

Universitatea “Politehnica” din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

**Studiul bazelor de date distribuite în sistemul de gestiune al bazelor de
date MariaDB**

Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de

Inginer în domeniul Calculatoare și Tehnologia

Informației

programul de studii de licență Ingineria

Informației (CTI – INF)

Conducător științific

Ș.l. Dr. Ing. Valentin PUPEZESCU

Absolvent


Bogdan - Petru MATRAGOCIU

Anul

2014

Universitatea "Politehnica" din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației
Departamentul Electronică Aplicată și Ingineria Informației

Aprobat Director de Departament :


Prof. Dr. Ing. Sever Pașca

TEMA PROIECTULUI DE DIPLOMĂ

a studentului Matragociu I. Bogdan-Petru, grupa 442A

1. Titlul temei: **Studiul bazelor de date distribuite implementate în sistemul de gestiune al bazelor de date MariaDB.**

2. Contribuția practică, originală a studentului va consta în:

Implementarea unei aplicații web medicale de gestionare a articolelor, știrilor și evenimentelor din domeniul stomatologic. Aceasta folosește baze de date distribuite în sistemul de gestiune de baze de date MariaDB. Se vor studia avantajele folosirii acestui tip de sistem de gestiune al bazelor de date față de alte sisteme (relaționale și NoSQL), precum și implementarea diverselor topologii de replicare disponibile în MariaDB.

3. Proiectul se bazează pe cunoștințe dobândite în principal la următoarele discipline: Programare Obiect-Orientată, Tehnologii de Programare în Internet, Proiectarea Bazelor de Date

4. Realizarea practică/ proiectul rămân în proprietatea: *UPB*

5. Proprietatea intelectuală asupra proiectului aparține: *UPB*

6. Locul de desfășurare a activității: *UPB, ETTI*

7. Data eliberării temei: 1 oct. 2013.

CONDUCĂTOR LUCRARE:

Ș.L. Dr. Ing. Pupezescu Valentin



STUDENT:

Matragociu I. Bogdan-Petru



Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul "*Studiul bazelor de date distribuite implementate în sistemul de gestiune al bazelor de date MariaDB*", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Calculatoare și Tehnologia Informației*, programul de studii *Ingineria Informației (CTI – INF)* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurărilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, 1.07.2014

Absolvent *Bogdan-Petru MATRAGOCIU*



Cuprins

1. Introducere, scopul lucrării
2. Baze de date
 - 2.1 Noțiuni generale
 - 2.2 SQL vs. NoSQL
3. MariaDB
 - 3.1 Introducere
 - 3.2 Replicarea
 - 3.2.1 Introducere
 - 3.2.2 Setarea replicării
 - 3.2.3 Implementarea replicării
 - 3.2.4 Topologii de replicare
 - 3.2.5 Administrare și mentenanță
4. Tehnologii utilizate
 - 4.1 HTML
 - 4.2 CSS
 - 4.3 JavaScript
 - 4.4 jQuery
 - 4.5 Java
 - 4.5.1 Introducere
 - 4.5.2 Sintaxa Java
 - 4.5.3 OOP
 - 4.5.4 Integrarea cu baza de date
5. Proiectarea aplicației
 - 5.1 Prezentare generală a interfeței
 - 5.1.1 Secțiunea de autentificare
 - 5.1.2 Secțiunea de știri și articole
 - 5.1.3 Secțiunea evenimente
 - 5.1.4 Secțiunea de întrebări și răspunsuri
 - 5.1.5 Secțiunea de administrator
 - 5.2 Structura bazei de date
 - 5.3 Arhitectura de replicare implementată
 - 5.4 Testarea replicării
6. Concluzii
7. Bibliografie
8. Anexe

Lista acronimelor

SDK - Software Development Kit (Platformă de dezvoltare software)

SQL - Structured Query Language (Limbaj de interogare structurat)

PHP - PHP: Hypertext Preprocessor (Limbaj preprocesare hypertext)

HTML - HyperText Markup Language (Limbaj de marcaje hypertext)

CSS - Cascading Style Sheet

SGBD - Sistem de Gestiune a Bazelor de Date

VARCHAR - Variable Length Character String (Șir de caractere cu lungime variabilă)

BLOB - Binary Large Object

XML - eXtensible Markup Language

JSON - JavaScript Object Notation

RDBMS - Relational DataBase Management System (Sistem de management al bazelor de date)

BSON - Binary Script Object Notation

UTF-8 - Universal Character Set Transformation Format - 8 bit

DB - DataBase

PNG - Portable Network Graphics

POO/OOP - Object Oriented Programming (Programare Obiect Orientată)

DML - Data Manipulation Language

DDL - Data Definition Language

JSP - Java Server Pages

URL - Uniform Resource Locator

DOM - Document Object Model

AJAX - Asynchronous JavaScript And XML

UI - User Interface (Interfața cu utilizatorul)

JPA - Java Persistence API

1. Introducere, scopul lucrării

În ultimul deceniu internetul și aplicațiile web au cunoscut o creștere exponențială. Aplicațiile clasice ce necesită descărcarea și instalarea pe calculatorul propriu au scăzut ca număr, implementările lor trecând în cloud. Tendința este clară de a oferi servicii utilizatorului în interiorul unui browser sau ca aplicație mobilă.

Aproape orice aplicație folosește o bază de date. Deși orice programator poate fi capabil să își definească propriul mod de organizare a datelor, bazele de date îi ușurează munca oferind un mod structurat, testat și tolerant la defecte pentru a stoca datele necesare aplicației. Tipurile de baze de date sunt numeroase și prin urmare folosirea celei potrivite pentru aplicația ce se vrea dezvoltată este o decizie ce trebuie luată cu grijă și simț de răspundere. În momentul de față există două mari tipuri de baze de date ce pot fi folosite, cele clasice SQL și sistemul mai nou apărut, NoSQL.

Lucrarea de față își propune prezentarea unei stive de tehnologii folosite pentru dezvoltarea unei aplicații web pentru gestionarea de articole, știri, evenimente și întrebări legate de domeniul medical și stomatologic. Tehnologiile folosite sunt:

- MariaDB - v. 10.0
- Java - v. 1.7
- HTML - v. 4.0/v. 5.0
- CSS - v. 2.0/v.3.0
- Bootstrap - v3.2.0
- jQuery (JavaScript) - v. 1.9

Instrumente folosite:

- Maven 3.0.5
- Git 1.8.4.msysgit.0
- BitBucket
- IntelliJ IDEA 135.908 EAP
- MySQL Workbench 6.0 CE
- Windows Azure
- Ubuntu Server 14.04 LTS
- Tomcat v. 7.0

Accentul va fi pus pe nivelul bazei de date și posibilitățile de evitare pierderii datelor și a distribuirii încărcării. Se va insista pe modalitățile de replicare, avantajele și dezavantajele acestora în sistemul de gestiune al bazelor de date MariaDB, acest sistem fiind la granița dintre SQL și NoSQL.

2. Baze de date

Bazele de date uneori numite și bănci de date sunt modalități de stocare a informației și a datelor pe un suport extern (dispozitiv de stocare), uneori și în memorie pentru acces mai rapid. Ele oferă posibilitatea extinderii ușoare și a regăsirii rapide a datelor. Deși aceste funcții pot părea banale, atunci când se lucrează cu milioane de elemente ce trebuie accesate simultan prin internet de către mii de utilizatori răspândiți pe tot globul și disponibilitatea aplicației trebuie să fie permanentă soluțiile cele mai bune nu sunt deloc simple.

De obicei o bază de date este memorată în unul sau mai multe fișiere ce sunt manipulate cu ajutorul sistemelor de gestiune a bazelor de date. Cel mai răspândit este modelul relațional în care datele sunt memorate în tabele. În afara de tabele o bază de date relațională mai poate conține indecși, proceduri stocate, declanșatori, utilizatori și grupuri de utilizatori, tipuri de date, mecanisme de securitate și gestiune a tranzacțiilor.

Alte tipuri de baze de date sunt cele NoSQL în care datele sunt salvate în format obiectual BSON (Binary JSON – JavaScript Object Notation) sau uneori XML.

2.1 Noțiuni generale

Un server de baze de date este un program ce poate stoca o cantitate mare de date și este disponibil, accesibil prin limbaje de programare cum ar fi Java sau PHP. Serverul, în cazul nostru MariaDB, poate fi folosit atât pe Windows cât și platforme Unix. Nici un sistem de baze de date nu este perfect dar trebuie să fie ales potrivit nevoilor aplicației.

2.2 SQL vs. NoSQL

Modelul relațional

Modelul relațional (SQL) introdus în anii 70 oferă un model matematic ușor adaptabil pentru a structura, salva și utiliza date. Datele sunt salvate în tabele și sunt definite relații între ele. Deoarece există de foarte mult timp pe piață, programatorii și administratorii de baze de date sunt foarte familiari cu ele. Chiar dacă impun destule condiții și constrângeri asupra datelor ele pot deveni flexibile ușor cu un minim de efort din partea utilizatorului.

Cele mai populare sisteme sunt:

- SQLite
- MySQL
- PostgreSQL

Model-less (Modelul NoSQL)

Acest model presupune eliminarea constrângerilor impuse de primul model păstrând capabilitățile de stocare, acces și utilizare a informației. Spre deosebire de sistemul relațional datele sunt păstrate de

obicei ca documente, obiecte. Un dezavantaj al lor este faptul ca fiecare tip de sistem folosește propriul mod de interogare spre deosebire de bazele relaționale care au un limbaj comun de extragere a datelor.

Cele mai populare sunt:

- MongoDB
- Apache Cassandra
- Couchbase
- Redis
- Neo4j

Diferențe notabile

Structura în care sunt ținute datele – bazele de date relaționale necesită o structura cu attribute bine definite pentru păstrarea datelor spre deosebire de NoSQL care oferă o structură liberă în funcție de necesități.

Interogarea – Indiferent de ce sistem de baze de date relaționale este folosit el implementează pana la un anumit nivel standardul SQL. Bazele de date NoSQL implementează un mod unic de interogare în funcție de tipul lor.

Scalabilitatea – Ambele sisteme sunt ușor scalabile prin creșterea performanțelor sistemului (scalare pe direcție verticală). Însă scalarea pe direcție orizontală (crearea unui cluster ce conține mai multe mașini) este mai ușor de implementat în sistemele NoSQL.

Siguranță – din acest punct de vedere bazele de date SQL oferă siguranță mai are a tranzacțiilor efectuate și integritate a datelor.

Suport și mentenanță – bazele de date relaționale având o istorie mai mare în spate și fiind foarte populare pot oferi de multe ori un suport mai bun. Dacă apar probleme ele pot fi ușor rezolvate spre deosebire de probleme care apar în NoSQL ce pot genera soluționări complexe.

Interogări complexe – bazele de date relaționale oferă disponibilitate mai mare în ceea ce privește interogările complexe.

3. MariaDB

3.1 Introducere

MariaDB este o ramură de dezvoltare desprinsă din MySQL care își menține statutul de software open source (cu sursa deschisă – licență GNU GPL). Este notabil faptul ca este un proiect condus de fondatorii MySQL după ce Oracle a cumpărat sistemul de gestiune de baze de date numărul unu în lume MySQL.

Scopul este de a menține o compatibilitate cât mai ridicată cu MySQL asigurând capabilitatea de a schimba sistemul de gestiune de baze de date de la MySQL la MariaDB fără efecte secundare (într-un singur pas), dar oferind performanțe mai bune și opțiuni mai numeroase de configurare.

Proiectul este susținut de Michael "Monty" Widenius, fondator al MySQL, cel care a dat numele sistemului bazei de date după prenumele celei mai mici fiice ale sale.

MariaDB oferă mai multe motoarele de stocare, pe lângă MyISAM, BLACKHOLE, CSV, MEMORY, ARCHIVE și MERGE disponibile în MySQL:

- Aria
- XtraDB
- FederatedX
- SphinxSE
- TokuDB
- Cassandra

O funcționalitate ce aduce MariaDB mai aproape de bazele de date NoSQL este posibilitatea de a avea coloane dinamice (dynamic columns). În exemplul următor este creat un astfel de tabel și sunt inserate valori.

create table Bunuri (

nume_articol varchar(32) primary key, -- *Atribute comune*

coloane_dinamice blob -- *Coloane dinamice*

);

INSERT INTO Bunuri VALUES

('Tricou MariaDB', COLUMN_CREATE('culoare', 'albastru', 'marime', 'XL')); -- *Inserarea de valori dinamice*

SELECT nume_articol, COLUMN_GET(coloane_dinamice, culoare as char) AS culoare FROM Bunuri; - *Vizualizare coloane statice si dinamice*^[11]

3.2 Replicarea

3.2.1 Introducere

Replicarea

Replicarea în MariaDB se realizează la fel ca în MySQL. Replicarea este fundația pentru a construi aplicații mari de performanță înaltă folosind MariaDB. Replicarea permite configurarea unui server sau a mai multor servere, replici ale altui server. Acest lucru nu este valabil doar pentru aplicațiile de înaltă performanță dar și pentru rezolvarea altor probleme cum ar fi, împărțirea (sharing-ul) de date cu mașini aflate la distanță, păstrarea unei copii de rezerva sau păstrarea unei copii pentru testare sau învățare (training). [\[1\]](#)

În continuare vor fi analizate toate aspectele replicării. Pentru început va fi prezentat modul de funcționare, apoi configurarea unui server, proiectarea unor configurații mai avansate de replicare, management-ul și optimizarea serverelor replicate. [\[1\]](#)

Problema replicării

Problema replicării se rezolvă prin punerea la dispoziție a unui server ce va fi sincronizat cu un altul, folosind arhitectura master-slave. Mai multe slave-uri se pot conecta la un singur master, iar un slave se poate comporta la rândul său ca un master. Serverele master și slave pot fi aranjate în diferite topologii. Se poate replica întregul server, doar anumite baze de date sau se pot alege doar anumite tabele pentru replicare. [\[1\]](#)

MariaDB suportă 2 tipuri de replicare : replicare „statement-based” sau replicare „row-based”. „Statement-based” sau replicare logică este disponibilă de la MySQL 3.23 și este cea mai folosită în acest moment. Replicare „row-based” este valabilă de la MySQL 5.1. Ambele tipuri de replicare salvează schimbările efectuate pe baza de date în așa numitul „binary log” iar acest fișier este derulat pe slave. Ambele tipuri de replicare sunt asincrone, asta însemnând că nu există certitudinea că datele copiate pe slave sunt actualizate la un anumit moment de timp calculat. Nu există nicio garanție asupra timpului de actualizare a datelor pe slave. Slave-ul poate garanta actualizarea datelor după secunde, minute sau chiar ore dacă query-urile au dimensiuni mari. [\[1\]](#)

Replicarea MySQL este cea mai compatibilă cu versiunile anterioare. Asta înseamnă că un server cu o versiune mai nouă poate fi slave pentru un server cu o versiune mai veche fără probleme. Totuși versiunile mai vechi în principiu nu pot fi slave pentru un server cu o versiune mai nouă (nu pot înțelege funcționalități noi apărute și pot fi diferențe între formatul de fișier folosit pentru replicare). Replicare nu presupune un timp adițional (overhead) mare pe master, presupune doar activarea funcției de „binary logging”. Fiecare slave poate adăuga o încărcare suplimentară dar infimă. [\[1\]](#)

Replicarea este în general folosită pentru a scala citiri care pot fi direcționate către un slave, dar nu este o modalitate bună de a scala scrieri. Atașarea multor slave-uri la un master presupune scrierea

datelor pe fiecare din mașini. Fiecare sistem funcționează la viteza de scriere pe care o are cea mai slabă mașina. [\[1\]](#)

Replicare este ineficientă când există mai mult de câteva slave-uri deoarece se duplică date fără rost. De exemplu un singur master cu 10 slave-uri va stoca 11 copii a acelorași date și va duplica mare parte din date în 11 cache-uri diferite. Acesta este analog cu „11 way RAID to 1” la nivel de server care specifică faptul ca acest tip de arhitectură nu este un mod de a utiliza eficient aparatura hardware. Totuși acest tip de arhitectură poate fi întâlnit destul de des. În continuare vor fi discutate moduri de a evita această problema. [\[1\]](#)

Probleme rezolvate de replicare

Cele mai întâlnite utilizări ale replicării:

- Distribuția datelor (Data distribution)

Replicare în MariaDB și MySQL nu folosește de obicei foarte mult din lungimea de bandă a rețelei și poate fi oprită sau pornită la cerere. Prin urmare este recomandată păstrarea unei copii a datelor într-un loc distant din punct de vedere geografic cum ar fi alt centru de date. Slave-ul de la distanță poate lucra chiar cu o conexiune intermitentă (intenționată sau nu). Totuși dacă este necesară o replicare fără timp de răspuns mare va fi nevoie de o conexiune stabilă. [\[1\]](#)

- Distribuirea încărcării (Load balancing)

Replicare poate ajuta la distribuția query-urilor de citire pe mai multe servere, lucru ce funcționează foarte bine pe aplicații cu operații intensive de citire. Se poate efectua „load balancing” doar cu câteva linii de cod. Pe scală mică se poate utiliza abordarea simplistă cum ar fi nume de host „hardcoded” sau DNS round-robin (care face ca mai multe IP-uri să poarte către aceeași gazdă). Există și abordări mai sofisticate. Soluții standard mai costisitoare sunt produsele pentru distribuție a încărcării pe rețea care funcționează bine împreună cu serverele MySQL. Proiectul server virtual Linux (LVS) se pliază pe această situație.

- Rezerva (Backup)

Replicarea este o tehnică bună pentru a păstra o copie de rezervă a datelor. Însă un slave nu este prin definiție un backup. [\[1\]](#)

- Disponibilitate mare și failover

Replicarea ajută la evitarea problemei ce presupune prăbușirea aplicației dacă există un singur punct care dacă pica poate face aplicația inutilizabilă (defectare hardware a mașinii ce ține serverul de baze de date). Un sistem bun de failover este deținerea de slave-uri replicate ce pot lua locul master-ului în caz de defectare. [\[1\]](#)

- Testarea actualizărilor (updates) aduse sistemului de gestiune de baze de date

Este o practică des întâlnită setarea unui slave care are o versiune actualizată de MariaDB/MySQL și folosirea ei pentru a testa buna funcționalitate înainte de a actualiza toate sistemele. [\[1\]](#)

Cum funcționează replicarea

Modul în care sunt replicate datele este prezentat în continuare. Văzută de la un nivel înalt replicarea presupune 3 pași:

1. Masterul înregistrează schimbările în fișierul de „binary log”. (Aceste modificări se numesc evenimente ale „binary log”-ului)
2. Slave-ul copiază evenimentele fișierului „binary log” de pe master în fișierul său „relay log”.
3. Slave-ul derulează evenimentele din „relay log”, aplicând modificările pe datele sale. [\[1\]](#)

Aceasta este vederea de ansamblu. Fiecare pas este destul de complex. În continuare va fi prezentată replicarea mai în detaliu. [\[1\]](#)

Prima parte a procesului este cea de „binary logging” care are loc pe master. Înainte ca orice tranzacție să înceapă să modifice date pe master, master-ul înregistrează schimbările în „binary log”. Tranzacțiile sunt scrise serial chiar dacă ele au fost intercalate în momentul execuției lor. După această scriere în fișierul de log, masterul indică motorului de stocare (storage engine) să salveze (to commit) tranzacțiile. [\[1\]](#)

Cum funcționează replicarea (fig. 3.1):

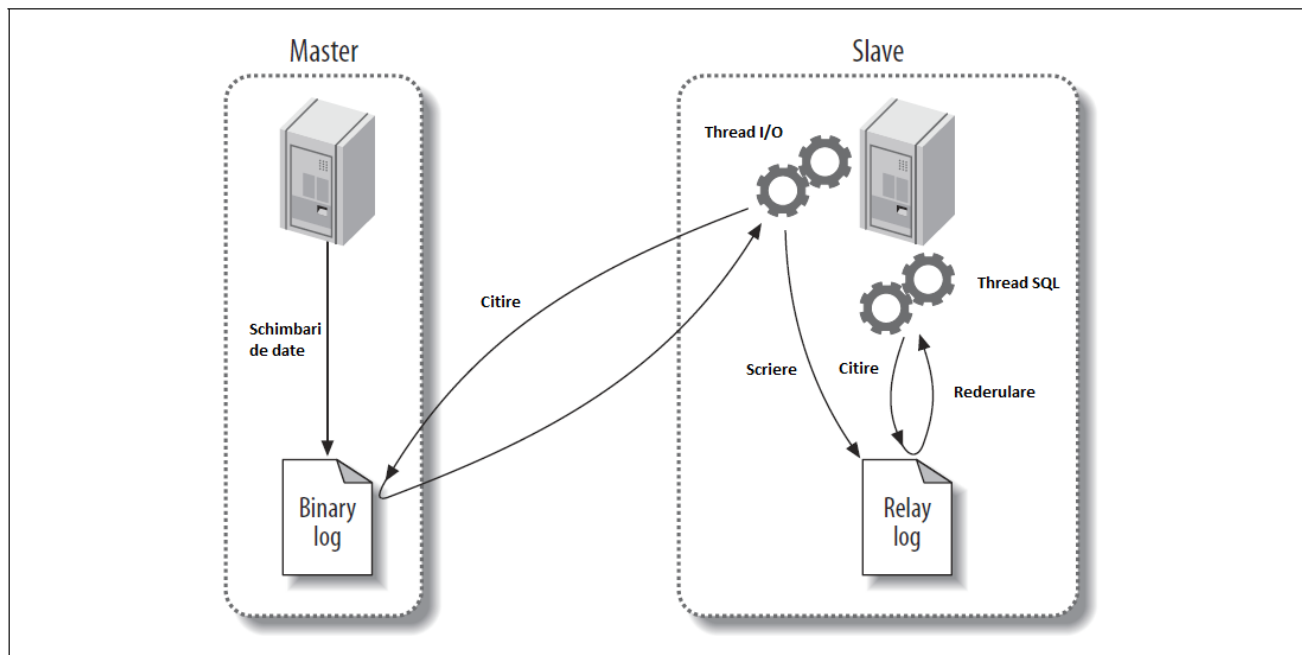


Fig. 3.1 Funcționarea replicării [\[1\]](#)

