Resiliència a nivell d’aplicació

Principis de Resiliència

La principal font de resiliència fins ara n’és, sense dubte, l’experiència. Aquesta és inherent i varia en funció de cada persona implicada en el desenvolupament del software. Un dels beneficis que dona l’experiència és la resiliència. Com ja hem mencionat, entenem per resiliència la capacitat del software a respondre en circumstàncies adverses i/o sortir d’elles de manera autònoma.

De la resiliència en l’àmbit del software no fa tant 2007[[1]](#footnote-1) que se’n parla, gairebé una dècada. En el seu llibre, *Release it*, Michael T. Nygard tracta el tema de la resiliència software des de la seva experiència. Cal dir que el terme de resiliència ha anat evolucionant. Hi ha tres termes que estan molt relacionats. Tal com es pot veure a la Figura 1. No hi ha dubte en la definició dels extrems, i sabem que la resiliència esta compresa entre els dos. Nosaltres, doncs, anomenem principi de resiliència qualsevol principi que doti el software amb capacitat de resistència, recuperació o inclús millora. Considerem com activador del principi qualsevol error; passat, present o futur.

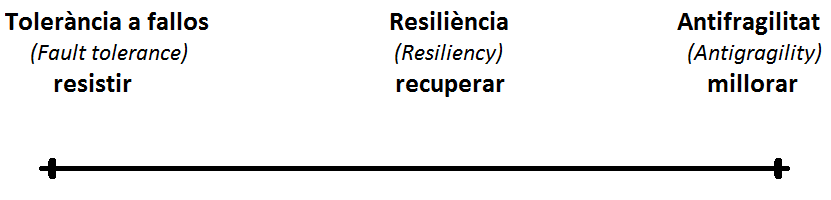


Figura 1. Mapa de conceptes

El primer és el concepte de *Fault Tolerance*, que consisteix en construir software robust. Es a dir, la fallida d’un component que no comporti la caiguda de tot el sistema. El segon concepte és el de la resiliència, tal com s’ha mencionat, és la capacitat recuperar-se. Finalment el concepte d’*Anti Fragility* que és la capacitat d’un sistema no només de tornar a l’estat normal sinó avançar cap a un estat millor.

Nosaltres entenem per aplicació resilient aquella que segueix donant servei a un determinat nivell de qualitat després d’haver patit errors i és capaç de tornar a l’estat normal recuperant la totalitat del servei. En el cas ideal aquesta recuperació és transparent de cara a l’usuari. En els altres casos s’informa l’usuari que la funcionalitat no esta disponible temporalment i es torna a informar quan s’hagi efectuat la recuperació.

# Release it

Ara analitzarem alguns dels patrons que Michael T. Nygard proposa en el seu llibre per aconseguir aplicacions resilient o més resilient. Alguns d’aquests patrons ja venen incorporats als *bons-costums* i/o a *frameworks* corresponents. Des del principi deixa clar que la principal motivació en construir software resilient, finalment arriba a ser la econòmica. El subtítol de la portada ho indica: “*Design and Deploy Production-Ready Doftware.* Segons sosté, una decisió de disseny és una decisió econòmica; i qualsevol *estalvi* que es vulgui fer en aquesta fase tindrà repercussions *cares* en producció. Per tant, la programació ha de ser pragmàtica, orientada a l’entorn de producció, no a l’entorn de proves o QA.

Encara que és difícil trobar-se dos vegades amb el mateix problema, tard o d’hora surten els anti-patrons. Són aquelles situacions sistemàtiques que porten a errors, i per tant es poden aplicar solucions generals. Enfocat en quatre grans temes el llibre estudia en profunditat els problemes que provoquen els anti patrons i els patrons corresponents a les solucions. Els primers dos grans temes compactes són l’estabilitat i la capacitat. Els darrers dos són principis generals de disseny i operacions. Encara i estant enfocat només en l’entorn de producció el llibre aconsegueix donar una visió prou completa del patrons que es podrien aplicar per aconseguir software resilient.

## Estabilitat

Com ja havíem dit el primer tema que tracta és l’estabilitat. El software resilient ha de ser estable. Un error d’una certa funcionalitat no pot ser que ens faci caure tot el sistema, deixant-nos sense poder fer res més abans de reiniciar l’aplicació o el servidor. L’autor identifica una gran varietat d’elements com a anti patrons en aquest tema. Aquests són: els punts d’integració, les reaccions en cadena, cascades d’errors, els usuaris, *threads* bloquejats, atacs d’auto denegació de servei, efectes d’escalat, capacitats no balancejades, respostes lents, SLA, respostes no determinades.

