UNIVERSITATEA “PETRU MAIOR” DIN TÂRGU-MUREŞ
FACULTATEA DE INGINERIE
SPECIALIZAREA: CALCULATOARE

LUCRARE DE DIPLOMĂ

Autentificarea, autorizarea şi controlul accesului în sisteme distribuite

Coordonator:
Asist. Ing. Genge Béla

Student:
Magyari István Attila

2009

UNIVERSITATEA "PETRU MAIOR" TÂRGU-MUREŞ
FACULTATEA DE INGINERIE
Specializarea CALCULATOARE Viza facultății

TEMĂ PROIECT DE DIPLOMĂ

Conducătorul temei: Candidat (a): **Magyari Attila**
Asist. ing. Genge Bela Anul absolvirii: 2009

a) Tema proiectului de diplomă: ***Autentificarea, autorizarea şi controlul accesului în sisteme distribuite***

b) Problemele principale care vor fi tratate în proiect:

- Introducere în domeniul sistemelor distribuite şi a securității;
- Stabilirea problemelor de securitate cu care se confruntă sistemele distribuite existente;
- Proiectarea componentelor sistemului prin formularea cerințelor de securitate şi evidențierea modulelor principale pentru satisfacerea acestor cerințe;
- Proiectarea protocoalelor de securitate implicate în stabilirea unei comunicări sigure între componentele sistemului şi verificarea independenței acestora;
- Proiectarea unei interfețe API pentru implementarea sistemului
- Implementarea sistemului propus prin utilizarea arhitecturilor Mozilla.

c) Desene obligatorii:

- Arhitectura sistemului propus;
- Diagrame de secvență pentru autentificarea aplicațiilor client;
- Protocoale de securitate proiectate.

d) Softuri obligatorii – Visual C++, NSPR, Qt.

Bibliografie recomandată:

1. Butler Lampson, Martín Abadi, Michael Burrows, Edward Wobber, Authentication in Distributed Systems: Theory and Practice, ACM Trans. Computer Systems 10, 1992.
2. Joshua D. Guttman, F. Javier Thayer Fabrega, Authentication tests and the structure of bundles, Theoretical Computer Science, Vol. 283, No. 2, pag.333-380, iunie 2002.
3. W. Stallings, Cryptography and Network Security, 4th edition, Prentice Hall, 2005.
4. A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.
5. J. D. Guttman, Security Protocol Design via Authentication Tests, 2002.
6. J. D. Guttman, Security goals: Packet trajectories and strand spaces, In R. Gorrieri and R. Focardi, editors, Foundations of Security Analysis and Design, volume 2171 of LNCS, Springer Verlag, 2001.
7. E. Gerck, Overview of Certification Systems: X.509, PKIX, CA, PGP & SKIP, 2000.
8. C. Szyperski, Component Software Beyond Object Oriented Programming, 2nd edition, Addison Wesley, pag. 3 – 47, 2002.

Termene obligatorii de consultații: - săptămânal

Locul practicii: laboratoarele specifice din facultate

Primit la data de: 15. 03. 2008
Termen de predare: 15. 06. 2009

Semnătura şefului de catedră Semnătura conducătorului

Semnătura candidatului

Cuprins

Capitolul 1 – Introducere

Importanța aspectelor de securitate in rețele de calculatoare a crescut odată cu extinderea prelucrărilor electronice de date şi a transmiterii acestora prin intermediul rețelelor. În cele mai multe sisteme distribuite şi rețele de calculatoare, protecția resurselor se realizează prin login direct folosind parole, cu transmiterea în clar a acestora. Aceasta autentificare are mai multe inconveniențe, din care cea mai problematică ar fi expunerea parolei la captură pasivă [1]. În zilele noastre vizitam zeci de website-uri, pe fiecare trebuie să ne cream un cont de utilizator, completând cu atenție formulare de înregistrare, să reținem parole, şi să le introducem de fiecare dată când vrem să accesăm conținutul. Acest lucru devine plictisitor şi repulsiv, şi în acelaşi timp informațiile noastre confidențiale sunt expuse la potențiale atacuri de fiecare dată. Mulți dintre noi refolosim acelaşi parole în mai multe locuri, crescând astfel riscul unui compromis neplăcut. În această lucrare propun un mecanism „Single Sign-On”, bazat pe certificate generate la solicitarea de către aplicația client. Sistemele Single Sign-On (SSO) permit ca utilizatorii să-şi introducă datele personale doar o singură dată, şi să primească acces la multiple resurse în sistem [2].

