Laborator 3. Consistența datelor şi toleranță la defecte

În cadrul acestui laborator sunt prezentate aspecte practice legate de două considerente importante în dezvoltarea aplicațiilor ce comunică prin intermediul rețelelor de calculatoare: consistența datelor şi toleranța la defecte. Problema consistenței datelor este legată de asigurarea corectitudinii operațiilor de transfer de date, de sincronizarea între procese distribuite, de implementarea soluțiilor de replicare în scopul eficientizării aplicațiilor distribuite. Toate soluțiile de optimizare a aplicațiilor distribuite folosind tehnici de replicare nu au sens fără un suport corespunzător de asigurare a consistenței datelor. Consistența datelor are însă un domeniu mult mai larg de aplicabilitate decât ca suport pentru replicare. Astfel, consistența datelor se aplică de exemplu atunci când dorim să asigurăm o ordonare logică a evenimentelor produse de procesele distribuite ale unei aplicații de exemplu. Toleranța la defecte reprezintă un alt subiect deosebit de important pentru dezvoltatorii de aplicații distribuite. Toleranța la defecte se ocupă de identificarea problemelor ce apar pe parcursul funcționării aplicației şi de implementarea de soluții adecvate pentru rezolvarea acestora. Defectele pot fi de diverse tipuri, hardware sau software, intenționate sau neintenționate, permanente sau tranziente, etc. Sistemele distribuite sunt prin natura lor complexe şi din acest motiv sunt mai predispuse la apariția unor defecte decât alte tipuri de aplicații. Totuşi în cazul sistemelor distribuite există posibilitatea limitării efectelor produse de apariția defectelor. De exemplu pot fi folosite două instanțe ale unui acelaşi proces rulând pe două stații diferite astfel încât dacă unul se defectează celălalt să îi poată prelua sarcinile. Astfel de soluții de asigurare a toleranței la defecte sunt prezentate în cadrul unor exemple de aplicații practice în cadrul acestui laborator.

4.1. Implementarea unui cache în Java

Vom prezenta în cele ce urmează un exemplu de soluție de replicare din categoria celor inițiate de clienți. Mai exact vom prezenta modul de implementare a unui cache web similar celor folosite de diverse aplicații tip browser, cache ce are meritul de a oferi o soluție de optimizare a timpului de acces. Implementarea aplicației prezentate poată fi consultată în listingul *example1*.

EXEMPLUL 1. Pentru exemplificare vom trata în cele ce urmează un exemplu de cache web.

Implementarea cache ține cont de câteva aspecte. Ca în orice soluție de cache, datele reținute local sunt păstrate un timp maxim limitat. Astfel, dacă o pagină devine prea veche, conținutul ei va fi automat reluat pornind de la pagina obținută de la adresa originală. Pentru optimizarea timpului de răspuns vom păstra datele cache-ului şi în memorie (timpul de acces la memorie este mai mic decât timpul de acces la disc). Însă, ca o soluție de toleranță la defecte, toate datele din cache le vom păstra şi pe discul local folosind o tehnică de checkpointing. Prin scrierea datelor şi pe disc avem avantajul regăsirii ultimelor valori din cache la o eventuală repornire a aplicației. Mai impunem o condiție suplimentară de funcționare, şi anume limitarea spațiului de stocare a datelor în cache. Aceasta înseamnă că şi în eventualitatea în care toate intrările ar fi corecte (nu a expirat timpul de reîmprospătare) se poate ajunge în situația în care să devină necesară ştergerea unor date din cache pentru a face loc altora noi. Problema aceasta este întâlnită în modul de funcționare al multor clienți web reali, dar şi atunci când dezvoltăm aplicații ce dispun de spațiu limitat de stocare. Dintre diversele soluții existente de alegere a datelor ce

sunt şterse pentru a face loc altora noi am ales algoritmul LRU (least recently used). Conform acestui algoritm vom şterge întotdeauna acele pagini care au fost accesate cel mai în trecut.

O intrare în tabela de pagini cache reprezintă o instanță serializată a clasei *WebCacheEntry*, definită astfel:

```
public class WebCacheEntry implements Serializable {

        // url-ul de unde a fost citita intrarea
        private final String url;
        // continutul paginii de web citita de la adresa data de url
        private String content;
        // momentul cand a fost facuta ultima actualizare a continutului
        private long writeDate = -1;
        // momentul cand a fost facuta ultima accesare a continutuli
        private long readDate = -1;
```

Datele corepunzătoare unei intrări în cache sunt:

- *url* – adresa corespunzătoare paginii din cache;
- *content* – conținutul paginii citite de la respectiva adresă;
- *writeDate* – momentul când a fost făcută ultima actualizare a conținutului paginii în cache;
- *readDate* – momentul când a fost făcută ultima accesare a conținului paginii din cache.

Variabila *writeDate* este necesară pentru informarea asupra valabilității conținutului paginii web din cache. Dacă valoarea aceasta este devine prea veche în raport cu momentul de timp curent atunci în mod automat se va realiza conținutul acesteia pe baza paginii originale.

Variabila *readDate* este necesară pentru implementarea soluției LRU de ştergere a unor intrări din cache ce nu au mai fost accesate de mai mult timp pentru a face loc unor noi intrări, realizându-se astfel limitarea spațiului de stocare la dimensiunea cerută.

Ambele variabile, *writeDate* şi *readDate*, sunt de tip *long*, fiind specificate în milisecunde, standard de timp în Java şi nu numai (a se vedea şi funcția *System.currentTimeMillis()*). Valoarea corespunzătoare paginii web, *content*, este de tip *String*, considerată a fi în formatul universal afişabil *html*.

Din implementarea clasei două funcții sunt importante, cea de accesare şi cea de actualizare a unei intrări cache:

```
public void setContent(String content) {
   this.content = content;
   // automat setam si data ultimei accesari si pe cea a ultimei actualizari
   writeDate = readDate = System.currentTimeMillis();
}

public String getContent() {
   // se actualizeaza si data ultimei accesari
   readDate = System.currentTimeMillis();
   return content;
}
```

În cadrul funcției *setContent* sunt setate valorile variabilelor *readDate* şi *writeDate*. În primul rând prin apelarea metodei specificăm o operație de actualizare, deci setăm şi momentul de timp corespunzător ultimei actualizări. Dar dacă deja avem conținutul înseamnă că deja am şi accesat conținutul (chiar dacă nu din cache) şi setăm corespunzător şi momentul de timp corespunzător ultimei accesări a conținutului. În cadrul funcției *getContent* nu este actualizat decât momentul de timp corespunzător ultimei operații de accesare a conținutului. Atenție: în cadrul acestui exemplu academic am folosit funcția *currentTimeMillis*, ce întoarce valoarea curentă de timp a ceasului sistemului. În general în aplicațiile distribuite nu este indicată folosirea momentului de timp local deoarece sistemele pot avea ceasuri locale cu un defazaj mare. În

general este mai indicată folosirea unor protocoale de sincronizare a timpului precum *ntp* (network time protocol) – consultați listingul *example3* pentru un exemplu – sau a ceasurilor logice pentru o sincronizare corectă a timpului sau evenimentelor produse între procesele distribuite.