Următorul pas este ca slave-ul să copieze „binary log”-ul de pe master pe hard-disk-ul său în așa numitul „relay log”. Pentru început pornește un fir de execuție „worker” numit firul de execuție I/O al slave-ului. Firul de execuție I/O deschide o conexiune obișnuită de la client către master, apoi începe o operație specială numită „binlog dump” (nu există o comandă SQL corespondentă). Acest proces citește evenimentele din „binary log”-ul masterului. El nu interoghează master-ul în vederea schimbărilor ci după ce este adus la curent cu masterul intra în „sleep” și așteaptă ca masterul să semnaleze noi evenimente. Acest fir de execuție scrie evenimentele în „relay log”. [\[1\]](#)

Firul de execuție slave SQL se ocupă de ultima parte a întregului proces. Acest fir citește și derulează evenimentele din „relay log” astfel încât datele de pe slave vor fi la curent cu cele de pe master. Atât timp cât acest fir de execuție tine pasul cu firul I/O, „relay log”-ul stă de obicei în cache-ul sistemului de operare, prin urmare „relay log”-urile au un timp adițional (overhead) foarte mic. Evenimentele pe care firul de execuție SQL le executa pot ajunge opțional în „binary log”-ul slave-ului, lucru folositor pentru scenariile de replicare ce vor fi prezentate ulterior. [\[1\]](#)

În figura de mai sus sunt prezente 2 fire de execuție pe slave dar conexiunea pe care slave-ul o deschide pe master pornește un nou fir de execuție și pe master. [\[1\]](#)

Acest proces de replicare decuplează procesele de detectare și derulare a evenimentelor pe slave ceea ce oferă posibilitatea ca ele să fie asincrone. Prin urmare, firul de execuție I/O poate lucra independent de firul de execuție SQL. De asemenea sunt plasate constrângeri pe procesul de replicare, cea mai importantă fiind ca replicarea este serializată pe slave. Asta înseamnă ca actualizări ce ar fi putut rula în paralel pe master nu vor fi paralelizate pe slave. Dat fiind acestui fapt pot exista întârzieri de tipul gât de sticlă (bottle-necks) când există multe date de prelucrat. [\[1\]](#)

3.2.2 Setarea replicării

Setarea replicării este un proces relativ simplu, dar există multe variațiuni ale procesului în funcție de scenariu. Cel mai simplu scenariu este atunci când avem un master și un slave. Procesul presupune următoarele etape în mare:

1. Configurarea conturilor de replicare pe fiecare server;
2. Configurarea masterului și a slave-ului;
3. Instruirea serverului să se conecteze și să replice din master. [\[1\]](#)

În principiu multe setări standard (default) nu vor trebui modificate, atât timp cât cele două instanțe sunt proaspăt instalate și conțin aceleași date. În continuare vom presupune ca cele două instanțe se numesc server1 (cu adresa IP 192.168.0.1) și server2 (cu adresa IP 192.168.0.2) și va fi explicată inițializarea slave-ului. [\[1\]](#)

Crearea conturilor pentru replicare

Există câteva privilegii ce trebuie setate pentru ca replicarea să funcționeze. Firul de execuție I/O de pe slave se conectează la master prin TCP/IP. Acest lucru înseamnă că va trebui creat un cont pe master și

se vor seta privilegiile corespunzătoare pentru ca firul I/O să se conecteze ca acel utilizator (user) să citească „binary log”-ul masterului. Cum se va crea acest cont pentru user-ul *repl*:

```
mysql> GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*
```

```
-> TO repl@'192.168.0.%' IDENTIFIED BY 'p4ssword';
```

Se va crea acest cont user și pe master și pe slave. Replicarea este constrânsă la rețeaua locală curentă (192.168.0.x) deoarece contul de replicare nu este sigur. [\[1\]](#)

Replicarea presupune doar setarea contului pe master dar s-a făcut setarea lui și pe client deoarece este mai ușoară monitorizarea ulterioară a schimbărilor de pe client și eventuala comportarea a slave-ului ca master pentru un nou slave. [\[1\]](#)

Configurarea masterului și slave-ului

Următorul pas constă în activarea anumitor opțiuni pe master, pe care îl numim server1. Trebuie activat „binary logging” și specificat un ID pentru server. În fișierul my.cnf trebuie adăugate sau modificate următoarele linii :

```
log_bin = mysql-bin
```

```
server_id = 10
```

Valorile sunt la alegerea utilizatorului. Ele au fost alese pentru simplitate. [\[1\]](#)

ID-ul de server trebuie să fie unic. De aceea a fost ales 10 în loc de 1. Unu este valoarea standard (default). Prin urmare utilizarea numărului 1 poate crea confuzii sau conflicte între servere care au același ID sau nu îl au setat. Un obicei comun este utilizarea ultimului octet din adresa IP a serverului presupunând că această nu se va schimba și ca este unică. [\[1\]](#)

Dacă opțiunea „binary logging” nu era specificată în configurare este necesară repornirea server-ului. Pentru a verifica faptul ca fișierul a fost creat pe master se poate rula comanda **SHOW MASTER STATUS** și se va obține un output similar cu următorul:

```
mysql> SHOW MASTER STATUS;
```

```
+-----+-----+-----+-----+
| File | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000001 | 98 | | |
+-----+-----+-----+-----+
```

1 row în set (0.00 sec)

Slave-ul necesită și el o configurare similară cu masterul în fișierul my.cnf și o repornire. [\[1\]](#)

```
log_bin = mysql-bin
```

server_id = 2

relay_log = mysql-relay-bin

log_slave_updates = 1

read_only = 1

Câteva din aceste opțiuni nu sunt strict necesare. Pe slave doar server ID-ul este necesar dar a fost activat și „binary logging” cu un nume explicit. În principiu vrem aceeași configurare pe toate mașinile astfel încât promovarea de la slave la master să fie facilă. La fel cum au fost create aceleași conturi și pe master și pe slave, vrem și aceleași setări. [\[1\]](#)

Există 2 parametri opționali: *relay_log* – care specifică locația și numele „relay log”-ului și *log_slave_updates* – care specifică slave-ului să adauge evenimentele primite și în „binary log”-ul propriu. Ultima opțiune cauzează procesare în plus pentru slave dar în continuare va fi explicată de ce adăugarea ei este o practică bună. [\[1\]](#)

Pornirea slave-ului

Următorul pas spune slave-ului cum să se conecteze la master și să înceapă derularea „binary log”-urilor primite. Pentru acest scop nu trebuie folosit fișierul my.cnf, ci comanda CHANGE MASTER TO. Această comandă înlocuiește setările corespundente din my.cnf în totalitate. În viitor permite și schimbarea masterului fără oprirea serverului. Următorul cod trebuie rulat pe slave:

```
mysql> CHANGE MASTER TO MASTER_HOST='server1',
```

```
-> MASTER_USER='repl',
```

```
-> MASTER_PASSWORD='p4ssword',
```

```
-> MASTER_LOG_FILE='mysql-bin.000001',
```

```
-> MASTER_LOG_POS=0;
```

Câmpul MASTER_LOG_POS este setat cu valoarea 0 deoarece indica începutul log-ului. După ce se rulează această comandă se poate verifica setarea cu comanda SHOW SLAVE STATUS ce produce următorul output:

```
mysql> SHOW SLAVE STATUS\G
```

```
***** 1. row *****
```

Slave_IO_State:

Master_Host: server1

Master_User: repl

Master_Port: 3306

Connect_Retry: 60

Master_Log_File: mysql-bin.000001

Read_Master_Log_Pos: 4

Relay_Log_File: mysql-relay-bin.000001

Relay_Log_Pos: 4

Relay_Master_Log_File: mysql-bin.000001

Slave_IO_Running: No

Slave_SQL_Running: No

...omitted...

Seconds_Behind_Master: NULL

Coloanele *Slave_IO_State*, *Slave_IO_Running*, și *Slave_SQL_Running* arată ca procesele de pe slave nu rulează. Se observă ca poziția log-ului este 4. Prin urmare 0 setată anterior nu este o poziție ci indică doar începutul fișierului de log. Primul eveniment este de fapt la poziția 4. [\[1\]](#)

Pentru a începe replicarea se execută comanda :

mysql> START SLAVE;

Această comandă nu ar trebui să producă erori sau output. Se poate verifica cu comanda SHOW SLAVE STATUS:

mysql> SHOW SLAVE STATUS\G

******* 1. row *******

Slave_IO_State: Waiting for master to send event

Master_Host: server1

Master_User: repl

Master_Port: 3306

Connect_Retry: 60

Master_Log_File: mysql-bin.000001

Read_Master_Log_Pos: 164

Relay_Log_File: mysql-relay-bin.000001

Relay_Log_Pos: 164

Relay_Master_Log_File: mysql-bin.000001

Slave_IO_Running: Yes

Slave_SQL_Running: Yes

...omitted...

Seconds_Behind_Master: 0

Se observă ca ambele fire de execuție I/O și SQL rulează și câmpul *Seconds_Behind_*

Master nu mai este NULL. Firul de execuție I/O așteaptă evenimente de la master ceea ce înseamnă că a preluat toate fișierele „binary log” de la master. Pozițiile de log s-au incrementat ceea ce înseamnă că există evenimente ce au fost preluate și executate. Modificarea masterului va face schimbări și pe slave.

Se pot vizualiza și firele de execuție pentru replicare în lista de procese atât pe master cât și pe slave. Pe master va fi o conexiune creată de firul I/O al slave-ului:

mysql> SHOW PROCESSLIST\G

******* 1. row *******

Id: 55

User: repl

Host: slave1.webcluster_1:54813

db: NULL

Command: Binlog Dump

Time: 610237

State: Has sent all binlog to slave; waiting for binlog to be updated

Info: NULL

Pe slave se pot observa 2 fire de execuție, cel I/O și cel SQL:

mysql> SHOW PROCESSLIST\G

******* 1. row *******

Id: 1

User: system user

Host:

db: NULL

Command: Connect

Time: 611116

State: Waiting for master to send event

Info: NULL

***** 2. row *****

Id: 2

User: system user

Host:

db: NULL

Command: Connect

Time: 33

State: Has read all relay log; waiting for the slave I/O thread to update it

Info: NULL [\[1\]](#)

Inițializarea unui alt slave de pe alt server

În configurația anterioară am presupus că și masterul și slave-ul aveau aceleași date inițial fiind nou instalate. Acesta este un caz ideal, de obicei un master a rulat mult timp și se dorește sincronizarea unui slave care nu are toate datele masterului. [\[1\]](#)

Există mai multe moduri de a inițializa, clona un server pe slave. Printre acestea se numără copierea datelor de pe master, clonarea unui slave de pe un alt slave, pornirea unui slave dintr-un backup. Pentru a sincroniza un slave cu masterul sunt necesare:

- O clona, „snapshot” al masterului la un anumit moment;
- Fișierul de log curent al masterului și poziția (offset) din log la care a fost făcută acea clonă. Aceste valori se numesc coordonate ale fișierului de log, deoarece identifică poziția în „binary log”. Coordonatele pot fi găsite cu comanda SHOW MASTER STATUS.
- „binary log”-ul masterului din acel moment până în prezent. [\[1\]](#)

Câteva moduri de a clona un slave dintr-un alt slave:

- Copie „la rece”

Aceasta presupune oprirea masterului și copierea fișierelor pe slave. Apoi masterul poate fi pornit din nou ceea ce începe un nou „binary log”. Folosind comanda CHANGE MASTER TO se va porni un slave de la începutul acelui „binary log”. Dezavantajul evident este necesitatea de a opri masterul până când se va termina copierea. [\[1\]](#)

- Copie „la cald”

Dacă se folosesc numai tabele MyISAM, se poate folosi *mysqlhotcopy* pentru a copia fișiere în timp ce serverul rulează. [\[1\]](#)

- Folosind *mysqldump*

Dacă se folosesc numai tabele InnoDB se poate folosi următoarea comandă pentru a face „dump” tuturor datelor de pe master, încărcarea lor pe slave și schimbarea coordonatelor slave-ului la poziția corespunzătoare din „binary log”. [\[1\]](#)

\$ mysqldump --single-transaction --all-databases --master-data=1

--host=server1 | mysql --host=server2

Opțiunea *--single-transaction* impune operației să citească datele așa cum erau la începutul tranzației. Se poate folosi și *--lock-all-tables* pentru a avea un „dump” consistent al tuturor tabelor. [\[1\]](#)

- Folosind *LVM snapshot* sau *backup*

Atât timp cât se știu coordonatele corespunzătoare din „binary log”, se poate folosi un *LVM snapshot* sau *backup* al masterului pentru a inițializa slave-ul (ex: InnoDB Hot Backup, backup-ul presupune existența tuturor „binary log”-urilor de la data de când a fost făcut backup-ul).

Backup-ul va fi rulat pe slave cu comanda *CHANGE MASTER TO* se pornește slave-ul. [\[1\]](#)

- Folosind un alt slave

Orice metodă menționată mai sus poate fi folosită pentru a clona un slave dintr-un alt slave. Dacă se folosește comanda *mysqldump*, opțiunea *--master-data* nu va funcționa. [\[1\]](#)

De asemenea, în locul comenzii *SHOW MASTER STATUS*, se va folosi *SHOW SLAVE STATUS*. [\[1\]](#)

Cel mai mare dezavantaj când se clonează un slave dintr-un alt slave este acela că dacă slave-ul nu este sincronizat cu masterul, se vor clona date invalide. [\[1\]](#)

3.2.3 Implementarea replicării

În continuare va fi explicat modul în care funcționează replicarea mai în amănunt. Vor fi discutate punctele forte și slăbiciunile și ulterior vor fi analizate configurații mai complexe de replicare.

Replicarea bazată pe statement (statement-based)

Numită și replicare logică, ea este primul mod de replicare folosit în MySQL. Acest mod nu este unul des întâlnit în lumea bazelor de date. Replicarea „statement based” funcționează înregistrând query-ul care a modificat datele de pe master. Când slave-ul citește evenimentele din „relay log” și le execută va face același lucru pe care l-a făcut masterul executând query-urile SQL. Acest tip de replicare are atât avantaje cât și dezavantaje.

Avantajul imediat observabil este acela că este ușor de implementat. Simpla înregistrare și reexecutare a query-urilor ce au modificat datele în teorie va ține slave-ul sincronizat cu masterul. Un alt beneficiu al replicării „statement based” este acela că evenimentele în „binary

log” sunt compacte. Deci acest tip de replicare teoretic nu va face un trafic mare pe rețea – un query care actualizează giga octeți de date poate avea doar câțiva octeți în „binary log”. Totuși acest tip de replicare nu este la fel de simplu în practica pentru ca multe schimbări de pe master pot depinde de alți factori diferiți de query-ul efectiv. De exemplu instrucțiunile se vor executa la o diferență de timp cel puțin mica sau posibil mai mare între slave și master. Ca rezultat în „binary log” sunt ținute și informații (meta data) cum ar fi un *timestamp*. Chiar și așa unele instrucțiuni nu se pot replica corect cum ar fi funcția *CURRENT_USER()*. Procedurile stocate și trigger-ele sunt și ele problematice în replicarea „statement-based”. O altă problema la acest tip de replicare este acela că modificările trebuie să fie serializabile. Asta presupune un mare volum de lucru la configurare, funcții extra ale server-ului. InnoDB folosește replicarea „statement based”. Nu toate motoarele de stocare folosesc acest tip de replicare deși până la MySQL 5.1 acest lucru era posibil. [\[1\]](#)

Replicarea pe baza de rând (row-based)

Replicarea „row-based” înregistrează în „binary log” exact ce modificări au fost aduse datelor, similar cum oferă alte produse de tip baze de date de pe piață. Acest tip de replicare are și ea avantajele și dezavantajele sale. Cel mai mare avantaj este acela că se va replica fiecare schimbare corect și unele instrucțiuni pot fi replicate mult mai eficient. Dezavantajul primar este acela că „binary log”-ul va deveni considerabil mai mare și există o vizibilitate mai proastă asupra căror instrucțiuni au modificat datele. Prin urmare nu se va mai putea folosi *mysqlbinlog* pentru audit. [\[1\]](#)

Replicarea „row-based” din păcate nu este compatibilă backward (compatibilă cu versiunile mai vechi). *mysqlbinlog*, care poate citi fișierele „binary log” aducându-le într-o formă descifrabilă pentru oameni, va ieși din program afișând eroare dacă se încearcă citirea unui „binary log” dintr-o distribuție mai veche a software-ului. [\[1\]](#)

Unele schimbări pot fi replicate mai eficient deoarece salve-ul nu va trebui să reexecute query-urile ce au făcut schimbări pe master. Reexecutarea de query-uri poate fi costisitoare. Ca exemplu poate fi luat acest query ce sumarizează date dintr-un tabel mare în unul mai mic:

```
mysql> INSERT INTO tabel_sumar(col1, col2, sum_col3)
-> SELECT col1, col2, sum(col3)
-> FROM tabel_enorm
-> GROUP BY col1, col2;
```

Sa presupunem că există trei combinații unice dintre coloanele col1 și col2 din tabelul *tabel_enorm*. Acest query va scana multe rânduri din tabelul sursă și vor rezulta doar 3 rânduri în tabelul destinație. Replicarea acestui eveniment ca „statement based” va obliga slave-ul să facă din nou toată procesarea pe care a făcut-o master-ul doar pentru a genera câteva rânduri însă replicarea „row based” va simplifica semnificativ procesarea slave-ului. În acest caz replicarea „row based” este mult mai eficientă. [\[1\]](#)

Pe de altă parte următorul eveniment este mult mai simplu de replicat prin „statement based”:

```
mysql> UPDATE tabel_enorm SET col1 = 0;
```


Replicarea row-based va fi foarte costisitoare în acest caz deoarece fiecare schimbare va fi scrisă în „binary log”, acesta devenind semnificativ mai mare. Prin urmare încărcarea master-ului va fi mai mare și la înregistrare și la replicare.

Deoarece nici unul dintre moduri nu este perfect pentru fiecare situație, ambele moduri pot fi folosite alternativ dinamic. Ca și standard (default) este folosită replicarea „statement based” dar când sunt detectate evenimente ce nu pot fi replicate corect se trece la replicare „row based”. Modul poate fi controlat prin schimbarea variabilei *binlog_format*.

În teorie replicarea row-based rezolvă câteva din problemele menționate anterior, dar în producție încă se folosește considerabil replicarea „statement-based”. Prin urmare este prea devreme să concluzionăm asupra replicării „row-based”. [\[1\]](#)

Fișiere folosite la replicare

În afara fișierelor de „binary log” și „relay log” există și câteva alte fișiere folosite de replicare. Unde sunt plasate ele depinde de configurarea instanței. Versiuni diferite pot plasa aceste fișiere în directoare diferite. Pot fi găsite în directorul destinat datelor sau în cel care conține fișierul *.pid* al serverului (*/var/run/mysqld* – sistemele Unix). Aceste fișiere sunt:

- *mysql-bin.index*

Un server care are opțiunea de „binary logging” activată va avea de asemenea un fișier numit la fel ca acela numai ca extensia va fi *.index*. Acest fișier menține evidenta fișierelor de „binary logging” care există pe disc. Nu este un index în sensul index-ului din tabele; fiecare linie din acest fișier conține numele unui fișier de „binary log”. [\[1\]](#)

Poate părea ca acest fișier este redundant și poate fi șters, însă serverul nu se va uita pe disc ci în acest fișier pentru a cunoaște existența fișierelor de „binary log”. [\[1\]](#)

- *mysql-relay-bin.index*

Acest fișier are același rol pe care îl are cel de mai sus numai că pentru „relay log”. [\[1\]](#)

- *master.info*

Acest fișier conține informație folosită de către slave pentru a se conecta la master. Formatul este text simplu (o valoare pe linie) și variază în funcție de versiunea serverului. Nu trebuie șters deoarece un slave nu va mai ști cum să se conecteze la master după restart. Fișierul conține parola utilizatorului de replicare ca și text simplu deci accesul la el ar trebui să fie restricționat. [\[1\]](#)

- *relay-log.info*

Acest fișier conține coordonatele slave-ului pentru „binary log” și „relay log” (poziția pe master a slave-ului). Nu trebuie șters deoarece slave-ul nu va mai ști de unde să replice după repornire și este foarte posibil să încerce replicarea unor instrucțiuni ce deja au fost executate. [\[1\]](#)

Trimiterea evenimentelor de replicare către slave-uri

Opțiunea *log_slave_updates* permite folosirea unui slave ca master pentru alt slave. Se impune server-ului să scrie evenimentele pe care le firul de execuție SQL le execută în propriul „binary log” care va fi preluat și executat de serverele slave ale sale. Următoarea figură (fig. 3.2) ilustrează acest mod de lucru.