Dezvoltarea rapidă a internetului şi faptul că acesta este accesibil unui număr foarte mare de utilizatori a determinat un volum mare de date private aflat în circulație punându-se astfel problema securității datelor şi a limitării accesului la date. Securitatea informației înseamnă protejarea informațiilor şi a sistemelor informatice împotriva acțiunilor neautorizate în ceea ce priveşte accesul, folosirea, copierea, modificarea sau distrugerea datelor. Transmisia de date dintre client şi server se va realiza folosind canale de comunicații securizate, cu ajutorul protocoalelor de securitate şi a criptografiei. Pentru a permite o conectare rapidă la multiple resurse, nu am folosit protocoalele complexe existente, ca SSL [3], sau versiunea mai nouă, TLS [4], ci am construit un set de protocoale noi, pe baza testelor de autentificare descrise de Guttman [5], [6]. Aceste protocoale au fost implementate folosind librăriile de securitate OpenSSL [7], oferind performanță şi confidențialitate.

Pentru a pune în practică modelul propus, am dezvoltat un *middleware* care implementează aceste protocoale de securitate şi mecanismul single sign-on. Majoritatea mecanismelor SSO existente pe care le-am analizat au un dezavantaj major: sunt implementate să funcționeze doar pe o singură platformă, cum ar fi Active Directory [8] pentru Microsoft Windows, sau eDirectory [9] pentru sistemele Unix. Un alt dezavantaj frecvent întâlnit este arhitectura centralizată, de exemplu LDAP [10], adică serverele de

autentificare trebuie să fie conectați la un server central pentru a autentifica utilizatorii. Noutatea modelului propus de mine constă în folosirea componentelor XPCOM [11], oferit de platforma Mozilla pentru a încapsula nivelul de transport. Astfel middleware-ul nu va fi restricționat pe o singură platformă şi mecanismul single sign-on va fi disponibil pe orice platformă care suportă Mozilla. Pentru a nu depinde de un server central la autentificare, vom implementa componenta în fiecare server din rețea, adăugarea cărora se va putea face cu uşurință în caz de nevoie. Clienții se vor putea deplasa liber la aceste servere, distribuind sarcina produsă de procesul de autentificare pe toate nodurile din rețea.

1.1 Structura documentului

Primul capitol prezintă câteva noțiuni introductive referitoare la problema dată, scopul lucrării şi setul de cerințe.

Capitolul 2 prezintă câteva noțiuni despre criptografie, tipurile acestei, diferențele dintre cea simetrică şi asimetrică, şi folosirea acestora. Se continuă cu descrierea protocoalelor de securitate, principii de proiectare, testele de autentificare şi independența protocoalelor. Urmează o prezentare a infrastructurii cheilor publice, şi descrierea certificatelor X.509. Capitolul continuă cu un subcapitol despre mecanismul Single Sign-On, şi principiile ingineriei programării bazate pe componente. În încheierea acestui capitol este o prezentare generală a platformei Mozilla, arhitectura acestei platforme şi câteva noțiuni despre tehnologiile folosite şi cum pot fi acestea utilizate în dezvoltarea de aplicații.

Capitolul 3 este dedicat proiectării sistemului şi începe cu formularea cerințelor şi prezentarea arhitecturii generale a sistemului. Se identifică componentele sistemului şi se prezintă rolul acestora. Urmează proiectarea fiecărei componente cu ajutorul şi sunt prezentate interfețele acestora. Urmează proiectarea şi verificarea protocoalelor de securitate.

Capitolul 4 conține implementarea sistemului şi descriere fiecare componentă cu ajutorul diagramelor de clase corespunzătoare şi a diagramelor de stare ale principalelor clase.

În final sunt prezentate şi explicate măsurări a performanței sistemului, şi concluziile referitoare la sistemul prezentat.