Toate operațiile de sincronizare a cache-ului cu discul local sunt implementate de către clasa *WebCacheFile*. Organizarea cache-ului pe disc este următoarea. Toate intrările cache sunt stocate în cadrul unui director. Pentru optimizarea accesului fiecare intrare cache este stocată într-un fişier separat având numele URL-ului corespunzător. Datele din fişiere reprezintă instanțe serializate de obiecte *WebCacheEntry*.

În aplicație constructorul clasei *WebCacheFile* primeşte ca argument numele directorului în care sunt scrise intrările cache. În constructor se fac o serie de verificări minimale de corectitudine: dacă directorul nu exista anterior, atunci acesta este creat; dacă numele directorului desemnează un fişier deja existent pe disc, atunci acesta este redenumit şi în locul acestuia este creat directorul de cache.

```
public WebCacheFile(String workingDir) {
   this.workingDir = workingDir;
   // verifica si directorul
   try {
      File f = new File(workingDir);
      if (!f.exists()) { // daca inca nu exista directorul de cache
         f.mkdirs(); // atunci acesta este creat
      } else if (f.exists() && !f.isDirectory()) {
         // daca deja exista, dar desemneaza un fisier
         try {
            File nf = File.createTempFile(workingDir, "tmp");
            byte b[] = new byte[1024];
            int nb;
            FileOutputStream fos = new FileOutputStream(nf);
            FileInputStream fis = new FileInputStream(f);
            while ((nb = fis.read(b)) > 0) {
               fos.write(b, 0, nb);
               fos.flush();
            }
            fos.close();
            fis.close();
            System.err.println("File "+workingDir+" already exists, renamed
               to "+nf.getAbsolutePath());
         } catch (Throwable t) {
            t.printStackTrace();
            System.exit(-1);
         }
         f.delete(); // atunci acesta este sters
         f.mkdirs(); // si se creaza un director pentru intrarile de cache
      }
   } catch (Exception e) { }
}
```

O primă metodă implementată este *init*, metodă ce are rolul de a citi intrările curente cache de pe disc şi de a forma hash-ul corespunzător pentru păstrare în memorie. Funcția este utilă în cazul în care aplicația este repornită, caz în care la inițializare ultimele intrări din cache sunt refăcute plecând de la informația existentă pe disc:

```
public Hashtable init() {
   Hashtable h = new Hashtable();
   File f = new File(workingDir);
   if (!f.exists() || !f.isDirectory())
      return h;
   File fs[] = f.listFiles(); // continutul directorului de cache
   if (fs == null || fs.length == 0) // daca nu exista nici o intrare
      return h;
   for (int i=0; i<fs.length; i++) {
```

```
      // pentru fiecare intrare reface intrarea cache in memorie
      if (!fs[i].isFile()) continue; // nu e de interes, il sarim
      try {
         ObjectInputStream ois = new ObjectInputStream(new
            FileInputStream(fs[i]));
         WebCacheEntry entry = (WebCacheEntry)ois.readObject();
         h.put(entry.getURL(), entry);
         ois.close();
      } catch (Exception e) {
         // am gasit un fisier corupt continand o intrare cache
         fs[i].delete();
      }
   }
   return h;
}
```

În cadrul funcției existența altor directoare este tratată prin simpla omitere a acestora. În plus există posibilitatea ca unele fişiere să conțină date corupte, de exemplu în cazul în care aplicația a fost întreruptă chiar în momentul în care actualiza conținutul unui fişier pe disc. În acest caz ştergem intrarea cache coruptă de pe disc, forțând astfel verificarea intrării într-o interogare ulterioară.

O altă funcție utilă este cea de actualizare a conținutului unei intrări cache pe disc:

```
public void updateEntry(WebCacheEntry entry) {
   String fileName = workingDir+File.separatorChar+formName(entry);
   File f = new File(fileName);
   if (f.exists()) // vechiul fisier, daca deja exista, este sters...
      f.delete();
   // scriem continutul pe disc
   try {
      ObjectOutputStream oos = new ObjectOutputStream(new
          FileOutputStream(f));
      oos.writeObject(entry);
      oos.close();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Funcție scrie o intrare cache primită ca argumetn într-un fişier corespunzător prin serializarea în forma serializată a obiectului *WebCacheEntry*. Funcția *formName* întoarce un nume unic de fişier plecând de la URL-ul corespunzător intrării cache:

```
URL u = new URL(entry.getURL());
return u.getHost()+"_"+(u.getPort() > 0 ? u.getPort()+"_" :
                   "")+u.getPath().replace(File.separatorChar, '_');
```

Clasa principală a aplicației cache web este *WebCache*. În cadrul acestei clasei se face tratarea cererilor de pagini web, clasa punând la dispoziția utilizatorilor o unică metodă publică, *public String requestURL(String url)*, funcție ce întoarce conținutul unei pagini web pe baza unui URL furnizat ca argument.

Clasa este inițializată prin furnizarea a două argumente: dimensiunea maximă pe disc a datelor din cache şi timpul maxim de păstrare a unei intrări cache înainte de o revalidare a conținutului cu pagina accesată de la URL-ul corespunzător.

