Aspecte ale proiectarii unui *Order Request Broker* (ORB)
(Partea a II-a)

Claudiu VINTE
Goldman Sachs Ltd., Tokyo, Japonia

Asa cum am vazut în prima parte, pentru a realiza o platforma tranparenta de comunicatie utilizând XQuark, acesta trebuie sa fie instalat si sa ruleze în fiecare nod al retelei, la nivel logic, al sistemului.
Întregul flux de mesaje inter-aplicatii este transmis prin intermediul acestor procese XQuark la care trebuie sa se conecteze toate aplicatiile care compun sistemul. Daca doua aplicatii client se afla pe aceeasi masina fizica si exista o instanta XQuark care ruleaza pe aceasta masina (si cele doua aplicatii client sunt conectate la aceasta instanta) atunci un mesaj trimis de una dintre aplicatiile client catre cealalta dintre ele este manipulat de XQuark în asa fel încât mesajul nu ajunge de fapt în retea, nefiind necesar, cei doi clienti se afla pe aceeasi masina fizica.
Cuvinte cheie: *arhitectura, comunicatie, aplicatie, sistem, mesaje.*

Componentele interne ale platformei de comunicatie XQuark

Procesul XQuark are, în principal, doua functii:

- sa puna la dispozitia aplicatiilor client un mecanism prin care sunt preluate mesajele de la acestea pe baza unei adrese definite la nivel logic (*App ID* si, cum am vazut, optional, *AppSerial ID*) si sa distri-buie în mod corespunzator aceste mesaje catre destinatarii aflati în retea si depre ca-re aplicatiile nu au nevoie sa stie unde sunt localizati, ce adrese fizice de retea au masinile pe care ruleaza acesti destinatari – altfel spus sa asigure o tranparenta totala, din perspectiva aplicatiilor client, a modu-lui în care traficul se realizeaza la nivel de retea;
- sa puna la dispozitia unui operator posibilitatea de a monitoriza si gestiona de la distanta starea si configuratia sistemului creat de nodurile în care ruleaza procesele XQuark si aplicatiile client care sunt co-nectate la acestea.

Programul care implementeaza procesul XQuark consta dintr-un singur obiect *XQu arkApp* care contine firul de executie principal din care sunt lansate, în cascada, celelalte fire de executie. Clientii locali sunt deserviti de un fir de executie spe-cializat, care este o instata a clasei *Local Listener*, iar clientii aflati la distanta sunt satisfacuti de un alt fir de executie specia-lizat, care este o instanta a clasei *Network Listener*. Firul de executie principal (*XQu arkApp)* asteapta sa primeasca mesaje de la ambele fire de executie care "asculta" cli-entii locali, respectiv reteaua (clientii aflati la distanta). Când soseste un mesaj acesta este prelucrat dupa cum urmeaza:

- daca este un mesaj de control, atunci este prelucrat intern de catre procesul XQuark;
- daca este un mesaj între doua aplicatii client, atunci este fie rutat catre retea fie catre clientii locali, depinzând de expedi-torul si destinatarul mesajului;
- daca este un mesaj de broadcast de la un client local, atunci este trimis în retea catre toti ceilalti clienti si, de asemenea, tuturor entitatilor de pe masina locala, cu exceptia expeditorului mesajului. În cazul unui mesaj de broadcast primit din retea, acesta este copiat si expediat catre toti clientii locali (figura 1).

Firul de executie *NetworkListener*

Obiectul de tipul *NetworkListener* creeaza o instanta a clasei *NetworkInterface*. Mesajele primite din retea sub forma unei succesiuni de pachete sunt convertite într-un obiect *NetMessage* corespunzator. Simul-tan, se realizeaza conversia de aliniere a octetilor de la ordinea folosita în retea la cea folosita de sistemul de operare de pe masina locala. De asemenea, unele mesaje de stare (de exemplu, *SYS_MAP_CHAN GED* si *PROGRAM_DIED*) sunt prelucrate intern de acest obiect. Altfel, acesta nu face decât sa translateze în mod corespunzator mesajele venite din retea si sa le adauge la coada de mesaje ale firului de executie principal *XQuarkApp*. Mesajele de trimis în retea sunt prelucrate dupa aceeasi sche-ma, conversia de aliniere a octetilor facân-du-se de acesta data de la ordinea folosita pe masina locala catre cea a retelei.

