wikipedia.* [En línea] [Citado el: 20 de 2 de 2020.] https://en.wikipedia.org/wiki/Code_coverage.

4. Software deployment. *Wikipedia.* [En línea] [Citado el: 20 de 2 de 2020.] https://en.wikipedia.org/wiki/Software_deployment.

5. Importance of Software testing. [En línea] [Citado el: 22 de 2 de 2020.] https://www.testdevlab.com/blog/2018/07/importance-of-software-testing.

6. Desarrollo ágil de software. *Wikipedia.* [En línea] [Citado el: 22 de 2 de 2020.] https://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software.

7. MareNostrum 4 User's Guide. [En línea] [Citado el: 02 de 3 de 2020.] https://www.bsc.es/support/MareNostrum4-ug.pdf.

8. Jacoco. *eclemma.* [En línea] [Citado el: 8 de 3 de 2020.] https://www.eclemma.org/jacoco/.

9. Coverage.py. *readthedocs.io.* [En línea] [Citado el: 8 de 3 de 2020.] https://coverage.readthedocs.io/en/coverage-5.1/.

10. Subprocess. *docs.python.org.* [En línea] [Citado el: 14 de 2 de 2020.] https://docs.python.org/3/library/subprocess.html.

11. Argparse. *docs.python.org.* [En línea] [Citado el: 14 de 3 de 2020.] https://docs.python.org/3/library/argparse.html.

12. Polling. *github.* [En línea] [Citado el: 20 de 3 de 2020.] https://github.com/justiniso/polling.

13. Pandas. *pydata.* [En línea] [Citado el: 20 de 3 de 2020.] https://pandas.pydata.org/.

14. Javadoc. *oracle.* [En línea] [Citado el: 6 de 4 de 2020.] https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html.

15. Docstrings. *python.* [En línea] [Citado el: 6 de 4 de 2020.] https://www.python.org/dev/peps/pep-0257/.

16. PlantUML. [En línea] [Citado el: 6 de 4 de 2020.] https://plantuml.com/.

17. devla. plantuml-generator. *github.* [En línea] [Citado el: 10 de 4 de 2020.] https://github.com/devlauer/plantuml-generator.

18. Sphinx. [En línea] [Citado el: 17 de 4 de 2020.] https://www.sphinx-doc.org/en/master/.

19. Javasphinx. *readthedocs.io.* [En línea] [Citado el: 17 de 4 de 2020.] https://bronto-javasphinx.readthedocs.io/en/latest/.

20. sphinxcontrib-plantuml. *pypi.* [En línea] [Citado el: 17 de 4 de 2020.] https://pypi.org/project/sphinxcontrib-plantuml/.

21. sphinx-rtd-theme. *readthedocs.io.* [En línea] [Citado el: 17 de 4 de 2020.] https://sphinx-rtd-theme.readthedocs.io/en/stable/.