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]](#endnote-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. 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]](#endnote-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 és encara pitjor. Aquest patró proposa vigilar les fonts probables d’errors i avançar-se amb la resposta en cas que la crida és determina 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.

.... Handshaking, Test Harness & decoupling midleware....

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]](#endnote-3), però recupera molts dels principis que proposa Michael Nygard. També menciona i recomana el llibre que encara es considera com la bíblia de la resiliència.

# Resilience is by design

# Principis proposats

# Comentaris finals

Resum final – criticar totes les fonts.

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

I will probably mention akka,

1. Considerem el llibre Release it! com el primer en el que no només es parla sino que es tracta la resiliència software d’una manera clara i contundent. [↑](#endnote-ref-1)
2. Aquest és un 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. [↑](#endnote-ref-2)
3. De l’any 2015 [↑](#endnote-ref-3)