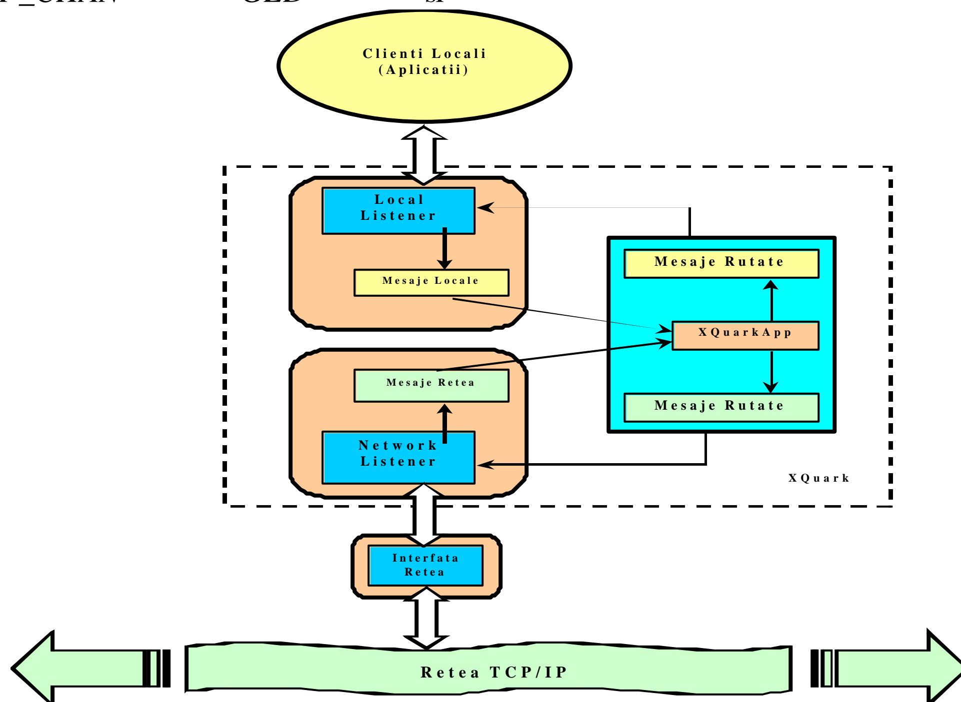

Fig. 1

Firul de executie *LocalListener*

Acest fir de executie creaza un *socket* si asteapta deschiderea de noi conexiuni din partea aplicatiilor client locale. Imediat dupa ce un nou client este conectat, primul mesaj care se asteapta este *NEW_ID_RE QUEST*. În acest moment un nou si unic *ID* este generat si trimis înapoi clientului de catre acest obiect. De asemenea, starea sistemului (*system map*) este actualizata si toata lumea este informata despre aceasta noua entitate intrata în sistem. Fiecare aplicatie client trebuie sa aiba asignat un identificator unic – *App ID*. Acesta este un "*short integer*" si poate avea valori între 2 – 254 inclusiv (valoarea 1 este o valoare speciala si desemneaza procesul XQuark însusi). În consecinta, pot coexista în sis-tem un numar de maximum 253 aplicatii diferite (trebuie însa remarcat ca numarul de instante pentru fiecare aplicatie este, în mod practic, nelimitat, el fiind specificat prin componenta *AppSerial ID* care este un "integer". Asignarea *App ID* este realizata în functia care controleaza fluxul principal de prelucrare al aplicatiei client (uzual *main()*) si trebuie facuta înainte de initierea procedurii de conectare la XQuark. Este responsabilitatea programatorului de apli-catie sa asigure unicitatea *App ID* pentru fiecare aplicatie distincta din sistem, lucru care se realizeaza prin colectarea tuturor identificatorilor de aplicatii într-un singur fisier *header* care va fi inclus de fiecare aplicatie din sistem.

Fire de executie multiple folosite la gestiunea comunicarii

În biblioteca XQnw (*XQuarkNetWare*) este pusa la dispozitia programatorului de apli-catii o colectie de clase care implemen-teaza serviciile cerute pentru comunicarea aplicatiei client cu XQuark. Interfata catre nivel retea este realizata de catre obiectul *NetworkInterface*, care este creat de firul de executie *NetworkListener* al procesului XQuark si manipuleaza întreaga comunica-tie cu reteaua (figura 2). Nivelul retea opereaza numai în termeni de mesage brute (succesiune de pachete), încapsulate în clasa *RawMessage* si toate conversiile între mesaje de tipul *NetMessage* si cele de tipul *RawMessage* sunt realizate de catre obiec-tul *NetworkListener*.