Capitolul 2 – Aspecte teoretice

2.1 Criptografia

Criptografia reprezintă o ramură a matematicii care se ocupă cu securizarea informației precum şi cu autentificarea şi restricționarea accesului într-un sistem informatic. În realizarea acestora se utilizează atât metode matematice (profitând, de exemplu, de dificultatea factorizării numerelor foarte mari), cât şi metode de criptarea cuantică. Criptologia este considerată ca fiind cu adevărat o ştiință de foarte puțin timp. Aceasta cuprinde atât criptografia - scrierea secretizată - cât şi criptanaliza. Criptarea se împarte în doua mari categorii: criptarea simetrică si cea asimetrică.

Înainte de epoca modernă, criptografia se ocupa doar cu asigurarea confidențialității mesajelor (criptare) - conversia de mesaje dintr-o formă comprehensibilă într-una incomprehensibilă, şi inversul acestui proces, pentru a face mesajul imposibil de înțeles pentru cei care interceptează mesajul şi nu au cunoştințe secrete adiționale (anume cheia necesară pentru decriptarea mesajului). În ultimele decenii, domeniul s-a extins dincolo de problemele de confidențialitate şi include, printre altele, şi tehnici de verificare a integrității mesajului, autentificare a emițătorului şi receptorului, semnătură electronică, calcule securizate.

Criptografia, folosită intr-un protocol de securitate, doreşte să asigure următoarele deziderate fundamentale pentru securitatea informației: confidențialitate, integritatea datelor, autenticitatea şi ne-repudierea.

2.1.1 Criptografia simetrică

Criptografia cu chei simetrice se referă la metode de criptare în care atât trimițătorul cât şi receptorul folosesc aceeaşi cheie (sau, mai rar, în care cheile sunt diferite, dar într-o relație ce la face uşor calculabile una din cealaltă). Acest tip de criptare a fost singurul cunoscut publicului larg până în 1976. Problema fundamentală a utilizării criptografiei în rețele este aceea a găsirii unor modalități de distribuție sigură şi periodică a cheilor criptografice, fiind necesar ca acestea să fie schimbate cât mai des. Uzual, se foloseşte un protocol de schimbare de chei între participanți, sau criptografia cu chei publice. Fiindcă

securitatea criptării simetrice depinde mult de protecția cheii criptografice, administrarea acestora este un factor esențial şi se referă la:

- ***generarea cheilor***, adică mijloacele (pseudo)aleatoare de creare a succesiunii de octeți (biți) ai cheii
- ***distribuția cheilor***, adică modul în care se transmit şi se fac cunoscute cheile tuturor utilizatorilor cu drept de acces la informațiile criptate
- ***memorarea cheilor***, adică stocarea lor sigură pe un suport magnetic sau pe un card, de obicei criptate sub o altă cheie de cifrare a cheilor, numită şi *cheie master*.

Algoritmii de criptare cu chei simetrice (cele mai populare includ: Twofish, Serpent, AES (Rijndael), Blowfish, CAST5, RC4, TDES, IDEA) se pot împărți în două categorii: cifru pe blocuri şi cifru pe flux. Cel mai celebru cifru simetric pe blocuri, DES (Data Encryption Standard) are deja peste 20 de ani. El este primul standard dedicat protecției criptografice a datelor de calculator. Progresele tehnologice au impus înlocuirea DES-ului care a devenit vulnerabil. S-a demonstrat de curând că, folosind o maşină paralelă complexă, se poate găsi, într-un timp de aproximativ 60 de ore, o cheie de 56 de biți cu care a fost criptat un bloc de text clar. Din acest motiv, în 2000, organizația guvernamentală americană NIST (National Institute of Standards and Technology) a selectat algoritmul Rijndael (AES) [12], dezvoltat de doi criptografi belgieni, Joan Daemen şi Vincent Rijmen, să fie noul standard în criptografie simetrică.