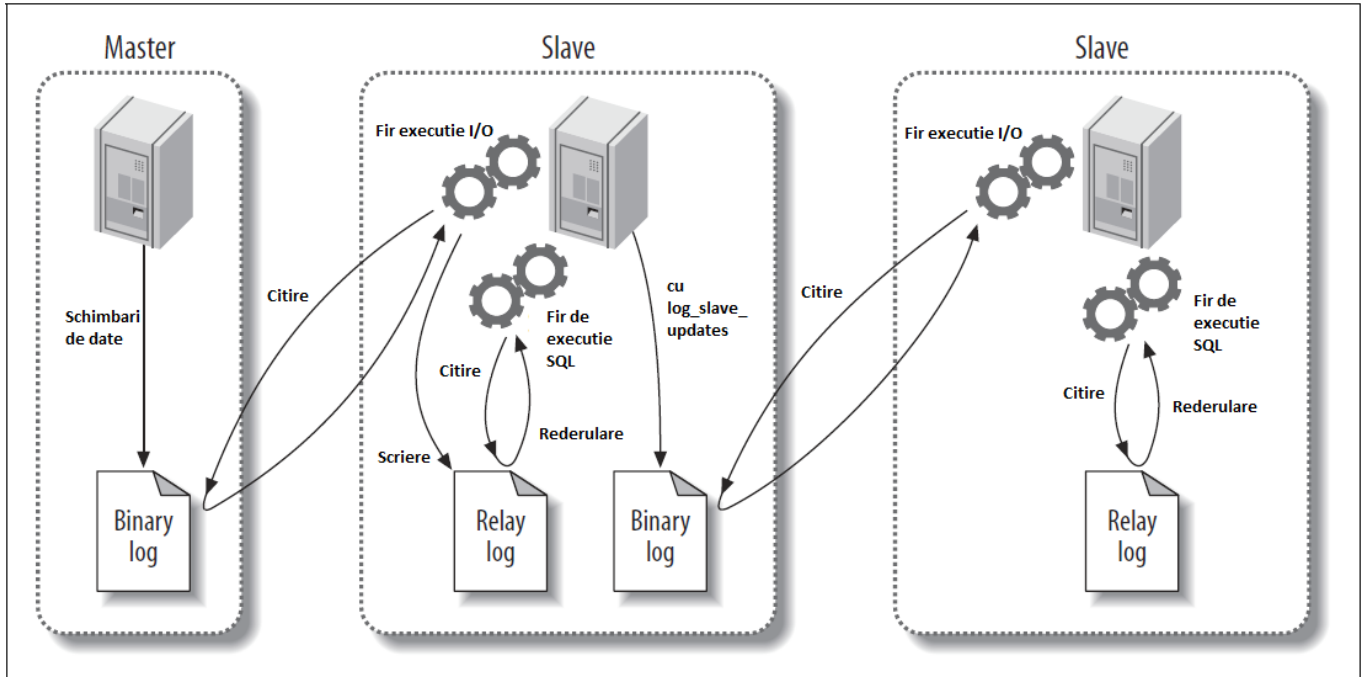


Fig. 3.2 Înlănțuirea de mai multe slave-uri^[1]

În acest scenariu o schimbare pe master cauzează un eveniment scris în „binary log”. Primul slave preia evenimentul și îl execută. În acest punct, de obicei ciclul de viață al evenimentului ar fi luat sfârșit însă deoarece este activată opțiunea *log_slave_updates* slave-ul va scrie modificările în „binary log”. Acum cel de-al doilea slave va putea prelua modificările în propriul său „relay log” și să le execute. Acest lucru presupune ca schimbările de pe serverul master se pot propaga către mașini ce nu sunt direct conectate la master. Opțiunea *log_slave_updates* de preferabil trebuie activată deoarece permite conectarea unui slave fără repornirea serverului. ^[1]

Când primul slave scrie în „binary log” un eveniment apărut pe master, acel eveniment cu siguranță va avea o poziție diferită față de cel de pe master – un alt fișier de log sau altă poziție numerică. Asta înseamnă ca nu se poate presupune ca toate serverele care sunt în același punct al replicării vor avea aceleași coordonate de log. ^[1]

Dacă din greșeală nu a fost asignat un ID de server diferit pentru fiecare mașină, acest tip de configurație poate cauza erori subtile, poate chiar cauza oprirea replicării. Una din cele mai întâlnite întrebări legate de replicate este de ce trebuie specificat acest ID. Când firul de execuție SQL citește din „relay log” va ignora evenimentele care au un server ID identic cu al său. Acest lucru împiedică apariția buclelor infinite în replicare. Acestea pot apărea la replicarea master-master. ^[1]

Filtre de replicare

Filtrele de replicare pot constrânge replicarea doar a anumitor părți ale datelor de pe master. Există două tipuri de filtre de replicare: acelea care filtrează evenimentele din „binary log”-ul masterului și acelea care filtrează evenimentele din „relay log”-ul slave-ului. Figura următoare (fig. 3.3) ilustrează aceste două tipuri:

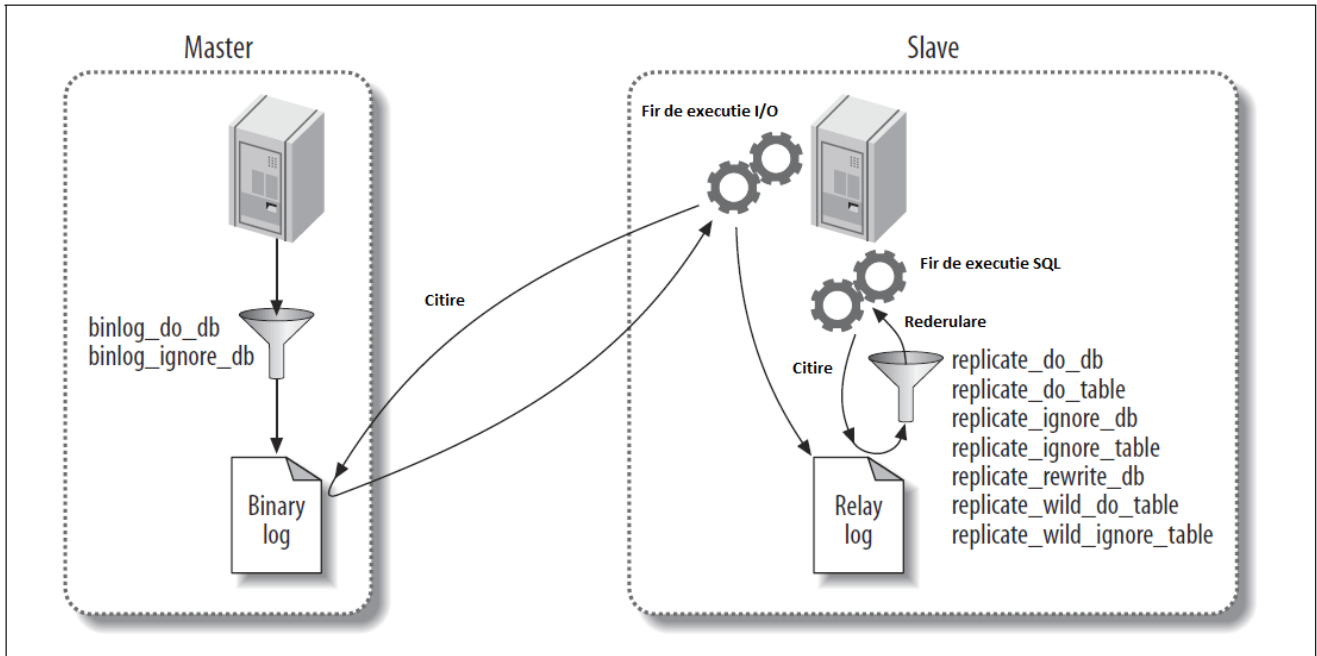


Fig. 3.3 Filtre de replicare [\[1\]](#)

Opțiunile care controlează filtrarea „binary log”-ului sunt `binlog_do_db` și `binlog_ignore_db`. De obicei nu trebuie activate după cum va fi explicat în continuare. [\[1\]](#)

Pe slave opțiunile `replicate_*` filtrează evenimente pe care firul de execuție SQL le citește din „relay log”. Se pot replica sau ignora una sau mai multe baze de date, rescrierea unei baze de date în altă bază de date și replicarea sau ignorarea unor tabele prin identificarea lor cu instrucțiunea `LIKE`. [\[1\]](#)

Cel mai important lucru de reținut în legătură cu aceste opțiuni este faptul că `*_do_db` și `*_ignore_db`, atât pe master sau slave pot să nu se manifeste așa cum ar fi de așteptat. Ar putea filtra după numele bazei de date dar de fapt filtrează după baza de date setată curent „default”. Dacă se execută următoarele instrucțiuni:

```
mysql> USE test;
```

```
mysql> DELETE FROM sakila.film;
```

parametrii `*_do_db` și `*_ignore_db` for filtra instrucțiunile `DELETE` pe baza de date `test` nu pe `sakila`. De obicei nu se vrea acest lucru deoarece instrucțiuni nedorite pot fi replicate sau ignorate.

Parametrii `*_do_db` și `*_ignore_db` pot avea utilizări dar acestea sunt limitate și rare, folosirea lor trebuie să fie făcută cu grija. Folosirea lor poate duce foarte ușor la desincronizare. [\[1\]](#)

Oprirea instrucțiunilor GRANT sau REVOKE de la replicare către salve-uri este o practică utilizată frecvent în filtre. Problema frecventă este ca administratorul folosește GRANT pentru a da privilegii unui utilizator pe master și apoi observă ca acestea au fost propagate și pe slave, unde acel utilizator nu ar trebui să aibă acces. Următoarele opțiuni pot preveni acest lucru:

replicate_ignore_table1`=mysql.columns_priv

replicate_ignore_table=mysql.db

replicate_ignore_table=mysql.host

replicate_ignore_table=mysql.procs_priv

replicate_ignore_table=mysql.tables_priv

replicate_ignore_table=mysql.user

Se poate aplica mai simplu un filtru pentru toate bazele de date care încep cu *mysql* cu o regulă de tipul:

replicate_wild_ignore_table=mysql.%

În acest fel cu siguranță instrucțiunile de tip GRANT nu vor fi replicate, dar anumite evenimente sau rutine pot fi excluse din replicare de asemenea. Prin urmare acesta este motivul pentru care utilizarea filtrelor trebuie să fie făcută cu grija. O idee mai bună poate fi prevenirea anumitor instrucțiuni specifice de a fi executate setând *SET SQL_LOG_BIN=0*, dar și acest mod are deficiențele sale. [\[1\]](#)

Ca și concluzie folosirea filtrelor trebuie făcută cu atenție deoarece replicarea de tipul „statement based” poate fi întreruptă foarte ușor. Replicare „row based” poate rezolva anumite probleme dar nu este încă dovedit 100% acest lucru. [\[1\]](#)

3.2.4 Topologii de replicare

Replicare poate fi setată pentru aproape orice tip posibil de configurare master-slave cu limitarea că o instanță slave poate avea un singur master. Prin urmare diferite topologii complexe pot fi dezvoltate, iar cele simple sunt destul de flexibile. O anumită topologie poate avea diferite utilizări. [\[1\]](#)

După cum a fost prezentat anterior setarea unui singur master cu un singur slave a fost acoperită. În continuare vor fi discutate diferite alte topologii cu avantajele și dezavantajele lor. De reținut sunt următoarele reguli:

- O instanță slave poate avea un singur master
- Fiecare server trebuie să aibă un server ID unic
- Un master poate avea mai multe salve-uri (un slave poate avea mai mulți copii la rândul său)

- Un slave poate propaga schimbările de pe master și poate fi la rândul său master pentru alte slave-uri dacă este activată opțiunea *log_slave_updates*. [\[1\]](#)

Un master cu mai multe slave-uri

Această topologie este foarte simplă deoarece implică atașarea unui nou slave la topologia clasică master-slave. Cele 2 sau mai multe slave-uri nu vor comunica între ele ci doar cu masterul ca în următoarea figură (fig. 3.4):

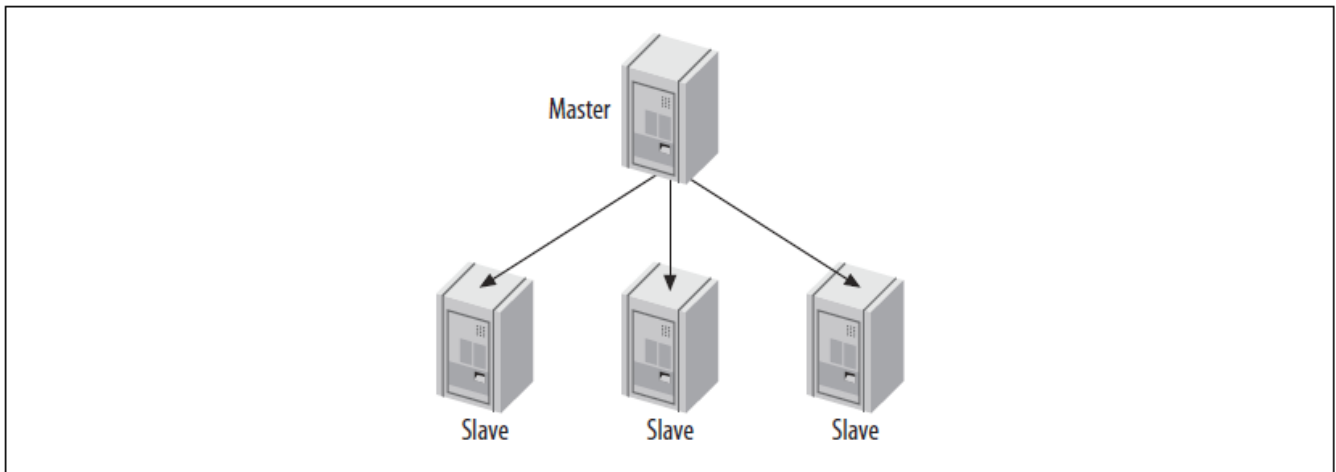


Fig. 3.4 Un master cu mai multe slave-uri [\[1\]](#)

Această topologie este folositoare și avantajoasă atunci când se execută multe citiri. Aceste citiri pot fi distribuite pe mai multe servere atunci când încărcarea pe master este prea mare sau încărcarea rețelei de la master către slave-uri este foarte mare. Pot fi setate mai multe slave-uri o dată sau adăugarea lor pe parcurs în funcție de necesar, folosind aceiași pași prezentați anterior la setarea unui slave. [\[1\]](#)

Deși este o topologie foarte simplă este destul de flexibilă pentru a satisface diferite probleme cum ar fi:

- Folosirea a diferite slave-uri pentru diferite roluri (de exemplu folosirea a diferite motoare de stocare pe slave-uri)
- Setarea unui server de replicare fără alt tip de trafic în afara de replicare
- Punerea unui slave într-un centru de date extern pentru recuperarea în caz de dezastru
- Întârzierea replicării pe anumite slave-uri pentru recuperare în caz de dezastru
- Folosirea unuia dintre slave-uri pentru *backup*, pregătire personal sau ca server de dezvoltare (development). [\[1\]](#)

Unul dintre motivele pentru care acest tip de replicare este foarte răspândit este acela că evita multe complexități ce apar în celelalte tipuri de replicare. De exemplu este ușor de comparat un slave cu altul ca și poziția a „binary log”-ului deoarece ea trebuie să fie aceeași mereu după replicare. Dacă toate slave-urile vor fi oprite la același moment de tip logic în replicare ele vor citi din nou de la același punct. Acest lucru rezolvă probleme administrative cum ar fi promovarea unui slave ca master.

Acest lucru este posibil între slave-uri de același nivel (frați). Este mai complicată compararea între slave-uri care nu sunt la același nivel față de master. Multe din modurile următoare de replicare cum ar fi arborele sau distribuția de master fac mult mai grea depistarea secvenței de unde se face replicarea. ^[1]

Master-master în modul activ-activ

Replicarea master-master (numită și dual master sau replicare bidirecțională) implică doua servere ambele master și slave unul pentru celălalt. În alta ordine de idei 2 perechi master-slave ca în figura următoare (fig. 3.5):

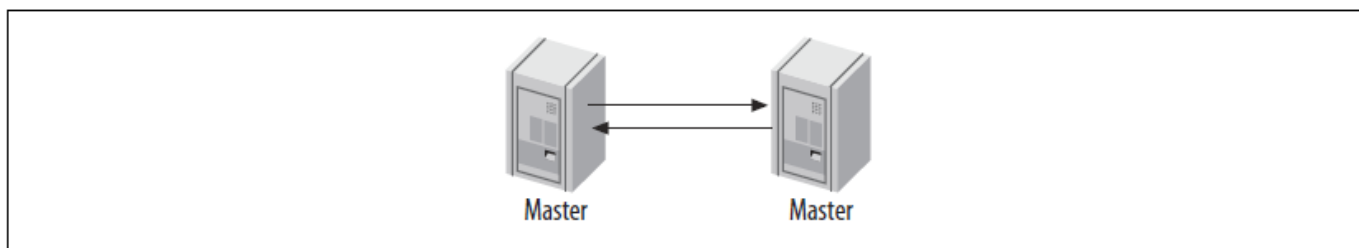


Fig. 3.5 Modul master-master^[1]

Replicarea master-master în modul activ-activ are utilizări speciale. Una din utilizări poate fi pentru birouri separate geografic unde fiecare birou are nevoie de copia sa de date locala.

Cea mai mare problema cu acest tip de configurare este rezolvarea de conflicte. Lista de probleme poate fi foarte lungă. Problemele apar de obicei când doua query-uri (unul pe prima mașină, celălalt pe a doua mașină) schimbă același rând simultan pe ambele servere sau inserează un rând cu *AUTO_INCREMENT* în același timp. ^[1]

Replicare multi-master nu este suportata

Replicare multi-master reprezintă posibilitate ca un slave să replice de pe mai multe mastere. Deși alte tipuri de sisteme de baze de date suportă acest lucru topologia din figura următoare (fig. 3.6) nu este posibilă:

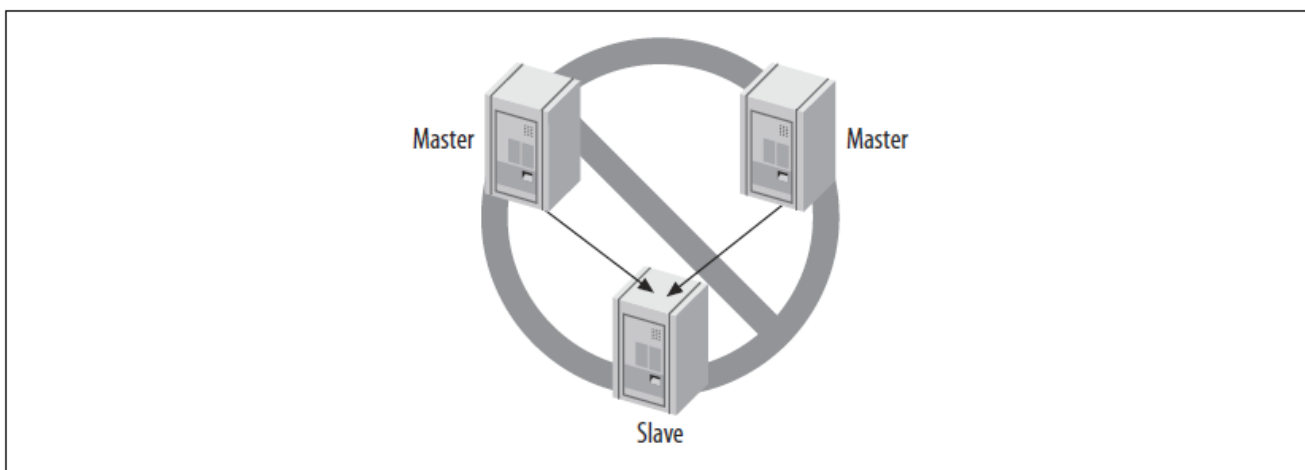


Fig. 3.6 Multi-master^[1]

Totuși acest tip de replicare poate fi emulat după cum va fi explicat ulterior.

