MINISTERUL EDUCAȚIEI, CERCETĂRII, TINERETULUI ȘI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

RePSMi MIDDLEWARE RECONFIGURABIL BAZAT PE PARADIGMA PUBLISH-SUBSCRIBE

LUCRARE DE LICENŢĂ

Absolvent: **Cornel MARIAN**

Coordonator ştiinţific: **S. L. Ing. Cosmina IVAN**

2011

MINISTERUL EDUCAȚIEI, CERCETĂRII, TINERETULUI ȘI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

DECAN,
Prof. dr. ing. Sergiu NEDEVSCHI

VIZAT,
ŞEF CATEDRĂ,
Prof. dr. ing. Rodica POTOLEA

Absolvent: **Cornel MARIAN**

RePSMi MIDDLEWARE RECONFIGURABIL BAZAT PE PARADIGMA PUBLISH-SUBSCRIBE

1. **Enunţul temei:** *Proiectarea unei arhitecturi pentru un middleware reconfigurabil bazat pe paradigma publish-subscribe de tipul bazat pe conținut, capabil să tolereze scenariile dinamice prezente în rețelele peer-to-peer, unde nodurile se pot deconecta și conecta în orice moment, prin reconfigurarea rețelei de brokeri distribuiți și prin rutarea eficientă a evenimentelor.*

2. **Conţinutul lucrării:** *Introducere, Obiectivele proiectului, Studiu bibliografic și concepte introductive, Analiză și fundamentare teoretică, Proiectare de detaliu și implementare, Testare și validare, Manual de utilizare și instalare, Concluzii și direcții de dezvoltare, Bibliografie, Anexe.*

3. **Locul documentării:** Universitatea Tehnică din Cluj Napoca

4. **Data emiterii temei:** 1 noiembrie 2010

5. **Data predării:** 24 iunie 2011

Absolvent: _____

Coordonator ştiinţific: _____

MINISTERUL EDUCAŢIEI, CERCETĂRII, TINERETULUI ŞI SPORTULUI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
CATEDRA CALCULATOARE

Declaraţie

Subsemnatul Cornel MARIAN, student al Facultăţii de Automatică şi Calculatoare, Universitatea Tehnică din Cluj-Napoca, declar că ideile, analiza, proiectarea, implementarea, rezultatele şi concluziile cuprinse în această lucrare de licenţă constituie efortul meu propriu, mai puţin acele elemente ce nu îmi aparţin, pe care le indic şi recunosc ca atare.

Declar de asemenea că, după ştiinţa mea, lucrarea în această formă este originală şi nu a mai fost niciodată prezentată sau depusă în alte locuri sau alte instituţii decât cele indicate în mod expres de mine.

Data: 24 iunie 2011

Absolvent: **Cornel MARIAN**
Număr matricol: 2010660
Semnătura:_____

Cuprins

1. Introducere

Tendinţele recente din domeniul informaticii au arătat o creştere semnificativă în interes faţă de aplicaţiile distribuite, care operează în medii dinamice, cum ar fi reţelele cu un număr mare de noduri şi cu diverşi furnizori de servicii.

Problemele de comunicare ivite din lipsă de fiabilitate sunt agravate de faptul că host-urile se conectează şi se deconectează de la reţea în momente imprevizibile. Modelul unei reţele de servere a căror rată de deconexiune e scăzută nu mai este actual. Cu toate acestea un număr mare de aplicaţii din domenii diferite încearcă să ţină pasul cu extinderea internetului.

Cercetătorii şi programatorii s-au alăturat provocării survenite din creşterea numărului de utilizatori de internet prin dezvoltarea de aplicaţii distribuite. Procesul de creare de aplicaţii este în permanenţă influenţat de reţelistică.

Modelul publish-subscribe e o abordare promiţătoare în a trata cerinţele aplicaţiilor distribuite în ceea ce priveşte flexibilitatea şi decuplarea dintre componente. Paradigma comunicaţiilor de tipul publish-subscribe a apărut ca o metodă eficientă prin care producătorii de evenimente pot trimite asincron notificări consumatorilor. O notificare e un mesaj trimis în rețea care informează o parte interesată că un eveniment a avut loc.[2]

Sisteme publish-subscribe, dezvoltate de industrie cât şi universităţi, se concentrează pe scalabilitate şi sunt construite având în vedere reţele stabile unde nodurile nu au probabilitate de deconexiune mare. Prin urmare, aceste sisteme nu furnizează nici un mecanism pentru a rearanja reţeaua de brokeri. Principalul dezavantaj al sistemelor similare este lipsa reconfigurării care trebuie să apăra în medii dinamice.

În ciuda flexibilității paradigmei, sistemele publish-subscribe disponibile în prezent oferă doar un sprijin limitat pentru scenarii dinamice. Unele sisteme permit componentelor client să migreze de la un broker la un altul în diferite locaţii logice sau fizice, în timp ce alte sisteme se bazează pe topologii redundante sau pe reîmprospătarea periodică a informaţiilor de rutare.

Următoarea provocare pentru middleware-urile publish-subscribe este, prin urmare, aceea de a tolera scenarii dinamice prin reconfigurarea rețelei de brokeri distribuiţi într-un mod eficient. Motivaţiile sunt numeroase. De exemplu, reţelele peer-to-peer sunt un bun exemplu unde conectarea sau deconectarea unui nod poate să apară în orice moment.

Lucrarea își propune construirea unui middleware reconfigurabil bazat pe paradigma publish-subscribe, care sa fie capabil să funcționeze intr-o rețea dinamică, peer-to-peer.

2. Obiectivele proiectului

Obiectivul principal al acestei lucrări poate fi rezumat în felul următor: construirea unui middleware de tipul content-based publish-subscribe, care să fie capabil să funcţioneze într-o reţea dinamică, scalabilă, peer-to-peer. Având acest scop în vedere, în cele ce urmează se vor descrie principalele caracteristici ale unui asemenea sistem ce se constituie ca obiective specifice ce vor fi urmărite în etapa de analiză cât şi în cea de implementare.

Prima cerintă esenţială a sistemului este aceea că acesta trebuie să aibă ca ţintă reţelele peer-to-peer. Aceste reţele au host-uri care în orice moment se pot conecta sau deconecta, concretizând astfel un mediu dinamic. Middleware-ul trebuie să fie flexibil, să nu depindă de noduri de reţea fixă, ci să fie reconfigurabil, operaţiile să nu fie influenţate de conectivitatea reţelei.

Arhitectura propusă va trebui să fie în aşa fel structurată încât toate nodurile sau majoritatea să participe activ la rutarea mesajelor. Toate nodurile se comportă ca brokeri. În context dinamic, nu se cunoaşte momentul realizării conexiunii nodurilor, acestea se pot conecta la reţea în orice moment. Astfel brokerii se pot conecta si deconecta în orice moment, neafectând sistemul, respectiv aplicația ce utilizează middleware-ul.

A doua cerintă importantă a sistemului cu impact asupra proiectării şi implementării sistemului o constituie strategia de rutare a mesajelor. O topologie în care există o structură de tip arbore rutarea necesită reconfigurare pentru a păstra integritatea comunicaţiei. Noţiunea de reconfigurare e a două caracteristică importantă a sistemului.

În această topologie de tip arbore mesajele pot ajunge la destinaţii, bazându-se doar pe legăturile dintre noduri.

O altă cerinţa este în mod clar menţinerea unei topologii conectate şi aciclice care să rămână aşa în ciuda schimbărilor arbitrare apărute la nivelul reţelei fizice.

3. Studiu bibliografic și concepte introductive

Acest capitol este dedicat prezentării unei imagini de ansamblu asupra middleware-ului publish-subscribe a cărui scop este de a îmbunătăţi modul de rutare a evenimentelor intr-un asemenea sistem. Deoarece această lucrare se axează pe prezentarea implementării unui sistem publish-subscribe, cititorul trebuie familiarizat în prealabil cu noţiunile care vor fi vehiculate în restul lucrării.

3.1. Concepte introductive

În această secţiune sunt prezentate şi definite concepte de bază, care sunt folosite în această lucrare: middleware, reţea peer-to-peer.

3.1.1. Middleware

Conceptul de middleware a fost introdus pentru a facilita comunicarea dintre entităţi în sistemele distribuite. Middleware e un nivel intermediar între sistemul de operare al nodurilor individuale şi aplicaţia distribuită. Acest nivel intermediar are de a face cu aspecte legate de comunicare şi încearcă să ofere aplicaţiei distribuite o privire omogenă a exteriorului. E larg răspândit şi a dovedit că e o abstractizare eficientă, care ajută la designul şi conceperea unui sistem distribuit complex.

Middleware-ul este un set de servicii independente de aplicaţii care permit aplicaţiilor şi utilizatorilor să interacţioneze prin reţea. În esenţă, middleware este software-ul localizat între reţele şi aplicaţii.

Middleware-ul oferă servicii generale care permit execuţia distribuită a aplicaţiilor. Este software-ul poziţionat între sistemul de operare şi aplicaţie. Ca o ”față-de-masă” care se întinde peste o reţea eterogenă, ascunzând complexitatea tehnologiilor care permit aplicaţiilor să fie rulate în respectivul mediu. Sarcinile middleware-ului sunt: încapsulează starea şi comportarea şi poate fi accesat numai printr-o interfaţă bine definită, interfaţa ascunde detaliile care sunt specific implementării, şi astfel ajută la încapsularea tehnologiilor diferite, comunică între ele prin schimb de mesaje.

O definiţie mai formală, consideră middleware-ul ca un nivel al software-ului al cărui scop constă în mascarea eterogenităţii platformei hardware şi software, precum şi furnizarea unui model de programare comod dezvoltatorilor de aplicaţii. El este format din procese sau obiecte ce se regăsesc pe un grup de calculatoare, şi care interacţionează între ele pentru a asigură implementarea comunicării şi partajarii resurselor în aplicaţiile distribuite. Altfel, aplicaţiile distribuite ar trebui să apeleze direct la interfaţa de programare furnizată de sistemul de operare al reţelei. Pentru a simplifica dezvoltarea şi integrarea aplicaţiilor distribuite, majoritatea soluţiilor middleware se bazează pe un anumit model, care descrie aspectele privind distribuirea şi comunicarea. Cele mai utilizate astfel de modele sunt: apelarea procedurilor de la distanţă (Remote Procedure Call), distribuirea obiectelor şi distribuirea documentelor. Cele mai cunoscute soluţii middleware sunt Sun RPC, CORBA (Common Object Request Broker Architecture), Java RMI (Java Remote Object Invocation) si DCOM (Distributed Component Object Model).

Middleware-ul trebuie să ofere mecanismele care să suporte conceptele incorporate în modelul obiect. De asemenea, trebuie să permită interactiunea operațională între două obiecte si trebuie să permită interacțiunea între două obiecte localizate în spații de adresare diferite. Middleware-ul oferă support si pentru integrarea unor tehnologii diferite. Eterogenitatea e tratată de către middleware prin faptul ca oferă aceeași funcționalitate la toate punctele de

acces, aplicațiile au access la funcționalitatea lor printr-un API. API-urile sunt adaptate la condițiile fiecărui limbaj de programare care este suportat de către middleware.

Un programator de aplicații în mod tipic vede middleware-ul ca o biblioteca de programe sau ca un set de unelte. Acetea sunt dependente de mediul de dezvoltare pe care programatorul il utilizeaza si este afectat de asemenea de compilatorul/interpretorul utilizat pentru dezvoltarea aplicatiei distribuite. In contextual middleware-ului, un standard trebuie sa stabileasca interfetele intre diferite componente pentru a permite interactiunea dintre ele. Un middleware furnizează un set standard de interfețe pentru o colecție de resurse distribuite, eterogene si proprietare.

3.1.2. Rețea peer-to-peer

Sistemele distribuite pe infrastructuri sunt clasificate în reţele peer-to-peer şi reţele bazate pe server. Într-o reţea peer-to-peer nu există servere dedicate şi nici o organizare ierarhică a calculatoarelor - toate calculatoarele sunt egale. De aceea se şi numeşte reţea peer-to-peer (de la egal la egal). Fiecare calculator are atât rol de client cât şi de server. Evident că acest lucru înseamnă că nu există un administrator care să se ocupe de reţea, ci fiecare utilizator trebuie să aibă grijă de propriul sistem (fiecare utilizator este atât administratorul propriei maşini cât şi utilizatorul acesteia).

3.2. Concepte generale legate de sistemele publish-subscribe

În această secţiune se vor prezenta concepte generale legate de sistemele publish-subscribe: elementele unu sistem publish-subscribe, modelele de subscripţii, cât şi paradigma generală a unui sistem publish-subscribe.

3.2.1. Elementele sistemului

Un sistem generic publish-subscribe[1](uneori denumit în literatură de specialitate că Event Service sau Notification Service) e compus dintr-un set de noduri distribuite într-o reţea de comunicaţii. Clienţii sistemului sunt divizaţi în funcţie de rolul pe care îl deţin în publishers, care se comportă ca producători de informaţie şi subscribers, aceştia fiind consumatori de informaţie. Clienţii nu comunică direct între ei, ci sunt mai degrabă decuplaţi: interacţiunea are loc prin nodurile sistemului publish-subscribe. Decuplarea e o caracteristică dezirabilă pentru un sistem de comunicaţii, deoarece aplicaţia poate să fie creată mai independent din punctul de vedere al problemelor de comunicare, evitându-se aspecte precum sincronizarea sau adresarea directă de la subscriberilor la publishers.

Operaţional, interacţiunea dintre nodurile client şi sistemul publish-subscribe are loc prin un set de operaţii care pot fi executate de clienţi pe sistemul publish-subscribe şi viceversa.

Un *publisher* trimite informaţie (un eveniment notat e) la sistem prin execuţia operaţiei publish(e).

Un *eveniment* în general e compus dintr-o pereche de atribute. Fiecare atribut are un nume format dintr-un şir de caractere simplu şi un tip. Tipul e în general un tip primitiv definit în limbajele de programare (integer, real, strîng).

Subscriberul e interesat de un anumit eveniment prin subscripţii(subscriptions). O subscripţie , e un filtru peste o porţiune a conţinutului evenimentului, exprimat printr-un set de constrângeri care depind de limbajul subscripţiilor. Un subscriber instalează sau elimină o subscripţie din sistem prin execuţia operaţiilor subscribe(e) şi unsubscribe (e).

Spunem ca o notificare se potriveşte unei subscripţii (matching) dacă satisface toate constrângerile declarate în atributele corespunzătoare.

3.2.2. Modele de subscripţii

Diferitele moduri de a specifica evenimentele au dus la crearea de variante distincte a paradigmei publish-subscribe. Modelele de subscripţii, care au apărut sunt caracterizate în funcţie de puterea de expresivitate: modelele cele mai expresive oferă subscriberilor posibilitatea că să îşi aleagă cu precizie evenimnetele de care sunt interesaţi. În cele ce urmează sunt prezentate cele mai folosite modele de publish-subscribe.

În *modelul bazat pe subiect*, notificarile sunt grupate pe subiecte. Un subscriber îşi declară interesul pentru un subiect anume şi va primi toate evenimentele care au legătură cu acel subiect. Fiecare subiect corespunde unui canal logic care conectează fiecare posibil publisher la toţi subscriberii interesaţi de acel subiect.[4]

Modelul bazat pe subiect a fost o soluţie adoptată de sisteme precum : TIB/RV, iBUS, SCRIBE, Bayeux şi CORBA Notification Service.

Marele dezavantaj al acestui model este expresivitatea limitată oferită subscriberilor. Un subscriber interesat de un set particular de evenimnete, care sunt corelate de un anumit subiect, primeşte notificare de la toate evenimentele care au legătură cu acel subiect. De

exemplu: un subiect B poate fi definit ca un subiect particular al subiectului A. Evenimentele care sunt corelate la B vor fi primite de către toţi clienţii subscrişi atât la A cât şi de B.

Un alt model este *modelul bazat pe conţinut*[6]. În acest model subscriberii îşi exprimă interesul pentru anumite evenimente, specificând condiţii. Altfel spus, un filtru într-o subscripţie este o interogare (query) formată dintr-un set de constrângeri aplicate valorilor atributelor notificării. Natura constrângerilor depinde şi de tipul limbajului folosit. Exemple de sisteme bazate pe acest model sunt: Gryphon, SIENA, JEDI, LeSubscribe, Ready, Hermes, Elvin.

În modelul publish-subscribe,bazat pe conţinut, evenimentele nu sunt clasificate în funcţie de un criteriu predefinit (numele subiectului), dar mai degrabă prin proprietăţile evenimentelor. Se poate observă ca un model bazat pe subiect poate fi emulat foarte uşor printr-un model bazat pe conţinut folosind un filtru format dintr-o condiţie de egalitate. Expresivitatea modelului bazat pe conţinut, produce un dezavantaj prin preţul mare al consumului de resurse necesar calculării condiţiilor.

Un al treilea model este *modelul bazat pe tip* (type-based model). Variantele de sisteme publish-subscribe bazate pe tip sunt obiecte aparţinând unui specific tip, care poate encapsula atribute cât şi metode.

Un alt model este *modelul bazat pe concept*. Acest model se bazează pe structură evenimentului, pe o sintactică şi o semantică. El permite descrierea evenimentelor că o schemă abstractă folosind ontologii, care oferă o cunoaştere a lumii prin metadata şi funcţii de mapare.

Modelul bazat pe Xml are caracteristică flexibilitatea, deoarece permite ierarhizarea evenimentelor cât şi o interoperabilitate, independenţa de implementare şi extensibilitate. Un dezavantaj este timpul de procesare a algoritmilor care folosesc date structurate sub formă de xml.

3.2.3. Paradigma programării bazate pe evenimente

În paradigma Publish-Subscribe [7] clientul interacţionează prin publicarea de evenimente şi prin subscrierea la evenimentele de care este interesat. Event-based systems oferă un nivel mai ridicat de flexibilitate, permiţând clienţilor să precizeze clasele de evenimente de care sunt interesaţi, folosind facilităţi lingvistice. Un număr de sisteme care folosesc paradigma publish-subscribe sunt disponibile, diferind unul faţă de celălalt prin design-ul Event Dispatcher-ului, componenţa responsabilă cu colectarea subscripţiilor şi cu trimiterea evenimentelor celor subscrişi.

Publish-subscribe permite interacţiune între aplicaţii, acestea având numele de clienţi. Aceşti clienţi pot publică notificări de evenimente pentru a exprimă apariţia unui eveniment anume şi se pot abona la notificarile altor clienţi. Cu toate că acest model de comunicaţie a fost definit în contextul notificarilor bazate pe evenimente, continutull notificării poate fi orice mesaj. Din acest motiv în cele ce urmează vom folosi termenul eveniment(event) cu sensul de mesaj cu excepţia cazului în care această ar putea duce la o confuzie.

Subscripţii pot fi emise nu doar pentru evenimente specifice ci şi pentru un set de evenimente definite printr-un tip de filtru. Evenimentele publicate sunt expediate la toţi clienţii care au reuşit să se subscrie cu succes, îndeplinind condiţiile filtrelor.

Abonarea la un eveniment poate fi revocată prin apelarea operaţiei unsubscribe, corespunzătoare evenimentului.

Event-dispatcher-ul mediază comunicarea, dintre publishers şi subscribers, oferind decuplarea caracteristică paradigmei. Evenimentele publicate sunt trimise la toţi clienţii care s-au abonat la acel tip de mesaj până în momentul în care ei revoca abonarea prin operaţia

unsubscribe. Publishers şi subscribers nu ştiu unul de existenţa celuilalt. Evenimentele pot fi publicate fără a ţine cont de numărul de subscripţii existenţe în sistem. Comunicarea este asincrona, nici publisher-ul şi nici subscriber-ul nu e blocat când produc notificări sau acceptă evenimente.

3.3. Standarde bazate pe evenimente

În această secţiune vor fi descrise standardele, care au diferite implementări, existente bazate pe evenimente: CORBA Event and Notification Service, Java Message Service(JMS).

Serviciul de evenimente (Event Service) permite obiectelor să înregistreze sau să anuleze interesul lor pentru evenimente specifice. Serviciul defineşte un obiect denumit event channel care colectează şi distribuie evenimente pentru obiecte care nu se cunosc reciproc.

Notification service e o extindere a serviciilor existente Event service, adăugând la aceasta: transmiterea evenimentelor în formă de structură de date, subscriere la eveniment; descoperirea, QoS, şi, opţional, un repository de tip eveniment.

Java Messaging Service (JMS)[3] este un serviciu de messaging permite obiectelor distribuite să comunice într-o manieră asincronă, dar sigură. Prin faptul că mesajele sunt trimise asincron, în loc să fie transmise sincron, scalabilitatea sistemului creşte. Procesele pot răspunde la mesaje când sunt în execuţie, dar există situaţia în care s-ar putea să nu se afle în execuţie în momentul în care mesajul este trimis iniţial. Specificaţia Java Messaging Service (JMS) defineşte un serviciu portabil de messaging. Prin utilizarea unui API comun, obiectele distribuite pot comunică într-o manieră tranzacţională, tolerantă la căderi, asincronă, dar cel mai important este că ele comunică într-o manieră independenţă de vendor.

3.4. Soluții existente publish-subscribe

În continuare vor fi prezentate câteva soluţii existenţe de sisteme publish-subscribe: TIB\Rendezvous, Siena, LeSubscribe, JEDI, Gryphon, Hermes, Xnet, Rebeca, Elvin.