```
private Hashtable cache;
private WebCacheFile cacheFile;
private long maxTimeToKeepPage = -1;
private long maxSizeOnDisk = -1;

public WebCache(long maxSizeOnDisk, long maxTimeToKeepPage) {
   this.maxSizeOnDisk = maxSizeOnDisk;
```

```
    this.maxTimeToKeepPage = maxTimeToKeepPage;
    String workingDir =
          System.getProperty("user.home")+File.separatorChar+"cache_test";
    cacheFile = new WebCacheFile(workingDir);
    cache = cacheFile.init();
}
```

După cum se poate observa, în constructorul clasei se apelează şi metoda *init* a clasei *WebCacheFile* pentru formarea cache-ului în memorie pornind de la intrările cache deja existente pe disc.

Metoda *requestURL* implementează următoarele:

```
public String requestURL(String url) {
   // stergem intrarile prea vechi a mai fi valabile
   purgeOldPages();
   if (cache.containsKey(url)) // daca deja avem in cache continutul
      return ((WebCacheEntry)cache.get(url)).getContent();
   // altfel facem o interogare a adresei
   String content = getWebPage(url);
   if (content == null) // daca interogarea a esuat
      return content;
   // formam o noua intrare cache
   WebCacheEntry entry = new WebCacheEntry(url);
   entry.setContent(content);
   // o adaugam intre intrarile pastrate in memorie
   cache.put(url, entry);
   // si o scriem si pe disc pentru toleranta la defecte
   cacheFile.updateEntry(entry);
   // verificam daca cu ce am adaugat in cache nu am depasit
   // dimensiunea maxima a cache-ului pe disc
   if (maxSizeOnDisk > 0l) {
      long sizeOnDisk = cacheFile.getSizeOnDisk();
      while (sizeOnDisk > maxSizeOnDisk) {
         WebCacheEntry wce = null;
         for (Enumeration en = cache.keys(); en.hasMoreElements(); ) {
            String su = (String)en.nextElement();
            WebCacheEntry e = (WebCacheEntry)cache.get(su);
            if (wce == null) wce = e;
            else if (wce.getLastRead() > e.getLastRead()) wce = e; // (LRU)
         }
         if (wce == null) break;
         cache.remove(wce.getURL());
         cacheFile.removeEntry(entry);
         long newSizeOnDisk = cacheFile.getSizeOnDisk();
         if (sizeOnDisk == newSizeOnDisk) break;
         sizeOnDisk = newSizeOnDisk;
      }
   }
   return content;
}
```

În cadrul metodei se începe prin ştergea intrărilor cache prea vechi, intrări a căror valabilitate a expirat. Cu alte cuvinte, paginile sunt ținute în cache atât timp cât utilizatorul nu inițiază o acțiune de accesare a conținutului acestora. Desigur, soluția implementată nu este singulară. O altă soluție ar fi fost să folosim un thread separat care la intervale de timp să verifice valabilitatea paginilor ținute în cache. Această implementare însă se depărtează de concentrarea acțiunilor spre client şi impune folosirea unor mijloace de sincronizare a accesului la cache între threadul principal şi acest thread auxiliar. Lăsăm însă ca exercițiu cea de-a doua soluție de implementare.

În cadrul funcției se continuă cu verificarea existenței paginii în cache. Dacă pagina există atunci este întors direct conținutul acesteia.

Dacă însă în cache nu avem o intrare corespunzătoare paginii cerute se încearcă interogarea serverului web la distanță. Dacă interogarea reuşeşte se continuă prin scrierea noii înregistrări în

cache şi întoarcea conținutului paginii. Odată cu scrierea unei noi intrări în cache apare însă şi problema posibilei depăşirii a spațiului maxim de stocare a intrărilor cache. În acest caz, după cum se poate observa, se impune şi o verificare, iar paginile cele mai vechi accesate sunt şterse, conform algoritmului LRU.

Funcția de interogare a serverului de la distanță este următoarea:

```
private String getWebPage(String u) {
   URL url;
   try {
      url = new URL(u);
   } catch (MalformedURLException e) {
      System.err.println("ERROR: invalid URL " + u);
      return null;
   }
   // ne limitam la protocolul http - cache web
   if (url.getProtocol().compareTo("http") != 0)
      return null;
   try {
      URLConnection urlConnection = url.openConnection();
      urlConnection.setAllowUserInteraction(false);
      InputStream urlStream = url.openStream();
      // citirea continutului paginii web de la respectiva adresa
      byte b[] = new byte[1000];
      int numRead = urlStream.read(b);
      String content = new String(b, 0, numRead);
      while (numRead != -1) {
         numRead = urlStream.read(b);
         if (numRead != -1) {
            String newContent = new String(b, 0, numRead);
            content += newContent;
         }
      }
      urlStream.close();
      return content;
   } catch (IOException e) {
      System.err.println("ERROR: couldn't open URL " + u);
   }
   return null;
}
```

În cadrul implementării folosim clasele *URL* şi *URLConnection* puse la dispoziție de Java pentru accesarea paginilor web. Desigur, implementarea are un scop pur academic, în practică fiind necesară verificarea unor parametrii de conectivitate (setarea unui *timeout* corespunzător de exemplu), sau setarea unor parametrii de proxy dacă aceasta este singura modalitate prin care stația de lucru poate accesa pagini la distanță, etc.

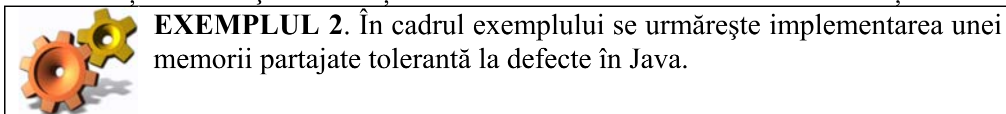
Aceasta este implementarea aplicației de cache web. Exemplul complet se poate consulta în cadrul listingului *example1*. În plus, listingul conține şi un exemplu de client, implementat cu ajutorul clasei *ClientTest*:

```
private static final String urls[] = new String[] {
   "http://google.com", "http://www.yahoo.com", "http://www.pub.ro",
   "http://www.acs.pub.ro", "http://www.acasa.ro"
};
...
WebCache cache = new WebCache(size, time);
Random r = new Random(System.currentTimeMillis());
while (true) { // incercam cateva citiri aleatoare de adrese
   int poz = r.nextInt(urls.length);
   String page = cache.requestURL(urls[poz]);
   System.out.println(page);
   try { Thread.sleep(1000); } catch (Throwable t) { }
}
```

Rularea exemplului se realizează cu *ant*. Parametrii de execuție (dimensiunea spațiului pe disc, timpul limitat de valabilitate a unei intrări cache) se pot specifica modificând intrările corespunzătoare din fişierul *build.xml*.

4.2. Memorie partajată tolerantă la defecte

În continuare prezentăm un exemplu de soluție de implementare a unei memorii partajate între procese distribuite. În cadrul exemplului sunt descrise mecanisme atât de asigurare a consistenței datelor, actualizările făcute asupra unor valori fiind corect văzute de toate procesele participante, dar şi de asigurare a toleranței la defecte. Astfel, în cadrul sistemului dorim să nu avem nici un single-point-of-failure care să ducă la oprirea completă a funcționării proceselor rămase în viață, dar dorim de asemenea şi ca în cazul defectării unor procese să implementăm mecanisme de revenire din eroare. Soluția completă de implementare poate fi urmărită în cadrul listingului *example2*. Pentru mai multe detalii legate de soluțiile alese pentru asigurarea consistenței datelor şi a toleranței la defecte se recomandă consultarea notițelor de curs.



EXEMPLUL 2. În cadrul exemplului se urmăreşte implementarea unei memorii partajate tolerantă la defecte în Java.

Aplicația implementată respectă arhitectura prezentată în Figura 1. În figură sunt descrise principalele entități implicate şi tipurile de mesaje folosite.

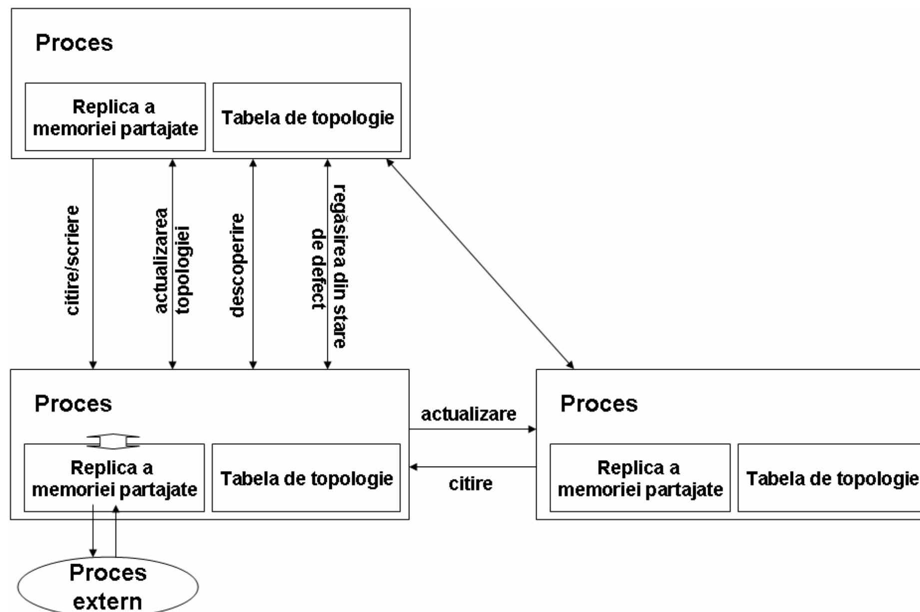


Figura 1. Arhitectura aplicației ce implementează funcționalitatea memoriei partajate.

În cadrul aplicației memoria partajată este reprezentată ca o colecție de variabile (obiecte serializabile), reprezentate prin numele şi valoarea curentă asociată. Toate procesele sistemului dețin local câte o copie completă a memoriei partajate. Pentru fiecare variabilă există câte un proces având un rol special, cel de *proprietar* al ei. Atunci când o variabilă este scrisă prima dată în memoria partajată procesul ce inițiază scrierea este proprietarul respectivei variabile.

Pentru asigurarea consistenţei datelor convenim să folosim o schemă simplă în care un proces ce doreşte scrierea sau citirea unei variabile trebuie să interogheze procesul owner al respectivei variabile. În acest fel se asigură o consistenţă corectă a datelor în funcţionare prin implementarea mecanismului de excludere mutuală a operaţiilor de scriere şi citire direct la client.

Până în acest moment avem o funcţionalitate de tip remote procedure call: un proces care doreşte accesul la o variabilă face o cerere la distanţă la instanţa deţinătoare a respectivei variabile. Totuşi, implementarea de memorie partajată propusă realizează şi mai mult. Menţinerea unei replici locale conţinând toate variabilele curent cunoscute în sistem de către fiecare proces ajută la soluţionarea a cel puţin două probleme: procesul de descoperire şi procesul de asigurare a toleranţei la defecte.

În general în aplicaţiile distribuite (şi nu numai) pentru a avea acces la informaţii referitoare la identitatea proceselor, a unor caracteristici sau entităţi funcţionale, se pune problema asigurării unui mecanism de descoperire adecvat. Conform soluţiei descrise, menţinând locală o copie a fiecărei variabile cunoscute în sistem eliminăm necesitatea unei operaţii de descoperire pentru fiecare operaţie în parte, ştim întotdeauna în sistem cine este procesul cu care trebuie să stabilim o interogare. În plus, toate procesele deţin replici complete ale memoriei partajate şi acest lucru facilitează adoptarea unei soluţii de asigurare a toleranţei la defecte prin redundanţă.

În cazul soluţiei prezentate cea mai gravă problemă ce apare atunci când apare un defect în funcţionarea unui proces este aceea că toate variabilele deţinute de respectivul proces rămân fără proprietar. Aceasta înseamnă că atunci când un proces încearcă o scriere sau citire a unei variabile el nu va găsi procesul proprietar cu care să realizeze negocierea accesului. Pe de altă parte, este clar că un proces ce încearcă un acces trebuie să încerce o comunicare cu procesul proprietar, ceea ce duce la apariţia unei excepţii de I/O şi deci avem un mecanism de detectare a defectelor.

Soluţia completă de asigurare a toleranţei la defecte este următoarea. În momentul în care un proces descoperă că un alt proces iese din sistem, el informează toate procesele despre dispariţia procesului defect (el este ignorat în sistem din acest moment, revenirea ulterioară însemnând reluarea tuturor paşilor de iniţializare). Totodată procesul iniţiază şi un mecanism de vot prin care îşi exprimă dorinţa de a deveni noul proprietar al tuturor variabilelor anterior deţinute de procesul defect. Un proces care primeşte un mesaj de vot poate decide pe baza unei ordonări a identificatorilor de proces că el doreşte să fie de fapt noul proprietar al variabilelor. Procesul de vot este similar celui de tip *bully*. În final procesul de vot duce la desemnarea unui proprietar unic şi toate variabilele din memoria cache sunt actualizate corespunzător în sensul desemnării proprietarului corect. Aceasta reprezintă o soluţie clasică de asigurare a toleranţei la defecte ce constă în ideea asigurării redundanţei informaţiei: în acest caz dacă un proces iese din sistem un altul îi poate lua imediat locul.