Din păcate există o confuzie în ceea ce privește terminologia. Multe persoane folosesc termenul multi-master pentru a descrie alte tipuri de topologie care conțin mai mult de două mastere. [\[1\]](#)

Funcționalități ce rezolvă problemele acestui tip de replicare au fost adăugate în MySQL 5.0 cum ar fi opțiunile *auto_increment_increment* sau *auto_increment_offset*. Aceste opțiuni pot autogenera valori care să nu aibă conflicte la INSERT. Totuși operația de scriere pe ambele mastere rămâne periculoasă. Actualizări ce au loc în ordine diferită pe cele 2 mașini pot duce la corupere de date fără nici un fel de avertizare. Ca exemplu putem lua un singur tabel cu o singură coloană cu valoarea 1. Aceste două query-uri se vor executa simultan:

- Pe primul master:
mysql> UPDATE tabel SET col=col + 1;
- Pe al doilea master:
mysql> UPDATE tabel SET col=col * 2;

Rezultat va fi acela ca pe un server vom avea valoarea 4 pe altul 3 și nicio eroare dată de replicare.

Desincronizarea datelor este doar începutul. Replicarea normală se poate opri cu eroarea dar aplicațiile nu se opresc din a scrie pe ambele servere. Rezolvarea acestei probleme devine foarte complicată deoarece ambele servere au modificări ce ar trebui propagate pe celălalt server. [\[1\]](#)

Setarea master-master în modul active-active devine foarte dificilă dar nu imposibilă folosind date partiționate corect și privilegii setate corect. De obicei există o modalitate mai bună decât aceasta pentru a răspunde nevoilor. [\[1\]](#)

În general a lăsa libertatea de a scrie pe ambele servere poate aduce mai multe probleme decât beneficii. Prin urmare configurarea active-pasivă ce va fi prezentată în continuare este mai indicată. [\[1\]](#)

Master-master în modul activ-pasiv

Acest mod evită problemele ce au apărut la modul discutat anterior, este chiar un mod foarte indicat și puternic de a proiecta un sistem tolerant la defecte și cu disponibilitate mare. Diferența este aceea că unul din servere este pasiv în modul „read-only” ca în figura următoare (fig. 3.7):

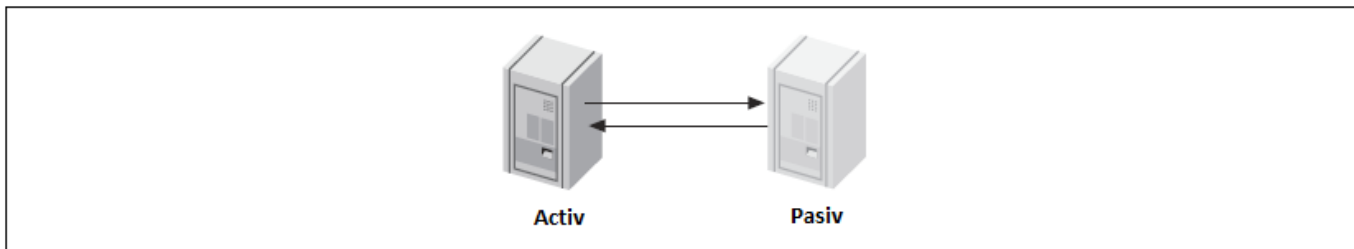


Fig.3.7 Master-master în modul activ-pasiv [\[1\]](#)

Acest mod permite interschimbarea între *activ* și *pasiv* foarte ușor deoarece configurațiile serverelor sunt simetrice. Asta face sistemul tolerant la defecte. De asemenea mentenanța, optimizarea tabelelor și actualizarea sistemului de operare sau a aplicațiilor, a hardware-ului devine ușor de realizat. ^[1]

De exemplu folosirea instrucțiunii *ALTER TABLE* presupune închiderea tabelului (lock), blocarea scrierii și citirii către tabel. Acest lucru poate fi de durată și poate întrerupe serviciul. Prin urmare configurația master-master permite oprirea firelor de execuție slave pe serverul activ și nu va procesa schimbări de pe serverul pasiv. Se vor altera datele de pe serverul pasiv apoi se vor schimba rolurile și se va reporni procesul slave pe serverul anterior activ. Acest server va citi „relay log”-ul și va executa aceleași instrucțiuni de *ALTER TABLE*. Acest lucru poate fi de durată dar nu contează pentru că serverul nu oferă disponibilitate pentru query-uri. ^[1]

Topologia *activ-pasiv master-master* poate fi ușor de folosit instalând utilitarul *MySQL Master-Master Replication manager Tool* (<http://code.google.com/p/mysql-master-master/>). Acesta automatizează diferite operațiuni cum ar fi recuperarea și resincronizarea, setarea unor slave-uri noi și așa mai departe. ^[1]

Pași pentru a configura o topologie master-master simetrică. Setul de pași trebuie realizat pe ambele mașini pentru a deveni simetrice.

1. Activarea „binary logging”, alegerea unui server ID unic, adăugarea conturilor de replicare.
2. Activarea *log_slave_updates*. Acest lucru este vital pentru toleranța la defecte.
3. Opțional serverul pasiv poate fi configurat ca „read-only” pentru a preveni schimbări ce ar putea genera conflicte pe serverul activ.
4. Ambele mașini trebuie să conțină exact aceleași date.
5. Pornirea fiecărei instanțe de MariaDB.
6. Configurarea fiecărei mașini ca slave pentru cealaltă, începând de la „binary log”-ul recent creat. ^[1]

În continuare vom urmări ce se întâmplă când o modificare este făcută pe serverul activ. Schimbarea este scrisă în „binary log” și ajunge prin replicare în „relay log”-ul slave-ului. Acest server pasiv execută instrucțiunile și scrie modificările în propriul „binary log” deoarece am activat *log_slave_updates*. Serverul activ primește aceleași schimbări prin replicare în propriul „relay log” dar le ignora deoarece server ID-ul lor este propriul ID. Interschimbarea activ-pasivă va fi prezentată ulterior. ^[1]

Setarea unei topologii *activ-pasiv master-master* presupune crearea unei copii de rezervă dar aceasta poate fi folosită pentru a spori performanța. Poate fi folosită pentru a citi din baza de date, *backup*, mentenanță și așa mai departe. Ea nu poate fi folosită pentru a obține performanța mai mare la scriere cum se obține cu un singur server. În următoarele topologii vom reveni la acest tip deoarece este o modalitate foarte comună în replicare. ^[1]

Master-master cu slave-uri

O configurație asemănătoare poate presupune adăugarea unui slave la fiecare master ca în figura următoare (fig. 3.8):

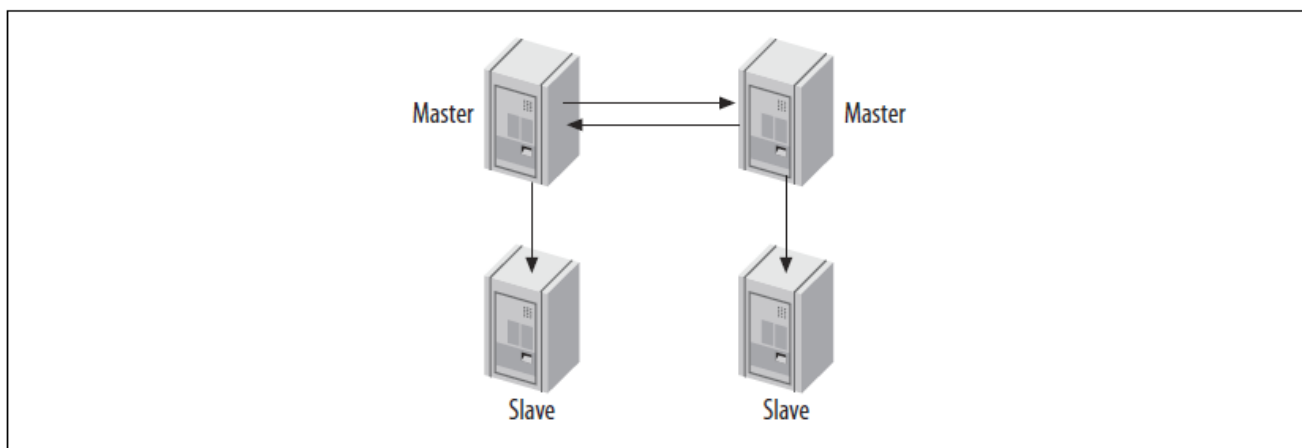


Fig. 3.8 Adăugarea de slave-uri la modul master-master^[1]

Avantajul este extra redundanță. Într-o rețea distribuită geografic elimină posibilitatea de defectare într-un singur punct ce duce la defectare totală. De asemenea se poate distribui citirea intensă pe slave-uri.

Dacă această topologie master-master este folosită local, această configurație este încă folosită. Promovarea unui slave ca master este posibilă în cazul defectării, deși este o operație mai complexă. Această operație este la fel de costisitoare ca schimbarea masterului unui slave. Această complexitate este de luat în considerare. ^[1]

Topologia inel

Topologia dual-master este de fapt un caz special al topologiei de replicare tip inel. Un inel are 3 sau mai multe mastere. Fiecare server este slave pentru cel de dinaintea sa în inel și master pentru serverul de după el. Această topologie se mai numește și replicare circulară (fig. 2.9).

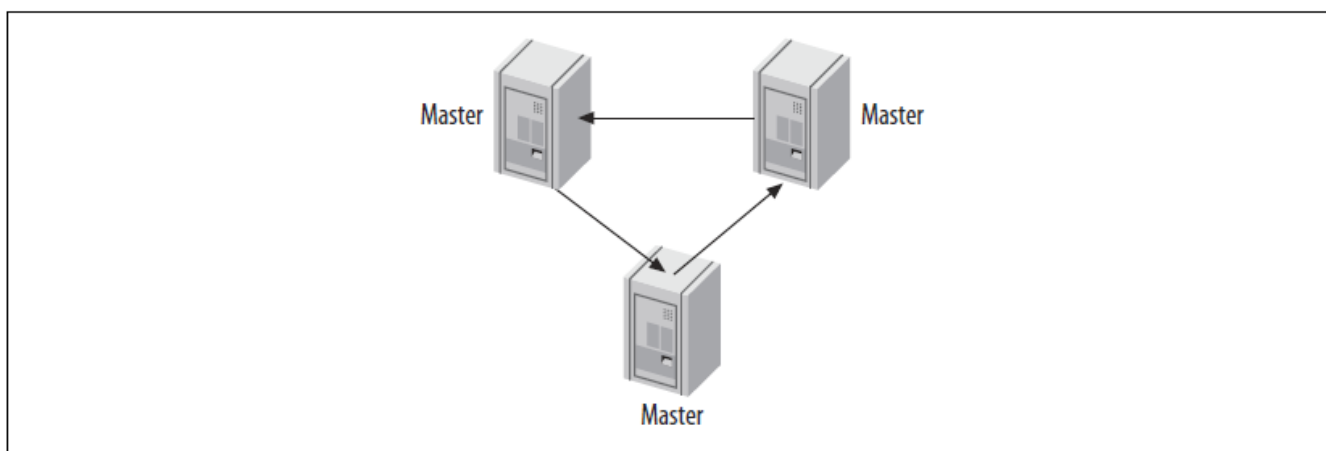


Fig. 3.9 Topologia inel^[1]

Topologia inel nu are câteva beneficii ale topologiei master-master cum ar fi simetria configurației și toleranța la defectare. Rețeaua depinde complet de fiecare nod din inel ceea ce crește șansa de defectare a întregului sistem. Dacă se scoate un nod din inel, evenimentele replicate din el pot intra într-o buclă infinită. Vor circula la infinit prin topologie pentru că singurul server care poate filtra acele evenimente este cel care le-a inițiat. În general inelele sunt de evitat. ^[1]

Se pot evita anumite probleme ale acestei topologii adăugând slave-uri la fiecare nod oferind redundanță ca în figura următoare (fig. 3.10):

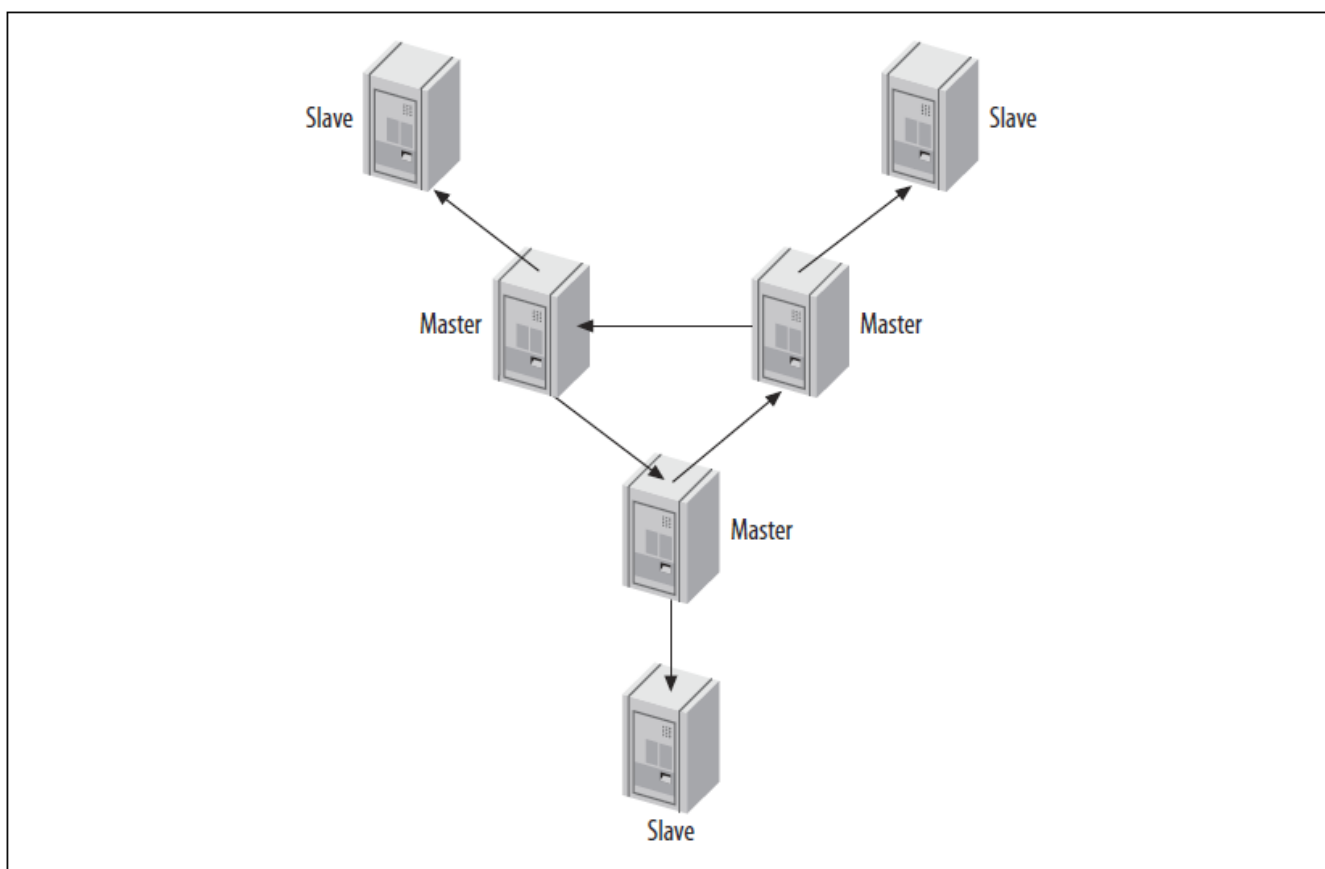


Fig. 3.10 Adăugarea de slave-uri la topologia inel^[1]

Master, distribuția masterului și a slave-urilor

Am menționat anterior că slave-urile pot produce o încărcare pe master dacă sunt destul de multe. Fiecare slave creează un nou fir de execuție pe master, care execută comanda specială *binlog dump*. Această comandă citește datele din „binary log” și le trimite către slave. Acest lucru se repetă pentru fiecare slave. ^[1]

Dacă există multe slave-uri configurate și un „binary log” cu o dimensiune considerabilă cum ar fi un fișier imens *LOAD DATA INFILE*, încărcarea pe master poate crește semnificativ. Masterul poate chiar să rămână fără memorie și să se defecteze pentru că toate slave-urile cer același eveniment cu o dimensiune foarte mare în același timp. Pe de altă parte dacă fiecare slave cere un alt eveniment care nu

există în memoria cache a sistemului de fișiere vor apărea căutări pe disc care pot de asemenea să intervină în performanța masterului. ^[1]

Din acest motiv dacă sunt necesare multe slave-uri este o practică bună înlăturarea încărcării de pe master și folosirea unui master de distribuție. Un master de distribuție este un master al cărui singur scop este să citească și să distribuie „binary log”-urile de pe master. Multe slave-uri se pot conecta la masterul de distribuție care preia încărcarea de pe masterul original. Pentru a bloca executarea evenimentelor pe acest server va trebui modificat motorul de stocare la *Blackhole storage engine* ca în figura următoare (fig. 3.11):

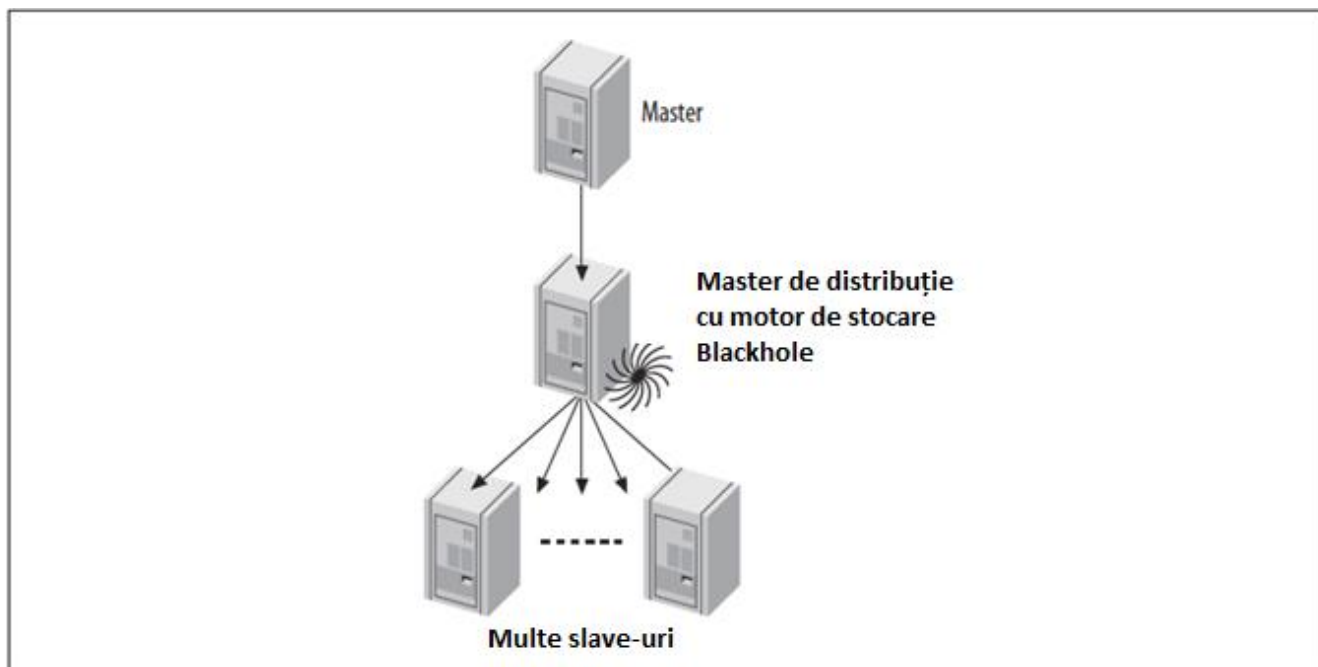


Fig. 3.11 Master de distribuție ^[1]

Este destul de greu de spus câte slave-uri poate susține un master până atunci când va fi nevoie de un master de distribuție. Ca și regulă generală dacă masterul rulează la o capacitate aproape de maxim, nu este indicată conectarea a mai mult de 10 slave-uri la el. Dacă scrierile sunt puține sau se replică doar o parte din tabele masterul foarte probabil poate susține mai mult de 10 slave-uri. În plus poate fi folosit mai mult de un master de distribuție. Pot fi folosite mai multe dacă există un număr mare de slave-uri sau se poate folosi distribuția piramidă a masterelor.