TIB/Rendezvous e un sistem comercial publish-subscribe creat de TIBCO. Evenimentele sistemului TIB/Rendezvous cuprind un set de date bazate pe tip, într-o manieră record-based. Limbajul subscripţiilor e bazat pe subiect. Se pot filtra evenimente pe bază valorii dintr-un câmp care e considerat subiect. Dispecerul exploatează o arhitectură de tip arbore. Fiecare nod de reţea care rulează un client, trebuie de asemenea să ruleze un daemon(un program care rulează în background) . TIB\Rendezvouse e responsabil cu filtrarea evenimentelor clienţilor de pe acel nod. Daemon-ul comunică prin mesagerie broadcast în acelaşi subnet. Comunicarea e asigurată prin două nivele de rutare daemon: subnet routing daemons şi wide-area routing daemons.

Siena a fost dezvoltat la Politehnica din Milano şi la Universitatea din Colorado. Obiectivul major al acestei versiuni este acela de a maximiza expresivitatea mecanismului de subscripţie, fără să sacrifice scalabilitatea. Notificarile evenimentelor sunt definite că seturi de atribute caracterizate de un tip, un nume, şi o valoare. Limbajul de subscripţie e content-based. Event dispatcherul exploatează o arhitectură distribuită şi rutarea e efectuată folosing o strategie care conţine anunţuri(advertisments).

LeSubscribe e un sistem content-base dezvoltat la INRIA. Proiectul îşi propune să suporte aplicaţii formate dintr-un număr mare de componente distribuite. Diferit de variantă

propusă de noi, LeSubscribe se focalizează pe problemă legată de filtrare pentru a permite asocierea eficientă a evenimentelor la un număr mare de subscripţii.

JEDI (Java Event-based Distributed Infrastructure) e o infrastructură de tipul object-oriented, event-based dezvoltat la Politehnica din Milano. JEDI e un sistem tuple-based notification: evenimentele sale sunt definite că seturi de şiruri de caractere, unde primul şir de character e numele evenimentului, şi celelalte şiruri sunt paramatrii evenimentului. JEDI conţine atât un event dispatcher centralizat cât şi distribuit, strategia de rutare fiind bazată pe forwarding ierarhic.

Hermes e un middleware publish-subscribe scalabil şi reconfigurabil, care foloseşte tehnici peer-to-peer pentru a construi şi întreţine reţeaua să de rutare overlay. Hermes prevede o formă uşor limitată de rutare content-base numită „type and attribute based". [14]

Gryphon e un rezultat al proiectului de cercetare iniţiat de IBM, care îşi propune construirea unui broker robust publish-subscribe. Nucleul conţine un algoritm de filtrare, bazat pe arbori de căutare paralelă, care permit brokerilor să determine ruta minima a mesajelor. Arhitectura event dispatcherului e distribuită în topologia unui arbore şi replicarea e folosită pentru a îmbunătăţii fiabilitatea sistemului.

Xnet e un sistem content-base publish-subscribe, dezvoltat de Eurecom. Proiectul se concentrează pe contruirea unui sistem scalabil şi fiabil pentru distribuirea de conţinut XML la un număr mare de consumatori. Protocolul sau de rutare exploatează agregarea subscripţiilor pentru a limită dimensiunea tabelelor de rutare fără a mari traficul, care s-ar produce din notificări.

Rebeca [5] e un serviciu de notificare care încorporează un număr de strategii de rutare, bazate pe subscripţiile arborelui de rutare al brokerilor. Abordarea bazată pe contracte de leasing permite sistemului să se autostabilizeze, fără nici un mecanism special, dar necesită consumatorilor să îşi reinoiasca contractele de leasing la intervale de timp. Această abordare măreşte traficul cât şi faptul că consumatorii pot primii notificări de care nu mai sunt interesaţi, deoarece subscripţiile rămân valide până contractele de leasing expiră.

Elvin e un sistem de publish-subscribe dezvoltat la Distributed Systems Technology Centre. Elvin conţine un content-base subscription language pentru a filtra notificarile consistând din multiple domenii. Sistemul poate fi distribuit la o reţea wide-area prin interconectarea de clusteri de brokeri. Scalabilitatea e facilitată de notification-quenching, o abordare care permite publisherilor să genereze doar evenimente de care cel puţin un subscriber e interest.

Aceste sisteme se concentrează pe scalabilitate şi sunt construite având în vedere reţele stabile unde nodurile nu au probabilitate de deconexiune mare. Prin urmare, aceste sisteme nu furnizează nici un mecanism pentru a rearanja reţeaua de brokeri. Principalul dezavantaj al sistemelor similare este lipsa reconfigurării care trebuie să apăra în medii dinamice. Sistemele similare studiate sunt open-source şi au fost un bun model pentru implementarea sistemului publish-subscriber, care are o arhirtectura modulară pe nivele şi tratează reconfigurarea.

3.5. Extensii ale sistemului publish-subscribe

Flexibilitatea paradigmei publish-subscribe este determinată de simplitatea interfeţei pe care o oferă. Interfaţa este formată din trei operaţii de bază: publish, subscribe şi unsubscribe. Unele aplicaţii necesită unele extensii ale celor trei operaţii, implementate cu un nivel deasupra paradigmei publish-subscribe.

Datorită acestei necesităţi paradigma publish a subscribe a fost extinsă cu câteva operaţii noi. Aceste extensii au două scopuri. Primul scop este acela de a îmbunătăţi comunicarea cu middleware-ul, permiţând folosirea unor operaţii mai complexe. Al doilea scop este acela de a oferi posibilitatea middleware-ului să îşi îndeplinească funcţiile mai uşor. În continuare se vor prezenţa extensiile cele mai importante.

3.5.1. Anunţurile

Anunturile (advertisments) sunt un mecanism pentru a optimiza mecanismul de subscripţii. Într-un sistem publish-subscribe, aplicaţiile pot publica noi notificări de evenimente prin invocarea operaţiei publish pe middleware. Advertisments(Anunturile) schimbă acest comportament prin faptul că solicită abonaţilor să informeze sistemul că au de gând să publice evenimente care se potrivesc cu un anumit filtru.

Dacă un publisher nu anunţă nici un eveniment care se potriveşte unei subscripţii particulare, atunci nici unul din evenimentele sale nu va trebui să verifice potrivirea la o anumită subscripţie. Acest lucru ajută middleware-ul publish-subscribe să economisească timp şi efort atunci când încearcă potrivirea evenimentelor cu subscripţiile.

3.5.2. Răspunsurile

Notificarile de evenimente suportă comunicare într-un singur sens, de la publishers la subscriberi. Sistemele publish-subscribe cu răspunsuri fac comunicarea bidirecţională prin permiterea receptorilor să trimită răspunsuri la publishers folosind aceeaşi infrastructură de comunicare. În acest fel, se permite sistemelor publish-subscribe să fie folosite de către aplicaţiile care necesită semantica cerere- răspuns.

În principiu, posibilitatea de a răspunde unei cereri s-ar putea construi într-un nivel supererior faţă de nivelul middleware-ului. Dezavantajul ar fi faptul că subscriberii ar trebui să contacteze publisherii direct, violând astfel anonimatul dintre publisher şi subscriber.

Încorporarea răspunsurile la cereri în middleware permite păstrarea aceluiaşi nivel de decuplare, regăsit în paradigma originală. Atunci când răspunsuri sunt trimise prin middleware-ul publish-subscribe, subscriberii pot răspunde la o notificare a unui eveniment, fără necesitatea cunoaşterii celui care a iniţiat cererea.

3.5.3. Identificarea locaţiei

Abilitatea de a subscrie la informatia generată într-un anumit loc sau de a răspândi informația doar într-o anumită regiune de interes pare a fi o extensie necesară a paradigmei publish-subscribe.

Într-o aplicație de monitorizare a incendiilor, de exemplu, sistemul de control trebuie sa notifice aspersoarele şi alte elemente de acționare doar atunci cand a fost detectat foc in interiorul camerei în care acestea sunt instalate. În aplicaţiile auto, conducătorii auto doresc să fie notificați de accidente sau blocajele de trafic care apar de-a lungul traseului, pe care intenţionează să-l urmeze şi nu in partea opusă a oraşului.

In ciuda numărului mare de potențiale aplicații, totuși, soluții care să trateze identificarea locației au fost doar recent propuse în contextul paradigmei publish-subscribe. Acest lucru se datorează faptului că sistemele publish-subscribe,bazate pe continut , permit ca locațiile să fie tratate în aceeasi manieră ca orice alte atribute, permitând clientului sa decidă în momentul aplicării condițiilor. Atunci când publica o notificare de eveniment clienții pot specifica locația fizică pentru care notificarea e relevantă. In mod similar, la subscriere , se poate exprima interesul pentru notificările din regiunea respectivă.

4. Analiză și fundamentare teoretică

Scopul acestei lucrări poate fi rezumat în felul următor: construirea unui middleware scalabil de tipul publish-subscribe bazat pe continut, care este capabil să funcţioneze într-o reţea dinamică, scalabilă, peer-to-peer. Având acest scop în vedere, în cele ce urmează se vor descrie principalele caracteristici ale soluţiei propuse.

4.1. Problema reconfigurării, caracteristicile sistemului

Prima caracteristică esenţială a sistemului este aceea că acesta trebuie să aibă că ţintă reţelele peer-to-peer. Reţele a căror host-uri în orice moment se pot conecta sau deconecta, fiind astfel un mediu dinamic. Middleware-ul trebuie să fie flexibil, să nu depindă de noduri de reţea fixă, ci să fie reconfigurabil, operaţiile să nu fie influenţate de conectivitatea reţelei.

Definim arhitectura ca una în care toate nodurile, sau majoritatea, participă activ la rutarea mesajelor. Toate nodurile se comportă ca brokeri. Brokerii se pot conecta şi deconecta în orice moment, neafectând sistemul. Nu se cunoaşte momentul realizării conexiunii nodurilor, acestea se pot conecta la reţea în orice moment .

A două caracteristică a sistemului e strategia de rutare. Într-o topologie de tip arbore mesajele ar ajunge la destinaţii bazându-se doar pe legăturile dintre noduri. Această strategie nu este potrivită deoarece într-un mediu dinamic unde nodurile cad în orice moment posibilitatea de a avea un arbore intact e improbabilă. O topologie în care există o structură de tip arbore rutarea necesită reconfigurare pentru a păstra integritatea comunicaţiei. Noţiunea de reconfigurare e a două caracteristică importantă a sistemului.

4.2. Analiza și proiectarea arhitecturii middleware-ului

La sistemele publish-subscribe, bazate pe conţinut, rutarea e gestionată de o reţeaua de brokeri. Interconexiunile dintre brokeri sunt stabilite static, doar informaţia de rutare e stabilită dinamic ca subscripţie, unsubscription, sau posibile anunţuri[8]. Atunci când topologia reţelei e dinamică, infrastructura de rutare trebuie să reconfigureze sistemul în timp ce acesta rulează. El trebuie să se adapteze la schimbările apărute la nivel fizic, fără a întrerupe operaţiile normale ale sistemului.

Soluţia propusă rezolvă problema reconfigurării, prin soluţionarea unor subprobleme, care adunate rezolvă problema reconfigurării într-o topologie de reţea dinamică. Acestea sunt:

1. Menţinerea interconexiunilor dintre brokeri, rezolvarea problemei deconectărilor fără a crea bucle.
2. Reconcilierea subscripţiilor deţinute de fiecare broker şi folosite că mesaje de rutare, păstrarea consistenţei sistemului, acesta să nu fie influenţat în operaţiile de subscribe şi unsubscribe.
3. Recuperarea mesajelor pierdute în timpul procesului de reconfigurare.

Pentru a rezolva aceste trei aspecte se propune arhitectura middleware-ului prezentată în Figura 1, ce se constituie in jurul protocolului Strawman. Aceasta se compune din două nivele: overlay, routing. Împărţirea pe aceste două nivele duce la izolarea fiecărei probleme cât şi structurarea codului pentru a putea fi uşor de întreţinut în timp. De asemenea se poate înlocui un nivel cu o altă soluție, cu o altă abordare sau se poate extinde mult mai uşor.

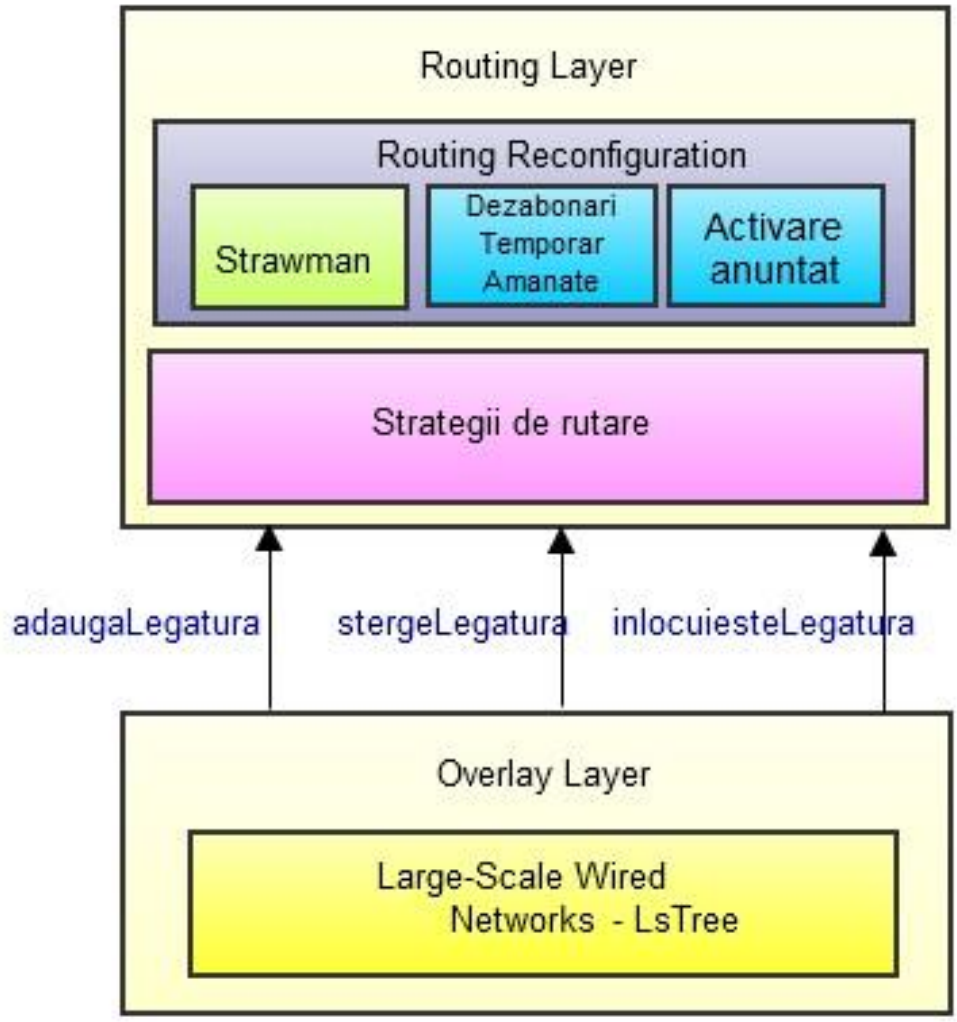


Figura 1. Arhitectura pe nivele a middleware-ului publish-subscribe

În continuare sunt descrise pe scurt cele trei nivele prezentându-se scopul, utilitatea şi funcţionalitatea fiecărui nivel.

Nivelul de infrastructură de bază(Overlay Layer). Nivelul de overlay[9] se situează la baza întregii arhitecturi şi constituie interfaţa middleaware-ului cu reţeaua şi sistemul de operare. Acest nivel reprezintă abstractizarea nodurilor unei reţele într-o topologie de tip arbore fără rutare. Overlay-ul e un subset, un arbore format din graful complet reprezentat prin reţeaua fizică, acest arbore e creat prin mecanismul de rutare IP. Nivelul gestionează şi implementează toate aspectele legate de reţea şi oferă o interfaţă uniformă nivelului de rutare. Acest nivel poate fi uşor înlocuit cu altă implementare specifică, de exemplu, reţelelor fără fir. Nivelul de overlay e construit deasupra nivelului TCP/IP al reţelei cât şi pe bază unei topologii de tip arbore.

Nivelul de rutare(Base Routing Layer). Construit deasupra nivelului de overlay, nivelul de rutare e componentă cheie a sistemelor publish-subscribe bazat pe conţinut. Un nivel de rutare e responsabil pentru distribuţia corectă a mesajelor, folosindu-se de o stategie care în cele mai multe cazuri depind de scenariul aplicaţiei destinaţie. În această lucrare, arhitectură prezentată e bazată pe o stategie de rutare de tip expediere de abonamente(subscriptions). Aspectul urmărit în această lucrare este abilitatea sistemului de a se adapta la schimbările apărute în nivelul de overlay. Un manager de overlay dinamic prevede o topoligie care se schimbă odată cu necesităţile reţelei fizice. Nivelul de rutare răspunde la schimbările din niveul inferior, prin reconfigurarea rutelor, în timp ce sistemul continuă să ruleze şi să opereze.

O caracteristică importantă a interfeţei este faptul că fiecare nod din nivelul de rutare cunoaşte identitatea tuturor nodurilor vecine din nivelul de overlay. Informaţia despre vecini surprinde simetria. Dacă un nod A recunoaşte un nod B că fiind vecin, atunci B trebuie de asemenea să recunoască pe A că vecin. În mod similar dacă un nod A îşi pierde conexiunea la B, atunci şi conexiunea e la B la A e ştearsă. În schimb, dacă un nod A apare deconectat de la un nod B , dar e recunoscut că un nod valid de către C, atunci situaţia e valida.

Orice nod din middleware este conştient de existenţa legăturilor dintre vecinii din nivelul de overlay. Nivelul de rutare exploatează informaţia despre identitatea vecinilor din ovelay pentru a schimbă subscripţii, unsubscriptions şi notificări despre evenimente cu fiecare dintre ei. De fiecare data când conexiunile din reţeaua fizică se schimbă nivelul de overlay reacţionează şi modifică topologia overlay-ului. În această situaţie, nivelul de rutare, notifica care legături la nodurile vecine au apărut sau dispărut. Aceste notificări permit că nivelul de rutare să menţină informaţia despre rutare consistentă cu nivelul de overlay.

Notificarile despre apariţia sau dispariţia de legături apare în multe circumstanţe. În toate cazurile, nivelul de overlay, informează adăugarea sau ştergerea unei legături la un vecin.

Apelul adaugaLegatura informează nivelul de rutare de existenţa unui nou vecin, apelul stergeLegatura informează pierderea unui vecin şi inlocuiesteLegatura informează faptul că legătură la vecin a fost pierdută şi va fi înlocuită cu un alt nod specificat că şi parametru.

4.3. Nivelul de rutare

În această secţiune este prezentat nivelul de rutare. Acest nivel ia deciziile importante legate de rutarea mesajelor, hotărând unde transmite anumite evenimente specifice.

4.3.1. Protocolul Strawman

Protocolul Strawman[11] se bazează pe două observaţii. Prima observaţie este că atunci când o legătură se rupe, mesajele nu mai pot fi trimise pe calea aceea şi toate subscripţiile recepţionate anterior nu mai sunt folositoare şi trebuie şterse.

A două obseravatie e aceea că atunci când un broker e anunţat că un nou nod urmează să fie adăugat, acesta trebuie să se asigure că orice eveniment care e interesat de el şi de faptul că e generat de celălalt capăt al legăturii e transmis corect la noua legătură. Ce e important la protocolul Strawman, e faptul că observaţiile de mai sus pot fi îndeplinite în totalitate cu mesaje de abonare şi dezabonare.

4.3.1.1. Descrierea protocolului

Atunci cand un broker e anunțat că o legatură la unul din vecinii săi s-a întrerupt, se comportă ca ți cum ar fi primit un mesaj de dezabonare pentru toate abonarile precedente. In acest fel se șterg rutele întrerupte. În mod similar atunci când o nouă legatură e adaugată, brokerul trimite mesaje de abonare pentru toate combinațiile posibile din tabelul de abonare. În acest fel se informează noul vecin al evenimentului că trebuie sa fie trimis prin aceasta legatură. Aceste abonări și dezabonări se propagă în arbore actualizând informația despre abonamente.

În continuare e prezentat pseudocodul executat pe un broker pentru aceste operații de reconfigurare, împreuna cu operațiile normale de abonare, dezabonare și procesare de evenimente. Fiecare broker detine un tabel de abonare subTab conținând abonamenete sub forma (n,p) pentru a ține evidenta vecinilor, n reprezentând brokerul și p pattern-ul.

Fiecare operație din pseudocod e executată local pe broker. O singura operație poate fi executată în acelați timp. Operatia *evenimentReceptionat*, *abonareReceptionata*, *dezabonareReceptionata* sunt actionate de receptionarea mesajului corespunzator de către broker.