2.1.2 Criptografia asimetrică

Criptografia asimetrică, sau criptografie cu chei publice, este un tip de criptografie în care utilizatorul are o pereche de chei, una publică şi una privată, dependente una de cealaltă, dar aproape imposibil de calculat una din ele dacă se cunoştea cealaltă. Astfel, una dintre chei se poate face publică şi stocată în domeniul public iar cealaltă va fi cheia privată, cunoscută numai de către posesor. Folosind cheia publică se poate cripta un mesaj care nu va putea fi decriptat decât cu cheia pereche, cea privată. O analogie foarte potrivită pentru proces este folosirea cutiei poştale. Oricine poate pune în cutia poştală a cuiva un plic, dar la plic nu are acces decât posesorul cheii de la cutia poştală. Dacă mesajul a fost criptat cu cheia privată, se poate decripta de oricine cu cheia publică. Acesta se numeşte semnătură digitală, pentru că se cunoaşte destinatarul mesajului (având în posesie cheia secretă pentru a genera mesajul), şi se poate dovedi că mesajul este nemodificat, precum arată şi Fig. 1. O analogie pentru

semnăturile digitale ar fi sigilarea unui plic folosind un sigiliu personal. Plicul poate fi deschis de oricine, dar sigiliul personal este cel care verifică autenticitatea plicului. Matematic, cele două chei sunt legate, însă practic nu se pot deriva una din cealaltă. Avantajul evident constă în faptul că cheia secretă este cunoscută doar de o singură entitate, şi nu trebuie trimisă niciodată, fiind astfel aproape imposibil de atacat cu succes, în cazul în care este folosit corect.