Masterul de distribuție poate fi folosit și în alte scopuri cum ar fi aplicarea filtrelor și rescrierea regulilor evenimentelor în „binary log”. Acest lucru este mai eficient decât repetarea filtrării și a celorlalte operații pe fiecare slave.

Dacă se folosesc tabele Blackhole pe masterul de distribuție el poate susține mai multe slave-uri decât de obicei. Masterul de distribuție va executa query-urile însă ele vor fi deloc costisitoare pentru că motorul de stocare Blackhole nu salvează datele în tabele.

O problemă des întâlnită este cum se asigură că masterul de distribuție folosește motorul de stocare Blackhole. Soluția este setarea lui pe întreg mediu:

Storage_engine = blackhole

Acest lucru va afecta doar instrucțiunile CREATE TABLE care nu specifică direct un motor de stocare. Dacă această problemă nu poate fi controlată mediul poate deveni fragil. Motorul InnoDB poate fi dezactivat însă tabelele vor folosi MyISAM cu opțiunea *skip_innodb*, MyISAM nu poate fi dezactivat.

O altă dificultate ar fi aceea că nu poate fi înlocuit masterul cu unul din slave-urile terminale. Este dificilă promovarea unui slave în locul său pentru că masterul intermediar de distribuție va avea aproape sigur coordonate diferite ale „binary log”-ului decât masterul original. [\[1\]](#)

Arbore sau piramidă

Dacă un master trebuie replicat către un număr mare de slave-uri, chiar dacă rețeaua trebuie distribuită geografic sau se încearcă o creștere a capacității de citire se poate folosi cu succes o proiectare de tip piramidă ca în figură (fig. 3.12):

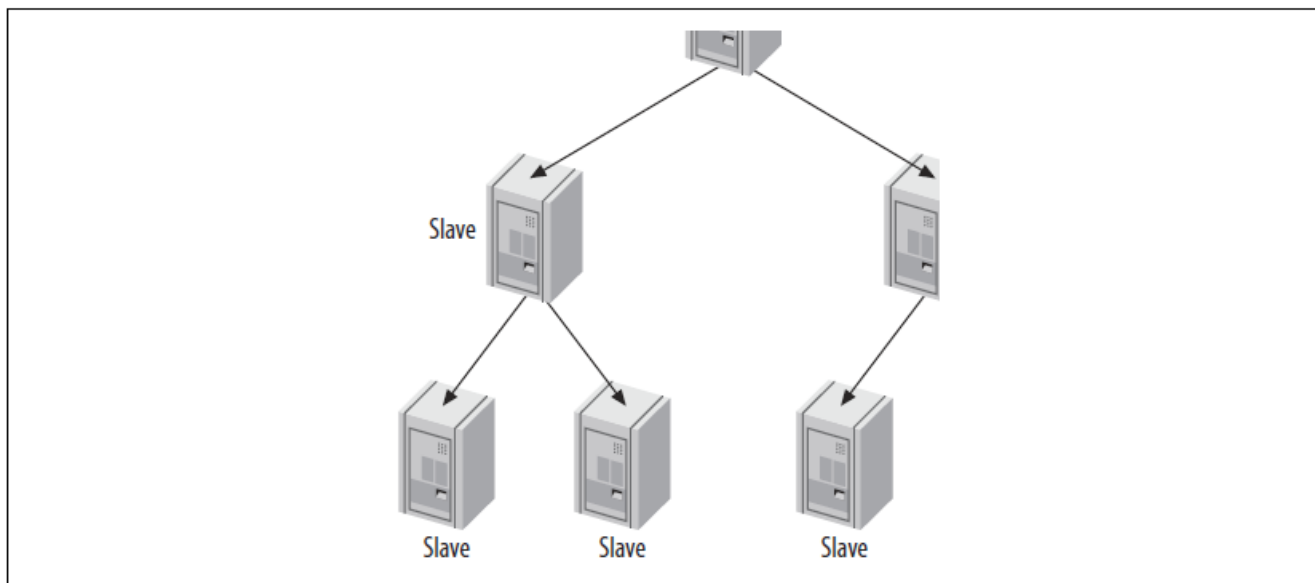


Fig. 3.12 Topologia arbore sau piramidă [\[1\]](#)

Avantajul acestui tip de proiectare este scăderea încărcării pe master la fel cum făcea masterul de distribuție din arhitectura anterioară. Dezavantajul este că orice defectare în nivelurile intermediare va afecta multiple servere aflate mai jos în ierarhie, lucru care nu s-ar fi întâmplat dacă ele erau conectate la master direct. De asemenea, cu cât sunt mai multe niveluri, cu atât este mai complicată repararea defectărilor. [\[1\]](#)

3.2.5 Administrare și mentenanță

Setarea replicării probabil nu este un lucru care se va face constat, doar dacă există multe servere. O dată ce a fost setată, monitorizarea și administrarea replicării topologiei implementate va fi un proces repetitiv, oricât de multe servere ar fi.

Aceste lucruri trebuie automatizate cât mai mult. Se pot folosi programe cum ar fi Nagios, MySQL Enterprise Monitor sau MonYOG. [\[1\]](#)

Monitorizarea replicării

Replicare crește nivelul de complexitate al monitorizării. Deși replicarea se realizează atât pe master cât și pe slave, lucrul intens este desfășurat pe slave și aici apar cel mai des problemele. Toate salvările trebuie să replice corect fără erori și cu întârzieri cât mai mici. Informații privind aceste aspecte sunt generate automat și creșterea performanței este la mâna administratorului. [\[1\]](#)

Pe master se poate folosi comanda SHOW MASTER STATUS care arată poziția curentă în „binary log” și configurarea. Se poate interoga masterul în legătură cu ce „binary log”-uri există pe disc. [\[1\]](#)

```
mysql> SHOW MASTER LOGS;
```

```
+-----+-----+
| Log_name | File_size |
+-----+-----+
| mysql-bin.000220 | 425605 |
| mysql-bin.000221 | 1134128 |
| mysql-bin.000222 | 13653 |
| mysql-bin.000223 | 13634 |
+-----+-----+
```

Această informație este folosită în determinarea parametrilor dați comenzii PURGE MASTER LOGS. Se pot vizualiza evenimente de replicare din „binary log” cu comanda BINLOG EVENTS. Ca exemplu va fi creat un tabel pe un server nefolosit. Pentru ca aceasta este un singur eveniment se va putea vizualiza astfel:

```
mysql> SHOW BINLOG EVENTS în 'mysql-bin.000223' FROM 13634\G
```

```
***** 1. row *****
```

```
Log_name: mysql-bin.000223
```

```
Pos: 13634
```

Event_type: Query

Server_id: 1

End_log_pos: 13723

Info: use `test`; CREATE TABLE test.t(a int) [\[1\]](#)

Măsurarea întârzierii pe slave

Una din cele mai comune lucruri care trebuie monitorizate este cât de întârziat este slave-ul față de master. Chiar dacă coloana *Seconds_behind_master* din SHOW SLAVE STATUS teoretic arată întârzierea slave-ului, nu este întotdeauna corectă din mai multe motive:

- Slave-ul calculează *Seconds_behind_master* comparând timpul de pe server cu cel din „binary log” deci slave-ul nu poate calcula întârzierea dacă procesează un eveniment.
- Slave-ul va raporta NULL dacă procesele de pe slave nu rulează.
- Unele erori sau rețeaua instabilă între master și slave pot împiedica replicarea și pot opri firele de execuție de pe slave și *Seconds_behind_master* va indica 0 și nu o eroare.
- O tranzacție foarte mare poate cauza fluctuație a întârzierii. De exemplu dacă o tranzacție care actualizează date este deschisă timp de o oră și apoi se realizează, întârzierea va fi înregistrată după ce ea a avut loc. Când slave-ul procesează instrucțiuni va raporta că este cu o oră în urma apoi va ajunge din nou la 0.
- Dacă un master de distribuție are slave-uri întârzierea raportată va fi 0 chiar dacă există întârziere relativă la primul master în ierarhie. [\[1\]](#)

Soluția pentru această problemă este ignorarea câmpului *Seconds_behind_master* și măsurarea întârzierii în alt mod prin care se poate observa direct. O soluție este *heartbeat record* (o instanță de timp ce va fi actualizată la fiecare secundă pe master). Pentru a calcula întârzierea se va scădea acest timp din timpul curent de pe slave. Această metodă este tolerantă la toate problemele enumerate mai sus și arată la ce moment de timp datele vor fi sincronizate pe slave. O implementare a replicării *heartbeat* este *pt-heartbeat*, inclusă în *Percona Toolkit*. [\[1\]](#)

Niciuna din aceste metrice nu pot spune cu exactitate cât îi va lua slave-ului să ajungă la curent cu masterul în medie. Acest lucru depinde de o mulțime de factori cum ar fi puterea slave-ului sau câte evenimente de scriere se transmit. [\[1\]](#)

Determinarea consistenței slave-ului cu masterul

Într-o lume perfectă, un slave trebuie să fie o copie exactă a masterului. Dar în realitate, erori în replicare pot cauza ca datele să fie corupte, nesincronizate cu masterul. Chiar dacă aparent nu sunt erori, slave-urile pot deveni nesincronizate pentru că anumite funcționalități nu sunt replicate corect, bug-uri, pierderi pe rețea, încheieri neașteptate ale programului, întreruperea alimentării sau alte defecte.

Aceasta este o regulă și nu o excepție ceea ce înseamnă ca verificarea slave-urilor ca și consistență trebuie să fie un proces de rutină. Acest lucru este în special important dacă replicarea se folosește pentru *backup*, pentru ca nu se dorește coruperea datelor de pe un *backup*. [\[1\]](#)

Nu există o funcționalitate care să verifice acest lucru în sistemul de baze de date. Există câteva sume de control cum ar fi CHECKSUM TABLE dar nu este recomandată verificarea slave-ului cu masterul în timp ce serverele rulează. [\[1\]](#)

Percona Toolkit oferă o metoda care rezolvă această problemă și alte câteva : *pt-table-checksum*. Acest instrument are cateva funcții de luat în seama, comparație în paralel a mai multor servere în același timp, dar cea mai de folos este aceea de a verifica dacă un slave este sincronizat cu masterul. Funcționează folosind instrucțiuni INSERT ... SELECT pe master. [\[1\]](#)

Aceste instrucțiuni fac o suma de control a datelor și inserează rezultatul într-o tabela. Acestea sunt replicate și se execută pe slave. Se pot compara rezultatele de pe master și de pe server și se poate observa o diferență. Deoarece acest proces funcționează prin replicare, nu este nevoie ca tabelele să fie închise (locked) pe ambele servere. [\[1\]](#)

Un mod de folosire al acestui instrument este:

```
$ pt-table-checksum --replicate=test.checksum --chunksize 100000 --sleep-coef=2
```

```
<master_host>
```

Această comandă face suma de control a tuturor tabelelor și încearcă procesarea pe bucăți de aproximativ 100000 de rânduri și inserează valorile în *test.checksum*. Se oprește după fiecare bucată, și așteaptă de doua ori mai mult timp decât a durat execuția bucății anterioare. Acest lucru asigură ca nu vor fi blocate operațiile normale de pe baza de date. [\[1\]](#)

După ce replicarea are loc un simplu query poate verifica dacă slave-ul este în concordanță cu masterul. *pt-table-checksum* poate descoperi slave-urile rulează query-ul pe fiecare server și expune rezultatul automat. Următoarea comandă va pătrunde pe o adâncime de 10 nivele în ierarhia slave-urilor, rulată pe master și va arăta tabelele care diferă de master:

```
$ pt-table-checksum --replicate=test.checksum --replcheck 10 <master_host> \[1\]
```


4. Tehnologii utilizate

4.1 HTML

HTML este o formă de marcare orientată către prezentarea documentelor text pe o singură pagină, utilizând un software de redare specializat, numit agent utilizator HTML, cel mai bun exemplu de astfel de software fiind browser-ul web. HTML furnizează mijloacele prin care conținutul unui document poate fi adnotat cu diverse tipuri de meta date și indicații de redare. Indicațiile de redare pot varia de la decorațiuni minore ale textului, cum ar fi specificarea faptului că un anumit cuvânt trebuie subliniat sau că o imagine trebuie introdusă, până la script-uri sofisticate, hărți de imagini și formulare. Meta datele pot include informații despre titlul și autorul documentului, informații structurale despre cum este împărțit documentul în diferite segmente, paragrafe, liste, titluri etc. și informații cruciale care permit ca documentul să poată fi legat de alte documente pentru a forma astfel hiperlink-uri (sau web-ul). ^[9]

HTML este un format text proiectat pentru a putea fi citit și editat de oameni utilizând un editor de text simplu. Totuși scrierea și modificarea paginilor în acest fel solicită cunoștințe solide de HTML și este consumatoare de timp. Editoarele grafice (de tip WYSIWYG) cum ar fi Macromedia Dreamweaver, Adobe GoLive sau Microsoft FrontPage permit ca paginile web să fie tratate asemănător cu documentele Word, dar cu observația că aceste programe generează un cod HTML care este de multe ori de proastă calitate. ^[9]

HTML se poate genera direct utilizând tehnologii de codare din partea serverului cum ar fi PHP, JSP sau ASP. Multe aplicații ca sistemele de gestionare a conținutului, wiki-uri și forumuri web generează pagini HTML. ^[9]

HTML este de asemenea utilizat în e-mail. Majoritatea aplicațiilor de e-mail folosesc un editor HTML încorporat pentru compunerea e-mail-urilor și un motor de prezentare a e-mail-urilor de acest tip. Folosirea e-mail-urilor HTML este un subiect controversat și multe liste de mail le blochează intenționat. ^[9]

HTML este prescurtarea de la Hyper Text Mark-up Language și este codul care stă la baza paginilor web. Paginile HTML sunt formate din etichete sau tag-uri și au extensia .html sau .htm . În marea lor majoritate aceste etichete sunt pereche, una de deschidere <eticheta> și alta de închidere </eticheta>, mai există și cazuri în care nu se închid, atunci se folosește <eticheta />. Browserul interpretează aceste etichete afișând rezultatul pe ecran. HTML-ul nu este un limbaj case senzitiv (nu face deosebirea între litere mici și mari). Pagina principală a unui domeniu este fișierul index.html respectiv index.htm Această pagină este setată a fi afișată automat la vizitarea unui domeniu. De exemplu la vizitarea domeniului www.numero.ro este afișată pagina www.numero.ro/index.html. ^[9]

Unele etichete permit utilizarea de attribute care pot avea anumite valori: <eticheta atribut="valoare"> ... </eticheta>

Componenta unui document HTML este:

1. versiunea HTML a documentului
2. zona head cu etichetele <head> </head>

3. zona body cu etichetele <body> </body> sau <frameset> </frameset>

Versiunea HTML poate fi:

- HTML 4.01 Strict

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

- HTML 4.01 Transitional

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

- HTML 4.01 Frameset

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
```

- HTML 5

```
<!DOCTYPE HTML>
```

Toate paginile HTML încep și se termină cu etichetele <html> și </html>. În interiorul acestor etichete găsim perechile <head>, </head> și <body>, </body>. ^[9]

head conține titlul paginii între etichetele <title> și </title>, descrieri de tip <meta>, stiluri pentru formatarea textului, scripturi și link-uri către fișiere externe (de exemplu scripturi, fișiere de tip CSS sau *favicon.ico*). ^[9]

Etichetele de tip meta conțin cuvinte cheie, descrierea paginii, date despre autor, informații utile motoarelor de căutare și au următorul format: <META NAME="nume" CONTENT="continut">.

Exemplu: link către un fișier extern CSS: <link rel="stylesheet" type="text/css" href="css.css">.

body găzduiește practic toate etichetele afișate de browser pe ecran. ^[9]

Exemplu: o pagină HTML cu titlul *Exemplu* iar conținutul *Conținut* pagină:

<html> <head> <title>Titlu</title> </head> <body> Continut pagina </body> </html> și în HTML poate fi introdus un comentariu, care bineînțeles nu va fi afișat de browser.

Mai jos sunt tipurile de elemente de marcare în HTML:

- Marcare structurală. Descrie scopul unui text. De exemplu:

```
<h1>Fotbal</h1>
```

Direcționează browserul pentru a reda "Fotbal" ca pe cel mai important titlu. Marcarea structurală nu are un anumit stil predefinit, dar cele mai multe browsere web au standardizat modul în care acestea sunt afișate. De exemplu, titlurile importante (h1, h2, etc.) sunt aldine și mai mari decât restul textului. De notat că "h1" este folosit doar o singură dată per pagină deoarece cu el este marcat titlul ei. ^[9]

- Marcare pentru prezentare. Descrie cum apare un text, indiferent de funcțiile sale. De exemplu:

`îngroșat`

Va afișa textul "îngroșat" cu litere groase, aldine. Html a început în ultimii ani să înceapă să nu mai folosească acest gen de tag-uri pentru că "b" nu dă sens paginii, pe când tag-ul "strong" (adică strong emphasis) dă un înțeles paginii, și mai important, asemenea tag-uri pentru prezentare doar încarcă o pagină cu informații și o fac astfel mai greu de încărcat, iar apoi dacă atașezi un document CSS la pagină, o singură modificare la CSS (de ex: de la "font-weight:italic" la "font-weight:bold" va schimba tot textul selectat, și de exemplu, link-urile vor trece de la text înclinat la text îngroșat, plus că în CSS avem avantajul de a putea preciza cât de mari sau mici să fie literele în pixeli «px», în puncte «pt», etc.)avem același efect ca și când am avea de schimbat toate tag-urile de "i" de pe pagină în tag-uri de "b", muncă care chiar și la un website mic este enormă, ce să mai vorbim de unul de genul wikipedia. Așa că dacă vrei să începeți o carieră în html sau un hobby (și să aveți succes) nu folosiți aceste taguri, nu degeaba s-a inventat CSS-ul. ^[9]

- Marcare pentru hyperlink. Leagă părți ale unui document cu alte documente. De exemplu:

`Wikipedia Românească`

Va reda Wikipedia românească ca hyperlink către un URL specificat.

Elemente speciale (widget). Creează obiecte, cum ar fi butoanele și listele.

Doar marcatorii de prezentare (împreună cu foile de stiluri - CSS) determină cum conținutul din interiorul marcatorului va fi prezentat. Ceilalți marcatori spun browser-ului ce obiecte să redea sau ce funcții să execute. ^[9]

4.2 CSS

CSS (Cascading Style Sheets) este un standard pentru formatarea elementelor unui document HTML. Stilurile se pot atașa elementelor HTML prin intermediul unor fișiere externe sau în cadrul documentului, prin elementul `<style>` și/sau atributul `style`. CSS se poate utiliza și pentru formatarea elementelor XHTML, XML și SVG. ^[9]

CSS3 reprezintă un upgrade ce aduce câteva attribute noi și ajută la dezvoltarea noilor concepte în webdesign. ^[9]

Unele dintre cele mai importante segmente (module) noi adăugate acestui standard pentru formatarea elementelor HTML aduc un plus considerabil în dezvoltarea activității webdesign. ^[9]

Mai jos sunt prezente în listă cele mai importante modulele adăugate în CSS3:

- Selectors - selectoare
- Box Model – model cutie
- Backgrounds and Borders – fundal și margini
- Image Values and Replaced Content – valori imagini și conținut înlocuit

- Text Effects – efecte pe text
- 2D/3D Transformations – transformări 2D/3D
- Animations - animații
- Multiple Column Layout – layout pe multiple coloane
- User Interface - interfața utilizator

Deși au apărut unele deficiențe de compatibilitate între browsere, majoritatea proprietăților CSS3 au fost implementate cu succes în toate variantele browserelor noi. [\[9\]](#)

- CSS3 - Borduri

Acum CSS3 oferă posibilitatea de a crea borduri cu colțurile rotunjite fără a folosi elemente grafice de fundal așa cum se folosea anterior acestui upgrade. [\[9\]](#)

Proprietatea CSS3 *border-radius* definește prin valorile exprimate în pixeli cât de rotunjite vor fi colțurile unui element HTML sau unei imagini. Fiecare colț poate avea o altă valoare exprimată în pixeli diferită de un alt colț al aceluiași element. Prin urmare putem folosi până la 4 valori diferite atribuite unui element HTML sau imagine. [\[9\]](#)

Exemplu:

border-radius: 5px ;

- definește valoarea de 5px radius pentru toate cele 4 colțuri ale elementului.

border-radius: 5px 7px 12px 4px;

- aceste valori multiple definesc cât de mult vor fi rotunjite colțurile elementului HTML, iar pentru fiecare colț este specificată valoarea. Colțul stanga-sus are valoarea border-radius de 5px, colțul dreapta-sus are valoarea border-radius de 7px, colțul dreapta-jos al elementului HTML are valoarea de 12px iar colțul din stanga-jos are valoarea de 4px. [\[9\]](#)

- CSS3 - Borduri Rotunjite - Optimizat

Varianta ne-comprimată sau ne-optimizată:

border-radius-left: 5px;

border-radius-right: 7px;

border-radius-top: 12px;

border-radius-bottom: 4px;

Varianta mimificată, compresată/optimizată:

border-radius: 5px 7px 12px 4px;

Ambele variante sunt corecte și acceptate de clientul browser.

- CSS3 - Borduri Rotunjite - Compatibilitate Browser

Pentru compatibilitatea cu diferite browsere se folosesc prefixe: -webkit- , -moz- , -o-

Compatibilitate: Internet Explorer (IE) - 0.9 , Chrome folosește prefixul -webkit- pentru 4,0 , Firefox folosește prefixul -moz- pentru versiunea 3.0, Safari folosește prefixul -webkit- pentru versiunea 3.1, Opera 10.5 prefix -o-

Exemplu CSS3 border-radius:

```
div {  
  
border: 2px solid #333333;  
  
padding: 10px 40px;  
  
background: #dddddd;  
  
width: 300px;  
  
border-radius:25px;  
  
}
```

Elementul HTML div este definit de următoarele proprietăți CSS: dimensiunea în lungime este redată de valoarea în pixeli a proprietății width, folosește o bordură de 2 pixeli, o bordură solidă de culoare gri-închis definită de valoarea HEX #333333. Culoarea de fundal este gri deschis definită de HEX #dddddd. Bordura rotunjită este de 25 pixeli pentru toate cele 4 colțuri. [\[9\]](#)

Pentru interfața grafică ce folosește CSS s-a folosit în aplicația proiectată tehnologia Bootstrap care oferă elemente și design modern testat și ușor de integrat în aplicațiile web. [\[9\]](#)

4.3 JavaScript

JavaScript (JS) este un limbaj de programare orientat obiect bazat pe conceptul prototipurilor. Este folosit mai ales pentru introducerea unor funcționalități în paginile web, codul Javascript din aceste pagini fiind rulat de către browser. Limbajul este bine-cunoscut pentru folosirea sa în construirea site-urilor web, dar este folosit și pentru accesul la obiecte încastate (embedded objects) în alte aplicații. A fost dezvoltat inițial de către Brendan Eich de la Netscape Communications Corporation sub numele de Mocha, apoi LiveScript, și denumit în final JavaScript. [\[9\]](#)

În ciuda numelui și a unor similarități în sintaxă, între JavaScript și limbajul Java nu există nicio legătură. Ca și Java, JavaScript are o sintaxă apropiată de cea a limbajului C, dar are mai multe în comun cu limbajul Self decât cu Java. [\[9\]](#)

Până la începutul lui 2005, ultima versiune existentă a fost JavaScript 1.5, care corespunde cu Ediția a 3-a a ECMA-262, ECMAScript, cu alte cuvinte, o ediție standardizată de JavaScript. Versiunile de Mozilla începând cu 1.8 Beta 1 au avut suport pentru E4X, care este o extensie a limbajului care are de a face cu XML, definit în standardul ECMA-357. Versiunea curentă de Mozilla, 1.8.1 (pe care sunt construite Firefox și Thunderbird versiunile 2.0) suportă JavaScript versiunea 1.7. [\[9\]](#)

Utilizare

Cea mai des întâlnită utilizare a JavaScript este în scriptarea paginilor web. Programatorii web pot îngloba în paginile HTML script-uri pentru diverse activități cum ar fi verificarea datelor introduse de utilizatori sau crearea de meniuri și alte efecte animate. ^[9]

Browselele rețin în memorie o reprezentare a unei pagini web sub forma unui arbore de obiecte și pun la dispoziție aceste obiecte script-urilor JavaScript, care le pot citi și manipula. Arborele de obiecte poartă numele de Document Object Model sau DOM. Există un standard W3C pentru DOM-ul pe care trebuie să îl pună la dispoziție un browser, ceea ce oferă premisa scrierii de script-uri portabile, care să funcționeze pe toate browselele. În practică, însă, standardul W3C pentru DOM este incomplet implementat. Deși tendința browserelor este de a se alinia standardului W3C, unele din acestea încă prezintă incompatibilități majore, cum este cazul Internet Explorer. ^[9]

O tehnică de construire a paginilor web tot mai întâlnită în ultimul timp este AJAX, abreviere de la „Asynchronous JavaScript and XML”. Această tehnică constă în executarea de cereri HTTP în fundal, fără a reîncărca toată pagina web, și actualizarea numai anumitor porțiuni ale paginii prin manipularea DOM-ului paginii. Tehnica AJAX permite construirea unor interfețe web cu timp de răspuns mic, întrucât operația (costisitoare ca timp) de încărcare a unei pagini HTML complete este în mare parte eliminată. ^[9]

4.4 jQuery

jQuery este o platformă de dezvoltare JavaScript, concepută pentru a ușura și îmbunătăți procese precum traversarea arborelui DOM în HTML, managementul inter-browser al evenimentelor, animații și cereri tip AJAX. jQuery a fost gândit să fie cât mai mic posibil, disponibil în toate versiunile de browsere importante existente, și să respecte filosofia "Unobtrusive JavaScript". Biblioteca a fost lansată în 2006 de către John Resig. ^[9]

jQuery se poate folosi pentru a rezolva următoarele probleme specifice programării web:

- selecții de elemente în arborele DOM folosind propriul motor de selecții open source Sizzle, un proiect născut din jQuery
- parcurgere și modificarea arborelui DOM (incluzând suport pentru selectori CSS 3 și XPath simpli)
- înregistrarea și modificarea evenimentelor din browser
- manipularea elementelor CSS
- efecte și animații
- cereri tip AJAX
- extensii (plugins)
- utilități : versiunea browser-ului, funcția each.

Cunoscutul program "Hello world" în jQuery.