Algoritmul Strawman prezentat in pseudocod:

```
// Proceseaza eveniment primit de la vecinul n
evenimentPrimit( n, EVENIMENT)
    trimite EVENIMENT la toti abonatii inscrisi la acest eveniment mai putin n.

// Dezabonare primita de la vecinul n
dezabonarePrimita(n, UNSUB(p))
    if(n,p) ∈ subTab then
          proceseazaUnsubDela(n,p)

// Subscriptie primita de la vecinul n
subscriptiePrimita(n, SUB(p))
   if (n,p) not ∈ subTab then
      subTab ← subTab ∪ {(n,p)}
      if aceasta este prima subscriptie primita pentru p then
            trimite SUB(p) la toti vecinii, mai putin la n
      else if daca e a doua subscriptie primita de la p then
            trimite SUB(p)  la primul abonat

// Sterge legatura la vecinul n
stergeLink(n)
  for all (n,p) ∈ subTab do
     proceseazaUnsubDela(n,p)
  vecini ← vecini – {n}

// Adauga legatura la noul vecin n
adaugaLegatura(n)
   vecini ← vecini ∪ {n}
    for all (n,p) ∈ subTab do
       trimite SUB(p) la n

// Proceseaza unsub pentru pattern n de la vecinul n
processUnsubDela(n,p)
   subTab ←subTab – {(n,p)}
   if nu mai sunt subscriberi pentru p in subTab then
      trimite UNSUB(p) la toti vecinii mai putin la n
  else if  o singura subscriptie (n',p) in subTab then
       trimite UNSUB(p) la n'
```

4.3.2. Inţelegerea propagării si reconfigurării

În această secţiune se vor sublinia problemele protocolului Strawman şi de asemenea se vor prezenta câteva observaţii care sunt importante în rezolvarea problemei reconfigurării.

Problemă fundamentală: Abonamentele sunt şterse şi imediat reintroduse. Protocolul Strawman este cel mai natural protocol atunci când reconfigurarea implică doar legături izolate, care urmează să fie adăugate sau şterse. Cazul cel mai frecvent într-o reţea dinamică este acela când o legătură întreruptă este în scurt timp înlocuită cu una nouă. În acest caz, Strawman este ineficient, caz ilustrat în Figura 2.

Dacă dezabonarile se propagă printr-un subarbore înainte că procesul de abonare să înceapă(cazul cel mai probabil) , efectul e că multe dintre abonarile din subarbore sunt şterse iar apoi în scurt timp sunt readaugate. Acest fenomen afectează performanţă, impactul e proporţional cu mărimea sistemului şi gradul de reconfigurare necesar. În partea următoare a lucrării se vor prezenţa protocoale, care nu sunt afectate de aceeaşi problemă şi care au o performanţă mai bună la acest capitol decât Strawman. Înainte de a descrie în detaliu protocoalele se vor face câteva observaţii, care vor stă la bază alegerii.

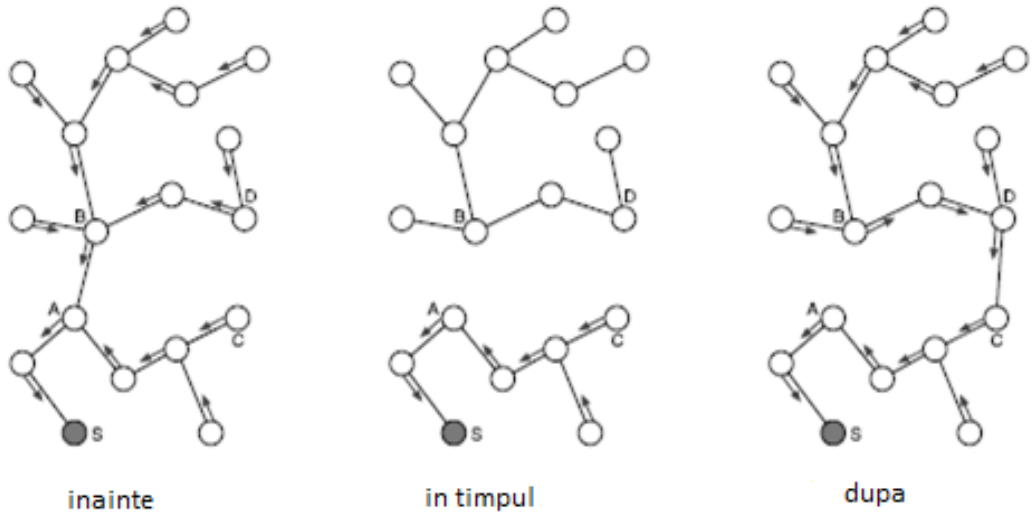


Figura 2. O rețea de noduri : înainte, în timpul și după reconfigurare folosind algoritmul Strawman.

Observaţia 1: Cu cât densitatea abonaţilor e mai mare, cu atât propagarea abonamentelor e mai scurtă. Pentru a înţelege această observaţie, e util să analizăm cum abonamentele se propagă în arbore.

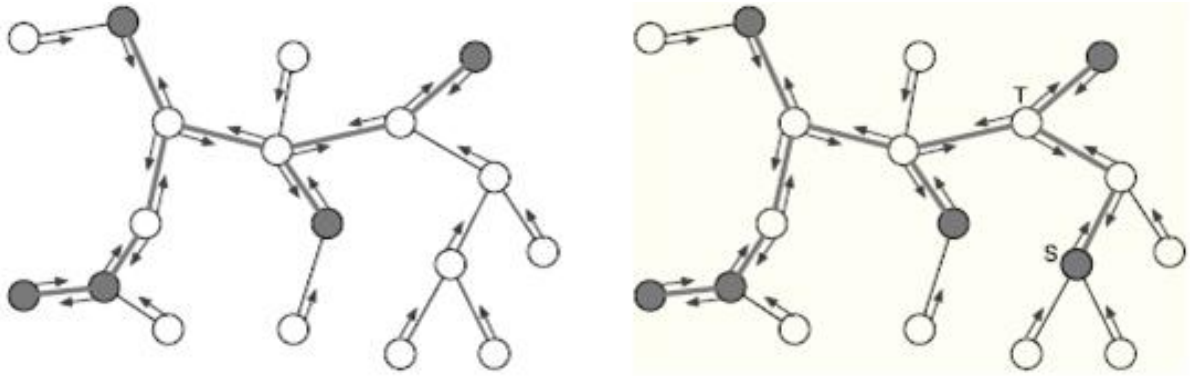


Figura 3. Propagarea subscripțiilor.

Se poate observa următoarea regulă: un abonament pentru un filtru p se propagă de-a lungul rutei unice dacă această există, altfel în tot arborele. Dacă un abonament pentru p deja există în arbore, atunci nici un abonament nu trebuie trimis. Aceeaşi regulă se aplică şi pentru dezabonari.

Observaţia 2: Abonamentele care trec peste vechea legătură pot să nu necesite propagare. Cea de-a două observaţie provine din analizarea procesului de reconfigurare a noii legături. În protocolul Strawman, capetele noii legături trimit abonamente la toate nodurile din tabelul de abonamente, pentru a se asigură că toate evenimentele de interes sunt transmise corect. Dacă acest aspect e suficient ,nu înseamnă că e şi necesar.

Să considerăm, de exemplu, situaţia prezentată în Figura în care doar un singur broker e abonat la un anumit mesaj. Se poate întâmplă că dezabonarea brokerului B(care va şterge săgeată dintre D şi B) să aibă loc încet, ajungând la D după ce noua legătură a apărut şi după ce D a schimbat abonamentele cu C.

Acest aspect cauzează înserarea de abonamente care vor fi şterse de propagarea înceată a dezabonarilor provocate de B.Cu toate acestea protocolul restaurează corect informaţia de rutare, dar nu o face într-un mod eficient. Acest fenomen poate apărea în protocolul Strawman, dar e şi mai probabil să apăra dacă operaţiile de abonare şi dezabonare sunt inversate, aşa cum s-a explicat mai sus.

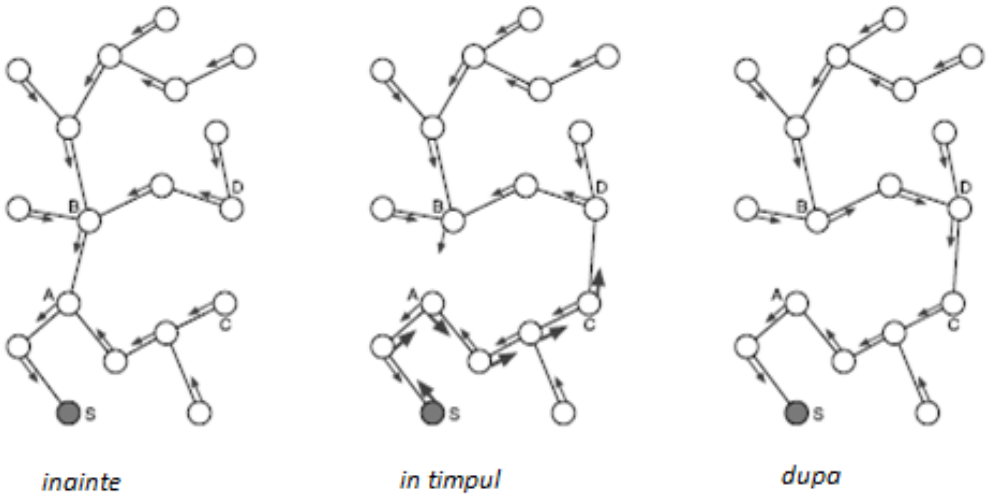


Figura 4. Subscripții create în timpul reconfigurării în situația în care abonările preced dezabonările

Observaţia cheie este aceea că legătura dintre C şi D nu e adăugată în mod izolat, dar mai degrabă că un răspuns la ştergerea legăturii dintre A şi B. Prin examinarea abonamentelor dintre A şi B, putem decide care abonamente ar trebuie schimbate între C şi D. Orice abonament care e prezent pe vechea legătură şi are ca rol doar să ruteze evenimente la un alt subarbore(de la B spre A) nu ar trebui să se propage pe noua legătură. Acest lucru e suficient pentru a preveni apariţia abonamentelor nenecesare generate în Figura 4.

Observaţia 3: Impactul reconfigurării este limitat la o cale bine-definită. Observaţia anterioară a avut ca scop reducerea propagării abonamentelor nenecesare, observaţia curentă se concentrează pe îngustarea domeniului de aplicare a reconfigurării din punctul de vedere al brokerilor implicaţi, ajutând la proiectarea de protocoale care limitează impactul reconfigurării. Pentru a găsi care brokeri sunt afectaţi, precizăm faptul că din punctul de vedere al rutării, evenimentele care intenţionau să traverseze legătură întreruptă trebuie să fie rerutate pe noua legătură pentru a ajunge pe cealaltă parte a arborelui. Observăm că doar tabelul de abonamente al brokerilor situaţi pe calea dintre vechea legătură şi noua legătură trebuie să fie schimbate[12]. Toţi ceilalţi brokeri trimit evenimentele pe drumul destinaţie din arbore, şi rămân neschimbaţi în timpul configurării. Ne referim la drumul din abore că la drumul reconfigurării şi îl definim că fiind concatenarea a două secvenţe de brokeri (lanţ de brokeri):

1. Drumul-1: începe cu primul capăt al legăturii şterse (capătul cu identificatorul cel mai mic) şi conţine lanţul de brokeri conectaţi la legătură noii legături, care se regăseşte în acelaşi subarbore, inclus că ultimul broker din capul drumului.

2. Drumul-2: începe cu cu celălalt capăt al noii legături, şi conţine brokerii conectaţi la al doilea capăt al legăturii şterse.

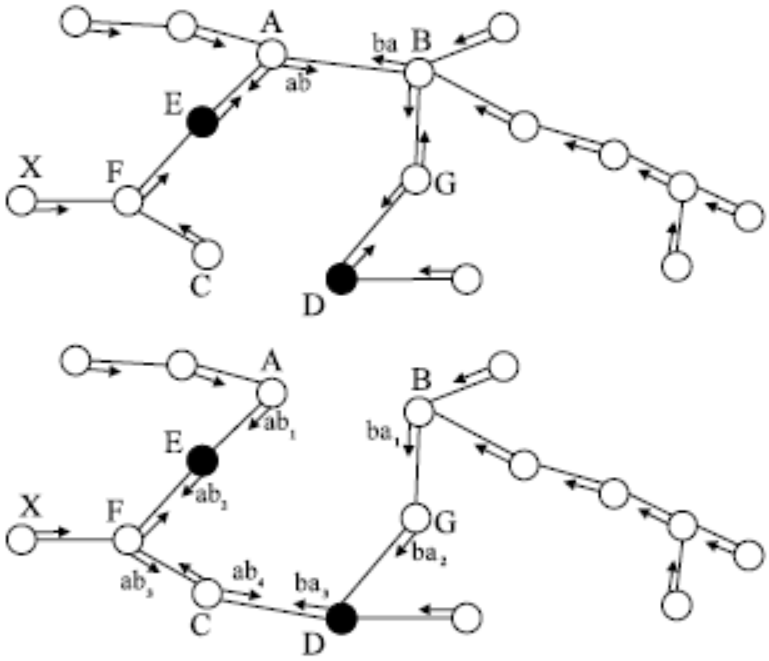


Figura 6. Reconfigurare

Figura 6 arată un exemplu de reconfigurare unde legătura (A,B) este înlocuită cu legătura (C,D). In acest caz (A,E,F,C) e drumul 1, şi (D,G,B) e drumul 2, rezultând drumul (A,E,F,C,D,G,B). Că un rezultat al reconfigurării, abonamentul ab, care se folosea de legătura ştearsă (A,B) pentru a ajunge în subarborele dinspre nodul B, e şters de reconfigurare şi e înlocuit de abonamentele ab1, ab2, ab3 şi ab4. În mod similar, din ba se obţine ba1, ba2, ba3.

Din exemplu putem deduce două considerente care ne ajută la înţelegerea mecanismului de reconfigurare. În primul rând, subarborele unui abonat întotdeauna conţine informaţia de rutare completă care să permită evenimentelor să ajungă abonatul din orice broker. În al doilea rând, unele din abonamentele, care permit evenimentelor să ajungă celalalat subarbore, pot fi deja prezente datorită altor abonamente. În acest exemplu, doar ab2, ab3 şi ab4 e necesar să fie adăugate în subarborele din stânga nodului A: ab1 poate deja rută evenimente dinspre A înspre E. Similar, în celălalt subarbore, doar ba3 e nevoie să pornească spre A, din moment ce ba1, ba2 sunt deja prezente datorită lui D.

Aceste considerente ne ajută să enunţăm un principiu general: abonamentele care înlocuiesc pe cele din primul capăt al vechii legături(din A spre B) sunt necesare doar pe drumul-cap. Similar, abonamentele care înlocuiesc pe cele din legătura veche (B spre A) sunt necesare doar pe drumul-coadă. Nici un alt broker nu e afectat de reconfigurare.

4.4. Protocoale optimizate

Bazându-ne pe observaţiile făcute în secţiunea anterioară, propunem şi descriem cele trei optimizări: DEZABONARI AMÂNATE(în două variante DEZABONARI TEMPORAR AMÂNATE şi DEZABONARI NOTIFICAT AMÂNATE), LEGĂTURI ACTIVATE CU NOTIFICARE, DRUM RECONFIGURAT. Fiecare protocol ţine cont de diferite ipoteze legat de mecanismul de gestionare a subarborelui. De exemplu protocolul DEZABONARI TEMPORAR AMÂNATE ţine cont de ipotezele regăsite în Strawman, în timp ce protocolul DRUM RECONFIGURAT se bazează pe ipoteză că notificarea trimisă de mecanismul de gestionare a subarborelui la sistemul de rutare conţine lista de brokeri din drumul reconfigurării. Protocoalele de asemenea diferă legat de complexitate şi de îmbunătăţirile aduse la performanţa sistemului.

4.4.1. Dezabonări amânate

Din observaţiile făcute în capitolul anterior rezultă că dezavantajul protocolului Strawman este faptul că procesul de dezabonare produs de ştergerea unei legături şi de procesul de adăugare de legături se desfăşoară în paralel. Protocolul DEZABONARI AMÂNATE se bazează pe observaţia 1 : cu cât densitatea abonaţilor e mai mare, cu atât propagarea abonamentelor e mai scurtă. Protocolul înlocuieşte operaţiile conventionalede abonare şi dezabonare din Strawman prin faptul că le procesează în ordine inversă: abonarile produse de apariţia unei noi legături sunt tratate imediat, în timp ce dezabonarile produse de ruperea unei legături sunt amânate. Această strategie nu introduce nici un mecanism special pentru a limită aria de acţiune de aceea poate adaugă abonări care trebuie şterse imediat. Aceste abonări se propagă doar pe subarborele dorit, distanţă fiind mică atunci când arborele e dens de abonări. Este important de enunţat faptul că reconfigurarea produsă de acest protocol nu interferează cu procesul normal de evenimente de abonare şi dezabonare.

În continuare sunt descrise cele două variante ale protocolului care diferă prin mecanismul folosit pentru a întârzia dezabonarile.

4.4.2. Dezabonări temporar amânate

În prima variantă a protocolului DEZABONĂRI AMÂNATE din Figura 6, întârzierea e produsă de un timeout, care e iniţializat pe fiecare capăt de broker al legăturii rupte. Expirarea timeout-ului declanşează propagarea dezabonarilor, de aceea am numit protocolul DEZABONĂRI TEMPORAR AMÂNATE. În mod ideal această întârziere ar trebui să coincidă cu timpul restaurării conectivităţii arborelui plus timpul necesar propagării abonarilor. Singură prezumţie necesară este că sistemul să notifice capetele de apariţia legăturilor noi şi dispariţia legăturilor vechi.

Acest protocol are avantaje semnificative faţă de Strawman. Din observaţia 2, ştim că întârzierea dezabonarilor conduce la propagarea de abonări pe legătură noua. Din nefericire, în această variantă, ne asumăm faptul că sistemul de gestiune a subarborelui nu produce asocieri între legăturile noi şi vechi, făcând optimizarea imposibilă.

Cu toate acestea, în cazul în care un capăt al legăturii noi coincide cu un capăt al legăturii vechi, nu avem suficiente informaţii pentru a preveni trimiterea de abonări nenecesare. Cu toate că acest caz pare unul izolat, este de fapt chiar des întâlnit. Unul din

capetele legăturii şterse e în mod normal responsabil să repare integritatea arborelui şi în multe cazuri devine capătul legăturii adăugate în timpul procesului de reconfigurare.

Pentru a trata acest caz eficient, prevenim sistemul de reconfigurare să trimită abonări direct spre brokeri conectaţi prin legături care sunt acum întrerupte. Toate celelalte abonări, cele care vin de la clienţi legaţi la brokeri pe care îi împart şi cei legaţi la brokeri cu legături intacte, sunt transmise în mod normal. Acest comportament e evidenţiat prin operaţia manageLegaturaStearsa(n, rid) , care e apelată atunci când legătură dintre brokerul curent şi brokerul n se întrerupe. Când acest lucru are loc, filtrele asociate vecinilor n (mulţimea filtrelor subTab[n]) sunt şterse imediat din tabele abonamentelor şi stocate într-o tabela de aşteptare. În acest mod sunt ignorate şi atunci când se procesează alte abonare(dezabonari) concurente şi atunci când se propagă abonamente la legături noi adăugate. În schimb, atunci când timpul de aşteptare (timeout time) expiră, filtrele din tabela de aşteptare se propagă, această fiind ideea de bază al protocolului optimizat şi anume propagarea amânată a dezabonarilor rezultate din reconfigurare.

Pseudocodul urmator prezinta protocolul „dezabonări temporar amânate”:

```
// Sterge legatura la vecinul n
 stergeLegaturaLa(n)
   rld ← newRld()
   manageLegaturaStearsa(n, rld)

// Adauga legatură la vecinul nou n
 adaugaLegatura(n)
  vecini ←vecini ∪{n}
 for all (n', p) ϵ asteptare[rld] do
    proceseazaUnsub(n,p)
  asteptare ← asteptare – asteptare[rld]

// Gestioneaza legatura rupta n pentru reconfigurare rld
manageLegaturaRupta(n, rld)
 if  | vecini| >= 1 then
     // self e un broker care nu e frunza
     asteptare[rld] ← subTab[n]
    subTab[n] ← subTab – subTab[n]
    vecini ← vecini – {n}
```

Reconfigurarea nodurilor broker de tip frunză din arbore va fi prezentat in cele ce urmează. Un caz special în care noua şi vechea legătură împart acelaşi capăt al legăturii este atunci când un broker de tip frunză e desprins şi apoi legat la un broker diferit. Acest caz e destul de frecvent deoarece brokerii de tip frunză reprezintă un număr mare din totalul de brokeri ai sistemului, şi deoarece sunt la marginea întregului sistem sunt potenţiali clienţi ai reconfigurării.

Particularitatea acestui caz este faptul că amânarea dezabonarilor devine inutilă în cazul unui broker frunză detaşat. Un nod de tip frunză, prin definiţie, nu are vecini cărora să le poată trimite mesaje. Prin urmare, se poate dezabona local fără să actualizeze tabele de aşteptare şi fără să seteze timpul de amânare (timeout) . Această optimizare nu doar că simplifica procesul atunci când un broker frunză e implicat în reconfigurare , dar acest protocol evită de asemenea propagarea dezabonarilor inutile de-a lungul noii legături atunci când expiră timpul de amânare.