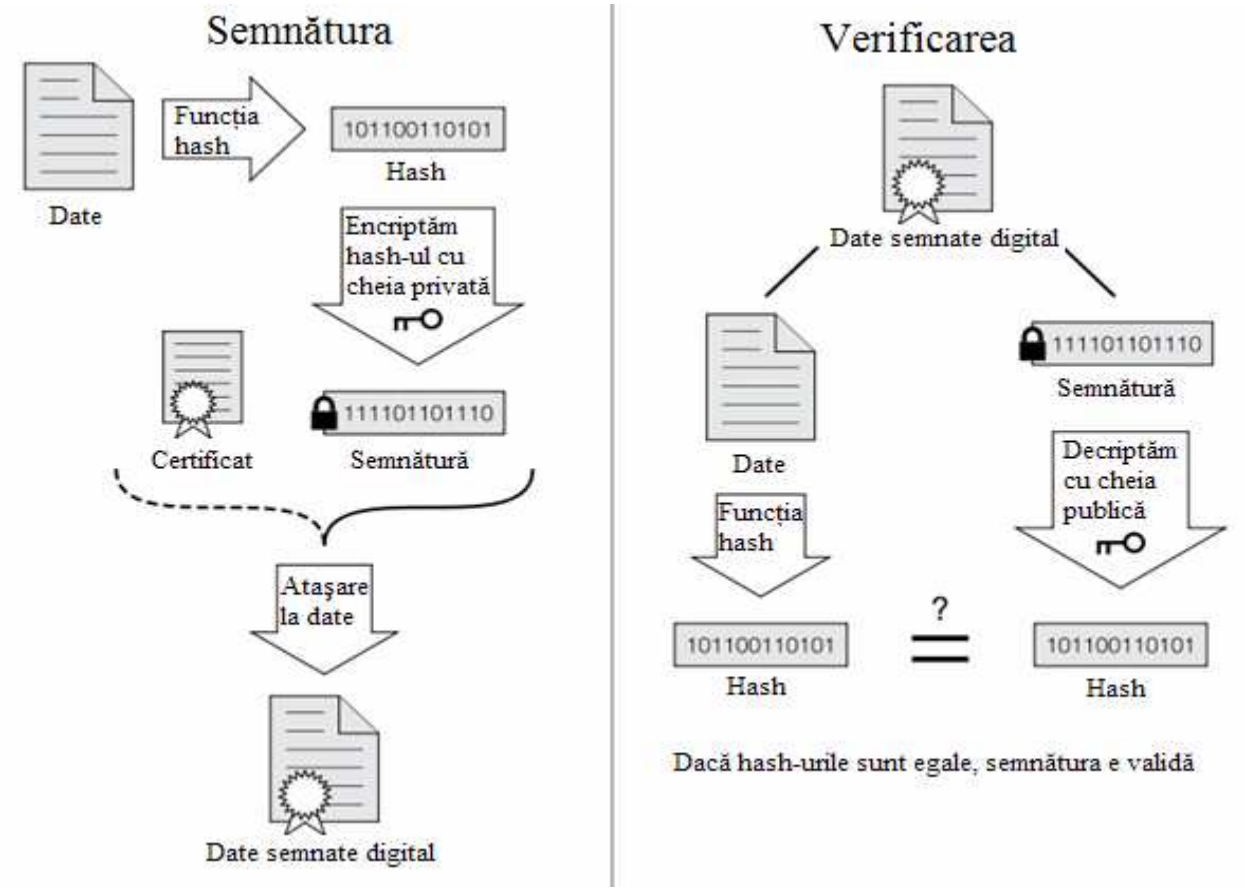

Fig. 1 – Modul de funcționare a semnăturii digitale

O problemă pentru criptografia asimetrică este dovedirea autenticității cheii publice, trebuie dovedit că nu a fost înlocuită de o a treia persoană malițioasă. O abordare comună este folosirea unei infrastructuri pentru chei publice (PKI – Public Key Infrastructure), în interiorul căreia, una sau multe „third-parties", cunoscute sub numele de autorități de certificare (CA – Certification Autority), certifică posesorul perechii de chei. O altă abordare, folosită de PGP (Prety Good Privacy), un program care garantează securitate criptografică şi autentificare, este metoda „web of trust" pentru a asigura autenticitatea perechii de chei, o metodă descentralizată prin care orice utilizator, prin intermediul unui certificat de identitate, poate garanta autenticitatea.

Criptosistemul RSA [13] este unul dintre cei mai cunoscuți algoritmi criptografici cu chei publice, şi este de asemenea primul algoritm utilizat atât pentru criptare, cât şi pentru semnătura electronică. Algoritmul a fost dezvoltat în 1977 şi publicat în 1978 de Ron Rivest, Adi Shamir şi Leonard Adleman la MIT şi îşi trage numele de la inițialele numelor celor trei autori. Algoritmul implică trei stări: generarea cheilor, criptarea şi decriptarea [14]. În acest moment, cea mai eficientă metodă de a realiza aceasta este descompunerea în factori primi a lui n, iar acesta înseamnă un nivel de dificultate similar cu problema factorizării. Cel mai mare număr factorizat vreodată avea 663 biți, iar cheile folosite de obicei au o lungime de 1024 sau 2048 biți, ceea ce demonstrează siguranța acestui algoritm. Perechea de chei se generează după următorii paşi:

1. Se generează două numere prime, de preferat mari, p şi q;
2. Se calculează $n = pq$ şi $\emptyset = (p-1)(q-1)$
3. Se alege un întreg aleator e, 1 < e < φ astfel încât cmmdc(e, φ) = 1. Perechea (n, e) este cheia publică.
4. Folosind algoritmul lui Euclid extins, se calculează întregul d, unicul cu proprietatea că $de \equiv 1 \ mod \ \emptyset$ constituie cheia secretă.

În general, deoarece se bazează pe o operație destul de costisitoare din punct de vedere al timpului de calcul şi al resurselor folosite, şi anume exponențierea modulo *n*, viteza RSA este mult mai mică decât a algoritmilor de criptare cu cheie secretă [15]. De aceea, în sistemele de comunicație în timp real, în care viteza de criptare şi decriptare este esențială (cum ar fi, de exemplu, aplicațiile de streaming video sau audio securizate), se foloseşte un sistem de criptare hibridă, adică RSA se foloseşte doar la începutul comunicației, pentru a transmite cheia secretă de comunicație, care ulterior este folosită într-un algoritm cu cheie secretă, cum ar fi 3DES sau AES.

2.1.3 Funcții hash

Se poate numi o funcție *hash* (message digest) orice procedură bine definită sau o funcție matematică care converteşte o cantitate mare de date (de dimensiuni variabile) într-un alt set de date, de obicei mult mai redus ca dimensiune, şi de lungime fixă. Funcția este ireversibilă, iar aceleaşi date de intrare va avea ca rezultat aceleaşi date de ieşire daca se aplică aceeaşi funcție hash pe ele. Pentru hash-urile bune, coliziunile (două texte clare diferite

care produc acelaşi hash) sunt extrem de dificil de găsit. Funcțiile hash sunt folosite în mai multe domenii, de exemplu pentru a accelera căutările în tabele, sau baze de date mari, ca sume de control, coduri de corectoare de erori, sau în criptografie, componente în schemele de semnătură digitală. Cele mai des folosite funcții hash in criptografie sunt MD5 (Message Digest Algorithm 5) şi SHA1 (Secure Hash Algorithm), cea din urmă fiind mai sigură. Puteți vedea rezultatul celei două funcții menționate aplicate pentru textul „Hello World" în Tabelul 1.