Din moment ce aplicatiile client nu comunica în mod direct în retea (ci în mod mij-locit, prin intermediul *Order Request Bro-ker*-ului XQuark) ele nu utilizeaza nici unul din aceste servicii de retea în mod ne-mijlocit.

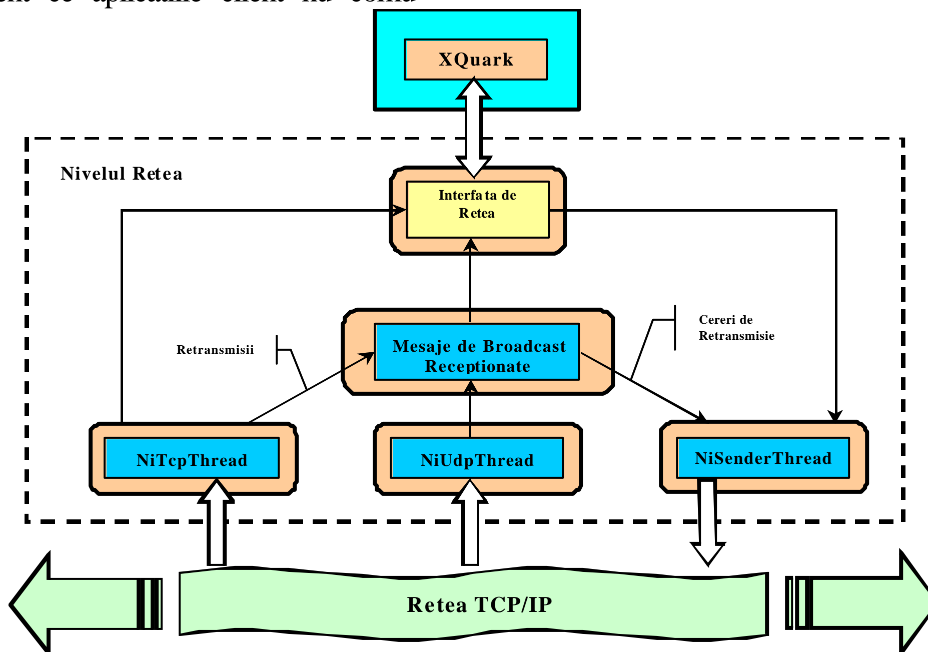

Fig. 2

Clasa *NetworkInterface* încapsuleaza toate protocoalele de comunicatie descrise ante-rior. Când un nou *host* este conectat în retea sau un *host* existent este deconectat (respectiv este întrerupta rularea procesului XQuark pe respectiva masina), obiectul *NiTCPThread* genereaza un pseudo-mesaj care este procesat de firul de executie *NetworkIterface* si apoi starea sitemului, pastrata în *system map*, este actualizata. Tot în aceasta clasa este implementata si o semantica pentru realizarea unui *broadcast* cu posibilitatea corectiei de pachete. Sec-venta pachetelor în cadrul mesajelor este verificata aici si se genereaza, în cazul detectarii unei inadvertente în ordinea de primire a pachetelor, un mesaj de cerere de retransmisie a pachetelor lipsa. Mesajele de broadcast foarte mari (peste 8 K*bytes*) sunt fragmentate si reasamblate de acest obiect.

Firele de executie multiple care realizeaza partea de comunicatie la nivel de retea sunt implementate în urmatoarele clase:

❑ Clasa *NiTcpThread* gestioneaza toate conexiunile TCP/IP din reteaua sistemului. Din moment ce o conexiune TCP/IP este dechisa de un XQuark pentru cu fiecare din celelalte procese XQuark din retea, limita maxima a numarului de conexiuni TCP/IP pe care un host le poate deschide este de aproximativ 64. Aceasta înseamna ca într-un segment de retea pot exista la un moment dat maximum 64 de noduri simul-tan (pot fi realizate mecanisme de rutare a pachetelor între segmente diferite de retea). Atunci când primeste o cerere pentru o noua conexiune, acest fir de executie va schimba cu noul *host* informatiile necesare actualizarii starii sistemului (în *system map*) si din acest moment "asculta" pe aceasta conexiune *socket* TCP/IP mesajele care vin de la *host*-ul conectat. Noile co-nexiuni si întreruperea conexiunii generea-za în fapt pseudo-mesaje care sunt prelu-crate la nivelul imediat superior de catre instata clasei *NetworkInterface*.