### Anti patrons

Els punts d’integració es van multiplicant conforme el sistema d’informació d’una organització va creixent. Cada cop hi ha més fonts i consumidors d’informació que es necessiten integrar, necessiten interaccionar. Per exemple, CRM, ERP, MRP, BPO entre d’altres. Per tant cada socket, procés, pipe o crida remota pot i arribarà a penjar-se.

Les reaccions en cadena tenen que veure amb temes d’escalabilitat a nivell horitzontal. La figura 1 mostra una granja amb vuit servidors darrere un balancejador de carrega.



Figura 1. Exemple d’escalabilitat horitzontal.

El problema apareix en cas de caiguda d’un servidor, els que queden s’han de repartir entre tots la seva feina. Depenent del tipus, l’error podria provocar la caiguda d’una altre servidor, fins arribar a caure tot el sistema.

Les cascades d’errors són semblants a les reaccions en cadena però a nivell de capes. Si els errors es propaguen d’una capa a un altra.

El comportament dels usuaris, tant de manera individual com general és prou demandant per no dir totalment imprevisible. A més el sistema escala en funció del hardware contractat i no en funció de la quantitat d’usuaris[[2]](#footnote-2). Per tant la pregunta és com reacciona el sistema quan la demanda supera la seva capacitat per respondre?

Els threads bloquejats apareixen a l’hora d’explotar el paral·lelisme de les CPU’s. El multithreading és complex i normalment no és factible provar l’aplicació amb un nombre suficientment alt de requests. Per tant son problemes que difícilment surten abans d’entrar en producció.

Els atacs d’auto denegació de servei: *self-denial attack* apareix quan el sistema com un tot, inclús els humans “conspiren” en contra d’ell mateix. Per exemple una campanya de màrqueting que atreu molts més clients dels que el sistema esta preparat per rebre.

Les capacitats no balancejades tenen a veure amb el gestor d’escalat i les diferències entre recursos frontend versus backend.

Les respostes lents apareixen normalment quan el sistema ja esta en un nivell de demanda excessiu, per culpa del *garbage collector* o *memory leaks*.

El service-level agreement és el contracte que regula les condicions de servei. També conté les clàusules de penalitzacions econòmiques en cas que el servei no compleix les solucions. El problema és que un sistema no pot tenir un SLA millor que el de la pitjor de les seves dependències.

S’ha de dissenyar amb escepticisme. En molts casos una aplicació tracta la seva base de dades amb massa confiança. Qualsevol dependència pot en un moment donat retornar una resposta no esperada. Per exemple la base de dades podria respondre amb un resultat considerablement més gran que normalment. Si l’aplicació no limita la quantitat d’informació que esta disposada a processar poden passar coses no desitjades, el temps que triga és massa i l’usuari perd l’interès, desbordaments de memòria, etc.

### Patrons

Per prevenir els escenaris problemàtics, en quant a l’estabilitat del sistema, enumerats més a dalt, Michael Nygard proposa vuit patrons. Com ja hem mencionats alguns ja estan implementats per les llibreries que és veuen actuant en dites circumstàncies. Per exemple, el primer patró és el timeout. Avui dia aquest principi ja està implementat en les llibreries, encara i així s’ha de ser conscient i configurar-ho pròpiament.

El següent patro s’anumena circuit breaker. Consisteix en monitoritzar el timeout i obrir el circuit si aquest salta molt sovint. Per tant, si el circuit està obert, ja sabem que no aconseguirem resposta, podem respondre que molt ràpidament. Un procés addicional es necessari en aquest cas per comprovar si el servei torna a estar disponible. De manera automàtica, l’aplicació pot detectar això i tancar el circuit tornant a l’estat normal.

Els *bulkheads* o mampares (veure Figura 3), separen l’espai una embarcació en compartiments. En cas de produir-se forats, el compartiment afectat es pot tancar i contenir la propagació de l’aigua a la resta del vaixell. Seguint aquest exemple l’aplicació hauria d’estar dividida en particions que no deixin propagar els a traves de les mampares.



Figura 2. Mampares d’una embarcació.

Com a exemple tenim a Baz com a dependència de Foo i de Bar. Per exemple un manteniment Baz seria impossible de realitzar degut a la impossibilitat de respectar els SLA de Foo i SLA de Bar a l’hora. En aquest cas, Baz hauria d’estar compartimentat protegint els clients. Evidentment s’ha d’estudiar bé la la mida dels compartiments, des les thread pools fins als servidors en un claster.



