THINK SYSTEM SOFTWARE ARCHITECTURES FOR FUTURE EASY HARDWARE SCALING

Author: Petre IORDANESCU, Date: December 2018
Categories: Software Design & Architectures

MOTIVATIE

De-a lungul timpului am observat faptul ca arhitecturile sistemelor, mai ales in sisteme ce implica si componente software (sincer nu am vazut pina acum unul fara software in industria IT), sunt tratate relativ cu simplitate si / sau indiferenta. Si aici ma rfer la faptul ca NU TOTI arhitecții si designerii de sisteme fac asa, dar cei mai multi FAC ASA... Oamenii focalizează pe partea de software development, pe termene, in general din start pe detalii iin detrimentul calității. **UN SISTEM CU UN DESIGN BINE FACUT aduce multiple calitati acestuia cit si procesului de realizare si întreținere ulterioara.**

Acest articol isi propune sa faca unele clarificări in ceea ce privește arhitectura si designul sistemelor prin prisma necesitatii scalarii hardware ulterioara a acestuia pentru a il menține in parametri acceptabili de performanta.

INTRODUCERE IN SCALAREA HARDWARE

Scalarea hardware inseamna pe scurt creșterea puterii de calcul brute (doar prin hardware fara modificări in software) si acest lucru se poate face in doua moduri ce nu se exclud intre ele:

- (**M1**) suplimentarea resurselor hardware ale fiecărei masini din sistem; acest lucru are efect NUMAI DACA SISTEMUL DE OPERARE „ȘTIE" SA FOLOSEASCA SI SA ALOCE NOILE RESURSE;
- (**M2**)s suplimentarea masinilor alocate sistemului fara a umbla in cele existente; acest lucru are efect NUMAI DACA SOFTWARE-UL PROIECTAT „ȘTIE" SA FOLOSEASCA NOILE RESURSE SA PARTI DIN EL POT FI MUTATETE PE NOILE MASINI.

Sa analizam avantajele fiecărei metode:

Criteriu	M1	M2
Cost	++	+
Down time	++	0
Relevanta sistemului de operare	++++	+
Timp reconfigurare	+++	++
Disponibilitate componente compatibile	++++	0
Efort design sistem	+	++++
Efort dezvoltare sistem	+	++
Upgrade sistem	+++	+
Întreținere sistem	+++	+
„Paralelizare" dezvoltare	0	++
Management dezvoltare sistem	++++	++

Din analiza tabelului de mai sus se observa vadit clar ca metoda M2 este de preferat metodei M1. Prezentul articol va focaliza mai departe pe cai de a „produce" un design si o arhitectura care sa faciliteze aplicarea metodei M2 cit mai usor cu putinta.

TEHNICI SI METODE

Principalele tehnici in atingerea scopului propus sunt:

- Modularizarea sistemului
- Componente „restless"
- Parametrizarea sistemului
- Utilizarea de standarde recunoscute, deschise si accesibile

Tehnicile enumerate mai sunt genera valabile in proiectarea sistemelor software, însă pentru a obtine un sistem care sa răspundă cit mai mult la (M2) trebuie sa fie aplicate într-o serie de direcții si sa urmărească o serie de recomandări specifice ce vor fi expuse in continuare.

De asemenea este important de menționat faptul ca tehnicile enumerate trebuiesc folosite TOATE si chiar daca sunt tratate individual (ceea ce este recomandabil pentru a evita „pierderea in detalii") ele se întrepătrund si fiecare componentă a sistemului va trebui sa „aiba o reprezentare" mai mare sau mai mica in fiecare dintre ele.

MODULARIZAREA SISTEMULUI

Modularizarea (in directia M2) are ca scop obtinerea de componente software care sa aiba functionalitati cu finalitate relevanta si utilizabila in mod independent de modul de functionare al celorlalte componente. Aceste componente trebuie sa fie cit mai „mici" (corect este termenul „light-weight" din engleza) astfel incit sa necesite un „foot-print" care sa le permita rularea pe un singur server.

Cu alte cuvinte se va urmări obtinerea de componente care se pot constitui in MICRO-KERNELE cu un grad de independenta cit mai ridicat.

O metoda des practicată DAR EXTREM DE DAUNATOARE este de a „se înghesui cit mai mult într-o componenta" generând astfel sisteme monolit, exact la capătul opus de ceea ce se dorește a se obtine.