4.4.3. Reducerea dependenţei de timer: Dezabonări notificat amânate

În protocolul Dezabonari Temporar Amânate un rol important îl are timpul de amânare. Dacă acest timp este prea mic, performanţa sistemului se apropie de cea a protocolului Strawman. Dezabonarile sunt trimise prea devreme înainte ca abonarile să fie restaurate, iar dacă timpul este prea mare, trasee învechite rămân în vigoare şi sunt trimise evenimente în locuri în care nu există abonaţi, afectând performanţa sistemului. Protocolul Dezabonari Notificat Amânate încearcă să îmbunătăţească protocolul descris anterior. Pentru această modificare, presupunem faptul că sistemul de reconfigurare al arborelui poate să asocieze legătură ştearsă cu legătură adăugată printr-un identificator unic. De asemenea apelurile la stergeLegatura sunt făcute înainte apelurile la adaugaLegatura pentru a asigură etichetarea corectă a legăturilor rupte prioritare trimiterii abonamentelor. Aceste asumpţii permit folosirea unui mesaj Flush, care e trimis de capătul legăturii noi spre capătul legăturii vechi, în momentul în care se termină propagarea abonamentelor. Toate lagaturile sunt FIFO, atunci când capetele legăturii vechi primesc mesajul flush, acestea ştiu că toate abonamentele de la noua legătură s-au propagat spre vechea legătură şi pot începe procesul de dezabonari.

Cu toate că mesajul flush ţine loc de timer, şi anume pentru a porni propagarea dezabonarilor, e posibil datorită reconfigurării concurente că mesajele să nu ajungă la capetele legăturii vechi. În acest caz, dezabonarile încep să se propage atunci când timer-ul expiră.

Pentru simplicitate, mesajul Flush e trimis broadcast în întreg arborele, cu toate că trebuie să se propage doar de-a lungul drumului de reconfigurat pentru a ajunge la capetele legăturii vechi. Sunt două modalităţi de a optimiza această propagare. Prima variantă constă în beneficia de informaţia legată de reconfigurare furnizată de sistemul de mentenanţa a arborelui. A două variantă e de a exploata propagarea abonamentelor trimise de capetele legăturii noi pentru a ghida propagarea mesajului de Flush.

4.4.4. Activarea anunţată a legăturii

Am arătat că reconfigurarea inutilă provine din două surse. În primul rând, potrivit observației 1, dezabonările de la legatura veche se pot propaga inutil și pot șterge temporar rute care sunt necesare după reconfigurare. In al doilea rând, potrivit obervatiei 2, abonările de la legatura noua se pot propaga nenecesar rutând informație, care e necesara doar înainte ca reconfigurarea să se producă.

Ambele protocoale, de tipul Dezabonări Amânate, rezolvă problema prin amânarea propagării dezabonărilor până când abonările legăturii noi se termină de propagat. A doua problemă e atunci când legatura nouă ți legătura veche impart un capăt. Legatura pe care o împart identifică abonarile care sunt pe legatura intreruptă și evită propagarea lor.

Algoritmul „activare anuntata a legaturii”:

```
// Inlocuire legatura (self,o) cu (n1,n2)
// rld e identificatorul reconfigurarii
inlocuiesteLegatura(o,n1,n2, rld)
  // n1 e in acelasi subarbore ca si self
```

```
   P ← {p| numai o e subscris la p}
   vecini ← vecini –{o}
   start unsubTimer(o,rld)
   trimite activare(rld,P,n2) la n1

// Timerul de dezabonare expira, n e vechiul vecin,
// rld e identidicatorul reconfigurarii
 unsubTimerExpira(n, rld)
 for all (n,p) ∈ subTab do
    proceseazaUnsubDela(n,p)

// Timer-ul de abonari expira
subTimerExpira(n, rld)
  for all p ∈ taggedSubs[rld] and n' difera de n then
      trimite SUB(p) la n
  anuleaza subTimer(n, rld)
  taggedSub[rld] = ∅
// Flush primit
 FlushPrimit(n, Flush(rId))
   If self has unsubTimer(n', rld) then
        // n' e un alt capat al vechii legaturi
         anuleaza unsubTimer(n' , rld)
  else
        // self nu e capat al vechii legaturi
       Trimite Flush(rId) la toti vecinii exceptand pe n

// Activate primit
activatePrimit( ACTIVATE(rId, P, n))
    vecini ← vecini ∪ {n}
     start subTimer(n,rld)
    for all p ∈ subTab do
       if p nu apartine subTab then
              trimite SUB(p) la n
       else
          taggedSubs[rld] ← taggedSubs[rld] + {p}
trimite FLUSH(rld) la n.
```

Acest protocol extinde comportamentul si functionalitatea protocolului de DEZABONARI AMANATE prin abordarea Observatiei 2 chiar si atunci cand capetele legaturii noi si vechi nu sunt partajate. Pentru a realiza acest aspect propaga informatia despre subscriptiile nenecesare de la vechiul nod spre noul nod, permitand capetelor noii legaturi sa recunoasca aceste subscriptii si sa evite propagarea lor. Protocolul se bazeaza pe faptul ca e chemata operatia *substituieLegatura* pe fiecare capat de legatura veche atunci cand apare un scenariu de reconfigurare.

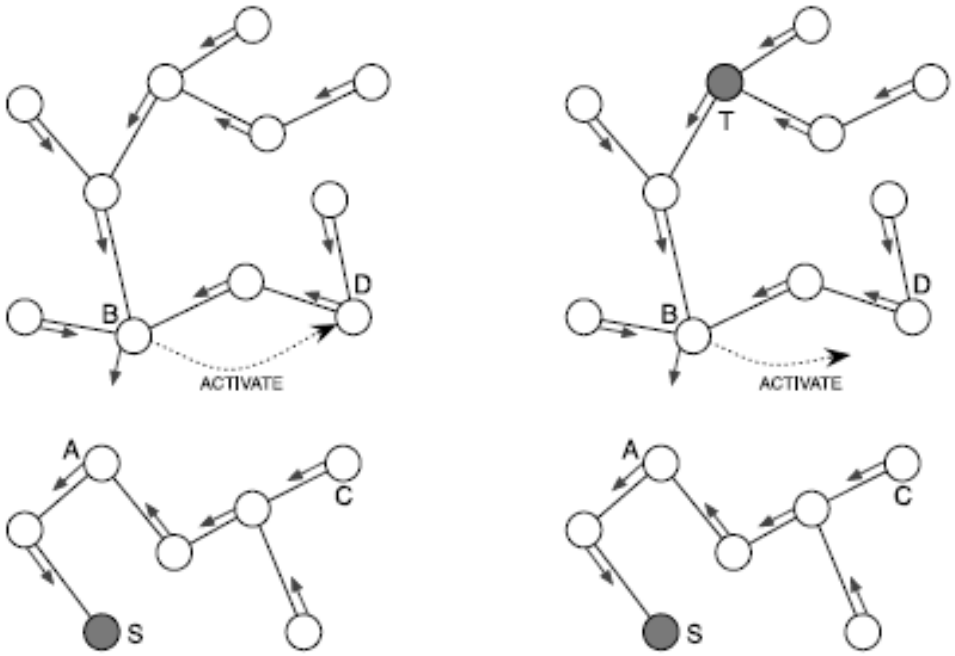


Figura 7: Scenarii ale protocolului *Activare anunțată a legăturii*.

În momentul operaţiei de substituire a legăturii vechi cu cea nouă un mesaj de ACTIVARE se trimite către noua legătură. Aceasta când primeşte mesajul propagă suscriptiile sale prin noua legătură. Procesul e similar cu cel întâlnit la operaţia adaugaLegatura întâlnit în protocoalele precedente, cu excepţia faptului ca aici nodul trebuie să fie local.

De exemplu în Figura 7 dacă legătura (A,B) se rupe, B trimite la D un mesaj de ACTIVARE unde P conţine pattern-ul pentru subscripţia lui B către A. Din momentul primirii acestui mesaj, D nu mai propagă subscripţii pe noua legătură, evitând în acest fel generarea de subscripţii străine, inutile.

Procesul de dezabonare e tratat în acelaşi fel că şi în cele 2 versiuni ale protocolului de DEZABONARI AMÂNATE.

Modalitatea folosită şi explicată e suficientă în condiţiile în care o reconfigurare apare izolată, în cazul apariţiei reconfigurarilor în paralel, între noduri apropiate care se influenţează unele pe celelalte, apăr propagări de abonări şi dezabonari nenecesare, pe care protocolul încearcă să le evite.

4.5. Nivelul de infrastructură de bază (overlay)

În această secţiune e prezentat nivelul de infrastructură de bază numit overlay. Acest nivel se situează la baza întregii arhitecturi şi constituie interfaţa middleaware-ului cu reţeaua şi sistemul de operare. Acest nivel reprezintă abstractizarea nodurilor de reţea într-o topologie de tip arbore fără rutare. Overlay-ul e un subset, un arbore format din graful complet reprezentat prin reţeaua fizică şi e creat prin mecanismul de rutare IP.

Acest nivel gestionează şi implementează toate aspectele legate de reţea. Nivelul de overlay de asemenea oferă o interfaţă uniformă nivelului de rutare. Acest nivel poate fi uşor înlocuit cu altă implementare specifică de exemplu reţelelor fără fir. Nivelul de overlay e construit deasupra nivelului TCP/IP al reţelei cât şi pe bază unei topologii de tip arbore.

Cerinţele necesare pentru implementarea corectă a nivelului de overlay sunt prezentate în continuare.

Prima cerinţă este în mod clar mentinerea unei topologii conectate şi aciclice care sa ramana asa în ciuda schimbărilor arbitrare aparute la nivelul reţelei fizice.

În mod evident, menţinerea conectivităţii în prezenţa de eşecuri este posibilă numai dacă un număr suficient de mare de noduri rămân în viaţă. Atunci când acest lucru nu se întâmplă, partiţiile poate să apară; cu toate acestea, protocoalele trebuie să fie capabil să se întoarcă într-o topologie conectata cât mai curând posibil.

Managementul numarului de vecini ai unui nod e a doua conditie. În mod special costul pentru un dispecer sa transmita o notificare este proporţională cu numărul de vecini ai dispecerului. Astfel, pentru a reduce acest cost, topologia nu ar trebui să aibă noduri cu un numar de vecini prea mare. Dacă se întâmplă acest lucru, nodurile cu vecini prea multi risca să fie împovărate cu costul de calcul asociat la mesajele de expediere, în timp ce cele cu cativa vecini, cum ar fi noduri frunza suportă costuri foarte reduse. Gradul maxim al fiecare nod ar trebui să fie limitat, prin urmare, prin luarea în considerare a puterii sale de procesare.

A treia cerinţă are ca scop să reducă la minim costurile reconfigurări la nivelul de rutare. Informaţii de rutare menţinută la fiecare nod trebuie să fie modificate ori de câte ori apar schimbări care stau la baza topologie. Prin urmare, modificările legate de deconectarea unui nod ar trebui să fie localizate la intr-o regiune în imediata apropiere a nodului deconectat. Această restricţie reduce efortul de reconfigurare, astfel acest middleware devine mult mai eficient.

Necesitatea de a continua raspandirea mesajului chiar şi atunci când in topologie apar modificări impune ca stratul de overlay să poată să se reconecteze la overlay repede după o deconectare sau un eşec. O scurtă perioadă de reconfigurare limiteaza numărul de notificări care ar putea fi pierdute din cauza reconfigurarii.

LSTree este o solutie pentru a răspunde cerinţelor descrise anterior prin păstrarea noduri conectate în medii caracterizate printr-o frecvenţă ridicată de eşecuri. Aspectul cheie al protocolului este capacitatea de a furniza middleware-ului o topologie de overlay care garantează atât eficienţa cat si fiabilitate. Acest lucru este realizat cu un proces de reconfigurare eficient care permite nodurilor să reactioneze la deconectarea veciniilor lor.Protocolul este deosebit de potrivit pentru a fi integrat în contextul publish-subscribe, datorită capacităţii sale de a gestiona gradul unui nod şi de a păstra reconfigurările localizate în regiuni mici, în jurul nodurilor care eșuează.

Cu protocolul LSTree, scopul este de a organiza nodurile de reţea într-o structură arborescentă înrădăcinată reacţionează propt la conectare şi deconectare de noduri.

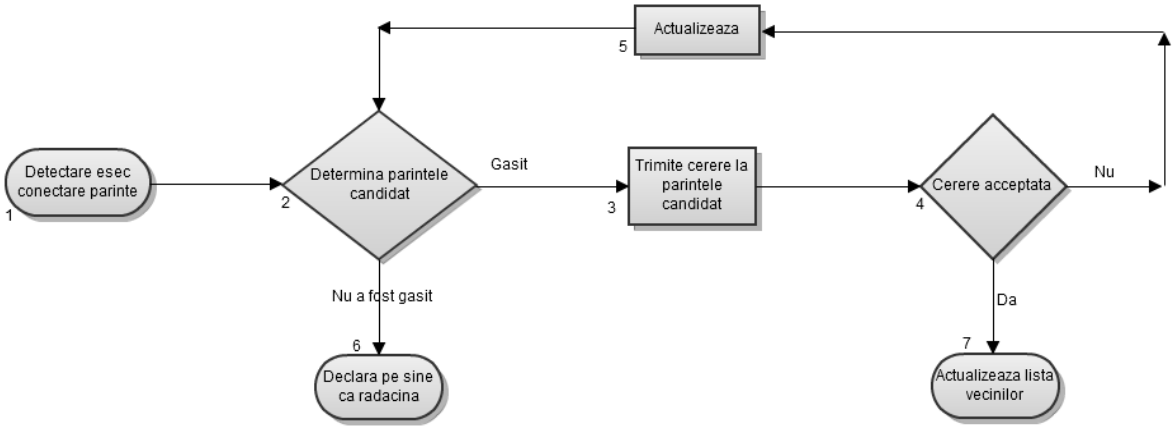


Figura 8. Determinarea unui nou părinte

Protocolul funcţionează într-un mod pe deplin distribuit. Fiecare nod înregistreaza identitatea vecinilor(parinte şi copii) într-o lista de vecini. La nivel de aplicaţie semnalizatoare sunt utilizate pentru a monitoriza membrii acestei liste si pentru a detecta deconectari. Atunci când un nod cu un singur părinte şi copii n frunze paraseste overlay-ul, protocolul repara arborele prin inserarea de noi n legături. Copiii nodului deconectat isi asume un rol activ şi fiecare dintre ie localizează un nou parinte. În mod similar, atunci când un nod nou se alătură overlay-ului, aceasta caută un părinte şi stabileşte o singura noua legătură.

Căutarea unui nou parinte constituie elementul central al protocolului de overlay şi este prezentat în Figura . Un nod detecteaza eşecul parintelui si determină identitatea unui nod care ar putea servi ca un înlocuitor (etapa 2) şi îi trimite un mesaj (pasul 3).Parintele candidat este responsabil pentru a decide dacă poate accepta cererea în condiţii de siguranţă, fără a crea cicluri şi fără a încălca alte cerinţe de acoperire. În cazul în care cererea este acceptată (pasul 5), cele două noduri isi actualizeaza lista lor de vecini, iar procesul se termină cu succes. În caz contrar, candidatul parinte trimite un mesaj de refuz specificând motivul pentru refuzul nodului solicitant, care reacţionează prin selectarea unui alt candidat şi reporneste procesului.

Aceşti paşi sunt iterati, fie până cand o cerere este acceptată sau până cand nodul solicitant este în imposibilitatea de a localiza orice noi candidaţi. În acest ultim caz, nodul îşi asumă faptul că nici un părinte adecvat există şi aceasta se alege pe el însuşi ca nod rădăcină (pasul 6). Se poate întâmpla în două cazuri. Primul şi cel mai comună este deconectarea unui nod radacina. Copiii de la rădăcină nu cooperează pentru a repara arborele astfel el se declare ca noua radacina. Al doilea caz apare în scenariile cu eşecuri de seturi mari de noduri, care conduc la partiţii temporare. Protocolul LSTree fuzionează astfel de partiţii avand noduri rădăcină care căuta periodic noduri în arbori diferiti.

Cele două aspecte-cheie ale protocolului sunt strategiile utilizate de către nodurile solicitant pentru a găsi noi candidaţi la pasul 2 şi capacitatea candidatilor părinţii să accepte şi să refuze solicitările de conectare la pasul 4.

Candidații părinți joacă un rol important în pasul 2 din protocol, deoarece decizia lor de a accepta sau respinge în cele din urmă afectează proprietăţile de acoperire. In cele ce urmeaza descriem aceste proprietăţi distinge între o cerinţă necesara, care trebuie să fie întotdeauna îndeplinita, şi o cerinţă optionala, care, deşi de dorit, pot fi incalcate pentru a permite reconectarea.

O cerinta necesara este aceea de a avea un singur arbore aciclic. Scopul principal este de a organiza nodurile de reţea într-un singur arbore, aciclic. Pentru a asigura libertatea de ciclu, vom defini o ordonare parţială n ! P („!" = se poate conecta la), care exprimă corectitudinea ca n să fie un copil pentru p, doar pe baza informaţiilor cunoscute de n şi P. Această proprietate este întotdeauna garantata între toate perechile parinte copil astfel pentru întregul arbore. De fapt, un candidat p parinte va respinge o cere de la n un alt nod în pasul 4 al protocolului, dacă n! P nu este adevarat. O definiţie viabilă a! se bazează pe o valoare întreagă menţinut la fiecare nod reprezentând distanţa sa actuală (numărul hopurilor) de la rădăcină. Un nod fără un părinte se poate conecta la orice nod cu un număr mai mic decat numarul de hopuri proprii. O astfel de conexiune se bazează doar pe informaţiile cunoscute, la cele două noduri şi nu poate introduce cicluri.

O cerintă opționala este gradul nodului. Acest aspect e tratat prin introducerea unei limite maxGrad care daca este atinsa influenteaza nodul ales ca candidat pentru a fi ales ca părinte să refuze si să oblige un alt nod sa fie selecționat.

5. Proiectare de detaliu si implementare

Reconfigurable Publish Subscribe Middleware e un framework de clase Java util pentru a construi aplicaţii de tipul publish-subscribe pentru reţele dinamice de dimensiune mare.

Aplicaţiile publish-subscribe sunt organizate ca o colecţie de componente, care interacţionează prin publicarea de mesaje şi prin abonarea la clasa de mesaje de care sunt interesate. Aceste aplicaţii sunt implementate pe baza unui middleware publish-subscribe, care dispune de un dispatcher, responsabil în colectarea abonamentelor şi trimiterea mesajelor de la publisheri către abonaţi.

Dispecerul e implementat distribuit[13] într-un set de brokeri conectaţi într-o reţea.

5.1. Caracteristicile arhitecturii

1. Arhitectura modulară permite programatorilor să adapteze cu uşurinţă sistemul propriilor necesităţi prin definirea propriului format de mesaje,filtre şi a mecanismul utilizat intern de fiecare broker pentru a compară şi trimite mesajele.

2. Suportă reconfigurare la schimbările topologiei reţelei, fiecare broker include două module: TopologyManager şi Reconfigurator, care conţin logică necesară pentru a gestiona în timpul rulării reconfigurarea reţelei, prin reacţionarea la schimbările apărute în reţeaua fizică, reconfigurarea în cazul apariţiei sau dispariţiei unui broker.

3. Suportă răspunsuri la mesaje, oferind o comunicare bidirecţională. Brokerii sunt capabili să menţină numărul de răspunsuri aşteptate pentru fiecare mesaj şi să verifice dacă toate au fost primite, o trăsătură care nu poate fi obţinută prin implementarea de răspunsuri la nivelul deaplicatie.

5.2. Broker API

Componentele unei aplicaţii construite bazate pe continut accesează serviciile publish-subscribe printr-un obiect care implementează interfaţă DispatchingService. Aceasta pune la dispoziţie metode ca subscribe şi unsubscribe , publish , primeşte mesaje, trimite răspuns. Programatorii îşi pot defini propriul lor format de mesaj şi propriul filtru prin moştenirea clasei Message şi implementarea interfeţei Flter.[14]

Dispecerul e organizat ca un set de brokeri, conectaţi printr-o reţea overlay. Fiecare broker e intern structurat în 2 nivele: nivelul de overlay şi nivelul de rutare.

Scopul nivel ului de overlay de a implementa mecanismul de management al nodurilor, al legăturilor fizice dintre brokeri. Conţine protocolul care menţine reţeaua de overlay conectată atunci când topologia reţelei se schimbă, datorită căderii de host-uri.

Cu toate ca o singură componentă ar fi putut oferi serviciul descris, totuşi sunt separate modalităţile cum informaţia e transportată de la un broker la altul şi cum se construieşte şi menţine reţeaua overlay. Nivelul de rutare interacţionează cu o singură componenţa care implementează interfaţa *Overlay*. O clasă abstractă *GenericOverlay*, implementează *Overlay* prin delegarea responsabilităţilor sale la componentele : Transport şi TopologyManager.

Clasa *Transport* e responsabilă de deschiderea şi închiderea conexiunilor la alţi brokeri ai reţelei şi prin managementul mesajelor schimbate direct cu brokerii conectaţi direct. De asemenea oferă metode pentru înregistrarea de ascultători care vor fi notificaţi atunci când o legătură noua e deschisă sau o legătură veche e inchisasau nu mai e accesibilă.

Clasa *TopologyManager* menţine informaţia despre vecinii nodului local şi implementează protocolul care menţine overlay-ul conectat. De asemenea implementează metode oferite de interfaţa *Overlay* pentru a înregistra ascultători care sunt notificaţi atunci când un vecin nou e adăugat sau un vecin deja existent e şters sau nu mai e accesibil.