❑ Clasa *NiUdpThread* implementeaza mecanismul de "ascultare" pentru mesajele de *broadcast* si actualizeaza coada de me-saje primite din retea în ordinea în care acestea ajung pe *socket*-ul UDP/IP. În cazul în care pe o masina ruleaza mai multe instante ale clasei *NetworkInterface* (sunt mai multe procese XQuark lansate în executie pe o aceeasi masina) este posibil sa se precizeze în constructorul obiectului *NiUdpThread* care dintre aceste interfete sa fie "ascultate". Altfel, mesajele de broad-cast trimise de pe aceeasi masina (implicit se asuma ca sunt trimise de acelasi XQuark) sunt ignorate. Fiecare mesaj de broadcast este transferat ca un singur pachet, daca nu este prea mare. Daca dimensiunea mesajului este mai mare decât dimensiunea maxima a pachetului UDP/IP (peste 8 K*bytes*), atunci mesajul este frag-mentat în pachete mai mici si reasamblat la receptie. Fiecare pachet transmis contine un antet urmat de corpul mesajului. Antetul contine un "numar magic", un numar de secventa a pachetului în cadrul mesajului si un cod de mesaj care indica daca pachetul este un început, o continuare sau termina-rea unui mesaj. Pentru mesajele care sunt continute în întregime într-un singur pachet acest ultim indicator (*flag*) este un "SAU" logic la nivel de *bit* între "start" si "end".

❑ Clasa *NiSenderThread* implementeaza un mecanism de asteptare pe coada mesa-jelor care trebuie transmise în retea. Ori de câte ori un mesaj este pus în aceasta coada acesta este extras si trimis catre *socket*-ul corespunzator (functie de adresa destina-tarului – fie UDP, fie TCP).

Utilizarea *Order Request Broker*-ului XQuark ca platforma unica de comuni-catie

Voi prezenta în continuare detaliile de im-plementare a doua aplicatii care se conec-teza la XQuark (pot fi localizate pe aceeasi masina fizica sau se pot afla la distata în retea – esential este ca pe masina pe care aceste aplicatii ruleaza sa fie instalata si lansata în executie o instata a procesului Xquark). Aplicatiile se numesc XQSender si XQReceiver si realizeaza urmatoarele functiuni: XQSender trimite în retea un mesaj de *broadcast* prin intermediul plat-formei XQuark, mesaj care poate avea con-tinutul unui ordin de vânzare sau cumpara-re în cadrul unui sistem de asistare a tranzactiilor bursiere (în cazul exemplului nostru, mesajul va contine numai un sir de caractere, reprezentând cheia de regasire într-o baza de date a respectivului ordin) si asteapta un mesaj de confirmare de la aplicatia client XQReceiver. Aceasta din urma, în momentul în care receptioneaza mesajul de *broadcast* îl prelucreaza si tri-mite un raspuns adecvat aplicatiei XQSen-der care a trimis mesajul originar. Se poate testa în acest mod viteza de

raspuns a conexiunii realizata pe baza platformei XQuark (figura 3).
Obiectul care încapsuleaza întregul meca-nism de conectare la platforma XQuark si de manipulare a mesajelor este implemen-tat de clasa *MainObject* care reprezinta o modalitate standardizata de acces si utili-zare a conexiunii, putând fi folosita de orice potentiala aplicatie client care se dezvolta în sistem. Ambele aplicatii client folosesc aceeasi clasa pentru realizarea conectivitatii.