Pentru asigurarea unei toleranţe la defecte ridicate se pune problema proiectării sistemului astfel încât să nu existe nici un *single-point-of-failure*. În general în aplicaţiile distribuite cel mai clasic astfel de punct unic de defecţiune este reprezentat de serverul central ce asigură mecanismul de descoperire a resurselor. În cazul aplicaţiei prezentate folosim următoarea soluţie. Convenim ca fiecare proces să deţină local o tabelă de topologie: o listă conţinând toate procesele curent existente în sistem. În momentul în care un proces doreşte să intre în sistem el va primi ca informaţii utile doar adresa unui alt proces deja conectat în sistem. Noul proces se va conecta la procesul respectiv şi va primi o copie a tabelei sale de topologie şi o copie a memoriei partajate. Urmează doar ca procesul nou intrat să se prezinte tuturor proceselor de a căror existenţa a aflat din tabela de topologie primită, iar fiecare proces ce primeşte un mesaj de prezentare să actualizeze corespunzător propria tabelă locală de topologie. O problemă imediată legată de această abordare este prezentată în Figura 2.

În cadrul figurii este prezentat un sistem compus din două procese, P1 şi P2. Fiecare proces ştie de existenţa celuilalt (tabela de topologie conţine intrările corecte în acest sens). La un moment dat în sistem se consideră intrarea a două noi procese, P3 şi P4. Pentru descoperirea proceselor existenţa P3 interoghează procesul P1, iar procesul P4 pe procesul P2, după cum este

prezentat în figură. P1 va răspunde cu propria tabelă de topologie conținând intrările P1 şi P2. După primirea tabelei de topologie, procesul P3 se va prezenta proceselor P1 şi P2, în final fiecare dintre procesele P1, P2 şi P3 cunoscând în acest mod o aceeaşi tabelă formată din cele trei intrări.

Considerăm acum că la un moment dat, în cursul acțiunilor întreprinse de procesul P3, şi procesul P4 doreşte să-şi anunțe intrarea în cadrul sistemului.

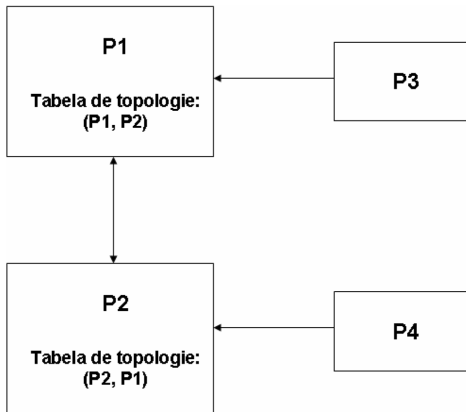


Figura 2. Exemplificarea unei situații de criză în algoritmul de descoperire.

Conform algoritmului, şi procesul P4 va primi din partea procesului P2 o tabelă de topologie conținând intrările P1 şi P2 (procesul P3 încă nu a apucat să se prezinte procesului P2). Urmează ca procesul P4 să se prezinte proceselor P1 şi P2 astfel descoperite, ca în final el să cunoască o tabelă de topologie conținând intrările P1, P2 şi P4. Însă în final procesele P1 şi P2 vor cunoaşte intrările P1, P2, P3 şi P4. Iată deci o situație de desincronizare a cunoştințelor despre topologia curentă de procese apărută din cauza faptului că mesajele trimise de procese sunt recepționate în orice ordine.

Soluția adoptată la această problemă este următoarea. De fiecare dată când un proces trimite un mesaj de introducere el primeşte din partea procesului respectiv înapoi întreaga tabelă de topologie cunoscută. De fiecare dată când procesul primeşte o tabelă de topologie, verifică dacă nu conține intrări ce nu îi sunt încă cunoscute, caz în care îşi actualizează corespunzător şi propria tabelă de topologie. Conform acestui algoritm, repetăm din nou paşii exemplului prezentat anterior.

Pentru început procesul P3 trimite un mesaj de introducere procesului P1. P1 adaugă în propria tabelă de topologie o intrare corespunzătoare procesului P3 şi trimite înapoi tabela de topologie nou formată. Procesul P3 primeşte tabela de topologie şi adaugă în propria tabelă de topologie intrările nou aflată, P2. Apoi el trimite un mesaj de introducere şi procesului P2.

Considerăm din nou că în acest moment procesul P4 doreşte şi el intrarea în sistem. El cunoaşte pentru început doar propria intrare şi intrarea corespunzătoare procesului P2. Procesul P4 trimite astfel un mesaj de introducere procesului P2.

Considerăm acum că procesul P2 primeşte întâi cererea din partea procesului P4. Trimite înapoi tabela de topologie proprie, conținând intrările P1, P2 şi P4. Procesul P4 descoperă un proces nou, P1, şi trimite în consecință un mesaj de introducere acestui proces. Între timp procesul P2 primeşte şi mesajul de introducere din partea procesului P3. Trimite înapoi propria tabelă de topologie, conținând intrările P1, P2, P3 şi P4. Procesul P3 află astfel de noua intrare, P4, şi trimite în consecință un mesaj corespunzător de introducere acestui proces.

În acest fel, la sfârşitul algoritmului, toate procese cunosc corect toate intrările de topologie, fără a se mai pune problema apariției unor defazaje apărute din cauza comunicației.

Să vedem în continuarea modalitatea de implementare practică a aplicației de memorie partajată. Am amintit faptul că în memoria partajată considerăm că avem o serie de variabile

partajate între procesele componente ale sistemului. Clasa corespunzătoare implementării unui variabile este *SharedMemoryVariable*:

```
public class SharedMemoryVariable implements Serializable {
       public LocalAddress owner;
       public final String name;
       public Serializable value;
```

Câmpul *owner* se foloseşte pentru a identifica proprietarul unei anumite variabile, procesul ce trebuie interogat pentru drepturi de acces asupra respectivei variabile. Câmpurile *name* şi *value* desemnează numele, respectiv valoarea curentă a unei variabile. Clasa *LocalAddress* conţine doar două câmpuri, adresa şi portul corespunzător unui proces distribuit din componenţa sistemului.

Clasa ce implementează memoria partajată este *SharedMemoryReplica*. Aceasta este clasa ce se foloseşte în mod direct de către utilizator sau o anumită aplicaţie, clasa punând la dispoziţie doar două metode:

- *read* – se foloseşte pentru citirea valorii curente a unei variabile din memoria partajată;
- *write* – se foloseşte pentru actualizarea valorii curente a unei variabile din memoria partajată.

```
public Object read(String varName) {
   if (variables == null || !variables.containsKey(varName)) {
      // daca nu exista variabila
      return null;
   }
   SharedMemoryVariable sh = (SharedMemoryVariable)variables.get(varName);
   // deleaga citirea propriu-zisa procesului corespunzator
   return process.read(sh);
}

public void write(String varName, Serializable varValue) {
   SharedMemoryVariable var = null;
   if (variables.containsKey(varName)) {
      var = (SharedMemoryVariable)variables.get(varName);
   } else {
      // o variabila noua
      var = new SharedMemoryVariable(process.localAddress, varName);
      variables.put(varName, var);
   }
   var.value = varValue; // setam noua valoare locala
   process.write(var); // delegam scrierea procesului
}
```

Fiecare replică a memorie partajate are ataşată un proces curent care se ocupă de toate schimburile de mesaje implicate. După cum se poate observa, ambele operaţii, scrierea şi citirea, nu influenţează în cadrul acestei clasei decât copiile locale, asigurarea consistenţei datelor între toate procesele implicate în sistemul de partajare a datelor memoriei partajate fiind delegată procesului local ataşat.

Clasa care implementează funcţionalitatea proceselor distribuite ce au rolul de a asigura consistenţa datelor corespunzătoare memoriei partajate şi toleranţa la defecte este *SharedMemoryProcess*.

Elementele importante reţinute de un proces sunt tabela de topologie curentă şi copia locală a memoriei partajate:

```
       // topologia curenta cunoscuta de procesul local
       private final LinkedList topology = new LinkedList();

       // replica locala procesului
       private SharedMemoryReplica replica;
```

În cadrul constructorului clasei se începe prin inițializarea unui server local ce va fi folosit pentru primirea unor mesaje de actualizare a stării sistemelor din partea celorlalte procese aparținând sistemului:

```
public SharedMemoryProcess(String discoveryAddress, int discoveryPort)
                       throws Exception {
   // initializam serverul
   try {
      if (discoveryAddress == null && discoveryPort > 0)
         server = new ServerSocket(discoveryPort);
      else
         server = new ServerSocket(0);
      (new Thread(new ServerThread())).start();
      localAddress = new
         LocalAddress(InetAddress.getLocalHost().getHostAddress(),
         server.getLocalPort());
   } catch (Throwable t) {
      t.printStackTrace();
      throw new Exception("Error in creating the server socket");
   }
```

Conform listingului, preluarea conexiunilor primite se realizează într-un thread separat. De fapt, pentru fiecare conexiune nou primită se porneşte câte un nou thread ce este ulterior răspunzător de tratarea mesajelor recepționate pe respectiva conexiune:

```
private class SocketThread implements Runnable {
   private Socket s;
   private ObjectInputStream ois;
   private ObjectOutputStream oos;
   public void run() {
      Thread.currentThread().setName(s.toString());
      while (true) {
         try {
            Byte command = (Byte)ois.readObject(); // citim comanda
            treatCommand(command, ois, oos); // si o tratam corespunzator
         } catch (Exception e) {
            break;
         }
      }
      try { ois.close();oos.close();s.close();} catch (Throwable t) { }
   }
}
```

În cadrul aplicației convenim ca orice mesaj să fie identificat printr-un câmp *Byte* care să îl preceadă. Astfel, tipurile posibile de mesaje implementate sunt:

```
private static final Byte sendDiscovery = Byte.valueOf((byte)0);
private static final Byte requestReplica = Byte.valueOf((byte)1);
private static final Byte readVar = Byte.valueOf((byte)2);
private static final Byte writeVar = Byte.valueOf((byte)3);
private static final Byte removeProcess = Byte.valueOf((byte)4);
private static final Byte newOwner = Byte.valueOf((byte)5);
private static final Byte changeValue = Byte.valueOf((byte)6);
```

Vom detalia semnificația fiecărui tip de mesaj în continuare. Revenind la constructorul clasei, pasul următor pornirii serverului este reprezentat de etapa de descoperire a topologiei curente:

```
synchronized (topology) {
   if (discoveryAddress != null) {
      LinkedList yetToSolve = new LinkedList();
      // incepem procesul de descoperire pornind de la adresa indicata prin
      // argumentele constructorului
```

```
      yetToSolve.addLast(new LocalAddress(discoveryAddress, discoveryPort));
      while (true) {
         if (yetToSolve.size() == 0) break;
         LocalAddress nextAddress = (LocalAddress)yetToSolve.removeFirst();
         // apelam initializarea descoperirii cu o anumita adresa
         LinkedList tmpList = init(nextAddress.address, nextAddress.port);
         if (tmpList == null) continue;
         if (!topology.contains(nextAddress) &&
                   !nextAddress.equals(localAddress))
         topology.addLast(nextAddress);
         for (Iterator it = tmpList.iterator(); it.hasNext(); ) {
            LocalAddress adr = (LocalAddress)it.next();
            if (!topology.contains(adr) && !adr.equals(localAddress))
               yetToSolve.addLast(adr);
         }
      }
   }
   if (!topology.contains(localAddress))
      topology.addLast(localAddress);
}
```

Implementarea mecanismului de descoperire urmează paşii prezentați anterior. Pentru implementare s-a convenit folosirea unei liste de adrese noi (adrese de procese ce au fost de abia descoperite şi cu care procesul curent nu a inițiat mecanismul de prezentare). Din această listă se scoate la fiecare pas câte un membru şi se trimite un mesaj prin care procesul se prezintă ca nou intrat în sistem procesului respectiv. Ca răspuns este primită topologia curent cunoscută de procesul respectiv, ce este folosită apoi pentru a verifica existența unor intrări ce nu sunt încă cunoscute local.

Metoda *init* folosită în listingul anterior implementează trimiterea tipului de mesaj *sendDiscovery* (de prezentare):

```
public LinkedList init(String discoveryAddress, int discoveryPort) {
   LinkedList tmpList = null;
   try {
      Socket s = new Socket(discoveryAddress, discoveryPort);
      ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
      ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
      // trimitem tipul de mesaj (descoperire)
      oos.writeObject(sendDiscovery);
      // si ne prezentam
      oos.writeObject(localAddress);
      // citim raspunsul - topologia curent cunoscuta de respectivul client
      tmpList = (LinkedList)ois.readObject();
      try { ois.close(); oos.close(); s.close(); } catch (Exception e) { }
   } catch (Throwable t) {
      t.printStackTrace();
   }
   return tmpList;
}
```

Ultimul pas în construcția constructorului este obținerea unei replici locale din partea unuia dintre procesele descoperite la pasul anterior:

```
public void requestReplica(String discoveryAddress, int discoveryPort)
                throws Exception {
   Socket s = new Socket(discoveryAddress, discoveryPort); // conectare
   ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
   ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
   // scriem mesajul - cerere copie curenta a replicii
   oos.writeObject(requestReplica);
   // ni se raspunde inapoi cu copia memoriei partajate
   replica = (SharedMemoryReplica)ois.readObject();
   try { ois.close(); oos.close(); s.close(); } catch (Exception e) { }
}
```

De data aceasta nu interesează obținerea unei liste exacte a variabilelor existente, actualizarea corectă are loc în momentul implementării funcțiilor de citire şi scriere.

În continuare detaliem etapele operației de citire a unei valorii curente a unei variabile:

```
public Object read(SharedMemoryVariable var) {
   if (var.owner.equals(localAddress)) // daca suntem owneri
      return ((SharedMemoryVariable)replica.variables.get(var.name)).value;
   // altfel interogam procesul detinator al respectivei variabile
   try {
      Socket s = new Socket(var.owner.address, var.owner.port);
      ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
      ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
      // tipul de mesaj (citire valoare variabila)
      oos.writeObject(readVar);
      // numele variabilei
      oos.writeObject(var.name);
      // memoria partajata
      SharedMemoryVariable v = (SharedMemoryVariable)ois.readObject();
      if (!v.owner.equals(var.owner)) {
         replica.variables.put(v.name, v);
         return read(v);
      }
      var.value = (Serializable)v.value;
      try { ois.close(); oos.close(); s.close(); } catch (Exception e) {}
      return v.value;
   } catch (Throwable t) {
      t.printStackTrace();
   }
   // daca am ajuns aici inseamna ca a aparut o problema...
   processLost(var.owner);
   return read(var);
}
```

Citirea valorii unei variabile presupune pentru început verificarea procesului proprietar asociat variabilei respective. Astfel, dacă chiar procesul curent este proprietarul, este întoarsă direct valoarea curentă citită din memoria locală. Altfel, se trimite un mesaj de solicitare a valorii curente procesului owner. Procesul răspunde prin propria valoare a variabilei, care este sigur ultima (o scriere actualizează întotdeauna prima dată copia locală procesului owner, deci se respectă o consistență strictă a datelor între două operații consecutive de scriere şi citire).

Problema apare în momentul în care există un defazaj între identificatorul procesului pe care procesul local îl consideră a fi proprietar al variabilei şi identificatorul procesului pe care procesul remote îl consideră a fi proprietar al variabilei. Acest lucru se întâmplă de exemplu în cursul unei operații de votare a unui nou proprietar al variabilei. În acest caz se consideră că procesul remote are dreptate şi se realizează o interogare a procesul ce a fost identificat ca owner. Nu este greu de văzut că, indiferent care ar fi posibilele ordonări de evenimente, algoritmul are finalitate şi în final atât procesul local cât şi cel de la distanță vor ajunge la un consens în privința proprietarului de drept a variabilei, care este de fapt şi proprietarul real.

O construcție interesantă este cea din finalul metodei, când se întâlneşte o excepție şi se trece la pasul necesar anunțării tuturor proceselor rămase încă în viață că a apărut un proces defect în sistem. În aceast caz procesul proprietar este scos din listă şi procesul curent încearcă să devină automat noul proprietar al tuturor variabilelor anterior deținute de procesul decedat (se inițializează procesul de vot prezentat anterior):

```
private void takeOwnerShip(SharedMemoryVariable var) {
   var.owner = localAddress; // ne declaram noul detinator
   synchronized (topology) {
     // informam toate procese despre schimbarea survenita
     LinkedList yetToSolve = new LinkedList();
```

```
     for (Iterator it = topology.iterator(); it.hasNext(); ) {
        LocalAddress a = (LocalAddress)it.next();
        if (a.equals(localAddress)) continue;
        yetToSolve.addLast(a);
     }
     // pentru fiecare proces cunoscut din topologia curenta
     // trimitem un mesaj de informare despre noul owner
     while (true) {
        if (yetToSolve.size() == 0) break;
        LocalAddress nextAddress = (LocalAddress)yetToSolve.removeFirst();
        try {
           Socket s = new Socket(nextAddress.address, nextAddress.port);
           ObjectInputStream ois = new
                 ObjectInputStream(s.getInputStream());
           ObjectOutputStream oos = new
                 ObjectOutputStream(s.getOutputStream());
           // tipul de mesaj
           oos.writeObject(newOwner);
           // si noua valoare a respectivei variabile
           oos.writeObject(var);
           LinkedList topo = (LinkedList)ois.readObject();
           for (Iterator it2 = topo.iterator(); it2.hasNext(); ) {
              LocalAddress a = (LocalAddress)it2.next();
              if (!topology.contains(a) && !localAddress.equals(a)) {
                 topology.add(a);
                 yetToSolve.addLast(a);
              }
           }
           try {
              ois.close(); oos.close(); s.close();
           } catch (Exception e) { }
       } catch (Exception e) {
           e.printStackTrace();
           processLost(nextAddress);
       }
    }
  }
}

private void processLost(LocalAddress processAddress) {
  synchronized (topology) {
    // actualizam corespunzator tabela de topologie locala
    topology.remove(processAddress);
    LinkedList yetToSolve = new LinkedList();
    for (Iterator it = topology.iterator(); it.hasNext(); ) {
      LocalAddress a = (LocalAddress)it.next();
      if (a.equals(localAddress)) continue;
      yetToSolve.addLast(a);
    }
    while (true) {
      if (yetToSolve.size() == 0) break;
      LocalAddress nextAddress = (LocalAddress)yetToSolve.removeFirst();
      try {
        Socket s = new Socket(nextAddress.address, nextAddress.port);
        ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
        ObjectOutputStream oos = new
                   ObjectOutputStream(s.getOutputStream());
        // trimitem tipul de mesaj (proces disparut)
        oos.writeObject(removeProcess);
                                      // trimitem adresa procesului disparut
          oos.writeObject(processAddress);
          // si propria adresa pentru a initia procesul de vot
          oos.writeObject(localAddress);
          LinkedList topo = (LinkedList)ois.readObject();
          for (Iterator it2 = topo.iterator(); it2.hasNext(); ) {
              LocalAddress a = (LocalAddress)it2.next();
              if (!topology.contains(a) && !processAddress.equals(a)
                   && !localAddress.equals(a)) {
                 topology.add(a);
                 yetToSolve.addLast(a);
              }
```