Figura 3. Aplicació del principi de mampares

El Steady-state és l’estat normal de l’aplicació. Aquest s’hauria de mantenir per si mateix sense necessitat d’intervenció humana diària. Pels problemes d’espai dels logs o neteja de la base de dades s’haurien de fer scripts que s’executin automàticament. Un altre aspecte a considerar per garantir un estat òptim de l’aplicació consisteix en controlar la memòria que la cache pot ocupar. Per últim els logs. Aquests, si s’han de conservar per llei és recomana no mantenir-los en servidors de l’entorn de producció.

Si una resposta lenta és pitjor que no donar cap resposta, llavors una resposta lenta i errònia o negativa és encara pitjor. Aquest patró proposa vigilar les fonts probables d’errors i avançar-se amb la resposta en cas que és pugui determinar que fallarà. Això no sempre es pot determinar, però si és el cas, no només estalvia temps de l’usuari sino recursos del sistema. Per tant abans de fer qualsevol crida, s’hauria de comprovar tot el que es pugui abans de fer-la. En primer lloc validar l’input i en segon comprovar, per exemple si el circuit breaker corresponent està tancat.

Les comunicacions són potencials fonts d’errors que s’han de tractar i protegir. La manera que proposa Michael Nygard és mitjançant el protocol de *handshaking*. Quan això no és possible, s’haurien de fer comprovacions d’estat: *Health-checks*, en cas que fer la comprovació sigui menys costosa que una crida que falla. També és recomanable utilitzar el handshaking per qualsevol protocol propi de baix nivell, per exemple a nivell de socket.

Test Harness representa un enfocament de desconfiança total amb respecte qualsevol dependència. Temps, format, contingut, mida de la resposta, o inclús el protocol de comunicació poden sortir del que s’havia especificat. Com tard o d’hora algun d’aquests problemes passaran s’ha d’estar preparat. Les proves del software ha d’incloure escenaris com els mencionats, i més.

Finalment un ben conegut patró de disseny: el baix acoblament, en aquest cas pel *middleware.* Aquell espai amb un desordre singular que permet la comunicació de sistemes que no s’havien dissenyat per treballar en conjunt. La Figura 5 mostra l’espectre d’acoblament pel middleware.



Figura 5. Els nivells d’acoblament i implementacions.

## Capacitat

La capacitat d’un sistema és el rendiment(*troughput*) màxim sostenible pel sistema amb un temps acceptable de resposta per cada petició. La capacitat d’un sistema es defineix en funció de tres conceptes: velocitat per petició, rendiment en quant a numero de peticions processades per unitat de temps i l’escalabilitat. En aquest cas s’entén per escalabilitat incrementar la capacitat. Un greu problema que apareix en la anàlisis de la capacitat és la falta de linearitat. Per exemple, si un sistema pot donar suport a 10.000 usuaris utilitzant un 50% de la CPU, és fals deduir que el sistema hauria de suportar 20.000 en total.

Segons l’autor hi ha una sèrie de problemes o circumstancies que amenacen la capacitat d’un sistema. Aquestes són: Resource Pool Contention, AJAX Overkill, Overstaying Sessions, Wasted Space in HTML, el botó de recarrega, Handcrafted SQL, Integration Point Latency.

### Anti patrons

Quan el *pool* de threads actius que demanen accés a la base de dades supera el numero de connexions disponibles apareix el problema de Resource Pool Contention. El coll d’ampolla del sistema és la limitació del numero de recursos disponibles, ja que normalment les pool de connexions bloquegen indefinidament els threads en cas que no es puguin servir. Aquest és clarament un problema que amenaça la capacitat.

L’ús en excés de l’AJAX es pot convertir en un problema. Podria fer veure que el navegador esta penjat, l’increment de comunicació per la xarxa podria disparar-se i suposar una carrega massa gran i innecessària tan pel servidor com per l’aplicació.

La gestió del temps de caducitat de les sessions d’usuari és un altre factor important en qüestió de capacitat. Un fet curiós és que els usuaris que és recursos necessiten son els usuaris mes desitjats pel negoci. Però la memòria del servidor és escassa.

Sempre que no és te cura de la mida del HTML, aquest acaba sent una mica més gran. El problema apareix per dos raons. Primer l’increment innecessari de la memòria dels servidors web, i segon l’ús de l’ample de banda addicional inútilment. Un altre efecte que això provoca té com a actors els navegadors. Aquests, en funció de la mida del HTML mantenen una connexió durant més o menys temps.