Nivelul de rutare oferă următoarea funcţionalitate sistemului: menţine abonamentele clienţilor şi foloseşte *Overlay*-ul pentru a rută mesaje de la publishers spre abonaţi. Include următoarele componente:

a. *Routerul* e elementul central al middleware-ului. Implementează procedura de rutare prin înregistrarea la nivelul de overlay a unor ascultători care notifica atunci când abonamente, mesaje şi răspunsuri ajung de la vecini. Foloseşte *SubscriptionTable* pentru a stoca informaţia despre vecinii abonaţi care sunt folositori pentru rutare şi delega operaţiile de bază către alte două componente: *RoutingStrategy*, care foloseşte strategia de rutare cea mai potrivită pentru a decide cum abonamentele şi mesajele sunt rutate în arborele reţelei, iar *ReplyManager* este responsabil de rutarea răspunsurilor către publisherii corespunzători, bazat pe informaţia stocată în replyTable.

b. *Reconfiguratorul* e responsabil de garantarea corectitudinii rutării şi păstrarea consistenţei sistemului atunci când topologia reţelei se schimbă prin adăugarea de noduri sau ştergerea acestora.

Similar altor sisteme publish-subscribe, sistemul adoptă o arhitectuara distribuită organizată ca un set de brokeri conectaţi într-o reţea dispatching-overlay. Pentru a suporta reconfigurare dinamică brikerii sunt structuraţi în două nivele: nivelul de infrastructura de bază(overlay layer) şi nivelul de rutare(routing layer). Cele două nivele sunt independente.

Nivelul de overlay are rolul de a gestiona topologia reţelei. Pune la dispoziţie servicii pentru a construi reţeaua overlay şi pentru a reorganiza nodurile în funcţie de mesajele primite de la nivelele superioare. Menţine conectate nodurile reţelei de brokeri atunci când apare o schimbare prin pierderea unei legături la un nod, schimbări fizice apărute în reţea de calculatoare (deoarece unele host-uri se pot deconecta în orice moment datorită pierderii conexiunii fizice). A fost ales un design care decuplează detaliie legate de cum sunt transportate datele între brokeri, rezultând un desing mai clar în acest fel.

Clasa *GenericOverlay* implementează interfaţa *Overlay* prin delegarea serviciilor sale componentelor *Transport* şi *TopologyManager*.

Componenta *Transport* asigură metode pentru deschiderea şi închiderea conexiunii (open, close) unei legături către alt broker şi metode pentru a trimite şi recepţionă mesaje între brokerii direct conectaţi. Pentru a asigura un punct de legătură pentru *TopologyManager* , clasa *Transport* are metode care asigură înregistrarea de listeners care să notifice când o legătură noua e adăugată sau când legături existenţe cad. Include două implementări pentru interfaţa *Transport* folosind *UDP* şi *TCP*. Mesajele sunt encodate folosind serializarea Java.

Clasa *TopologyManager* implementează protocolul care întreţine overlay-ul conectat şi face acest lucru garantând faptul ca constrângerile topologiei rămân acelea impuse de strategia protocolului de rutare(ex: că nici o buclă să nu existe în arbore) .

Această componenţa are de asemnea metode apelate de nivelul de rutare pentru a manipulă explicit overlay-ul(ex: adăugarea sau ştergerea de brokeri) şi pentru a accesă lista de brokeri vecini. De asemenea implementează metodele interfeţei *Overlay* pentru a se înregistra că ascultător la schimbările apărute în setul de vecini. Aceste metode sunt folositoare pentru a permite componentelor din nivelul de rutare să reacţioneze la schimbările apărute în topologie. Un manager de topologie care nu restricţionează prezenţa unei anumite topologii şi nu încearcă să păstreze reţeaua de brokeri conectată atunci când legături, brokeri cad. Restricţionează doar apariţia a mai mult de o conexiune spre acelaşi vecin.

Nivelul de rutare ia deciziile importante legate de rutarea mesajelor, decide unde transmite anumite evenimente specifice. Acest nivel stochează subscripţiile clienţilor şi foloseşte Overlay-ul pentru a rută mesajele de la publishers spre subscriberi.

Componenţa de bază a nivelului de rutare este clasa *Router* , care implementează mecanismele principale de rutare. Router-ul primeşte notificări de la componenţa Overlay despre abonamente, mesaje şi răspunsuri primite de la vecini (alţi brokeri sau clienţi). Clasa *GenericRouter* implementează interfaţa Router şi delegă funcţionalitate componentelor: *SubscriptionTable*, care e responsabil cu stocarea eficientă a subscripţiilor primite de la vecini. Atunci când un nou mesaj ajunge, *SubscriptionTable* e responsabil cu alegerea şi potrivirea mesajelor cu filtrele şi subscripţiile stocate deja pentru a determină vecinii subscrişi, implementând „stategia de potrivire”.

Descriptorul nodului encapsuleaza identificatorul unic şi alte informaţii suplimentare despre nod cum ar fi adresa URL folosită pentru a-l localiza sau flag-ul dacă e broker sau client.

Tabele *SubscriptionTable* e folosită de broker pentru a stoca informaţii despre subscripţiile (filtre) primite de la vecini. Este o interfaţă generică, clasele care implementează această interfaţă pot diferi prin stategia de rutare adoptată pentru a stoca subscripţiile.

GenericTable stochează toate filtrele primite ale fiecărui vecin şi verifică dacă mesajele primite din afară se potrivesc filtrelor existenţe invocând metodă Filter.match. Cu toate ca această metodă nu e cea mai eficientă totuşi nu face nici o presupunere legată de formatul mesajelor sau filtrelor. *PtreeTable* implementează un mecanism mai specializat şi eficient sub formă de perechi de atribute sau conjuncţii de predicate.

Interfaţa *Router* e „core-ul" brokerului. Clasele care implementează această interfaţă constituie componentele de bază ale brokerului. E înregistrată la *Overlay* pentru a primi notificări despre mesaje venite de la alte noduri ale reţelei. În cele mai multe cazuri delega funcţionalitatea la celelalte componente care alcătuiesc brokerul la *RoutingStrategy* şi *ReplyManager*. Deţine datele comune gestionate de acele componente cum ar fi: *SubscriptionTable* şi *ReplyTable*.

După cum sugerează şi numele *RoutingStrategy* implemeneaza strategia specifică de rutare, care determină cum subscripţiile şi mesajele sunt rutate de-a lungul reţelei. RoutingStrategy e modulul responsabil cu managementul rutării mesajelor. Metodele sale sunt chemate de overlay pentru a efectuă operaţiile de bază: procesarea subscripţiilor şi dezabonarilor şi trimiterea mai departe a mesajelor.

ReplyManager are rolul de a rută răspunsuri către publisherii corespunzători, bazându-se pe informaţia stocată în *ReplyTable*. Interacţionează cu structură de date ReplyTable în care sunt stocate toate înregistrările răspunsurilor cum ar fi Id-ul celui care a trimis, timeout-ul expirării, numărul de vecini la care a fost trimis mesajul.

Componenta *Router* guvernează implementările bazate pe topology statice. Componenta Reconfigurator are rolul critic de a garanta că informaţia de rutare şi conţinutul din tabele de subscripţii rămâne consistenţă atunci când topologia reţelei se schimbă. Acest lucru e realizat prin încorporarea unor protocoale adecvate care să reconcilieze asemnea informaţie.

În broker *Reconfigurator-ul* e responsabil cu reconfugurarea. El trimite subscripţiile, dezabonarile după cum este necesar. E înregistrat la *Overlay* pentru a primi notificări despre schimbările vecinilor sau a nodului local.

Reconfiguratorul „dezabonari amanate" întârzie dezabonarile pentru un broker care părăsește rețeaua. În acest fel Managerul de topologie are timpul necesar restaurării conectivităţii legăturilor şi permite subscripţiilor noi să se propage în reţea înainte că dezabonarile să înceapă, reducându-se în acest fel overhead-ul. *Reconfiguratorul* „Activarea anunţata a legaturilor" aduce de asemnea reducerea overhead-ului.

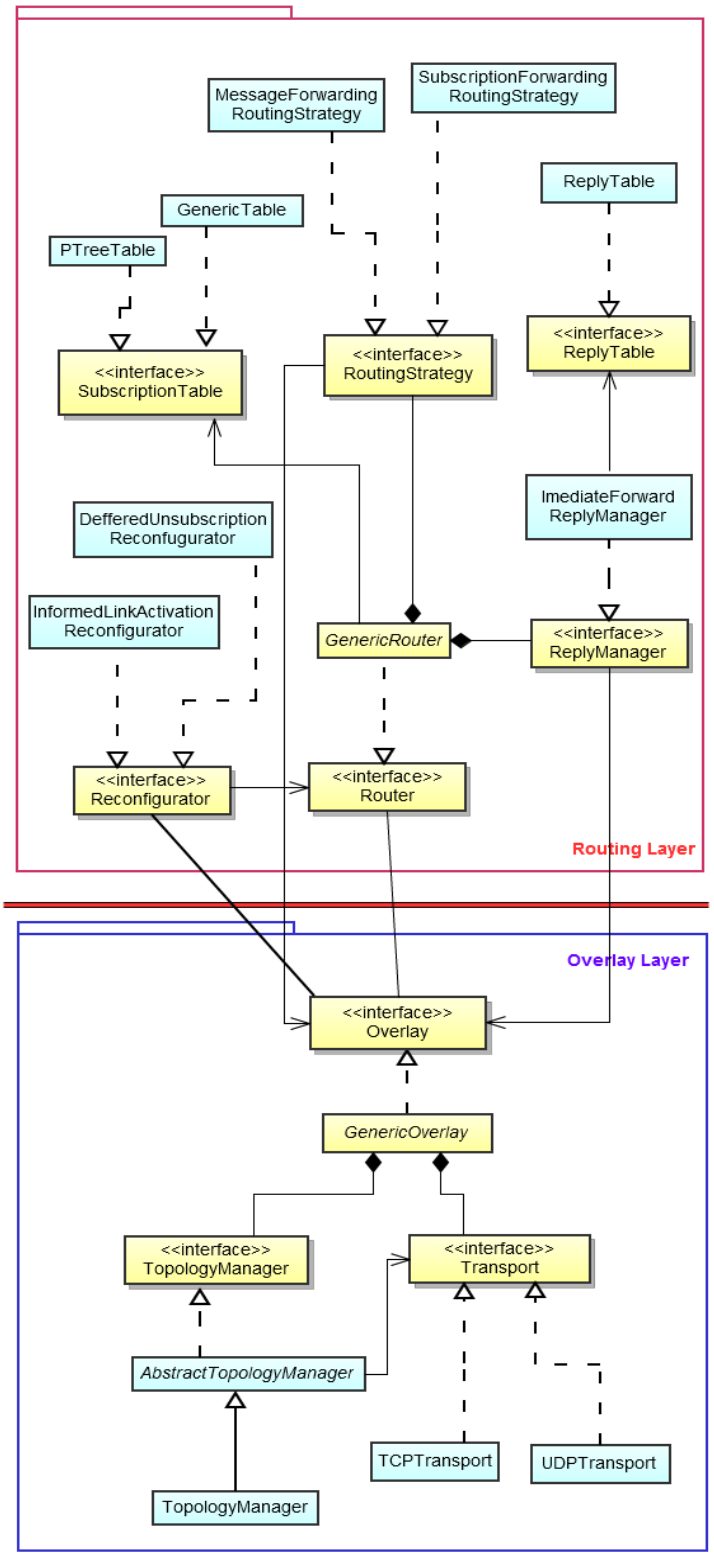


Figura 9. Diagrama de clase și interacțiunea dintre cele două nivele

5.3. Client API

Componentele aplicaţiei accesează serviciile puse la dispoziţie prin interfaţa DispatchingService. Aceasta oferă două implementări care diferă prin protocolul de transport folosit pentru a se conecta la brokeri(*UDP* şi *UTP*) şi de asemenea interfaţa ascunde toate detaliile legate de cum un clienţii sunt conectaţi şi comunică prin reţea. E independentă de formatul mesajelor şi a filtrelor. Acest aspect legat de filtre oferă flexibilitate prin faptul că se pot crea mesaje şi filtre customizate cu semantici diferite fără a afecta infrastructura reţelei. Pentru atingerea acestui scop e definită clasa abstractă pentru mesaje şi două interfeţe pentru filtre, care includ setul de metode necesare pentru publish-subscribe să funcţioneze.

Clasa abstractă *Message* incorporează identificatorul mesajului calculat la momentul publicării. Interfaţa *Filter* defineşte metoda matches care poate fi redefinita de clasa care implementează interfaţa cu logică de filtrare dorită[16]. În plus, metoda hash code şi equals sunt folosite intern de brokeri pentru a stoca şi compara filtre. Interfaţa filtrului *ComparableFilter* reprezintă filtre care sunt acoperite de alte filtre existente.

Frameworkul suportă de asemenea posibilitatea ca clienţii să poată trimite mesaje de răspuns la mesaje, oferind în acest fel comunicare bidirecţională. Brokerii ţin evidenţa de tranzitul de mesaje etichetate şi ţin rutele de întoarcere a mesajelor răspuns prin metoda reply a *DispatchingService*. Suportând răspunsuri nativ, brokerii sunt capabili să urmărească numărul de răspunsuri aşteptate pentru fiecare mesaj şi pot verifică dacă toate dintre ele au fost primite, trăsătură care nu se poate obţine la nivelul aplicaţiei.

Interfaţa *DispatchingService* e interfaţa client pentru a accesa serviciul de evenimente. Este o interfaţă mai generală. Clasele care implementează această interfaţă diferă prin protocolul folosit pentru a accesa dispatching network.

Metodele acestei interfete sunt:

- close(): închide conexiunea la dispatching network.
- getAllReplies(MessageID, repliableMessageID): returnează toate răspunsurile pentru un mesaj.
- getNextMessage(): returnează primul mesaj găsit şi-l şterge din coadă de mesaje primite.
- getNextMessage(Filter f): returnează primul mesaj care se potriveşte filtrului f şi-l şterge din coadă de mesaje primite.
- getNodeDescriptor(): folosit pentru a accesă identificatorul asignat clientului.
- hasMoreMessages(): verifică dacă mai sunt mesaje disponibile.
- hasMoreMessages(Filter f): verificaţi dacă există mesaje disponibile care se potrivesc cu filtru specificat.
- hasMoreReplies(MessageID repliableMessageID): verifică dacă există un răspuns în coadă de mesaje pentru mesajul specificat.
- isOpened(): verifică dacă dispatching service e deschis.
- open(): deschide conexiunea la dispatching network.
- publish(Message msg): publică un nou mesaj.
- reply(Message reply, MessageID repliableMessageID) : trimite un răspuns.
- subscribe(Filter filter): se subscrie la un mesaj care îndeplineşte condiţiile filtrului.

- unsubscribe(Filter filter): se dezabonează de la mesaje care îndeplinesc condiţiile filtrului.
- unsubscribeAll(): şterge toate abonarile făcute.

Interfata *Filter* e folosita pentru a te subscrie la o clasă specifică de mesaje, mai exact la mesaje care îndeplinesc condiţiile unui filtru specificat. E interfaţă cea mai generală. Clasele care implementează această interfaţă sunt filtre specializate.

Metodele definite de aceasta interfata sunt:

- equals(Object o): testează dacă un filtru e egal cu alt filtru.
- matches(Message msg): testează dacă mesajul îndeplineşte condiţiile acestui filtru.

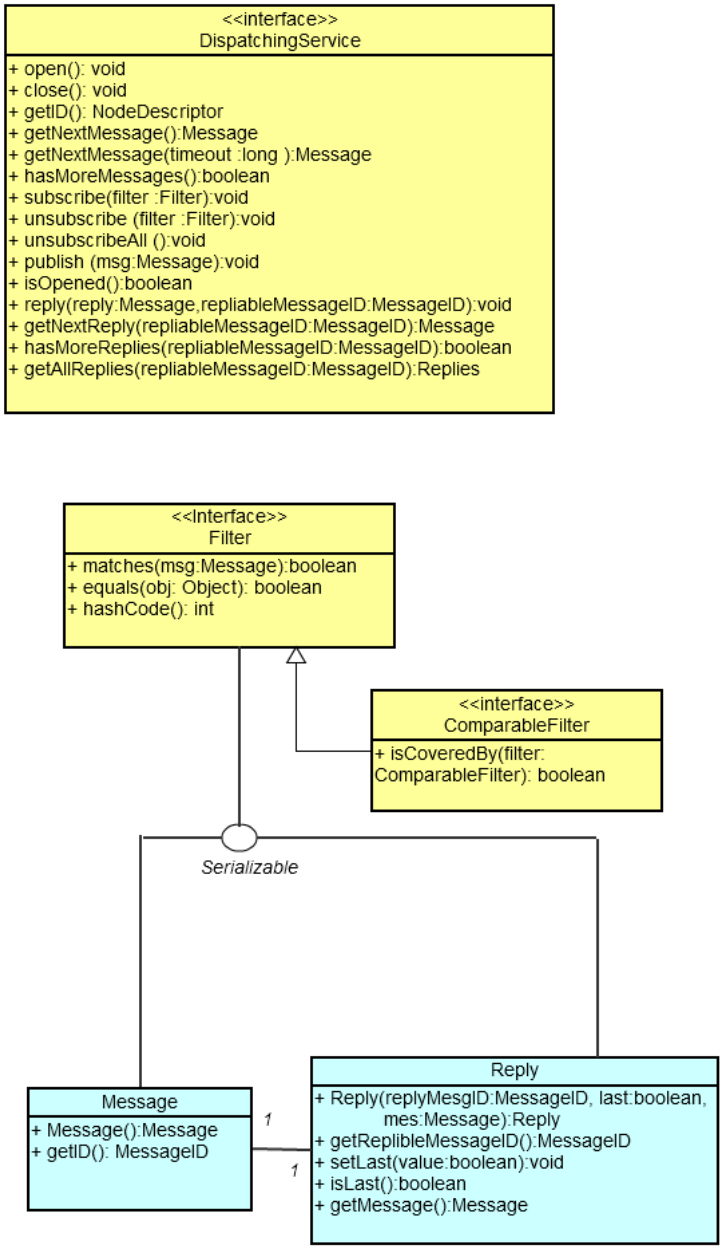


Figura 10. Client API

Repliable indica dacă un mesaj poate să returneze un răspuns sau nu.

Clasa *Message* e o clasa abstractă care defineşte un mesaj, are un ID unic MessageID.

MessageID reprezintă Id unic şi universal al mesajului. NodeDescriptor descrie un nod (broker sau client). Încapsulează identificatorul unic al nodului şi alte câteva informaţii despre nod(dacă este un client sau un broker şi URL care se foloseşte pentru a localiza nodul).

Clasa *Replies* reprezintă un set de răspunsuri la mesajele publicate. Dacă toate mesajele au fost recepţionate atunci *areAllReplies*() are valoarea true, altfel are faloarea false.

Clasa *Reply* reprezintă răspunsul pe care îl primeşte un broker. Fiecare răspuns are un ID.

TCPDispatchingService e clasa responsabilă de comunicarea tcp dintre noduri.

Excepţiile aruncate sunt *NotConnectedException* şi *ReplyTimeoutException*.

NotConnectedException apare atunci când se încearcă transmiterea de data la un broker vecin şi acesta nu e conectat. Extinde java.net.SocketException şi implementează interfaţă java.io.Serializable. *ReplyTimeoutException* e aruncată atunci când timeout-ul expiră şi cineva aşteaptă pentru un răspuns. Extinde java.lang.Exception şi implementează java.io.Exception.

Interfetele nivelului overlay sunt: *ConnectivityChangeListener*, *DataListener*,*Link*, *NeiggborChangeListener*, *Overlay*.

Interfeţa *ConnectivityChangeListener* e implementată de *AbstractTopologyManager*, *SimpleTopologyManager*. Subinterfata este *TopologyManager*. Componentele care implementează această interfaţă ascultă după schimbări apărute în conexiunea nodurilor şi legăturilor, cum ar fi: adăugarea, ştergerea, căderea de legături.

Metodele defite de interfata sunt:

- notifyLinkClosed(Link link): această metodă e apelată atunci când o legătură e închisă.
- notifyLinkCrashed(Link link): această metodă e apelată atunci când o legătură cade.
- notifyLinkOpened(Link link): această metodă e apelată atunci când o legătură noua e adăugată pentru un vecin al nodului local.

DataListener e interfata a carei componente care inplementeaza această interfaţă ascultă după primirea de date noi prin ceva Transport. E implementată de *AbstractTopologyManager*, *GenericOverlay*, *SimpleTopologyManager*, *TCPDispatchingService*.

Metodele definite de interfata sunt:

- notifyDataArrived(Strîng subject, Link source, Serializable data): această metodă e apelată în momentul în care o data noua e primită de la un nod vecin sau de la nodul local.

Link defineşte o legătură între doi borkeri. Fiecare legătură e asociată cu un *Transport*. Ascunde protocolul de trimitere şi primire(de orice obiect seriablizabil) dintre brokeri vecini. Fiecare are asociat un subiect(un string) care poate fi trimis la nivelul superior pentru a se face diferanta între două tipuri de date.