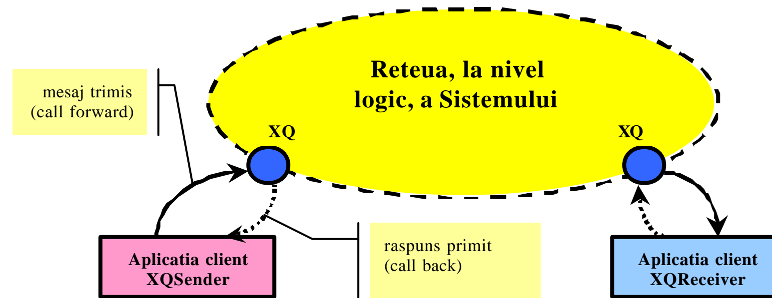

Fig. 3

Sunt definite, de asemenea, clasele care implementeaza mesajele vehiculate între aceste doua aplicatii:
1. *clientOrderMsg*;
2. *clientOrderListMsg*;
3. *clientOrderAckMsg*;
4. *clientOrderAckListMsg*.

Functia *main()* a aplicatiei XQReceiver furnizeaza un mecanism prin care permite aplicatiei sa-si continue fluxul principal al prelucrarii chiar daca pierde conexiunea cu platforma XQuark. Altfel spus, aplicatia nu înceteaza a rula în momentul în care pro-cesul XQuark si-a întrerupt activitatea si încearca periodic sa recâstige conectivi-tatea.

```
typedef SMessageQueue<NetMessage*> NetMsgPtrQueue;
class MainObject
{
public:
        ///
        MainObject();
        ~MainObject();
bool Connect();
        void Disconnect();
        void DispatchEvents();
bool NetLinkStopped();
        void SendMessage(ClientOrderListMsg *);
        void SendMessage(ClientOrderMsg *);
        void SendMessage(ClientOrderAckListMsg *);
        void SendMessage(ClientOrderAckMsg *);
bool IsAlive() { return alive; }
        void SetAlive (bool setalive) { alive = setalive; }
        void TimeOut (int millsec) { m_msgRxEvent.Wait(millsec); }
private:
        bool alive;
        ClientOrderListMsg *clientOrderListMsg;
        ClientOrderMsg *clientOrderMsg;
        ClientOrderAckListMsg *clientOrderAckListMsg;
        ClientOrderAckMsg *clientOrderAckMsg;
        ///event to be used for inter-thread communication
        SEvent m_msgRxEvent;
        ///queue for Net msgs
        NetMsgPtrQueue  m_netQueue;
        ///
        XQuarkLink *m_pNetLink;
        ///
        void ProcessNetMessage();
};
int main(int argc, char **argv)
{
```

```
MainObject  *g_main;
g_main = new MainObject();
while (1) {
        if (!g_main->IsAlive()) {
                cout << "ask for XQuark connection ........... ";
        if (g_main->Connect())
                        cout << "connected to XQuark" << endl;
                else
                        cerr << "cannot connect to XQuark!" << endl;
}
        while (!g_main->NetLinkStopped()) {
                g_main->DispatchEvents();

                // here should be placed the application main entry procedure
// g_main->Disconnect();
        }
        cout << "have not XQuark connection" << endl;
                g_main->TimeOut(7000); // wait 7 sec. and try to connect again

  // here should be placed the application main entry procedure

}
        delete g_main;
return 0;
}
bool MainObject::Connect()
{
if (m_pNetLink)
Disconnect();
        m_pNetLink = new XQuarkLink(m_msgRxEvent, m_netQueue, false, false);
        if (!m_pNetLink->Start()) {
                alive = false;
                return false;
        } else {
                alive = true;

m_pNetLink->RegisterMessageInterest(MsgNums::FIRST_APPL_MSGNUM,
MsgNums::LAST_APPL_MSGNUM);
                m_pNetLink->BroadcastIsOpen(); // send Open for Bussiness.

                // --- make DispatchEvent thread ---
                return true;
}
}
```

Metoda *MainObject::Connect()*, prezentata mai sus realizeaza stabilirea conexiunii cu platforma XQuark prin apelul functiei *Start()* din clasa *XQuarkLink*. Pentru a ne asigura ca avem întotdeauna numai o singura conexiune deschisa de o aplicatie client catre XQuark clasa *MainObject* tre-buie sa gestioneze corespunzator instata *XQuarkLink*. Daca pointerul *m_pNetLink* nu este *null* aceasta înseamna (respectând cele mentionate mai sus, ne asiguram ca în constructorul obiectului *MainObject* poin-terul *m_pNetLink* este initializat cu *null*) ca înca avem o conexiunea stabilita cu platforma XQuark si ca trebuie sa abando-nam mai întâi conexiunea curenta si apoi sa incercam sa ne reconectam, daca se doreste acest lucru, apelând în prealabil functia *MainObject::Disconnect()* prezen-tata mai jos:

```
void MainObject::Disconnect()
{
 alive = false;
 if (m_pNetLink) {
  m_pNetLink->Stop();
  while (!m_pNetLink->IsStopped()) {
   m_msgRxEvent.Wait(300); // wait until should_stop variable it is set on
  }
  m_pNetLink->Destroy();
  m_msgRxEvent.Wait(1000); // wait for a while, before trying to do anything else
  m_pNetLink = 0;
 }
```

```
}
```