```
$(document).ready(function(){  
    $('body').html('Hello world!');  
});
```

Plugin-urile sau extensiile sunt unele dintre cele mai interesante aspecte ale jQuery. Arhitectura sa permite programatorilor să dezvolte sub-aplicații bazate în biblioteca principală care extind funcțiile de bază jQuery cu funcții specifice plugin-ului. În acest fel biblioteca principală poate ocupa foarte puțin spațiu, iar extensiile necesare în anumite pagini web pot fi încărcate la cerere, doar când este nevoie de ele. Există un set de extensii principal numit jQuery UI (jQuery User Interface). jQuery UI oferă un set de extensii pentru interactivitate de bază, efecte mai complexe decât cele din biblioteca de bază și teme de culori. Avantajul jQuery UI față de alte extensii este că dezvoltarea și testarea acestor componente se face în paralel cu dezvoltarea bibliotecii principale, minimizând riscul de incompatibilitate. [\[9\]](#)

Orice programator poate crea o extensie și jQuery oferă publicare în catalogul de pe pagina proiectului în diversele categorii disponibile. [\[9\]](#)

4.5 Java

4.5.1 Introducere

Java este un limbaj de programare de nivel înalt, dezvoltat în anul 1995 de către SUN Microsystems, și utilizat astăzi pe scara largă atât în aplicațiile desktop sau internet cât și pentru cele destinate dispozitivelor mobile. [\[7\]](#)

Java a fost de la bun început un limbaj gândit pentru ușurința și eficiența creării de software, eliminând neajunsurile altor limbaje (de ex. C++). Calitatea sa principală este independența de platformă - un program scris o dată poate fi rulat pe diverse arhitecturi (Intel x86, SPARC etc.) fără a fi necesară ajustarea lui. [\[7\]](#)

Dintotdeauna, una din problemele majore ale limbajelor a fost portabilitatea. Java rezolvă această problemă prin faptul că programele, odată compilate, generează cod mașină care nu este destinat direct procesorului calculatorului gazdă, ci unui procesor virtual Java, simulat de către un software ce poartă numele de Java Virtual Machine (JVM). JVM „traduce” codul mașină Java în instrucțiuni înțelese de către procesorul gazdă. Existând câte o variantă JVM pentru toate sistemele de operare și arhitecturile de procesor - însă toate simulând același procesor virtual Java! - codul mașină Java este practic independent de platforma și arhitectură (fig. 4.1). [\[7\]](#)

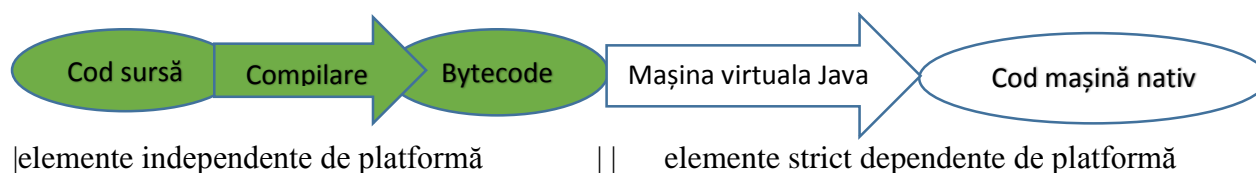


Fig. 4.1 Fluxul codului Java

1. Platforme disponibile

Date fiind direcțiile multiple de utilizare Java și necesitățile lor diferite, Oracle oferă în acest moment trei platforme Java:

- **Java SE** (Standard Edition) - platformă Java pentru aplicații desktop sau server. Cuprinde componentele necesare pentru orice aplicație Java, inclusiv design de interfețe grafice complexe. [\[7\]](#)
- **Java EE** (Enterprise Edition) - platforma Java dedicată aplicațiilor server, cu grad sporit de complexitate. Java EE include resursele disponibile în Java SE și adaugă elemente necesare serverelor de aplicație folosite pentru aplicații distribuite. [\[7\]](#)
- **Java ME** (Micro Edition) - platformă Java dedicată dispozitivelor portabile sau cu resurse limitate (telefoane mobile, PDA-uri etc). Platforma include facilități built-in de adaptare a platformei Java la platforma hardware pe care va rula, facilități de comunicare în rețea și de realizare de interfețe grafice etc. [\[7\]](#)

2. Crearea unui program - Compilare/Interpretare

Pentru a putea scrie și rula programe Java în mod eficient, programatorul are nevoie de următoarele resurse:

- **Java Runtime Environment** (JRE) - reprezintă totalitatea resurselor necesare pentru a putea rula programe Java. Printre ele se numără mașina virtuală și un set minimal de clase predefinite Java, corespunzătoare operațiilor des întâlnite în programare (ex: lucrul cu șiruri de caractere, operații matematice uzuale etc). [\[7\]](#)
- **Java Development Kit** (JDK) - pachetul conține resursele necesare creării de programe Java: compilator, generator de documentație, arhivator și o întreagă serie de alte utilitare implicate direct sau indirect în procesul de dezvoltare. În general JDK conține și JRE, pentru a putea rula aplicațiile create. [\[7\]](#)
- **un mediu de dezvoltare Java** (IDE - Integrated Development Environment). Deși codul Java poate fi scris folosind orice editor de text, un mediu de dezvoltare oferă multiple facilități care sporesc eficiența programatorului (un editor avansat, managementul facil al proiectelor și resurselor atașate acestora, debugging avansat etc). Putem privi IDE-ul ca pe o interfață la uneltele puse la dispoziție de JDK; IDE-ul se folosește de ele și nu poate funcționa în lipsa lor. [\[7\]](#)

Sunt disponibile două categorii de medii de dezvoltare:

- medii de dezvoltare open-source - sunt softuri public disponibile, care pot fi descărcate și rulate de către orice utilizator. Proeminente sunt NetBeans (care beneficiază de sprijin din partea Sun și ulterior Oracle) și Eclipse (proiect inițiat și sprijinit de IBM)

- medii de dezvoltare proprietare - sunt softuri disponibile contra cost. Exemple: JBuilder, IntelliJ Idea, MyEclipse etc.

Atât JRE cât și JDK sunt public disponibile pe <http://www.oracle.com/technetwork/java/javase/downloads/index.html> . JDK poate fi descărcat în cel puțin doua variante – de sine stătător („stand-alone”) sau la pachet cu mediul de dezvoltare NetBeans („bundle”). ^[7]

Imaginea de ansamblu:

Programele scrise în limbajul Java iau forma unui ansamblu de fișiere text cu extensia *.java*, ce conțin așa-numitul *cod sursă* (instrucțiunile Java ce compun programul). Aceste fișiere sunt apoi compilate cu ajutorul compilatorului *javac*, rezultând un set de instrucțiuni în cod mașină ce se va executa pe procesorul virtual Java și care poartă denumirea de *bytecode*. Bytecode-ul este plasat într-un fișier având același nume cu cel sursă, dar cu extensia *.class*. La rulare, mașinii virtuale i se pasează ca argument numele clasei ce reprezintă punctul de intrare al programului; mașina virtuală încarcă *bytecode*-ul clasei corespunzătoare (fișierul *.class*) și executa codul cuprins în fișier (fig. 4.2). ^[3]

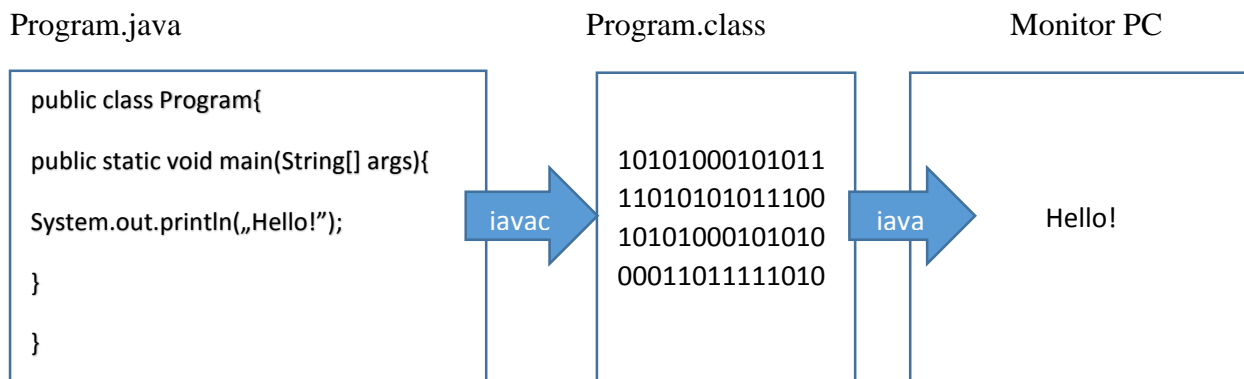


Fig. 4.2 Programul care afișează „Hello”

Conform imaginii de mai sus, deducem ca Java este atât un limbaj compilat (codul sursă este transformat în *bytecode*) cât și unul interpretat (*bytecode*-ul este interpretat de către mașina virtuală Java astfel încât să poată fi executat pe procesorul gazdă). ^[3]

4.5.3 Sintaxa Java

Java este un limbaj concurrent orientat pe clase și obiecte și special proiectat pentru a avea cât mai puține dependențe. În 2014 este cel mai utilizat limbaj de programare cu peste 9 milioane de programatori. Intenția este ca programatorii să poată scrie o dată și să ruleze oriunde („write once; run anywhere”). Limbajul este asemănător cu cel al C/C++ dar având mai puține funcționalități de nivel jos decât ele. ^[3]

4.5.4 OOP

Java a fost creat ca un limbaj obiect orientat încă de la început. Programarea în Java se realizează folosind ca structuri de date clasele și instanțele acestora (obiectele). Un principiu important în proiectarea orientată pe obiecte este moștenirea. Moștenirea presupune derivarea unei clase dintr-o clasă deja existentă, această nouă clasă păstrând funcționalitățile clasei pe care o moștenește. Acest principiu reduce duplicarea de cod.

Un alt principiu OOP este încapsularea. Acest principiu se respect atunci când toate atributele unei clase sunt declarate ca private (cu acces doar din clasa de care aparțin), accesul din exterior fiind făcut doar prin funcții *getter* și *setter*, metode care limitează citirea și scrierea acelei variabile exact cum acestea au fost gândite de către programator. [\[3\]](#)

4.5.5 Integrarea cu baza de date

Integrarea cu baza de date în mod clasic se face în Java folosind librăria JDBC. În aplicația proiectată s-a folosit JPA și Hibernate pentru a ușura accesul la baza de date. Întotdeauna a fost o problema trecerea datelor din format tabelar în obiecte. De aceea au fost create librăriile ORM (object relational mapping) care permit accesul mai ușor la baza de date din limbajul de programare. În Java există specificația JPA care este o serie de interfețe ce definește această mapare. Pentru această interfață există o serie de implementări diferite printre care cea mai cunoscută Hibernate. [\[6\]](#)

5. Proiectarea aplicației

Fluxul datelor în aplicație este reprezentat în figura următoare (fig. 5.1):

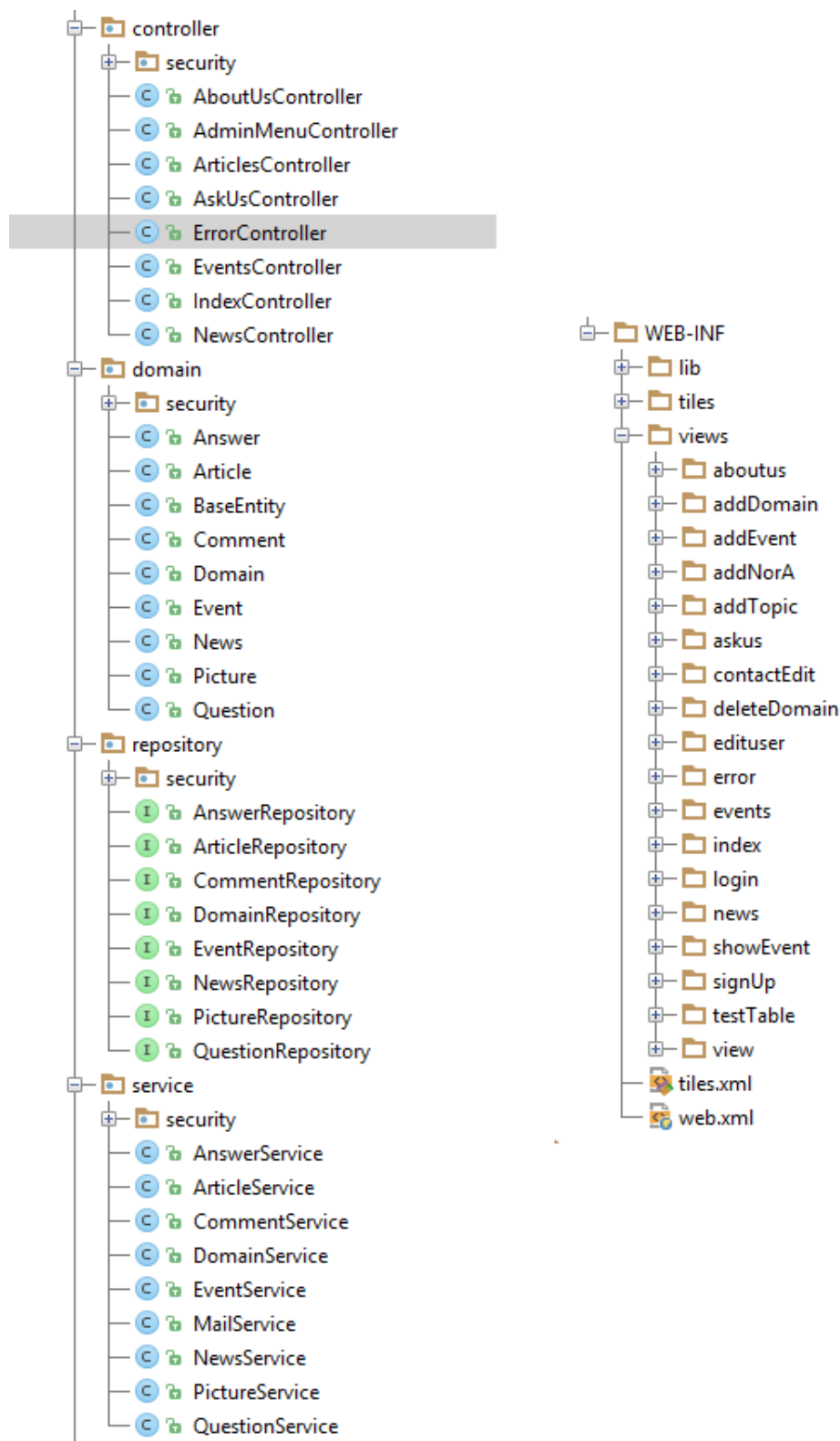


Fig. 5.1 Pachetele Java folosite si paginile JSP

Aplicația respectă modelul de proiectare (design pattern) MVC (Model-View-Controller). Astfel Fluxul datelor va trece prin următoarele pachete Java:

- domain – este pachetul Java care reprezintă modelul de date pentru aplicație. Aici fiecare tabela din baza de date are o clasa Java ORM corespondentă.
- repository – este pachetul Java care utilizează modulul Spring Data pentru a rula query-uri în limbajul HQL (Hibernate Query Language). Acest limbaj este similar cu SQL însă query-urile se realizează utilizând clase Java, rezultatul fiind obiecte Java modelate în pachetul anterior.
- service – este pachetul Java care apelează metodele din pachetul repository manipulează datele primite sau implementează anumite funcționalități (trimitere e-mail, scriere pe disc).
- controller – este pachetul Java care interceptează cererile primite de la utilizator în funcție de URL-ul cerut și răspunde corespunzător cu o pagina web sau date în format JSON.
- views – este directorul destinat paginilor JSP care creează conținut HTML în mod dinamic folosind datele returnate de controller.

Fluxurile de date pentru pagina de întrebări și răspunsuri și cea de știri pot fi vizualizate în anexă. [\[4\]](#)

5.1 Prezentare generala a interfeței

Aplicația dezvoltată este un sistem pentru gestionarea de știri, articole, evenimente, întrebări și răspunsuri din domeniul medical (stomatologie și medicina generală). Interfața conține pagini de vizualizare a acestor date pentru utilizatorii normali (users) și o secțiune de administrare a datelor pentru utilizatorii cu drepturi de editare și ștergere (admins). Meniul aplicației în limba română este următorul (fig. 5.2):



Fig. 5.2 Meniul aplicației

5.1.1 Secțiunea de autentificare



Fig. 5.3 Formularul de autentificare

Aplicația este disponibilă pentru două tipuri de utilizatori: administratori (*admin*) și utilizatori normali (*user*). Autentificarea în aplicație (fig. 5.3) se poate face în orice moment în colțul din dreapta sus al paginii. Dacă utilizatorul încearcă să acceseze o pagina ce poate fi vizualizată doar dacă el deține unul dintre aceste roluri de autentificare va fi redirecționat către o pagina de înregistrare.

Butonul *Cont nou* oferă posibilitatea oricărui vizitator al site-ului să își creeze un cont pentru a avea acces la mai multe funcționalități ale site-ului devenind utilizator de tip *user*.

Rolul de administrator poate fi oferit unui utilizator normal doar de către un alt administrator din meniul special accesibil doar utilizatorilor de tip *admin*. Meniul disponibil doar administratorilor (fig. 5.4):

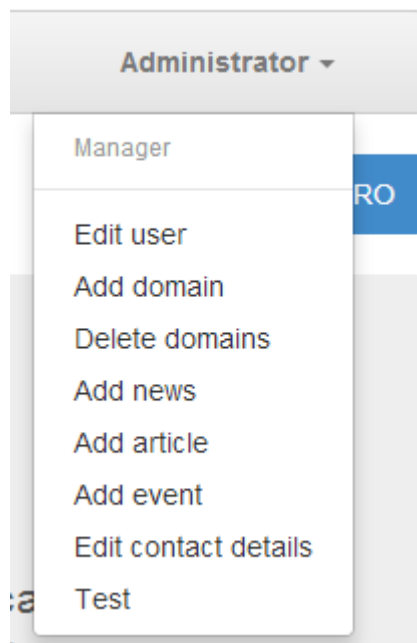


Fig. 5.4 Meniul administratorului

5.1.2 Secțiunea de știri și articole

Secțiunea de știri și articole (fig. 5.5) oferă informații vizitatorilor site-ului. Cele mai noi știri sau articole din domeniile selectate din manșeta din partea stânga a ecranului vor apărea în partea dreapta. Ele sunt accesibile oricărui tip de utilizator și se încarcă în calupuri de cate 10 știri. O dată ce utilizatorul ajunge la sfârșitul paginii se vor încărca încă 10 știri/articole pana când se va ajunge la final. Accesarea unei știri este disponibilă oricărui tip de vizitator însă vizibilitatea textului este limitată. Dacă utilizatorul nu are cont pe site va putea vizualiza doar prima parte a conținutului, el fiind redirecționat către pagina de autentificare în cazul în care dorește accesarea întregului conținut. De asemenea orice comentariu la o știre/ un articol este constrânsă de obținerea unui cont de utilizator.



Fig. 5.5 Vizualizarea știrilor sau articolelor

5.1.3 Secțiunea evenimente

Secțiunea de evenimente este asemănătoare cu cea de articole și știri însă evenimentele nu sunt sortate pe categorii. Vizualizarea lor este constrânsă la fel ca la secțiunea anterioară (fig. 5.6).

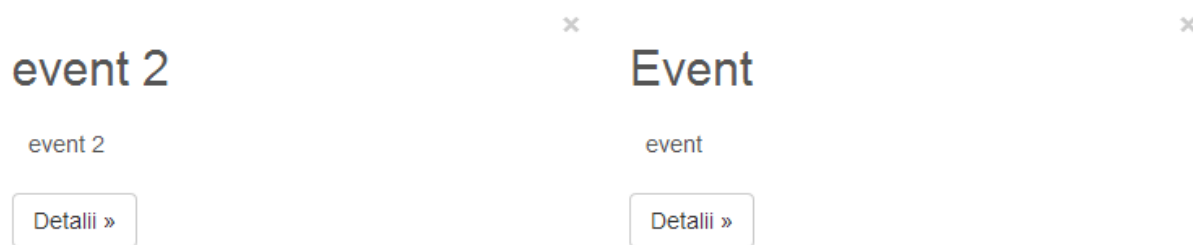


Fig. 5.6 Vizualizarea de evenimente

5.1.4 Secțiunea de întrebări și răspunsuri

Secțiunea de întrebări și răspunsuri (fig. 5.8) este dedicată tot utilizatorilor autentificați. Orice utilizator poate pune o întrebare la care poate aștepta răspuns din partea administratorilor sau a altui utilizator autentificat. Înregistrarea unei întrebări noi presupune trimiterea unui e-mail tuturor administratorilor pentru a îi înștiința de existența unei noi întrebări.

Utilizatorii de tip administrator pot șterge orice întrebare sau răspuns ce care poate avea un conținut neadekvat.

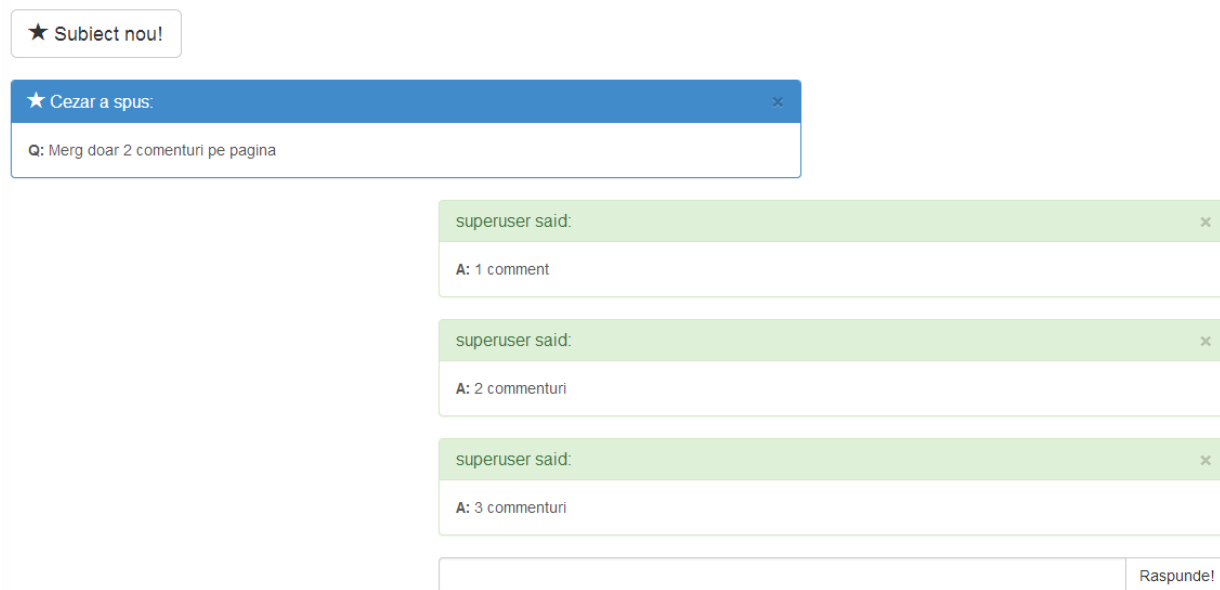


Fig. 5.8 Secțiunea întrebări și răspunsuri

5.1.5 Secțiunea de administrator

Secțiunea de administrator este un meniu special, disponibil numai utilizatorilor administratori cu drepturi de ștergere și editare. Acest meniu conține următoarele secțiuni:

- Editare utilizatori
- Adăugare categorie (articol/știre)
- Ștergere categorie
- Adăugare știre
- Adăugare articol
- Adăugare eveniment
- Editarea informațiilor de contact
- Pagina de test replicare

Secțiunea editare utilizatori afișează toți utilizatorii înregistrați și permite ștergea sau promovarea lor ca administratori.

Secțiunile de adăugare conțin formulare pentru adăugare de conținut. Formularele conțin reguli de validare pe câmpuri pentru a evita introducerea de date nedorite.

Secțiunea ștergere categorie prezintă lista de categorii disponibile și cate un buton de ștergere pentru fiecare din ele.

Pagina de editare a informațiilor de contact permite schimbarea acestor informații în cazul în care acestea vor diferi de-a lungul ciclului de viață al aplicației.

5.2 Structura bazei de date

În următoarea figură (fig. 5.9) este prezentată structura bazei de date în forma normală 3 și relațiile între tabele:

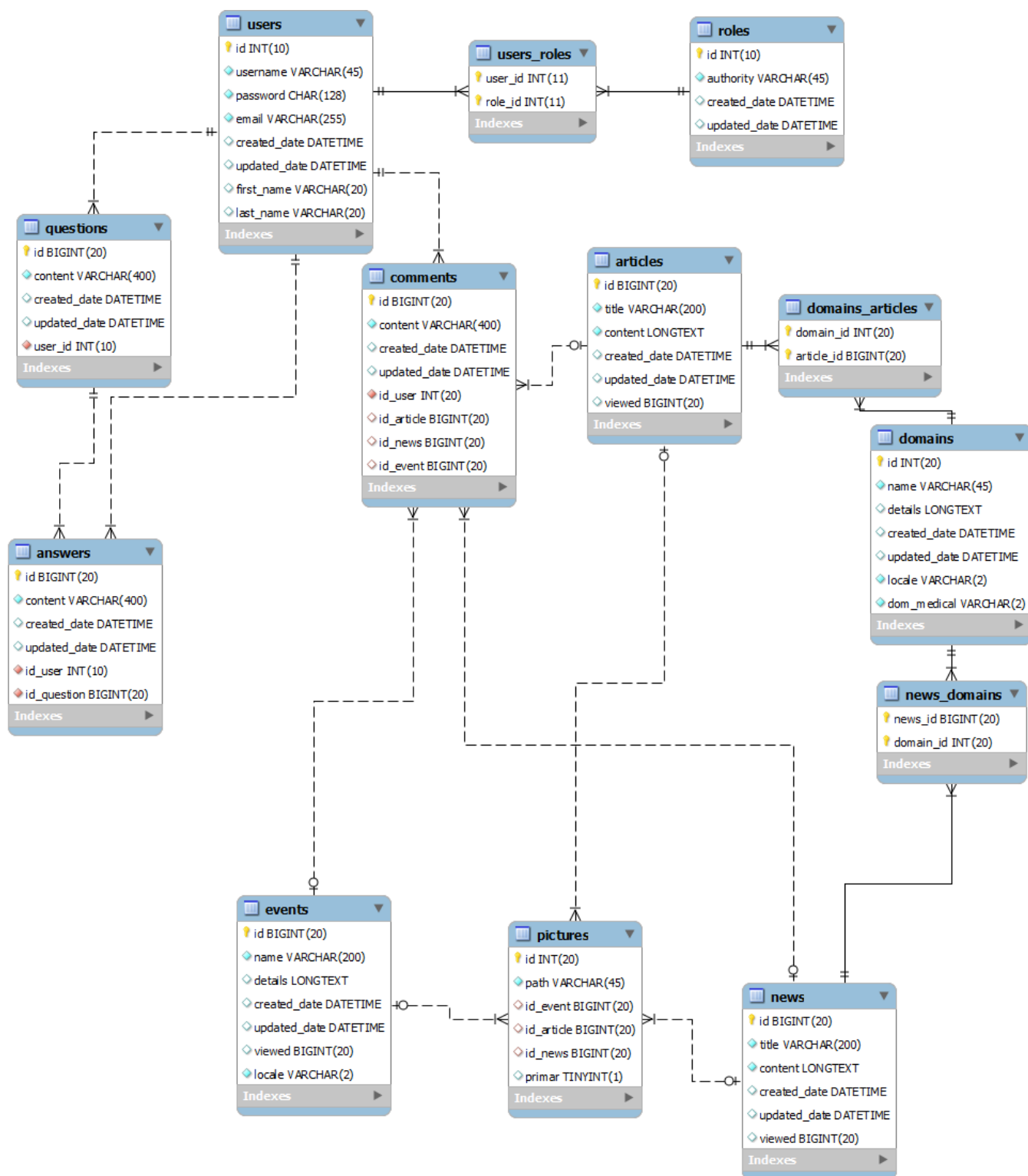


Fig. 5.9 Schema bazei de date

5.3 Arhitectura de replicare implementată

Deoarece pentru a realiza o arhitectură complexă distribuită este nevoie de un suport hardware puternic și costisitor, a fost aleasă o soluție virtualizare în acest sens. Proiectele de tip *cloud-computing* au luat amploare în ultima perioadă de timp, distribuitori de servicii precum IaaS (*Infrastructure as a Service* – Infrastructura ca Serviciu) sau PaaS (*Platform as a Service* – Platforma ca Serviciu) existând în număr destul de mare. Din acest motiv a fost aleasă platforma Windows Azure de tip IaaS ce oferă servicii de virtualizare în *cloud*.

A fost aleasă o platformă de tip IaaS deoarece oferă o flexibilitate mai mare decât cele de tip PaaS.

Utilizând aceste servicii a fost creată o rețea virtuală de calculatoare cu IP-uri în intervalul (172.16.0.4 -> 172.16.0.8). Fiecare dintre mașinile virtuale din rețea conține sistemul de operare Ubuntu Server 14.04 LTS. Configurarea fiecărei mașini a fost făcută prin protocolul de acces la distanță SSH.

Arhitectura rețelei de replicare tip arbore este următoarea (fig. 5.10):

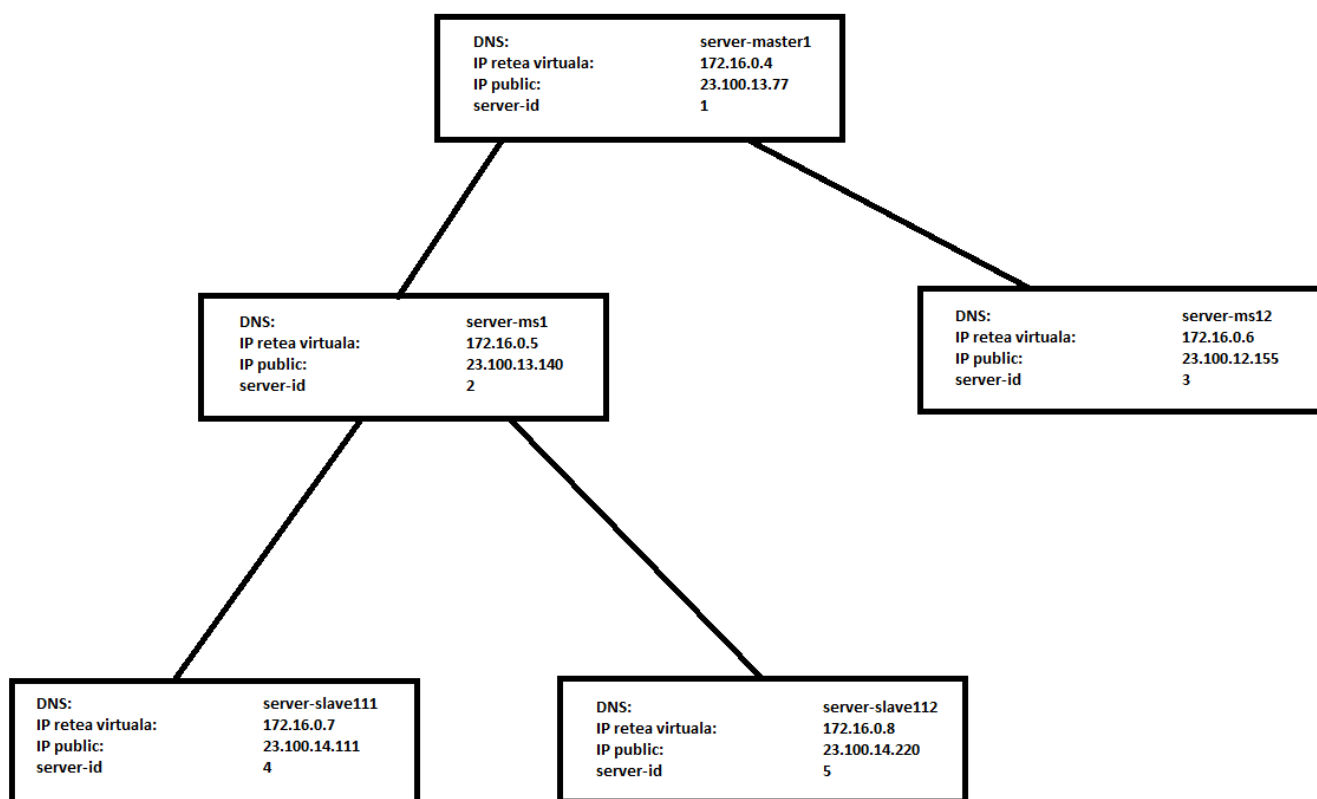


Fig. 5.10 Rețeaua de replicare implementată

Vârful arborelui conține o instanță de Tomcat v. 7.0 care găzduiește aplicația web proiectată și o instanță de server MariaDB folosită de către aplicația web. Fiecare nod copil conține câte o instanță de server MariaDB care se conectează la serverul de baze de date părinte și replică datele acestuia.

5.4 Testarea replicării

În meniul de administrare există pagina de testare replicare (fig. 5.11). Aceasta este creată în scop didactic pentru a vizualiza datele din tabela *comments* (comentarii) din baza de date curentă și un server ales.

The interface consists of three input fields at the top for connection details: 'localhost:3306', 'root', and 'root'. Below these is a blue 'Post' button. The main area is divided into two columns: 'Master' and 'Slave'. The 'Master' column contains a table with three rows of data: 'frumos', 'si mai frumos', and 'da'. The 'Slave' column is currently empty.

Master	Slave
frumos	
si mai frumos	
da	

Fig. 5.11 Testarea replicării

Pagina conține un formular pentru a seta informațiile de conexiune la serverul de baze de date ce replică baza de date curentă. În stânga jos se poate vizualiza tabela *comments* de pe serverul master absolut iar în partea din dreapta jos se vor vizualiza datele din aceeași tabelă de pe serverul slave ales după apăsarea butonului de *Post*.

Astfel se poate verifica dacă replicarea a fost făcută corect pe oricare din noduri.

6. Concluzii

Aplicațiile web care folosesc un framework de dezvoltare web (Spring, Rails, Django) sunt relativ ușor de implementat și sunt compatibile cu orice platformă datorită accesării lor prin intermediul unui browser. Astfel programatorul poate merge către o piață cât mai mare de utilizatori scriind cod o singură dată. De exemplu platforme ca cele de tipul PhoneGap oferă integrarea unei aplicații web mobile ca și aplicație nativă pentru aproape orice tip de dispozitiv mobil (Android, iOS, etc.).

Aplicațiile web oferă o flexibilitate mare atât în alegerea modului de stocare a datelor, în special a bazei de date cât și în alegerea diferitelor implementări de librării ce vor fi folosite în timpul dezvoltării. Având această libertate programatorul poate decide care este cea mai bună opțiune pentru el în cazul bazei de date. Dacă datele sunt structurate poate alege o bază de date relațională, dacă datele nu au un format bine stabilit poate merge către NoSQL.

În cazul librăriilor Java oferă cele mai multe librării scrise și bug-uri în număr destul de mic, ele ajungând la o maturitate deoarece există de o bună perioadă de timp pe piață. Însă framework-urile de tipul Rails (ce folosesc limbajul de programare Ruby) sau Grails (inspirat din Rails ce folosește limbajul de programare Groovy) este posibil să aibă un real succes în anii ce urmează deoarece automatizează multe procese de dezvoltare care devin repetitive (interfețe pentru operații de tip CRUD – create, read, update, delete – creare, citire, actualizare, ștergere). Însă acest lucru poate oferi flexibilitate mai mică.

Respectarea unei arhitecturi de tip MVC duce la o bună structurare a aplicației și la o bună înțelegere a celor care vor vrea să modifice aplicația în viitor.

Din punct de vedere al bazelor de date se observă multitudinea de opțiuni existente pe piață care pot chiar să copleșească programatorul, opțiunile fiind destul de multe, fiecare cu avantajele și dezavantajele sale. Baza de date studiată în această lucrare poate avea un real succes în viitor deoarece are un limbaj de interogare cu care mulți programatori sunt familiari. Ea a fost deja adoptată de Google în defavoare produsului Oracle MySQL, trecându-și produsele de la MySQL la MariaDB. De asemenea produsele de tip *open source* s-au bucurat dintotdeauna o încredere mai mare din partea programatorilor.

Faptul că MariaDB este aproape 100% compatibilă cu MySQL și în multe cazuri poate avea performanțe mai bune va atrage mulți programatori. Din punct de vedere al capacităților NoSQL, nu pare a fi cea mai bună soluție, decizia fiind destul de clară de a merge către un alt tip de bază de date cum ar fi MongoDB, în cazul în care se va lucra cu date greu de structurat.

Din punct de vedere al replicării MariaDB se comportă foarte bine crearea unor topologii de replicare fiind la îndemâna programatorului prin modificarea câtorva fișiere de configurare și setarea anumitor parametrii. În timpul implementării topologiei de replicare nu au fost găsite diferențe notabile față de MySQL. Topologiile ce au avut succes în MySQL pot fi folosite cu succes în MariaDB opțiunea cea mai bună fiind la alegerea programatorului în funcție de aplicația ce trebuie realizată.

7. Bibliografie

- [1] High Performance MySQL (3rd Edition), Editura O'Reilly, Baron Schwartz, Peter Zaitsev și Vadim Tkachenko, 2012
- [2] Baze de date relaționale și aplicații, Editura Tehnica, Felicia Ionescu, 2004
- [3] Java How to Program (9th Edition), Editura Deitel, Paul Deitel și Harvey Deitel, 2012
- [4] Note de curs, Inginerie software, Ștefan Stăncescu
- [5] Spring in action, Editura Manning, Craig Walls, 2011
- [6] Pro JPA 2, Editura Apress, Mike Keith și Merrick Schincariol, 2013
- [7] Java Oracle, <http://www.oracle.com/technetwork/topics/newtojava/unravelingjava-142250.html>
- [8] Java API, <http://docs.oracle.com/javase/6/docs/api>
- [9] HTML, CSS, Javascript, jQuery - <http://www.wikipedia.com>
- [10] HTML, CSS, Javascript, jQuery - <http://www.w3schools.com>
- [11] MariaDB - <https://mariadb.com/kb/en/>

8. Anexe

Diagrame UML pentru fluxul datelor în secțiunea întrebări și răspunsuri, știri (fig. 8.1, 8.2).

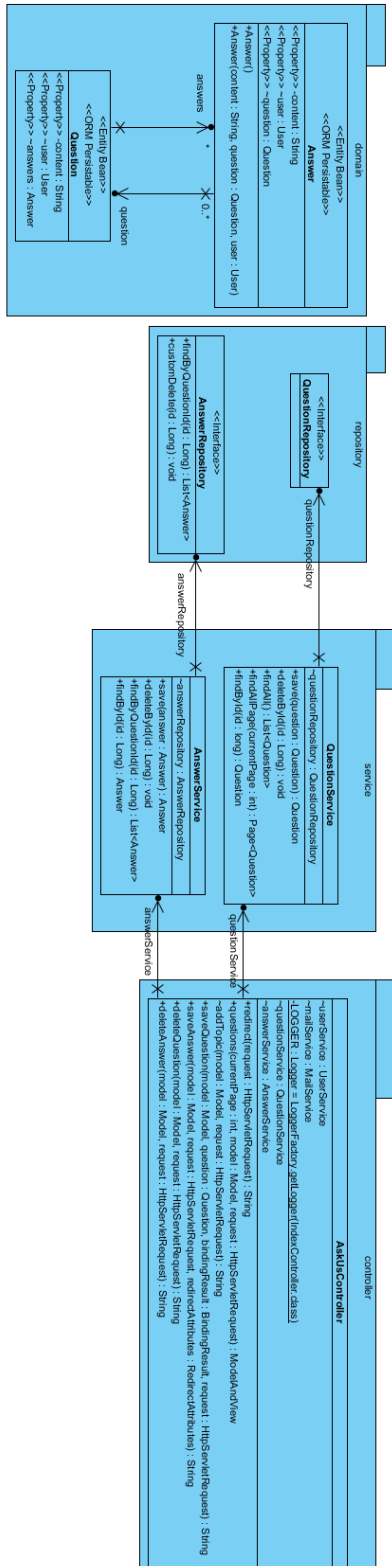


Fig. 8.1 Fluxul datelor în secțiunea întrebări și răspunsuri

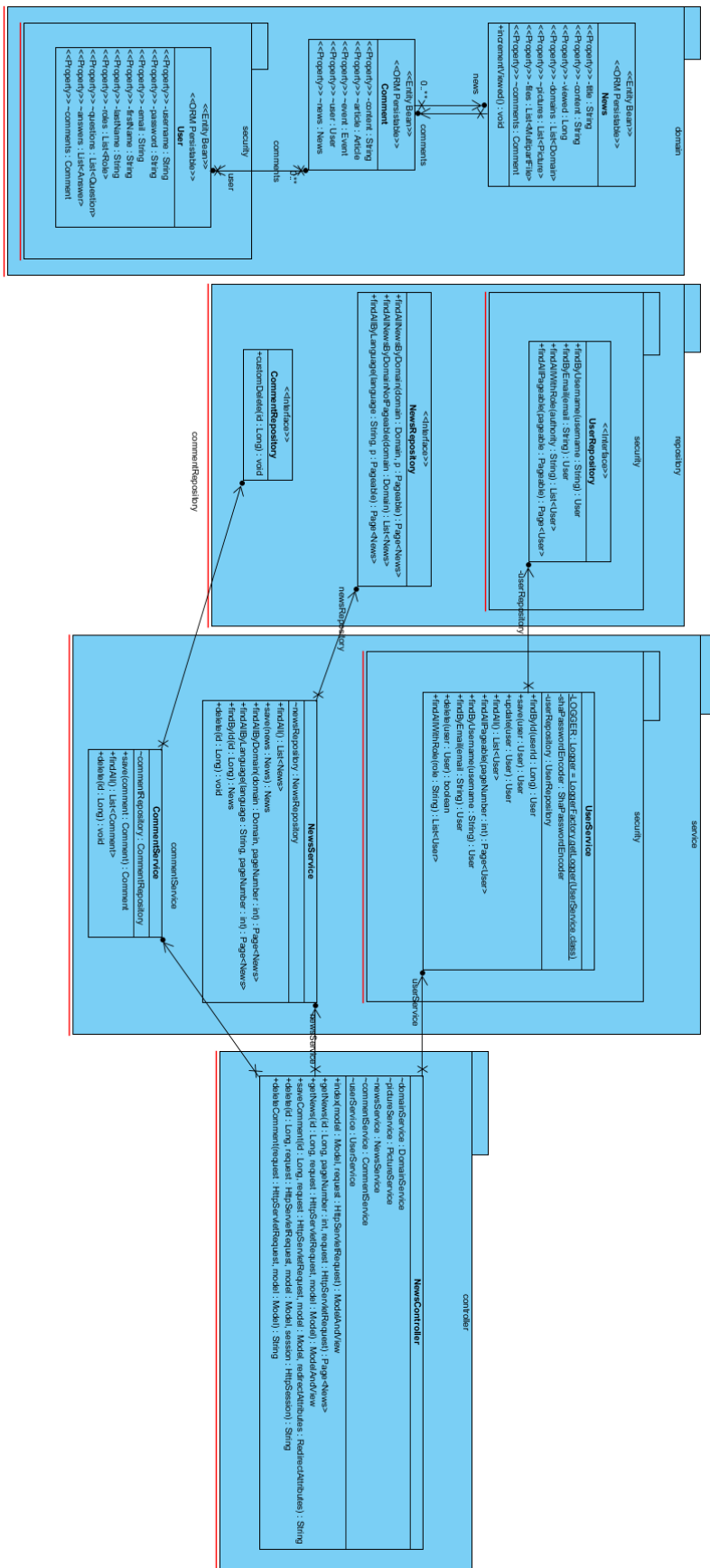


Fig. 8.2 Fluxul datelor în secțiunea știri

Cod sursă per pachete Java pentru fluxul datelor din tabela *comments* pana în interfața grafică în pagina de întrebări și răspunsuri (flow):

- Mapare entitate domain.*Comment.java*:

```
package my.app.stoma.domain;

import my.app.stoma.domain.security.User;
import org.hibernate.validator.constraints.NotEmpty;
import javax.persistence.*;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

/**
 * Created by bmatragociu on 3/26/2014.
 */

@Entity
@Table(name = "comments")
public class Comment extends BaseEntity {

    @Size(min = 1, max = 400)
    @NotNull
    @NotEmpty
    @Column(name = "content")
    private String content;

    @ManyToOne
    @JoinColumn(name = "id_user")
    User user;

    @ManyToOne
    @JoinColumn(name = "id_article")
    Article article;

    @ManyToOne
    @JoinColumn(name = "id_event")
    Event event;

    @ManyToOne
    @JoinColumn(name = "id_news")
    News news;
```

```
public Article getArticle() {
    return article;
}

public void setArticle(Article article) {
    this.article = article;
}

public User getUser() {
    return user;
}

public void setUser(User user) {
    this.user = user;
}

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}

public Event getEvent() {
    return event;
}

public void setEvent(Event event) {
    this.event = event;
}

public News getNews() {
    return news;
}

public void setNews(News news) {
    this.news = news;
}
```

```

    }
}

```

- repository.CommentRepository.java

```

package my.app.stoma.repository;

import my.app.stoma.domain.Comment;

import org.springframework.dao.DataAccessException;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Modifying;

import org.springframework.data.jpa.repository.Query;

import org.springframework.data.repository.query.Param;

import org.springframework.stereotype.Repository;

/**
 * Created by bmatragociu on 3/27/2014.
 */

@Repository

public interface CommentRepository extends
JpaRepository<Comment, Long> {

    @Modifying

    @Query("delete from Comment c where c.id = :id")

    void customDelete(@Param("id") Long id) throws
DataAccessException;

}

```

- service.CommentService.java

```

package my.app.stoma.service;

import my.app.stoma.domain.Comment;

import my.app.stoma.repository.CommentRepository;

import org.springframework.beans.factory.annotation.Autowired;
;

import org.springframework.stereotype.Service;

```

```

import
org.springframework.transaction.annotation.Transactional;

import java.util.List;

/**
 * Created by bmatragociu on 3/27/2014.
 */

@Service

public class CommentService {

    @Autowired

    CommentRepository commentRepository;

    @Transactional(readOnly = false)

    public Comment save(Comment comment) {

        return commentRepository.save(comment);

    }

    @Transactional(readOnly = true)

    public List<Comment> findAll() {

        return commentRepository.findAll();

    }

    @Transactional(readOnly = false)

    public void delete(Long id) {

        commentRepository.customDelete(id);

    }

}

```

- controller.AskUsController.java

```

package my.app.stoma.controller;

import my.app.stoma.domain.Answer;

import my.app.stoma.domain.Question;

import my.app.stoma.domain.security.User;

import my.app.stoma.service.AnswerService;

import my.app.stoma.service.MailService;

```

```

import my.app.stoma.service.QuestionService;

import my.app.stoma.service.security.UserService;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import
org.springframework.beans.factory.annotation.Autowired
;

import org.springframework.data.domain.Page;

import
org.springframework.security.access.annotation.Secured;

import
org.springframework.security.core.context.SecurityConte
xtHolder;

import
org.springframework.security.core.userdetails.UserDetail
s;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.validation.BindingResult;

import
org.springframework.web.bind.annotation.ModelAttribut
e;

import
org.springframework.web.bind.annotation.PathVariable;

import
org.springframework.web.bind.annotation.RequestMappi
ng;

import
org.springframework.web.bind.annotation.RequestMetho
d;

import
org.springframework.web.servlet.ModelAndView;

import
org.springframework.web.servlet.mvc.support.RedirectA
ttributes;

import javax.servlet.http.HttpServletRequest;

import javax.validation.Valid;

```

```

import java.io.UnsupportedEncodingException;

import java.util.List;

/**
 * Created by bmatragociu on 2/26/14.
 */

@Controller

@RequestMapping(value = "askus")

public class AskUsController {

    @Autowired

    QuestionService questionService;

    @Autowired

    AnswerService answerService;

    @Autowired

    UserService userService;

    @Autowired

    MailService mailService;

    private static final Logger LOGGER =
LoggerFactory.getLogger(IndexController.class);

    @RequestMapping(value = "", method =
RequestMethod.GET)

    public String redirect(HttpServletRequest request) {

        return "redirect:askus/0";

    }

    @RequestMapping(value = "/{currentPage}", method
= RequestMethod.GET)

    public ModelAndView questions(@PathVariable int
currentPage, Model model, HttpServletRequest request)
{

        Page<Question> page =
questionService.findAllPage(currentPage);

        if (page.getTotalPages() == 0) {

            model.addAttribute("Empty", true);

```

```

        return new ModelAndView("/askus",
model.asMap());
    }

    if (page.getNumberOfElements() == 0) {

        currentPage = currentPage - 1;

        return new ModelAndView("redirect:/askus/" +
currentPage);
    }

    List<Question> questions = page.getContent();

    model.addAttribute("allQuestions", questions);

    model.addAttribute("noOfPages",
page.getTotalPages());

    model.addAttribute("currentPage", currentPage);

    return new ModelAndView("/askus",
model.asMap());

    }

    @Secured({"ROLE_USER"})

    @RequestMapping(value = "/addTopic", method =
RequestMethod.GET)

    String addTopic(Model model, HttpServletRequest
request) {

        model.addAttribute("question", new Question());

        return "/addTopic";

    }

    @Secured({"ROLE_USER"})

    @RequestMapping(value = "/save", method =
{RequestMethod.POST, RequestMethod.GET})

    public String saveQuestion(Model model,
@ModelAttribute(value = "question") @Valid Question
question, BindingResult bindingResult,
HttpServletRequest request) throws
UnsupportedEncodingException {

        if (!bindingResult.hasErrors()) {

            UserDetails userDetails = (UserDetails)
SecurityContextHolder.getContext().getAuthentication().
getPrincipal();

```

```

            User currentUser =
userService.findByUsername(userDetails.getUsername())
;

            question.setUser(currentUser);

            questionService.save(question);

            mailService.sendToAllAdmin("New Answer!",
"New question posted: " + question.getContent());

            return "redirect:/askus";

        } else {

            return "/addTopic";

        }

    }

    @Secured({"ROLE_USER"})

    @RequestMapping(value = "/saveAnswer", method =
RequestMethod.POST)

    public String saveAnswer(Model model,
HttpServletRequest request, RedirectAttributes
redirectAttributes) throws
UnsupportedEncodingException {

        String content = request.getParameter("content");

        if (content == null || content.equals("") ||
content.length() > 400) {

            redirectAttributes.addFlashAttribute("error",
true);

            return "redirect:/askus/" +
request.getParameter("currentPage");

        }

        Answer answer = new Answer();

        answer.setContent(request.getParameter("content"));

        Question question =
questionService.findById(Long.parseLong(request.getPa
rameter("questionId")));

        UserDetails userDetails = (UserDetails)
SecurityContextHolder.getContext().getAuthentication().
getPrincipal();

```

```

        User currentUser =
userService.findByUsername(userDetails.getUsername())
;

        answer.setQuestion(question);

        answer.setUser(currentUser);

        answerService.save(answer);

        mailService.sendToAllAdmin("New Answer!",
"New answer posted: " + answer.getContent());

        return "redirect:/askus/" +
request.getParameter("currentPage");
    }

    @Secured({ "ROLE_ADMIN" })

    @RequestMapping(value = "/deleteQuestion", method
= RequestMethod.POST)

    public String deleteQuestion(Model model,
HttpServletRequest request) {

        Long id =
Long.parseLong(request.getParameter("id"));

        questionService.deleteById(id);

        return "redirect:/askus/" +
request.getParameter("currentPage");
    }

    @Secured({ "ROLE_ADMIN" })

    @RequestMapping(value = "/deleteAnswer", method
= RequestMethod.POST)

    public String deleteAnswer(Model model,
HttpServletRequest request) {

        Long id =
Long.parseLong(request.getParameter("id"));

        answerService.deleteById(id);

        return "redirect:/askus/" +
request.getParameter("currentPage");
    }
}

```

- askus/body.jsp

```

<% @ taglib prefix="sec"
uri="http://www.springframework.org/security/tags" %>

<% @ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>

<% @taglib uri="http://www.springframework.org/tags"
prefix="spring" %>

<div class="container">

    <c:choose>

        <c:when test="${Empty == true}">

            <a href="addTopic"><button type="button"
class="btn btn-default btn-lg"><span class="glyphicon
glyphicon-star"> </span> <spring:message
code="newTopic.label" text="New topic!"/>
</button></a>

            <br>

            <br>

            <spring:message code="noQuestions.label"
text="No questions!"/>

            </c:when>

            <c:otherwise>

                <br><br>

                <a href="addTopic"><button type="button"
class="btn btn-default btn-lg"><span class="glyphicon
glyphicon-star"> </span> <spring:message
code="newTopic.label" text="New topic!"/>
</button></a>

                <c:if

                    test="${error == true}">

                        <button class="btn btn-danger pull-right"
disabled>Please type an answer not longer than 400
characters!</button>

                    </c:if>

                    <br><br>

                <c:forEach var="question" items="${allQuestions}">

                    <div class="row col-lg-8">

```

```

<div class="panel panel-primary">
  <div class="panel-heading">
    <sec:authorize
access="hasRole('ROLE_ADMIN')">
      <form method="POST"
action="deleteQuestion">
        <button type="submit"
class="close">×</button>
        <input type="hidden" name="id"
value="${question.id}">
        <input type="hidden"
name="currentPage" value="${currentPage}">
      </form>
    </sec:authorize>
    <h3 class="panel-title"><span
class="glyphicon glyphicon-star"></span>
${question.user.username} &nbsp;<spring:message
code="said.label" text="said:"/></h3>
  </div>
  <div class="panel-body">
    <b>Q:</b>    ${question.content}
  </div>
</div>
</div>
<div>
  <c:forEach var="answer"
items="${question.answers}">
    <div class="row col-lg-8 col-lg-offset-4">
      <div class="panel panel-success">
        <div class="panel-heading">
          <sec:authorize
access="hasRole('ROLE_ADMIN')">
            <form method="POST"
action="deleteAnswer">
              <input type="hidden" name="id"
value="${answer.id}">
              <input type="hidden"
name="currentPage" value="${currentPage}">

```

```

        <button type="submit" class="close
pull-right">×</button>
      </form>
    </sec:authorize>
    <h3 class="panel-
title">${answer.user.username} said:</h3>
  </div>
  <div class="panel-body">
    <b>A:</b>    ${answer.content}
  </div>
</div>
</div>
</c:forEach>
<div class="row col-lg-8 col-lg-offset-4">
  <form action="saveAnswer" method="POST">
    <div class="input-group">
      <input type="text" name="content" class="form-
control">
      <input type="hidden" name="questionId"
value="${question.id}">
      <input type="hidden" name="currentPage"
value="${currentPage}">
      <span class="input-group-btn">
        <button type="submit" class="btn btn-default"
type="button"><spring:message code="answer.label"
text="Answer!"/></button>
      </span>
    </div>
  </form>
  <br>
</div>
</c:forEach>

```

```

<div class="row col-lg-12">
    <!-- Pagination: -->
    <ul class="pagination">
        <%--First li--%>
        <c:choose>
            <c:when test="{currentPage == 0}">
                <li class="disabled"><a><</a></li>
                <c:set var="begin" value="0"/>
            </c:when>
            <c:otherwise>
                <li><a href="{currentPage-1}"><</a></li>
                <c:set var="begin" value="{currentPage-1}"/>
            </c:otherwise>
        </c:choose>
        <%--Until when--%>
        <c:choose>
            <c:when test="{currentPage == noOfPages-1}">
                <c:set var="end" value="{currentPage}"/>
            </c:when>
            <c:otherwise>
                <c:set var="end" value="{currentPage+1}"/>
            </c:otherwise>
        </c:choose>
        <%--Iterating pages--%>
        <c:forEach var="i" begin="{begin}" end="{end}">
            <c:choose>
                <c:when test="{currentPage == i}">
                    <li class="disabled"><a>{i+1}</a></li>
                </c:when>
                <c:otherwise>
                    <li><a href="{i}">{i+1}</a></li>
                </c:otherwise>
            </c:choose>
        </c:forEach>
    </ul>
</div>
</div>
</c:choose>
</c:forEach>
<%--Last li--%>
<c:choose>
    <c:when test="{currentPage == noOfPages-1}">
        <li class="disabled"><a>></a></li>
    </c:when>
    <c:otherwise>
        <li><a href="{currentPage+1}">></a></li>
    </c:otherwise>
</c:choose>
</ul>
</c:otherwise>
</c:choose>
</div>
</div>

```