Metodele definite de interfata sunt:

- close(): închide legătură. După ce o legătură e închisă, aceasta e eliminată din transport şi nu mai poate fi redeschisă.
- isConnected(): verifică dacă legătura e conectată. Când e creată prin metoda Transport.openLink legătură e conectată. Pentru a deconectă trebuie apelată metodă close(). Returnează true dacă legătura nu e închisă, şi false altfel.
- send(String subject, Seriazible data) : trimite o informaţie cu un subiect la vecinul conectat prin legătură.
- getTransport():returnează transportul asociat legăturii.

Interfata *NeighborhoodChangeListener* e implementată de *DeferredUnsubscriptionReconfigurator*, InformedLinkActivationReconfigurator, TCPDispatchingService. Componentele care implementează această interfaţă ascultă după evenimente legate de ştergerea unui vecin sau a nodului local existent.

Metodele definite de interfata sunt:

- notifyNeighborAdded(NodeDescriptor addedNeighbor, Serializable reconfInfo): e apelată metodă când un nou vecin se conectează(sau e conectat). Parametrii: addedNeighbor a descrierea nodului nou adăugat, reconfInfo a informaţie suplimentară despre scrhimbarile din topologie. Clasa Reconfigurator foloseşte această informaţie pentru a optimiza procesul de reconfigurare. Informaţia conţinută în parametrii depinde de combinaţia dintre managerul de topologie şi metodă de reconfigurare folosită. Acest parametru poate fi null dacă managerul de topologie nu suportă transmiterea de informaţie suplimentară.
- notifyNeighborRemoved(NodeDescriptor removedNeighbor): metoda e apelată atunci când un vecin sau nodul local se deconectează(sau e deconectat). Parametrul removedNeighbor e descrierea nodului vecin care e deconectat.
- notifyNeighborDead: aceasta metodă e apelată dacă se pierde conexiunea la un vecin sau nodul local

Overlay-ul e componenta unui broker care are responsabilitatea de management al reţelei de brokeri deasupra căruia în nivelul de rutare apare reconfigurarea. Are metode de adăugare/ ştergere noduri vecine şi de trimitere/ primire de pachete(orice obiect serializabil). Fiecare pachet are un subiect asociat (un string) care e folosit de nivelul superior pentru a face distincţie între pachete de acelaşi tip.(publish vs subscribe). Un overlay poate gestiona diferite cozi pentru mesajele recepţionate, fiecare asociată cu un anumit subiect. Fiecare broker are descriptorul sau unic *NodeDescriptor* care e creat şi gestionat de componenta overlay.

Metodele definite de interfata overlay sunt :

- start(): pornete overlay-ul. Un overlay trebuie pornit înainte ca să poată trimite sau primi pachete.
- stop():opreşte overlay-ul.

- isRunning(): verifică dacă overlay-ul rulează(a fost pornit şi nu a fost încă oprit).
- addNeighbor(String url): se conectează la nodul situat la URL specificat. Dacă operaţie are succes atunci acel nod devine nod vecin şi cei înregistraţi la NeighborAddedListeners sunt notificaţi de noul vecin adăugat, altfel o excepţie e aruncată. Metodă aruncă excepţia NotRunningException dacă managerul de topologie nu rulează, java.net.MalformedURLException dacă URL are un format greşit, *AlreadyNeighborException* dacă nodul e deja vecin, java.net.ConnectException dacă conexiunea cade.
- tentativelyAddNeighbor: incearcă să se conecteze la nodul situat la URL-ul specificat. Dacă totul merge bine nodul specificat devine un vecin nou şi nodurile înregistrare la NeighborAddedListeners sunt notificate cu privire la noul vecin. În cazul în care conectarea cu nodul specificat încalcă unele constrângeri printre cele impuse de prezenţa TopologyManager, aceasta metoda nu face nimic (nu se deschide noua conexiune şi nu comunică ascultătorilor apariţia nodului).
- removeNeighbor(NodeDescriptor node): se deconectează de la vecinul specificat. Dacă totul merge bine conexiunea este închisă şi nodurile înregistrate la NeighborhoodChangeListeners sunt anunţate de ştergerea nodului. În cazul în care nodul specificat nu a fost un vecin al nodului local, această metodă nu face nimic.
- addNeighborhoodChangeListener(NeighborhoodChangeListener listener): Înregistrează un ascultător pentru modificările apărute în vecinătate. Înregistrează un ascultător pentru evenimentele de la adăugarea / ştergerea / căderea unor vecini. În cazul în care ascultătorul (de exemplu, un ascultător care este egal cu acesta) a fost adăugat înainte, această operaţiune nu are niciun efect.
- removeNeighborhoodChangeListener(NeighborhoodChangeListener listener):şterge înregistrarea la acel ascultător.
- getNeighbors(): returnează colecţia curentă de vecini.
- getAllNeighborsExcept: returnează colecţia actuală de vecini cu excepţia celui specificat.
- isNeighborOf(NodeDescriptor neighbor):verifică dacă nodul specificat ca parametru e vecin cu nodul local
- getNumberOfNeighbors: returnează numărul curent de vecini ai nodului local.
- getNumberOfBrokers: returnează numărul de brokeri conectaţi direct cu nodul local.
- getNumberOfClients: returnează numărul de clienţi conectaţi la nodul local.
- addPacketListener(PacketListener listener, java.lâng.String subject): înregistrează un ascultător pentru sosirea de noi pachete adresate unui anumit subiect. În cazul în care ascultătorul (de exemplu, un ascultător care este egal cu acesta) a fost adăugat înainte, această operaţiune nu are niciun efect.

- removePacketListener(PacketListener listener, java.lâng.Strîng subject): Elimină ascultător dat. Elimină prima apariţie PacketListener (printre cele înregistrate cele înregistrate pentru subiectul dat), care este egală cu cea din parametru.
- send(java.lang.String subject,java.io.Serializable packet,NodeDescriptor recipient): trimite un pachet cu subiectul specificat la destinatarul dat (care trebuie să fie un vecin al nodului local).

PacketListener e componentele care implementează această interfaţă pot asculta după sosirea de noi pachete.
Metodele definete sunt:

- notifyPacketArrived (String subject, NodeDescriotir source, Serializable packet) această metodă e apelată de Overlay atunci când un nou pachet soseşte de la un nod vecin sau de la nodul local.

TopologyManager e componenţa responsabilă de managementul topologiei de brokeri. Extinde interfaţă *ConnectivityChangeListener*.

Obiectele care implementează această interfaţă *Transport* gestionează conexiunea dintre nodul local şi vecinii lui, având metode pentru a trimite şi primi orice obiect serializabil. Fiecare pachet are asociat un subiect, care poate fi folosit de nivelele superioare pentru a distinge diferite tipuri de pachete. Clasele care implementează această interfaţă diferă de tipul protocol pe care îl folosesc pentru a se conecta şi pentru a face schimb de informative.

Clasa *AbstractTopologyManager* e un manager de topologie abstract, care furnizeaza metode commune la toti managerii de topologie. Atunci cand un nou transport e adaugat, managerul de topologie abstract isi inregistreaza un listener pentru toate evenimentele(adaugare, stergere, cadere de legaturi) notificate de transport. In mod similar atunci cand transportul e sters, managerul de topologie se deinregistreaza.

GenericOverlay e clasa care implementeaza interfata Overlay prin delegarea majoritatea operaţiunilor sale la un TopologyManager şi la unul sau mai multe *Transporturi*. Rolul său este de a decupla transportul de nivelurile superioare prin punerea intr-o coada a pachetelor primite şi livrarea acestora către stratul superior asincron Implementeaza *Overlay* si *DataListener*.

LinkSubstitutionInfo e clasa care conţine informaţii despre o nouă legătură, care a înlocuit o alta legatura în timpul unei operaţiuni de reconfigurare. Într-o reţea, o legatura poate înlocui o alta legatura în cazul în care ambele sunt conectate la acelaşi sub−arbori de acoperire.

Atunci când se utilizează în *Link* Activation ca protocol de reconfigurare, managerul de topologie utilizează această clasă în evenimentele *NeighborDead* de a informa reconfigurator atunci când o legătură existente anterior între broker local şi broker mort este înlocuit cu o noua legatura. Legatura noua poate fi între doi brokeri diferiti de cele din legatura vechi; între un broker care a fost în legătura vechi şi un broker care nu a fost în legatura veche.

Metodele sunt:

- getNewLinkBrokerInOtherSubTree(): returnează brokerul care este capat de legatura al legaturii noi în sub-arbore care conţine brokerul local.
- getNewLinkBrokerInOtherSubTree():returnează brokerul care este capat al legaturii noi în sub-arbore care conţine vecinul vechi al broker local. vecinul vechi este brokerul, care a fost legat de brokerul local prin legatura veche.

Clasa *ReconfigurationID* reprezinta ID-ul atribuit de către managerul de topologie la o operaţie de reconfigurare. ReconfigurationId poate fi exploatat de reconfigurator prin execuţia de protocoale avansate de reconfigurare.

SimpleTopologyManager e un manager de topologie care nu restricţionează overlay-ul şi poate lucra pe orice topologie şi nu încearcă să păstreze overlay-ul conectat atunci când legăturile/ brokerii se deconectează. Ea doar interzice existenţa de mai mult decât o singură legătură spre acelaşi vecin. Extinde clasa AbstractTopologyManager.

TCPTransport e un transport *TCP* folosit pentru a se conecta la brokerii vecini. Implementează Runnable şi extinde AbstractTransport. Constructorul *TCPTransport* crează un nou obiect care poate fi folosit pentru a deschide conexiuni de transport spre alte transporturi dar nu primeşte transporturi. Constructorul *TCPTransport*(int port): Creează un *TCPTransport* nou capabil de a acceptă noi conexiuni. Parametrul port specifică portul acest transport acceptă conexiuni.

AlreadyNeighbotException e exceptia aruncata pentru a indica faptul că nodul dorit sa fie adăugat ca vecin este deja un vecin al nodului local. Acesta include *NodeDescriptor* pentru nodul adăugat. Extinde clasa java.net.ConnectException si implementeaza interfata java.io.Serializable.

NotConnectedException reprezintă o excepţie care trebuie să fie aruncata atunci când o metodă responsabila de trimiterea datelor la un broker vecin este invocat în acelaşi timp cu brokerul care nu este conectat.Extinde java.net.SocketException si implementeaza java.net.SocketException.

NotRunningException e exceptia aruncat pentru a indica faptul că obiectul ţintă nu rula. Extinde java.lang.Exception si implementeaza java.io.Serializable.

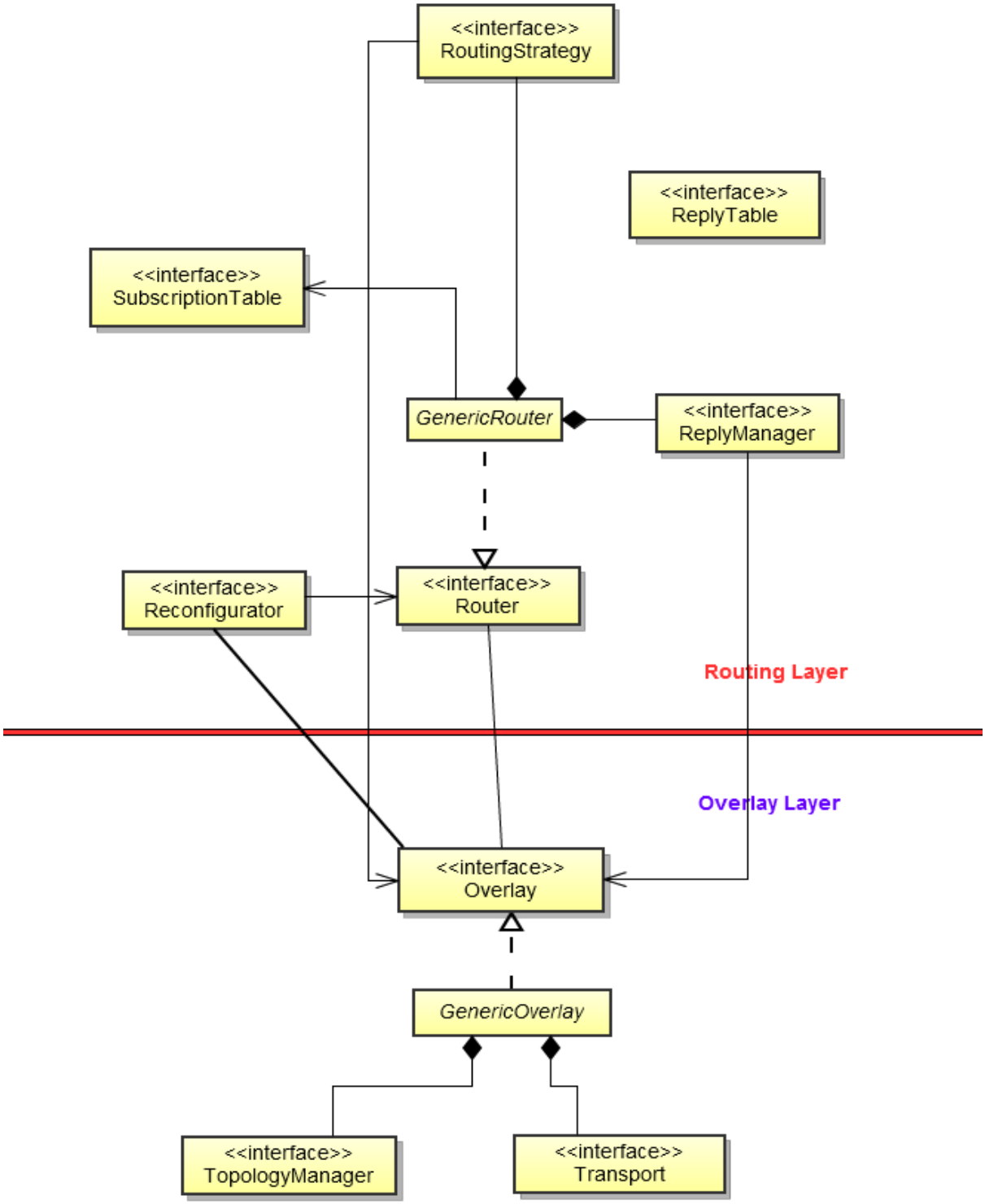


Figura 11. Interfețele celor două nivele: overlay și rutare.

Într-un broker *Reconfigurator* este modulul responsabil de gestionarea reţelei de expediere în prezenţa reconfigurări. În special, o transmite abonamente / unsubscriptions că este necesar. Este înregistrat la Overlay pentru a primi notificări despre vecinii de nodul local.

Metodele interfetei sunt:

- setRouter(Router r) : setează ruterul.
- Superinterfaces : NeighborhoodChangeListener.

Clase care implementează această interfaţă : *DefferedUnsubscriptionReconfigurator, InformedLinkActivationReconfigurator*.

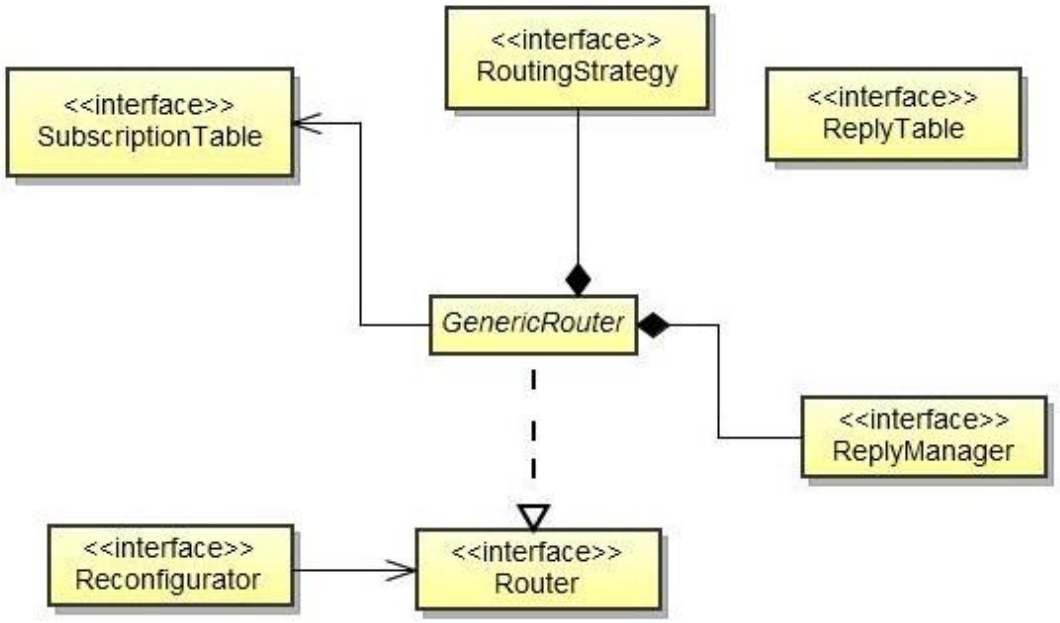


Figura11.1. Interfețele nivelului de rutare

Interfata *ReplyManager* este modulul care gestionează răspunsurile. Acestea interacţionează cu o structură de date, *ReplyTable,* în care sunt înregistrate toate datele care aparţin de răspunsuri, cum ar fi ID-ul expeditorului, timeout-ul de expirare şi numărul de vecini la care mesajul a fost dat. Clasa care implementează această interfaţă este ImmediateForwardReplyManager.

Metodele definite sunt:

- recordRepliableMessage(MessageID repliableMessageID, NodeDescriptor senderID, FutureInt numExpectedReplies): Adaugă o noua intrare în tabelul de răspuns locale de a gestiona un mesaj nou repliable.
- Dacă numExpectedReplies == 0, nici un răspuns va ajunge vreodată pentru acest mesaj. Acesta va transmite un răspuns fals trecut cu acelaşi repliableMessageID, ultima == adevărat şi sarcină utilă == null.

- forwardReply(Reply reply): trimite mesajul mai departe către nodul care a publicat evenimetul. Această metodă poate fi utilizată pentru a implementa un sistem fără nici o amânare, toate răspunsurile sunt trimise imediat şi niciodată stocate. În cazul în care NodeDescriptor nu este prezent, răspunsul este pierdut.

- setRouter(Router r): Acest metodă e setată în router în ReplyManager. ReplyManager are nevoie de router pentru a accesă ReplyTable. Utilizatorii nu trebuie să apeleze aceasta metoda deoarece routerul apelează această metodă în constructor.

ReplyTable este o structură de date pentru a gestiona mesajele de tip răspuns şi răspunsurile lor.

Metodele sunt:

- addEntry(MessageID repID, NodeDescriptor senderID, FutureInt numPendingReplies): Adaugă o noua intrare în tabelul de răspuns. Dacă unul din parametri nu sunt corecţi, (adică nuli), se aruncă o excepţie de tipul MalformedException.
- getSender(MessageID repliableMessage): returnează vecinul care a trimis mesajul.
- removeExpiredEntries(): Inspecteaza şi şterge toate elementele care au un timeout expirat.
- getNumberOfPendingReplies(MessageId repliableID): returnează numărul de răspunsuri care nu au ajuns încă. Dacă repliableId e null aruncă NullPointerException. Această metodă verifică dacă a fost setată valoarea, dacă nu a fost aşteaptă să fie setată. Dacă MessageID nu e prezent în tabel se aruncă excepţia NoEntryException.
- getExpiredTime(MessageID repliableMessageID): returnează momentul exact în care mesajul de răspuns expiră. Dacă MessageID nu este prezent în tabel atunci returnează ReplyTable.NO_ENTRY.
- decrementNumberOfPendingReplies(MessageId replyMessageID): Decrementează numărul de răspunsuri după care s-a aşteptat deja. Acest număr trebuie să fie pozitiv. Dacă, după decrementare devine 0, se elimină intrarea din tabel.Această metoda poate fi invocat numai atunci când valoarea a fost e setată, altfel se aruncă o IllegalStateException.
- removeCompletedEntries():Îndepărtaţi intrările care aşteaptă pentru nici un răspuns şi returnează Id nodului corespunzător, care aşteaptă un mesaj.
  Numărul de răspunsuri aşteptate pentru aceste noduri este zero.
- removeEntry(MessageID repliableMessageID): elimină intrarea din tabel.
- setTimeout(long timeout): setează cât de mult se aşteaptă pentru un răspuns.
- getTimeout(long timeout): returnează timeout-ul aşteptării pentru răspuns.

Interfaţa *Router* este "core-ul" unui broker. Clasele care implementează această interfaţă constituie principala componentă a unui broker. Este înregistrat la Overlay pentru a primi mesajele provenind de la alte noduri ale reţelei. În cele mai multe cazuri de delegaţi funcţiile sale principale pentru alte componente care compun un broker: şi anume, RoutingStrategy şi ReplyManager. Ea deţine structură de date comune gestionate de către acele componente: SubscriptionTable şi ReplyTable.

Metodele interfeţei ruter sunt:

- subscribe(NodeDescriptor neighborID, Filter filter): Subscrie vecinul specificat la mesajele care îndeplinesc condiţiile filtrului. Reconfiguratorii presupune că punerea în aplicare a acestei metode este sincronizat pe obiect router, astfel încât reconfiguratorul poate blocă atunci când are nevoie pentru a se asigură că nu se execută operaţii pe router concomitent cu acţiunile de reconfigurare.
- unsubscribe(NodeDescriptor neighborID, Filter filter): dezabonează vecinul specificat că parametru de la mesajele care se potrivesc filtrului specificat.