Functia *Disconect()* va apela metoda *Stop()* a clasei *XQuarkLink* care seteaza sub *mutex* variabila *should_stop* (din clasa *SThread*). În acest moment, metoda *Run()* a clasei derivate din *SThread*, verifica daca variabila *should_stop* a fost setata în afara ei si daca da, functia *Run()* returneaza controlul functiei care a apelat-o (firului de executie *SThread::ThreadEntryProc()*).

```
SThread::ThreadEntryProc(void *param)
{
auto_ptr<thread_params> p(static_cast<thread_params*>(param));
        auto_ptr<SThread> thread(p->thread_ptr);

#ifdef __SUNPRO_CC
        (void) set_terminate(abort);
        (void) set_unexpected(abort);
#endif

{
SMutex::AutoLock lock(const_cast<SMutex&>(thread->thr_mutex));
        thread->is_started = true;
        thread->thr_id = SThreadID::CurrentThread();
        thread->SetPriority(thread->Priority());
}
 if (thread->InitInstance()) {
        // Release main thread
        p->sem.Post();

        thread->FinishedInitialize(); // release waiters

        // run the main routine
        thread->Run();

        if (!thread->auto_del) {
                SMutex::AutoLock lock(const_cast<SMutex&>(thread->thr_mutex));
                thread->is_stopped = true;
        }
 } else {
        // don't delete object so that Start() can get at the had_error var
        thread->had_error = thread->is_stopped = true;
        p->sem.Post();
 }
#if defined(HAVE_WIN32_THREADS)
        _endthread();
#elif defined(HAVE_POSIX_THREADS)
        pthread_exit(0);
#elif defined(HAVE_SOLARIS_THREADS)
        thr_exit(0);
#endif
        // not reached anyhow
#ifndef HAVE_WIN32_THREADS
        return 0;
#endif
}
```

De remarcat ca în aceasta functie am folo-sit tipul *auto pointer* care, în contextul par-ticular al verificarii continutului unor vari-abile sub mutex, unde nu este posibila monitorizarea careia dintre aplicatii a generat abandonarea firului de executie, reprezinta o solutie eleganta pentru dealo-carea memorie în momentul în care firul de executie a fost creat. De asemenea, în ace-lasi context al conectivitatii, o alta caracte-ristica importanta pentru oferirea flexibili-tatii necesare aplicatiilor în ceea ce priveste conectarea si deconectarea la si de la procesul XQuark este posibilitatea setarii unei veriabile (*auto_del*) a clasei *SThread* în clasele derivate din acesta, respectiv *XQuarkLinkBase*, *XQuarkLink*, *XQuarkLin kWindows*:

XQuarkLinkBase(bool trace=false, bool auto_del=true);

Aceasta variabila booleana este în mod implicit initializata cu valoarea *true*, care semnifica faptul ca firul de executie se distruge

pe sine însasi, imediat ce termina secventa de procesare pentru care a fost creat, fara a fi necesar apelul functiei *Disconnect()*. Acest aspect este în mod special important când aplicatia client are fluxul principal de prelucrare bazat în mod nemijlocit pe conexiunea cu platforma XQuark si, implicit, pierderea conectivi-tatii aduce aplicatia în situatia de a nu-si mai putea exercita rolul pentru care a fost proiectata, fiind un non-sens sa mai continue sa ruleze în afara conectarii la XQuark (atunci aplicatia îsi termina exe-cutia, odata cu întreruperea conexiunii cu procesul XQuark). Pe de alta parte, exista situatii când, dimpotriva, se doreste ca aplicatia sa-si poata continua fluxul prin-cipal de prelucrare (daca acesta este altul decât numai comunicarea cu procesul XQuark) chiar daca a pierdut conexiunea cu XQuark-ul.