```
            }
            LocalAddress a = (LocalAddress)ois.readObject();
            if (!a.equals(localAddress)) {
               // daca am primit de la un proces un vot negativ
               return;
            }
         } catch (Exception e) {
            e.printStackTrace();
            processLost(nextAddress);
         }
       }
     }
     // daca am ajuns pana in acest punct
     // toate procesele au fost de acord cu votul nostru
     synchronized (replica.variables) {
       for (Enumeration en = replica.variables.keys();
               en.hasMoreElements(); ) {
          String varName = (String)en.nextElement();
          SharedMemoryVariable smv =
               (SharedMemoryVariable)replica.variables.get(varName);
          if (smv.owner.equals(processAddress)) {
             takeOwnerShip(smv);
          }
      }
    }
}
```

Procesul de votare implică la destinație tratarea mesajelor astfel:

```
if (command.equals(removeProcess)) {
  LocalAddress lostAddress = (LocalAddress)ois.readObject();
  LocalAddress requestAddress = (LocalAddress)ois.readObject();
  synchronized (topology) {
     topology.remove(lostAddress); // actualizam tabela de topologie
     oos.writeObject(topology); // trimitem propria topologie inapoi
  }
  // si trimitem valoarea propriului nostru vot
  if (localAddress.toString().compareTo(requestAddress.toString()) < 0) {
     // acordam votul procesului remote
     oos.writeObject(requestAddress);
   } else {
     // votam pentru preluarea de catre procesul local a ownership-ului
     oos.writeObject(localAddress);
     // initiem propriul proces de votare
     processLost(lostAddress);
   }
   return;
}
if (command.equals(newOwner)) {
   SharedMemoryVariable var = (SharedMemoryVariable)ois.readObject();
   replica.variables.put(var.name, var);
   oos.writeObject(topology);
   return;
}
```

Funcția de scriere a unei variabile este relativ similară celei de citire, în sensul că se trimite un mesaj de actualizare procesului proprietar al variabilei respective. În plus, la primirea unui mesaj de actualizare, procesul owner inițiază un mesaj de broadcast prin care anunță toate procesele curente din proces asupra schimbării de valoare. Acest broadcast devine util numai în procesul de declarare a unui proces ca ieşit din sistem, caz în care noul proces trebuie să cunoască ultima valoare corectă a tuturor variabilelor pe care le moşteneşte.

```
public void write(SharedMemoryVariable var) {

   // daca variabila este detinuta chiar de procesul local
   if (var.owner.equals(localAddress)) {
```

```
        replica.variables.put(var.name, var);
        broadcastNewVar(var);
        return;
    }

    // trimitem noua valoare procesului owner al respectivei variabile
    try {
        Socket s = new Socket(var.owner.address, var.owner.port);
        ObjectInputStream ois = new ObjectInputStream(s.getInputStream());
        ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
        // tipul de mesaje (actualizare variabila)
        oos.writeObject(writeVar);
        // si noua valoare
        oos.writeObject(var);
        // ni se raspunde prin confirmarea valorii actualizate
        Object o = ois.readObject();
        var.value = (Serializable)o;
        try { ois.close(); oos.close(); s.close(); } catch (Exception e) { }
        return;
    } catch (Throwable t) {
        t.printStackTrace();
    }
    processLost(var.owner);
}

// iar procesul de la distanta executa:

if (command.equals(writeVar)) {
    SharedMemoryVariable var = (SharedMemoryVariable)ois.readObject();
    SharedMemoryVariable localVar =
            (SharedMemoryVariable)replica.variables.get(var.name);

    if (localVar != null && var.owner != localVar.owner) {
        localVar.value = var.value; // noua valoare
        // daca owner-ul e altul, s-a produs un defasaj si informam
        // owner-ul corect asupra actualizarii de continut
        write(localVar);
        oos.writeObject(localVar.value);
    } else {
        replica.variables.put(var.name, var);
        oos.writeObject(var.value);
        broadcastNewVar(var);
    }
    return;
}
```

În acest mod asigurăm atât o consistență a datelor actualizate, cât şi o toleranță ridicată la defecte.

În cadrul listingului *example2* mai există şi un client de testare a aplicației de memorie partajată, *ClientTest*:

```
final String varNames[] = new String[] { "a", "b", "c", "d", "e" };
Random r = new Random(System.currentTimeMillis());
SharedMemoryProcess process = new SharedMemoryProcess(adr, port);
SharedMemoryReplica replica = process.getReplica();

while (true) {
    int poz = r.nextInt(varNames.length);
    int op = r.nextInt(100);

    if (op < 50) { // readOp
        i = (Integer)replica.read(varNames[poz]);
        System.out.println(System.currentTimeMillis()+" Read "+varNames[poz]+
              ":= "+i);
    } else {
        i = new Integer(r.nextInt(1000));
        replica.write(varNames[poz], i);
        System.out.println(System.currentTimeMillis()+" Write "+varNames[poz]+
```

```
            ":= "+i);
    }
}
```

Exemplul complet poate fi consultat în cadrul listingului *example2*. Rularea exemplului se realizează cu *ant*. Parametrii de execuție (adresa şi portul corespunzătoare unui proces la care facem conectarea pentru intrarea în sistem) se pot specifica modificând intrările corespunzătoare din fişierul *build.xml*.

4.3. Aplicație practică

Vă propunem modificarea exemplelor descrise în cadrul laboratorului cu alte construcții specifice consistenței datelor şi/sau toleranței la defecte.

Task1.

Modificați aplicația de cache web prezentată în cadrul punctului 1. astfel încât alegerea paginilor ce sunt şterse din cache pentru a face loc altora noi să se facă conform algoritmului LFU (least frequently used). Conform acestui algoritm, intrările cache păstrează un indicator al frecvenței acceselor efectuate asupra acestora. Alegerea paginii ce va fi ştearsă nu va mai lua în considerare pagina accesată cel mai în urmă, ci pagina accesată de cele mai puține ori.

Task2.

Modificați aplicația de cache web prezentată în cadrul punctului 1 pentru a folosi noțiunea de ceas sincronizat universal. În acest sens folosiți protocolul NTP şi sursele din listingul *example3*.

Task 3.

În cadrul aplicației de memorie partajată prezentată la punctul 2. apare o problemă atunci când un proces devine prea lent pentru a răspunde în timp util. În acest caz se poate ajunge la timeout, conexiunea să expire şi toate procesele să îl considere defect. Procesul lent poate ulterior să-şi revină, caz în care comportamentul său, conform a ceea ce există implementat, devine ambiguu. Propuneți şi implementați o soluție de tratare a acestui caz de excepție, împreună cu un posibil scenariu de test.