Comunicarea intre aceste componente trebuie sa se faca utilizând canale uzuale si existente in SO-urile curente de tip server (Linux, Unix, Windows). Exemple de astfel de canale sunt: socket-uri web, named pipes, brokere si cozi de mesaje 3rd party – AMPQ, REDIS, RABBIT-MQ, etc. Aceste canale pot fi usor securizate prin metode ultra-cunoscute fara a necesita nici un fel dezvoltare specifica sau recompilări / reinstalări ci doar prin configurări.

ATENTIE: nu ne oprește nimic ca descompunerea in componente sa fie facuta cit mai detaliat / adinc si chiar ESTE RECOMANDABIL sa se întâmple acest lucru. Însă mai sus ne refeream la componentele „mari" relevante la nivel arhitectural; modulele din fiecare din aceasta sunt relevante la nivel de software design si intern (adica nu pentru end-user) in dezvoltarea sistemului.

COMPONENTE „RESTLESS"

Componentele sistemului vor trebui gândite astfel incit sa gestioneze doar **STARILE PROPRII** in sensul de memorare / reținere a acestora (vezi si despre arhitecturi REST[1]). Componentele nu trebuie sa faca presupuneri sau sa memoreze stările altor componente – principiul „client / server[2], request / response" trebuie avut in vedere permanent in proiectare.

DE CE? In perioada actuala nu mai este nimic nou sau special in a ‚găsi computere la tot pasul' si a folosi masiv comunicațiile intre locații indiferent de distanța fizica intre ele. Memorarea stărilor presupune identificarea fara echivoc a proprietarului acestora, numai ca este extrem de dificil si de multe ori aproape imposibil de facut acest lucru. Cine este proprietarul unei stări? Un identificator de utilizator? Acelasi utilizator se poate conecta simultan de la mai multe echipamente... Identificatori din rețea (gen IP sau MAC)? Acestea nu sunt garantate a fi stabile si se pot schimba fara probleme si fara știrea nimănui (ginditi-va de exemplu la o adresa IP alocata automat prin DHCP). MAC? A trecut vremea cind era „arsa intr-un ROM pe placa de rețea. Un exemplu foarte la indamină si ce poate fi usor experimentat este o masina virtuala (VM) căreia i se poate schimba rapid adresa MAC...[3]

Concluzia noastra este ca o decizie înțeleaptă este sa gândim „restless".

[1] https://en.wikipedia.org/wiki/Representational_state_transfer
[2] https://en.wikipedia.org/wiki/Representational_state_transfer
[3] Pentru cei interesați de detalii privind acest aspect se poate incepe cu acest articol: https://en.wikipedia.org/wiki/Session_(computer_science)

PARAMETRIZAREA SISTEMULUI

Parametrizarea sistemului in sensul prezentului articol se refera la posibilitatea configurării „legăturilor” unei componente cu celelalte din fisiere externe (stocate extern si nu in cod) sistemului cit mai accesibile cu instrumente foarte banale. Cel mai simple este ca diversele configurări sa fie stocate si regăsibile in fisiere text standard (plain text). Convențiile de structurare a informației de configurare trebuie sa fie aliniate la practicile curente gen: model JSON, XML, INI, YAML, etc.

Parametrii vitali si care tin de masina pe care este / va fi instalata componenta, TREBUIE sa fie configurabili in acest fel, chiar daca uneori este necesara re-boot-area masinii (preferabil in loc de modificări direct in cod cu potentiale (d)efecte greu de cuantificat). Exemple de astfel de parametrii sun: adresa IP, portul(rile) cu care software-ul lucrează, protocoale, adresele altor componente accesate – de exemplu baza de date etc

UTILIZAREA DE STANDARDE

Utilizarea standardelor va permite implementarea de politici de securitate si interoperabilitate fara recompilarea codului sa alte modificări in cod, intr-un mod cu totul transparent si ne-necesar in etapa de dezvoltare a sistemului. Si aici ne referim la genul de astfel de politici pe care diversi utilizatori vor si isi permit sa le implementeze: load balancing, baza de date in mod „redundancy fail safe”, utilizarea protocoalelor SSL (ex https), etc.

Pentru a fi eficiente in acest sens, standardele trebuie sa fie:

- (re)cunoscute,
- deschise si
- accesibile

Petre IORDANESCU, 2018, decembrie