Din prima categorie de aplicatii client (din perspectiva platformei XQuark) poate face parte, de exemplu, un server care furnizea-za clientilor sai preturi în timp real de la bursa. Din a doua categorie de aplicatii poate face parte un client al acestui server de preturi furnizate în timp real, care realizeaza o analiza a acestor preturi gene-rând, de exemplu, rapoarte de corelatie dintre date si care poate sa-si continue acti-vitatea chiar daca nu primeste informatia de la piata furnizata de server, dar încearca periodic sa recâstige conexiunea cu XQu ark-ul pentru a-si putea împrospata pro-priile date. Aceasta din urma aplicatie este capabila sa detecteze faptul ca a pierdut conexiunea cu XQuark-ul si încearca periodic sa se reconecteze. Întorcându-ne la functia *MainObject::Disconnect()*, sec-venta urmatoare

```
while (!m_pNetLink->IsStopped()) {
  m_msgRxEvent.Wait(300); // wait until should_stop variable it is set on
}
m_pNetLink->Destroy();
m_msgRxEvent.Wait(1000); // wait for a while, before trying to do anything else
m_pNetLink = 0;
```

asigura faptul ca firul de executie nu este abandonat înainte ca variabila logica should_stop sa fie în mod corect setata sub *mutex* si pointerul *m_pNetLink* nu este pus pe *null* înainte ca zona de memorie pe care o pointeaza sa fie dealocata corespunzator.

Mergând mai departe, daca în functia *MainObject::Connect()* firul de executie care asigura conectarea la XQuark este creat prin initializarea în constructorul sau a variabilei *auto_del* cu valoarea *false,* se ofera în acest mod libertate deplina aplicatiei în ceea ce priveste controlul conexiunii cu procesul XQuark. Bucla in-terioara din functia *main()* este conditiona-ta de rezultatul returnat de functia *MainObject::NetLinkStopped(),* care setea-za, de asemenea, variabila *alive*, ce indica, la nivelul aplicatiei client daca conexiunea cu reteaua sistemului creat de procesele XQuark este viabila sau nu.

```
bool MainObject::NetLinkStopped()
{
  if (m_pNetLink) {
    if (m_pNetLink->IsStopped()) {
      alive = false;
      m_pNetLink->Destroy();
      m_msgRxEvent.Wait(1000); // wait for a while, before trying to do anything else
      m_pNetLink = 0;
  return true;
    }
  } else {
alive = false;  // it's gone
return true;
  }
  return false;
}
```

Aplicatiile XQSender si XQReceiver im-plementeaza strategii diferite în ceea ce priveste conectarea la platforma de comu-nicatie XQuark. Amândoua folosesc o instanta a clasei *MainObject* si ambele aplicatii sunt proiectate sa ruleze indepen-dent

de starea conexiunii cu procesul XQuark (pot fi privite ca aplicatii care nu îsi intrerup executia din perspectiva conectivitatii la XQuark – *never dye applications*).
Totusi, XQSender este o aplicatie Win-dows în care utilizatorul are optiuni dedicate operatiilor de conectare (deconec-tare) la (de la) XQuark.
Pe de alta parte, aplicatia XQReceiver este o aplicatie consola, pentru ca poate rula sub UNIX. Daca nu are conexiune cu plat-forma de comunicatie, va încerca în mod automat (periodic) sa realizeze aceasta conexiune.

Bibliografie

1. David Butenhof, Programming with POSIX Threads - Addison-Wesley, 1997
2. Michi Henning, Steve Vinoski, Advan-ced CORBA Programming with C++ - Addison Wesley Longman, Inc., 1999
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Pat-terns - Elements of Reusable Object-Oriented Software - Addison-Wesley Longman, Inc., 1995
4. Steve Kleiman, Devang Shah, Bart Smaalders, Programming with Threads - Prentice Hall, 1996
5. Bill Lewis, Daniel Berg, Multithreads Programming with Pthreads - Prentice Hall, 1998
6. Thomas J. Mowbray, Raphael C. Malveau, CORBA Design Patterns – John Wiley & Sons, Inc., 1997
7. W. Richard Stevens, UNIX Network Programming - volume 1, Networking APIs: Sockets and XTI, Second Editi-on, Prentice Hall, 1998
8. W. Richard Stevens, Advanced Progra-mming in the UNIX Environment - Addison-Wesley, 1992