Tabel 1 – Rezultatele funcțiilor hash MD5 şi SHA1

MD5	2ef7bde608ce5404e97d5f042f95f89f1c232871
SHA1	ed076287532e86365e841e92bfc50d8c

2.2 Protocoale de securitate

Protocoalele de securitate sunt o secvență de operațiuni ce folosesc transformări criptografice şi al căror principal scop este să asigure o serie de proprietăți de securitate, cum ar fi: autentificarea, integritatea, secretizarea, ne-repudierea. Proiectarea unui protocol corect este dificil, pentru că nu se pot verifica cu uşurință. Multe dintre protocoale de securitate propuse s-au dovedit mai târziu a avea defecte, de exemplu protocolul cu cheie simetrică Needham-Schroeder a fost dovedit defect de către Burrows, Abadi şi Needham, la 3 ani după publicare. În cazul descoperirii defectelor abia după ce protocoalele au devenit utilizate pe scară largă consecințele pot fi foarte grave, şi acest fapt a determinat multe persoane să fie sceptice referitor la protocoalele de securitate în general. Există câteva tehnici de formale pentru analiza şi verificarea protocoalelor de securitate. Printre acestea se numără: BAN Logic, NRL Analzyer, FDR, Spi calculus, metoda inductivă, teoria spațiilor Strand. Corectitudinea protocoalelor poate fi verificată parțial cu aceste metode.

O altă problemă în ceea ce priveşte proiectarea şi verificarea protocoalelor de securitate este dificultatea de a specifica criteriile de corectitudine sau obiectivele protocolului. Multe protocoale sunt proiectate greşit pentru că proiectanții lor nu ştiu exact ce obiective vor să atingă. Specificând cerințele protocoalelor se clarifică problema pe care trebuie să o rezolve proiectanții şi realizează o legătură între criteriile informale legate de corectitudine şi descrierea formală a protocolului. Totuşi încă nu există o definire precisă a proprietăților necesare unui protocol, la fel cum nu există o noțiune exactă a ceea ce înseamnă autentificarea.

Proiectarea protocoalelor de securitate este un proces repetat de proiectare şi verificare. Datorită anilor de cercetare în verificarea protocoalelor, analiza şi verificarea a devenit mult mai uşoară, cu toate acestea procesul depinde de reguli şi experiența proiectantului. Încă nu există o metodologie de proiectare.

2.2.1 Principii de proiectare

Protocoalele de securitate, cum sunt cele folosite pentru autentificare, sunt predispuse la greşeli de proiectare de multe tipuri. De-a lungul timpului, au fost propuse multe formalisme pentru a investiga şi analiza protocoalele cu scopul de a vedea dacă au defecte. Deşi uneori aceste formalisme s-au dovedit utile, acestea nu sugerează reguli de proiectare, nu sunt în mod direct utile pentru prevenirea defectelor.

În [16] sunt prezentate principiile proiectării protocoalelor de securitate. Aceste principii nu sunt necesare pentru a asigura corectitudinea, dar nu sunt nici suficiente. Ele sunt utile pentru că folosirea lor simplifică protocoalele şi previne repetarea unor confuzii şi greşeli deja descoperite şi publicate. Principiile sunt formulate pentru a evita anumite aspecte ale protocoalelor care sunt greu de analizat. Dacă aceste aspecte sunt evitate, devine mai puțin necesar să se recurgă la folosirea metodelor formale. Principiile sunt indicații informale şi sunt utile independent de orice logică.