- unsubscribeAll(NodeDescriptor neighborID): şterge toate abonarile vecinului specificat.
- publish(NodeDescriptor neighborID, Message message): publicarea mesajul provinenit de la vecinul specificat. În funcţie de politică de rutare a adoptată, acest lucru necesită transmiterea mesajului dat la unul sau la oricare dintre vecinii brokerului.Dacă mesajul este de tipul Repliable, Ruterul are nevoie să activeze ReplyManager pentru a înregistra mesajul în ReplyTable.
- getSubscriptionTable() : returnează tabelul abonamentelor folosit de acest ruter.
- getReplyTable(): returnează tabelul răspunsurilor folosit de acest ruter.
- setOverlay(Overlay o): înregistrează overlay-ul.
- getOverlay(): returnează overlay-ul instanţei.
- forwardReply(Reply reply): transmite răspunsul la expeditorul sau. În conformitate cu informaţiile cuprinse în tabelul raspunsurolor răspunsul va fi trimis la un vecin sau nu se face nimic în cazul în care nu există informaţii cu privire la expeditor, care nu este cuprins în tabelul de răspunsuri.
- getID(): returnează identificatorul unic al brokerului.

Într-un broker modulul RoutingStrategy are funcţia de a gestiona rutarea mesajelor. Metodele sale sunt apelate de către Overlay(prin ruter) pentru a efectuă operaţiile de bază: procesează subscripţii, dezabonari şi trimite mai departe mesaje.

Metodele interfeţei RoutingStrategy sunt:

- publish(NodeDescriptor neighbor, Message message): publică mesajul specificat venit de la vecinul dat că parametru.
- setRouter(Router router): asociază RoutingStrategy cu ruterul specificat.
- subscribe(NodeDescriptor neighbor, Filter filter): subscrie vecinul specificat că parametru la mesajele care se potrivesc pentru filtrul filter.
- unsubscribe(NodeDescriptor neighbor): şterge toate abonarile vecinului specificat că parametru.

Interfaţa SubscriptionTable e folosită de broker pentru a stoca informaţii legate de abonamentele primite de la vecinii săi. Clasele care implementează această interfaţă pot fi diferite prin strategia de rutare adoptată .

Metodele interfeţei SubscriptionTable sunt:

- addSubscription(NodeDescriptor n, Filter f): adaugă subscripţia specificată în tabel.
- removeSubscription(NodeDescriptor n, Filter f): şterge subscripţia specificată prin parametru din tabel.

- removeAllSubscriptions(NodeDescriptor n): şterge toate subscripţiile în tabel.
- clear(): şterge toate abonarile.
- isSubscribed(Filter filter): returnează adevărat dacă cel puţin un vecin e subscris la filtrul specificat că parametru.
- isFilterInTable(Filter filter): returnează adevărat dacă un vecin e abonat la filtrul dat ca parametru.
- matches(Message message): returnează o colecţie de noduri vecine care au cel puţin un filtru care se potriveşte mesajului dat ca parametru.
- getSubscribedNeighbors(Filter f): returnează colecţia de vecini care s-au subscris la filtrul dat ca parametru.

Clasa *SubscriptionTable* e folosită de broker pentru a stoca informaţiile despre subscripţiile(filters) pe care le primeşte de la vecini. Clasa *GenericTable* nu face nici o

presupunere legat de subscripţiile primite de la vecini, despre formatul filtrelor. Implementează interfaţa SubscriptionTable prin păstrarea unei liste pentru fiecare vecin, lista filtrelor primite de la vecin prin subscripţii, prin folosirea metodei Message.match [10] determină lista vecinilor subscrişi la un mesaj dat.

Metodele implementate de clasa *GenericTable* sunt:

- addSubscription(NodeDescriptor n, Filter f): adaugă o subscripţie la lista de subscripţii.
- removeSubscription(NodeDescriptor n, Filter f): sterga din lista subscripţia data prin filtrul specificat.
- stergeToateSubscriptiile(NodeDescriptor n): şterge toate subscripţiile asociate vecinului specificat ca parametru.
- clear(): şterge întreagă lista.
- isSubscribed(NodeDescriptor n, Filter f): returnează adevărat dacă nodul vecin dat ca parametru e subscris filtrului specificat.
- getAllFilters(NodeDescriptor n): returnează coletia de filtre asociate vecinului specificat.
- isFilterInTable(Filter filter): returnează adevărat dacă cel puţin un vecin e subscris la filtrul specificat.
- matches(Message message): returnează o lista de noduri vecine care au cel puţin un filtru care e asociat mesajului dat ca parametru.
- getSubscribedNeighbors(Filter f): returnează o colecţie de vecini subscrişi filtrului specificat ca parametru.

DefferedUnsubscriptionReconfigurator e clasa care se ocupă de reconfigurare prin implementarea celor două versiuni ale protocoalelor dezabonari amânate : Dezabonari temporar amânate şi dezabonari notificat amânate. Variantă de protocol folosită e aleasă în constructor. Dacă se crează această clasa cu timeout specificat de 0 atunci se comportă că şi protocolul Strawman. Constructorul are un parametru numit deferTimeout, implicit este 3000 ms.

Metodele aceste clase sunt:

- setDeferTimeout(long timeout): setează timeout-ul necesar pentru dezabonarile temporar amante. Noua setare va afea efect doar la următoarea reconfigurare.

- setRouter(Router r): asociază ruterul specificat.
- notifyNeighborAdded(NodeDescriptor addedNeighbor, reconfInfo): metode e apelată atunci când un nou vecin e adăugat sau se conectează.
- notifyNeigborRemoved(NodeDescriptor removedNeighbor): metodă e apelată atunci când un vecin al nodului local se deconectează sau e deconectat.
- notifyNeighborDead(NodeDescriptor deadNeighbor): metodă e apelată atunci când un vecin al nodului local e compromis.
- notifyPacketArrived(Strîng subject, NodeDescriptor source, Serializable packet): această metodă e apelată de către Overlay atunci când un nou packet a sosit de la un vecin la nodului local.

Clasa ImmediateForwardReplyManager e implementarea interfeţei ReplyManager.

Metodele implementate sunt:

- recordRepliableMessage(MessageID repliableMessageID, NodeDescriptor senderID, int numExpectedReplies): adaugă o înregistrare în lista locală de răspunsuri şi gestionează răspunsul nou adăugat.

- forwardReply(Reply reply): trimite răspunsul dat ca parametru spre destinatarul corespunzător al mesajului răspuns.

Clasa *InformedLinkActivationReconfigurator* gestionează reconfigurarea folosind versiunea îmbunătăţită a protocolului Strawman.
Metodele clasei sunt:

- setSubscriptionTimeout(long timeout): setează timeout-ul folosit de abonarile amânate. Noua setare are efect decât după apariţia următoarei reconfigurări.
- setUnsubscriptionTimeout(long timeout): setează timeout-ul folosit de dezabonari amânate. Noua setare are efect decât după apariţia următoarei reconfigurări.
- notifyNeighborAdded(NodeDescriptor addNeighbor): metodă e apelată pentru a notifica adăugarea unui nou vecin.
- notifyNeighborRemoved(NodeDescriptor removedNeighbor): această metodă e apelată atunci când un vecin se deconeacteaza.
- notifyNeighborDead(NodeDescriptor deadNeighbor): această metodă e apelată atunci când un nod vecin e compromis.

Clasa *SubscriptionForwardRoutingStrategy* implementează interfaţă RoutingStrategy şi specifică modalitatea de rutare folosită de reţeaua de rutere.

Metodele acestei clase sunt:

- subscribe(NodeDescriptor neighbor, Filter filter): subscrie vecinul specificat ca parametru la mesajele care se potrivesc filtrului filter.
- unsubscribe(NodeDescriptor neighbor,Filter filter): dezabononeaza nodul vecin specificat de la mesajele care se potrivesc filtrului specificat. În general e opusul metodei subscribe.
- unsubscribeAll(NodeDescriptor neighbor): şterge toate subscripţiile de la nodul vecin specificat prin parametru.
- publish(NodeDescriptor sourceID, Message message): publică mesajul venit de la vecinul specificat ca parametru. În funcţie de strategia de rutare adoptată e nevoie ca mesajul să fie trimis la o parte din vecini sau la toţi vecinii nodului.

Acest nivel gestionează şi implementează toate aspectele legate de reţea. Nivelul de overlay de asemenea oferă o interfaţă uniformă nivelului de rutare. Acest nivel poate fi uşor înlocuit cu altă implementare specifică de exemplu reţelelor fără fir. Nivelul de overlay e construit deasupra nivelului TCP/IP al reţelei cât şi pe bază unei topologii de tip arbore.

5.4. Analiza funcționării protocoalelor de rutare peste nivelul de overlay

In aceasta sectiune vom face un pas mai departe pentru integrarea nivelul de rutare cu nivelul de overlay pentru a forma un sistem complet capabil sa ruleze in retele de mari dimensiuni, dinamice.

Protocolul de overlay este în măsură să menţină o topologie conectată şi acyclică, în prezenţa de noduri care se alătură şi se deconectează de la reţea arbitrar. În mod similar, protocoalele propuse pentru nivelul de rutare sunt capabile să-si adapteze informaţiile de rutare la modificările topologiei cu costuri minime de comunicare, realizand reducerea overhead-ului în ceea ce priveşte abordarea strawman disponibila în literatura de specialitate.

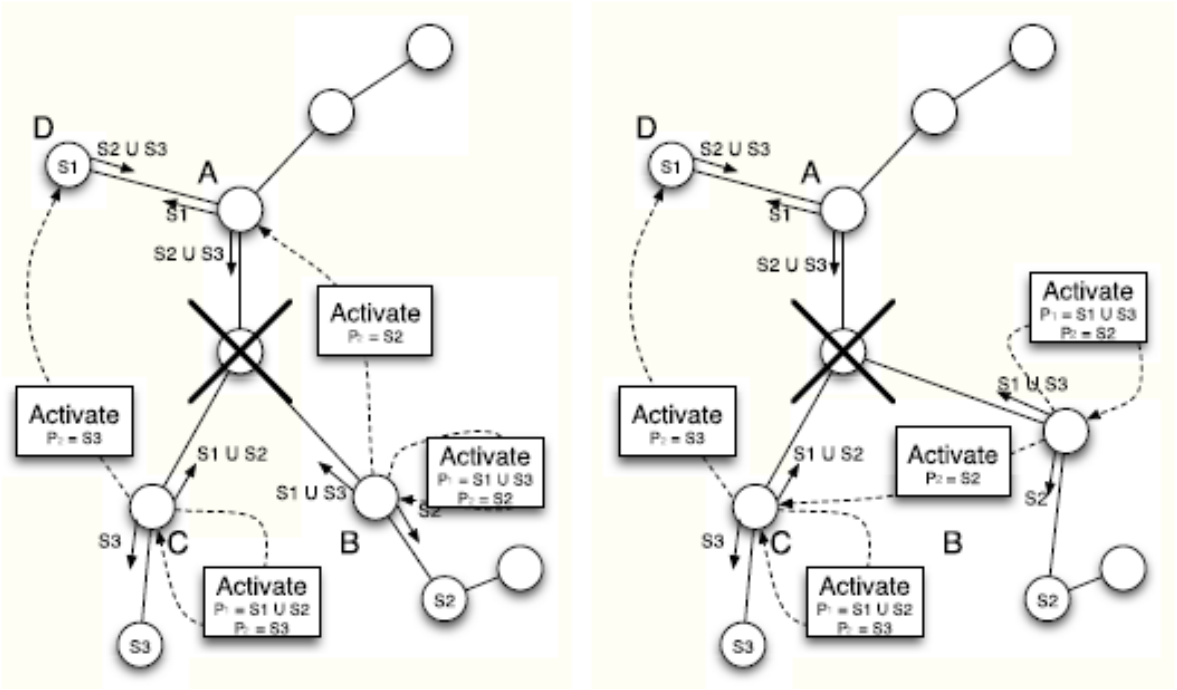
5.4.1. Integrare protocol "dezabonări temporar amante"

Protocolul "dezabonari temporar amanate" este care poate fi cel mai uşor de integrat cu nivelul de overlay. Interfaţa acestuia cu overlay-ul este, de fapt, identic cu interfaţa cu protocolul Strawman. Ambele au nevoie doar de overlaypentru a notifica fiecare dispecer de apariţia sau dispariţia de legaturi catre dispeceri vecine prin intermediul apelurilor addLinkTo,removeLinkTo.

LSTree furnizează aceste notificări. Un nod detecteaza dispariţia unui legătură a unui vecin, fie un părinte sau un copil, facand un apel la removeLinkTo. În mod similar, apelul addLinkTo se face prin protocolul de îndată ce o cerere de conexiune este acceptată.

5.4.2. Integrare protocol "activare informată a legăturii"

Să ne gândim la situaţia din figura 14.2 (a).Deconectare unui nod duce la dispariţia celor 3 legaturi la nodurile A, B, şi C, în timp ce overlayul re-stabileste o conexiune prin adăugarea a 2 link-uri noi: de la C la D, şi de la B la A. Pentru ca e un nod stânga doar 2 din cele 3 link-uri sunt înlocuite, şi celalalt este pur şi simplu pierdut. Astfel, pentru a integra protocolul, este suficient să selectam două dintre legaturile indepărtate şi sa asociem fiecare dintre ei la cele ale nodului nou.. În principiu, orice asociere de legaturi noi si vechi poate fi folosita. Cu toate acestea, utilizarea protocolului LSTree sugereaza in mod natural o astfel de asociere. Legatura de la nodul nu a reuşit să-parinte este cel care se pierde. Fiecare legatură la copiii săi este asociata la legatura nouă, găsită de către copilul corespunzator folosind protocolul LSTree.



(a) două mesaje de activare **(b) un mesaj de activare**

Figura 12 : Integrarea protoclului cu managerul de overlay .

Figura 12 arată că activarea a două mesaje utilizate de către protocolul poate fi înlocuită cu un singur mesaj care conţine toate informaţiile necesare.

Cu referire la figura 12 (a), legatura de la nodul cazut, care nu a reuşit la C se înlocuieşte cu C-D, în timp ce legătură spre B se înlocuieşte cu B-A. Odata ce legăturile noi au fost asociate cu cele eliminate, reconfigurarea poate fi fectuatăfolosind versiunea asimetrică a protocolului. Fiecare dintre copiii nodului interschimba lista P1 şi P2. Primul salt, de fapt, nu este necesar. Prin urmare, fiecare dintre copii nod nu a reuşit trimit un mesaj de activare, care conţine setul P2 la celalalt capat al legaturii noi. Apoi foloseşte P1 stabilit la nivel local, pentru a evita înmulţire unsubscriptions inutile la alte capete.

Cu referire la figura, lista P1permite nodului C evitarea propagarii de abonamente, S1 şi S2 la D, în timp ce lista corespunzătoare P2, permite nodului D prevenirea propagarii S3 la expeditorul C. Observăm că timeout-uri şi mesaje de culoare sunt gestionate astfel cum acestea ar fi în funcţionarea normală a protocolului, cu excepţia faptului că timeout-ul nu este în mod clar declanşat la nodul care a esuat. Raţionamentul prezentat permite în consecinta ca protocolul sa poată fi integrat cu LSTree fără modificări suplimentare.

Cu toate acestea, asocierea dintre legaturile noi şi vechi propuse de LSTree nu este întotdeauna perfectă. Figura 12 (b) arată o situaţie în care legătura la dispecerul B este înlocuita cu una nouă de la B la C. Abonament inutile de-a lungul legaturilor noi. Abonamente S2 nu sunt cuprinse în P2, şi de aceea sunt inutil. Această situaţie apare atunci când protocolul LSTree recuperează un eşec incepand de la un set frate.

6. Testare şi validare

Testarea Software reprezintă o investigație empirică realizată cu scopul de a oferi părților interesate informația vizavi de calitatea produsului sau serviciului supus testării, luând în considerație contextul operațional în care acesta din urma va fi folosit. Testarea pune la dispoziție o viziune obiectivă și independentă asupra produsului în dezvoltare, oferind astfel posibilitatea de a înțelege și evalua riscurile asociate cu implementarea produsului soft.

Tehnicile de testare includ, dar nu sunt limitate la, procesul de execuție a programului sau aplicației în scopul identificării defectelor/erorilor de software. Testarea mai poate fi definită ca un proces de validare și verificare a faptului că un program/aplicație/produs software corespunde business cerințelor și cerințelor tehnice care au ghidat proiectarea și implementarea lui; și rulează și se comportă corespunzător așteptărilor. Nu există un astfel de proces de testare ce ar permite identificarea tuturor defectelor posibile ce le poate conține un produs software.

Scopul primordial pentru procesul de testare este identificarea erorilor de software, izolarea și fixarea/corectarea defectelor care au cauzat aceste erori. Deseori este un exercițiu non-trivial, deoarece testarea nu poate demonstra cu certitudine de 100% procente ca produsul funționează corect în orice condiții; testarea doar poate demonstra că produsul nu funcționează corect în anumite condiții. În scopul testării deseori se include atât examinarea statică a codului sursă, cât și examinarea codului în execuție în diferite împrejurări și sub diferite condiții.

In vederea testării middleware-ului reconfigurabil bazat pe paradigma publish-subscribe s-au creat clase de test: *ClientPerturbator*, *ClientTester*, *ReplyCapableClientReplier* si *ReplyCapableClientSender*. Acestea au rolul de a testa functionalitatea si pe parcursul procesului de dezvoltare a software-ului au ajutat la identificarea erorilor si au condus la corectarea acestora.

Prin clasa *ClientPerturbator* e folosita pentru a produce caderi si a perturba clientul. Intr-un numar de iteratii se publica mesaje cu o anumita probabilitate si cu un numar de abonari si dezabonari cuprins intre o valoare minima si maxima de subscriptii.

Cu ajutorul clasei clasei *ClientTester* se testeaza daca mesajele publicate ajung la destinatie si daca subscriptiile sunt inregistrate corect. De asemenea se masoara timpul necesar trasmiterii.

Clasa *ReplyCapableClientReplier* testeaza daca un raspuns trimis de client ajunge sau nu inapoi la sursa.

Clasa *ReplyCapableClientSender* porneste un client care publică mesaje şi aşteaptă răspunsuri. Se foloseşte atât UDP cât şi TCP. Apelul clasei este urmatorul:

Pentru testarea si validarea aplicatiei s-a implementat o aplicatie care foloseste functionalotatea middleware-ului. Pe durata implementarii aceasta a avut rolul de a testa functionalitatea protocoalelor cat si a nivelului de overlay. Aplicatia e formata din Client GUI (Figura) si Broker GUI (Figura).

Brokerul are rolul de a crea reteaua overlay de dispeceri format dintr-un set de brokeri conectati. Interfata permite urmatoarea interactiune:

- Listen port: permite configurarea portului pe care clientul asculta dupa mesaje.
- Connect to: brokerul la care se doreste conexiunea si portul.
- Log-ul prezinta evenimentele care au loc pe broker.

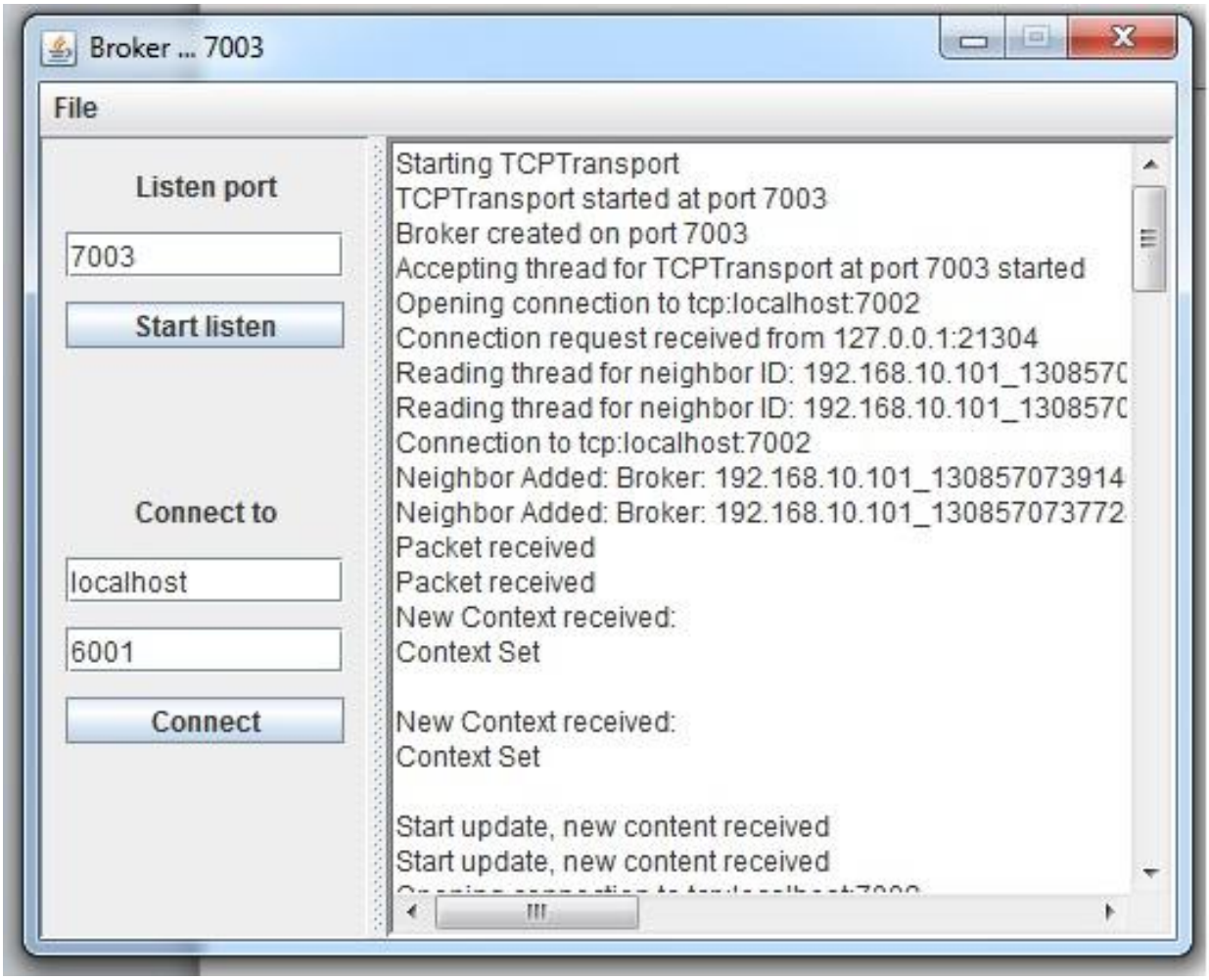


Figura 13. Broker GUI.