El botó de recarrega del navegador està en mans de l’usuari, i això no són bones noticies. Cada cop que és prem el navegador abandona la petició anterior, obre un socket i fa una petició nova. El problema es que ningú mata la petició anterior, el servidor no sap que la pot descartar.

Combinar el servei d’ORMs amb consultes fetes a ma, encara que prometen eficiència són molt impredictibles. Tota la configuració de la base de dades esta preparada per les consultes dels ORM. L’autor dona diverses raons per no executar consultes fetes a ma, o minimitzar el seu ús.

Qualsevol comunicació remota comporta una certa latència, que normalment és 1000 vegades més gran que una crida local. Encara que aquest és un problema d’eficiència per l’usuari, pel sistema acabarà sent un problema de capacitat.

### Patrons

Per evitar problemes de Resource Pool Contention s’han de configurar adequadament les Pool Connections. Aquestes poden a més d’evitar alentir tot el sistema, millorar la capacitat. A part, s’han de protegir tots els threads que demanin connexió a la base de dades.

Una bona implementació de cache pot reduir problemes de rendiment. Redueix la carrega del servidor de la base de dades. Però s’ha de verificar que aquest és el cas, s’ha de mesurar la tassa d’encerts i la freqüència d’ús dels contingut. En la configuració de la cache s’ha de tenir en compte el límit d’espai que pot ocupar la cache i implementar un bon mecanisme de flush.

En el mon web cada cop es requereix contingut dinàmic i específic. Portat a l’extrem arribem a trobar parts molt estàtiques a dins. Les parts estàtiques es poden pre-calcular.

Per últim, dins del tema de la capacitat, està el *Garbage Collector*. És la manera més rapida i fàcil per millorar la capacitat en aplicacions Java. Cal, doncs, analitzar el comportament del Garbage Collector, l’ús del *heap* i el temps que es triga per treure la brossa, i ajustar la mida del heap. Configurar-lo bé porta avantatges de capacitat a més, té la capacitat de descobrir *memory leaks.*

Com a resum esquemàtic dels principis de resiliència que proposa el llibre tenim la Figura 3. Aquesta mostra les interaccions de patrons i anti-patrons. Els quadrats representen els patrons i els ovals els anti-patrons.



Figura 3. Interacció entre patrons i antipatrons

El llibre esta enfocat en trobar els anti-patrons i proposar patrons per solucionar la varietat d’errors que provoquen els primers.

# Patterns of resilience

Uwe Frederichson té una sèrie de presentacions sobre la resiliència que són més recents[[3]](#footnote-3), però en gran part recupera els principis que proposa Michael Nygard. També menciona i recomana el seu llibre, *Release it* que encara es considera com la bíblia de la resiliència. Segueix el mateix enfocament a producció que trobem en el llibre però no es proposa estudiar els problemes sinó tracta directament els principis de resiliència. Aquests s’expliquen fins a arribar donar el codi. El resum visual dels patrons proposats per l’autor es pot veure a la Figura 14.



Figura 14. Patrons de resiliència i relacions.

# Resilience is by design

# Principis proposats

# Comentaris finals

Moltes de les propostes que s’han fet, començat pel llibre, estan enfocats a una resiliència que s’aconsegueix en la fase del disseny. Nosaltres però pensem que la resiliència ha de se transversal i inclús més enllà. Considerem que el pensament empàtic cap a l’usuari en qualsevol fase és la forma més adequada de generar principis de resiliència. Amb usuari, no ens referim simplement a l’usuari final, el mateix desenvolupador és un usuari de l’aplicació.

Encara que el cas ideal és fer resiliència a partir del disseny, la resiliència també pot ser afegida com una capa addicional a qualsevol aplicació. No tots els principis de resiliència requereixen el mateix nivell d’intrusisme, per tant, és possible implementar-la i afegir-la a posteriori al software.

Els principis que el present treball ha implementat estan explicats a continuació. (if necessary...)

1. Considerem el llibre Release it! com el primer en el que no només es parla sino que es tracta la resiliència software àmpliament i en detall. [↑](#footnote-ref-1)
2. Aquest és un exemple d’argument antiquat ja que des del segon trimestre del 2008 han començat a aparèixer serveis de host que proporcionaven un escalat en funció del nombre d’usuaris. [↑](#footnote-ref-2)
3. De l’any 2015. [↑](#footnote-ref-3)