Client GUI reprezinta clientul care acceseaza serviciile puse la dispoziţie prin interfaţa DispatchingService, folosing Client Api al frameworkului prezentat. Aceasta oferă două implementări care diferă prin protocolul de transport folosit pentru a se conecta la brokeri(UDP şi UTP) şi de asemenea interfaţa ascunde toate detaliile legate de cum un clienţii sunt conectaţi şi comunică prin reţea.

Frameworkul suportă de asemenea posibilitatea ca clienţii să poată trimite mesaje de răspuns la mesaje, oferind în acest fel comunicare bidirecţională. Brokerii ţin evidenţa de tranzitul de mesaje etichetate şi ţin rutele de întoarcere a mesajelor răspuns prin metoda reply a *DispatchingService*. Suportând răspunsuri nativ, brokerii sunt capabili să urmărească numărul de răspunsuri aşteptate pentru fiecare mesaj şi pot verifică dacă toate dintre ele au fost primite, trăsătură care nu se poate obţine la nivelul aplicaţiei.

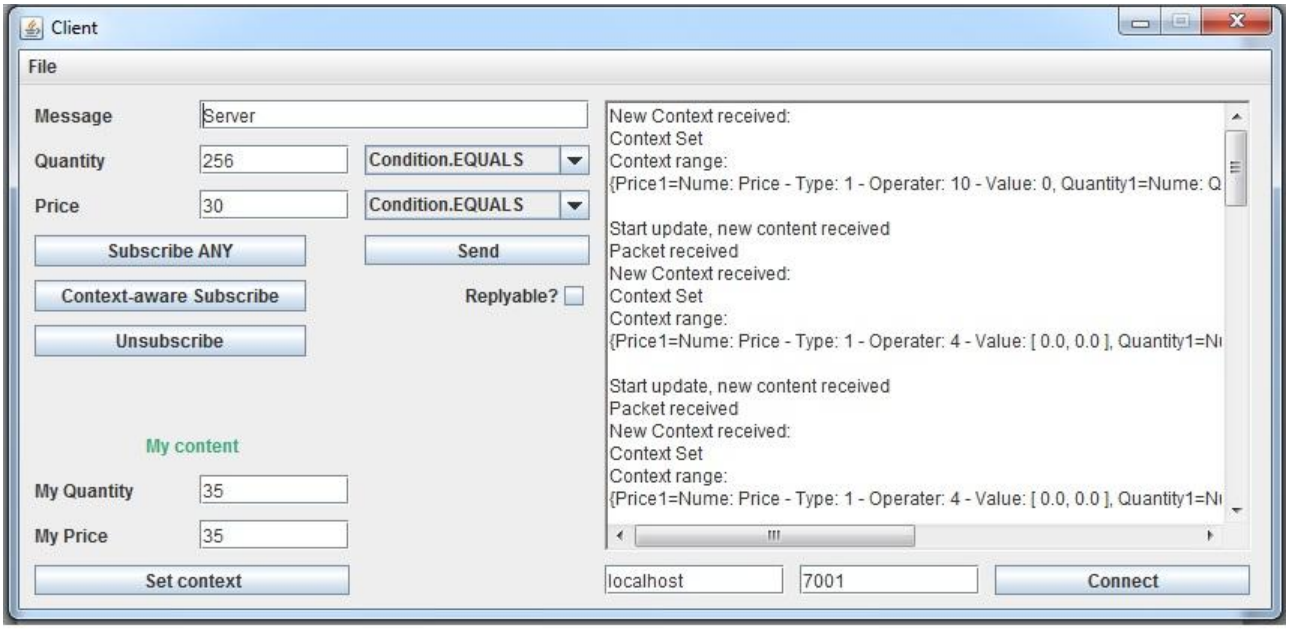


Figura 14. Client GUI.

Clientul are urmatoarea functionalitate:

- Message: se defineste context- message-ul
- Quantity: cantitatea
- Price: pretul
- Conditia : conditia dupa care se face matchingul.
- Subscribe Any: subscriere la toate mesajele care contin Message-ul.
- Context-aware subscribe: subscriere la mesaje care indeplinesc conditiile.
- Unsubscribe : dezabonare de la toate mesajele.
- Send : publica mesaj.
- Connect : conexiunea clientului la unul din brokeri.
- Set context: seteaza contextul actual al clientului.

Scopul primordial pentru procesul de testare e a fost identificarea erorilor izolarea și fixarea/corectarea defectelor. Testarea functionalității pe parcursul procesului de dezvoltare a software-ului a ajutat la identificarea erorilor si a condus la corectarea acestora.

7. Manual de instalare și utilizare

Secțiunea aceasta prezintă setările necesare pentru a putea utiliza și verifica sistemul RePSMi. Pentru aceasta trebuie instalate anumite unelte si trebuie urmariți câțiva pași necesari pentru a utiliza frameworkul.

Inainte de a putea rula aplicația, trebuie instalat produsul JDK, cel putin versiunea 6. Acest produs trebuie instalat deoarece frameworkul e scris in limbajul Java. Mașina virtuală Java este mediul în care se execută programele Java. În prezent, există mai mulți furnizori de JVM, printre care Sun, IBM, Bea, Oracle, FSF. JVM reprezinta un nivel care are rolul de translator intre programul compilat (bytecode) si codul in limbaj masina. Orice alt compilator creeaza fisiere specifice limbajului - masina al platformei respective. Astfel, executabilul produs va fi limitat la arhitectura masinii pe care a fost compilat. JVM interpreteaza codul compilat si il transforma in limbaj masina. Diferenta dintre cele doua abordari este considerabila in ceea ce priveste portabilitatea. Executabilele non-Java comunica direct cu setul de instructiuni al platformei respective, pe cand "executabilele" Java (codul compilat .class) comunica cu setul de instructiuni al masinii virtuale, care este apoi transformat in instructiuni specifice platformei.

Structura fisierelor e organizata pe pachete in felul urmator:

- broker.overlay: contine nivelul de overlay al RePSMi
- broker.routing: contine nivelul de rutare al RePSMi
- client: contine client API al RePSMi
- test: contine clasele de test ale RePSMi
- util: contine clasele ajutatoare ale RePSMi
- context: contine un exemplu de context
- context.gui: contine aplicatia de demo si de test pentru RePSMi
- context.routing

Framework-ul se poate utiliza fie prin importarea surselor intr-un proiect Java, fie prin folosirea ca referință a librăriilor jar din directorul /dist.

Framework-ul are ca cerinte de sistem asemenea celor rularii JDK, un exemplu pe care aplicatia rulează fără probleme este: Windows 7, procesor 1 Ghz, 1 Gb Ram, HD 20 Gb.

Aplicatia de test si demo poate fi rulata prin executia jar-urilor: BrokerGui.jar si ClientGui.jar. Scenariul de testare poate fi rulat prin MainClass.jar.

Scenariile de utilizare ar putea fi: jocurilor pe calculator, software utilizat în spitale pentru monitorizarea schimbării stării de sănătate, news sites, stock markets.

Industria jocurilor pe calculator a devenit rapid o forță economică majoră care a depășit industria filmului în ceea ce privește veniturile. Numeroase jocuri online în care mai mulți jucători să se poată juca în paralel, cum ar fi World of Warcraft, pot avea mai multe milioane de jucători. Jucătorii reacționează la evenimentele apărute în timpul jocului, cum ar fi la acțiunile celorlalți jucători. Evenimentele apărute în timpul jocului pot fi simple mișcări ale avatarelor, precum a ataca sau a fi atacat sau rezolvarea unor sarcini și achiziționarea unor arme sau abilități. Evenimentele din timpul jocului trebuie notificate acelor jucători care sunt

influențați de ele. Notificările trebuie să aibă loc prompt și la toate părțile implicate în paralel și în timpul cel mai scurt posibil din motive de corectitudine.

Domeniul sănătății are multe aplicații software de tipul sistemelor bazate pe evenimente, un exemplu constând în monitorizarea personală a stării de sănătate. Aplicația pentru monitorizarea stării de sănătate este un sistem personalizat care permite persoanei și îngrijitorului ei să monitorizeze starea de sănătate a persoanei. Această aplicație poate fi în mod deosebit utilă persoanelor bolnave cronic precum și cetățenilor în vârstă.

Datele pot fi colectate cu ajutorul senzorilor mobili situați pe persoană precum și cu ajutorul senzorilor ficși amplasați în căminul acestora. Sisteme de alarmă sunt montate pentru a anunța persoana, iar, în cazul în care este necesar, pentru a anunța și îngrijitorul. Senzorii colectează automat datele ce țin de starea de sănătate cum ar fi: ritmul cardiac și tensiunea, localizarea persoanei cu referire la un spațiu, sau administrarea medicamentelor. Acest sistem are ceva în plus, precum ar fi măsurători specifice regulate pentru anumite condiții de sănătate care sunt efectuate de persoană, iar rezultatele acestor măsurători sunt filtrate pentru cazuri de situații de urgență și păstrate pentru a putea fi observate pe termen lung. Datele trebuie să fie protejate de manipularea frauduloasă, de acces extern necalificat, precum și păstrate în siguranță și depozitate pe termen lung.

Aplicația de monitorizare a stării de sănătate prezintă multe provocări derivate din volumul mare de evenimente cu probabilitate mică de apariție și necesitatea de a obține evenimente cu probabilitate mai mare de apariție care să fie răspândite.

Protecția necesită o preocupare majoră, mai ales că monitorizarea datelor poate conduce la creșterea primelor asigurărilor de sănătate sau la încercarea de a limita servicii. Deși percepută public ca o amenințare majoră, confidențialitatea nu este raportată ca fiind o preocupare majoră pentru persoanele în vârstă și bolnavii cronici care percep faptul că beneficiile depășesc riscurile și teama de a rămâne fără ajutor după suferirea unui eveniment incapacitant care nu este observat.

8. Concluzii și direcții de dezvoltare

În această seciune sunt prezentate concluziile legate de lucrarea prezentată cât şi direcţiile de dezvoltare pentru middleaware-ul reconfigurabil publish subscribe descris în această lucrare.

8.1. Concluzii

Modelul de comunicaţii publish-subscribe se bucură de o popularitate în creştere, atât în cercetare cât şi în industrie. În acest model, aplicaţiile client interacţionează publicarea de evenimente şi prin abonarea la evenimentele de care sunt interesaţi. Sistemele publish-subscribe bazate pe conţinut oferă un nivel ridicat de flexibilitate, permiţând clienţilor să specifice filtre specializate folosind facilităţi lingvistice pentru a asocia un filtru cu conţinutul mesajelor.

Mai multe sisteme publish-subscribe există, dezvoltate atât de industria software cât şi de mediile universitare. Ele se concentrează pe scalabilitate şi sunt construite având în vedere reţele stabile unde nodurile nu au probabilitate de deconexiune mare. Prin urmare, aceste sisteme nu furnizează nici un mecanism pentru a rearanja reţeaua de brokeri ca răspuns la schimbările apărute în topologia reţelei . Principalul dezavantaj al sistemelor similare este lipsă reconfigurării care trebuie să apăra în medii dinamice.Sistemele similare studiate sunt open-source şi au fost un bun model pentru implementarea sistemului publish-subscriber RePSMi, care are o arhirtectura modulară pe nivele şi tratează reconfigurarea.

În această lucrare, am abordat această problemă şi am prezentat o abordare completă de reconfigurare pentru middleware-ul publish-subscribe şi am propus un set de optimizări ale protocolului Strawman.

Arhitectură aplicaţiei e formată din două nivele: overlay şi rutare. Scopul nivelui de overlay e de a implementa mecanismul de management al nodurilor, al legăturilor fizice dintre brokeri. Conţine protocolul care menţine reţeaua de overlay conectată atunci când topologia reţelei se schimbă, datorită căderii de host-uri. Nivelul de rutare oferă următoarea funcţionalitate sistemului: menţine abonamentele clienţilor şi foloseşte Overlay-ul pentru a rută mesaje de la publishers spre subscriberi.

Middleware content-based publish-subscribe (RePSMi- Reconfigurable Publish Subscribe) proiectat este proiectat pentru a face faţă reconfigurarilor dinamice, cum ar fi cele din reţele peer-to-peer. RePSMi e un framework de clase Java unde componente interacţionează prin publicarea de notificări de evenimente care au avut loc şi prin abonarea laacele evenimente de care este interesat. Sistemul construieşte o reţea de brokeri distribuiţi conectaţi într-o reţea overlay dispatching, care rutează mesajele de la publisheri la subscriberi folosind o versiune optimizată a protocolului Strawman şi prin folosirea de strategii de rutare. Diferenţă faţă de sistemele similare existenţe e trăsătură inovativa de reconfigurare care e necesară într-o reţea peer to peer[14], unde legăturile se pot rupe în orice moment.

O altă trăsătură a sistemului RePSMi e arhitectura modulară care permite definirea formatului mesajelor şi filtrelor, mecanism folosit intern de fiecare broker pentru a potrivi şi trimite mesaje cât şi alegerea stategiei de rutare.

În final, putem spune că proiectul a urmărit proiectarea, optimizarea şi exemplificarea unui middleware publish-subscribe reconfigurabil capabil să ruleze într-un mediu dinamic unde posibilitatea apariţiei căderii arbitrare a nodurilor este probabilă.

8.2. Direcții de dezvoltare

O directie de dezvoltare este implementarea protocoalelor de rutare pentru retele mobile ad hoc Manet. Pe viitor frameworkul va fi extins si cu suport pentru aceste retele mobile MANET deoarece este cat de cat simpla, fiind nevoie doar de implementarea algoritmilor de rutare specifici.

Integrarea sistemului intr-un framework de testare Omnet++, program folosit pentru construirea simulatoarelor de rețele. Se pot testa rețele cu fir cat si wireless, de asemenea se pot testa protocoale cât si performanța acestora comparată cu alte protocoale.

Inginerii de reţea utilizează simulatoare pentru depanarea sistemelor şi testarea fiabilităţii şi a calităţii serviciilor oferite de hardware-ul de reţea şi de protocoalele de telecomunicaţii. Astfel, simularea devine un pas important în dezvoltarea reţelelor de comunicaţii peer to peer. In scopul obţinerii unor rezultate relevante în urma simulărilor, este important ca modelul pe baza căruia este realizat simulatorul, să fie conform realităţii. Simulatoarele existente sunt OPNET Modeler, NS-2, GloMoSim, Omnet++.

Cu ajutorul simulatoarelor se va putea analiza imbunătățirea performanței protocoalelor optimizate, cât si prezentarea de grafice care sa măsoare optimizarea făcută in urma observatiilor facute.

Cu ajutorul framework-ului RePSMi se pot implementa aplicatii care ruleaza in retele peer-to-peer si folosesc paradigma publish-subscribe. Un exemplu de astfel de aplicatie este Stock Exchange implementata de Rebeca, care in timp real prezinta schimbarile aparute la bursa.

De asemenea paradigma publish-subscribe se poate folosi pentru construirea de aplicatii gen Google Alerts. Google Alerts ne ofera posibilitatea sa nu stam zilnic sa cautam informatii despre un anumit domeniu ci sa le primim pe email. Adaugam unul sau mai multe cuvinte cheie de care suntem interesati si le primim in email. Presupunem ca lucram la un site de inchiriat masini si pentru care trebuie sa adunam cat mai multe legaturi ce vin spre site. Vom adauga in Google Alerts cuvantul cheie si vom filtra ce rezultate sa primim, astfel zilnic vom primi notificari cu site-urile ce au fost indexate si contin cuvantul cheie pentru care am optat.

Bibliografie

[1] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. ACM Trans. on Computer Systems,
19(3):332–383, Aug. 2006.

[2] G. Muhl, L. Fiege, and P. Pietzuch. Distributed Event-Based Systems.
Springer, 2006.

[3] Sun Microsystems, Inc. Java Message Service Specification, Version 1.1, April 2002.

[4] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A Peer-to-Peer approach to Content-Based publish/subscribe. In In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03), June 2003.

[5] Jose Antollini, Mario Antollini, Pablo Guerrero, and Mariano Cilia. Extending rebeca to support Concept-Based addressing. In In Proceedings of the Argentinean Symposium on Information Systems (ASIS'04), Cordoba, Argentina, September 2004.

[6] E. Yoneki and J. Bacon. Content-based routing with on-demand multicast. In Proc. of the 24th Int. Conf. on Distributed Computing Systems Workshops(ICDCSW04), 2004.

[7] F. Fabret et al. Filtering algorithms and implementation for very fast publish/subscribe systems. ACM SIGMOD Record, 30(2):115–126, 2005.

[8] Baehni, S., Chhabra, C., Guerraoui, R.: Mobility friendly publish/subscribe. In: EPFL Technical Report 200488 (2006)

[9] Altherr, M., Erzberg, M., Maffeis, S.: iBus - a software bus middleware for the java platform. In: Proceedings of the International Workshop on Reliable Middleware Systems, pp. 49–65 (October 1999)

[10] Baldoni, R., Beraldi, R., Piergiovanni, S.T., Virgillito, A.: On the modelling of publish/subscribe communication systems. Concurrency and Computation: Practice and Experience 17(12), 1471–1495 (2005)

[11] Carzaniga, A.,Wolf, A.L.: Forwarding in a content-based network. In: Proceedings of ACM SIGCOMM 2003, pp. 163–174 (2003)

[12] Fiege, L., Gartner, F.C., Kasten, O., Zeidler, A.: Supporting mobility in content-based publish/subscribe middleware. In: ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), pp. 103–122 (2003)

[13] Ioan Salomie, Tudor Cioara, Ionut Anghel, Tudor Salomie - Distributed Computing and Systems, Albastra Publishing House 2008, ISBN 978-973-650-234-7

[14] Pesonen, L., Bacon, J.: Secure Event Types in Content-based, Multi-Domain Publish/Subscribe Systems. In: Proceedings of SEM 2005. ACM (2005)

[15] A. Tanenbaum, *Reţele de calculatoare* (ediţia a patra), Byblos, Tg.Mureş, 2003.

[16] M. Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley, 2002, ISBN-0321127420

Anexe

Lista figurilor

Glosar acronime

API - acronimul din limba engleză pentru Application Programming Interface - interfața pentru programare de aplicații. De obicei este vorba despre interfața dintre programele de aplicație și sistemul de operare, care stabilește în amănunt modul în care programele de aplicație pot accesa (apela) serviciile sistemului de operare sub care rulează.

CORBA - (Common Object Request Broker Architecture) – ansamblu de specificatii de bază pentru ORB (Object Request Broker) privind tehnologia client/server, implementate deja pe unele platforme, de exemplu: IBM, Digital

GUI - reprezintă Graphical User Interface (UI)

IP (**I**nternet **P**rotocol) - protocol Internet care permite interconectarea unui număr practic nelimitat de retele de calculatoare si functionarea lor ca una singură. Diferite retele care utilizează IP sunt conectate între ele prin calculatoare specializate numite routere, iar cele care nu sunt bazate pe IP (de exemplu, BITnet sau DECnet) sunt conectate prin calculatoare numite gateway, care realizează
transferul diferitelor servicii.

LAN (**L**ocal **A**rea **N**etwork) - 1. Grup de computere si alte dispositive dispersate pe o suprafată relativ limitată si conectate printr-o linie de comunicatie care permite oricărui dispozitiv să interactioneze cu altul din această retea.

TCP/IP (**T**ransport **C**ontrol **P**rotocol / **I**nterface **P**rogram) - protocol de software dezvoltat de departamentul Apărării al SUA pentru comunicatiile între computere.

UDP (User Datagram Protocol) - protocol de comuni-catie din categoria celor care nu necesită stabilirea unei conexiuni cu destinatarul înainte de a începe transmisia datelor.

URL (Uniform Resource Locator) - procedură pentru accesarea server-elor si trimiterea documentelor în Internet. URL lucrează independent fată de diferitele protocoale folosite în Internet.

VM (Virtual Machine) - masină virtuală.