Un protocol este un set de reguli sau convenții care definesc un schimb de mesaje între doi sau mai mulți parteneri. Aceşti parteneri sunt persoane, procese sau calculatoare, care vor fi numiți în continuare interlocutori. Într-un protocol de securitate toate sau o parte din mesaje sunt criptate în întregime sau parțial. În acest context criptarea şi decriptarea sunt definite ca fiind transformări dependente de cheie ale unor mesaje care pot fi inversate doar prin folosirea unei chei stabilite. Aceste chei pentru criptare şi decriptare sunt aceleaşi sau pot fi diferite în funcție de algoritmul de criptare folosit.

În continuare sunt prezentate aceste principii pentru proiectarea protocoalelor de securitate. Sunt prezentate două principii de bază, primul fiind preocupat de conținutul mesajului, iar al 2-lea este preocupat de circumstanțele în care un mesaj se comportă corect:

1. Fiecare mesaj ar trebui să specifice ce înseamnă, ce conține, interpretarea lui ar trebui să depindă doar de conținutul său. Ar trebui să se poată scrie o propoziție care să descrie conținutul mesajului.

2. Condițiile în care un mesaj se comportă conform aşteptărilor ar trebui specificate clar pentru ca oricine modifică un protocol să vadă dacă acestea sunt acceptabile sau nu.

Pentru o bună înțelegere a primului principiu luăm un exemplu: un server de autentificare *S* trimite un mesaj a cărui sens poate fi exprimat astfel: „După recepționarea unei secvențe de biți *P*, *S* trimite lui *A* o cheie de sesiune *K* pentru a fi folosită în conversația cu *B*”. Toate elementele trebuie reprezentate în mesaj pentru ca înțelesul său să poată fi refăcut de destinatar fără a fi nevoie de informații din context. Pentru exemplu dat, dacă *P*, *S*, *A*, *B* sau *K* sunt omise din mesaj şi se doreşte obținerea lor din context, există posibilitatea ca un mesaj să poată fi folosit în locul altui mesaj, cu scopul de a înşela.

Principiul doi menționează că nu este suficient ca mesajul să fie înțeles pentru a se comporta corect, trebuie să îndeplinească o varietate de alte condiții. Aceste condiții de cele mai multe ori constau în ce poate fi privit informal ca o declarație de încredere. Declarațiile de încredere nu pot fi greşit concepute, doar pot fi luate în considerare nepotrivit. De exemplu, dacă cineva consideră că alegerea cheilor de sesiune ar trebui făcută de un server de încredere şi nu de un participant la sesiune, atunci el nu va dori să folosească un protocol cum este Wide-mouthed-frog (protocol care distribuie o cheie de sesiune generata de unul din interlocutori prin intermediul unui server).

Pornind de la aceste doua principii au fost formulate alte principii mai specifice.

Principiul 3 este un caz particular al principiului 1: dacă identitatea unui interlocutor este esențiala pentru sensul mesajului, este prudent să se menționeze numele său explicit în mesaj. Numele relevante pentru un mesaj pot fi uneori deduse din alte date şi din ce cheie de criptare s-a folosit. Când această informație nu poate fi dedusă, omiterea ei din mesaj poate avea consecințe grave.

În continuare sunt prezentate notațiile folosite în exprimarea celorlalte protocoale:

K – cheia de criptare publică în criptarea asimetrică, şi cheia de criptare/decriptare în criptarea simetrică

K^{-1} - cheia privată în criptarea asimetrică

A, B, C, S – interlocutori (*S* fiind serverul)

$\{X\}_K$ - mesajul X este criptat cu cheia *K* şi oricine cunoaşte inversa lui *K* (este *K* în criptarea simetrică şi K^{-1} în cea asimetrică) poate decripta mesajul, iar dacă K este secret putem vedea $\{X\}_K$ ca un mesaj semnat digital.

Criptarea poate fi folosită pentru diferite scopuri:

- pentru asigurarea confidențialității. În acest caz când un interlocutor cunoaşte K^{-1} şi vede $\{X\}_K$ ştie că acest mesaj este pentru cineva care cunoaşte K^{-1}; şi poate va deduce că îi este adresat cu ajutorul informațiilor adiționale din mesaj.

- este uneori folosită pentru garantarea autenticității. În acest caz este de presupus că doar emițătorul corect cunoaşte cheia secretă pentru a cripta mesajul. Criptarea contribuie clar la sensul general al mesajului.
- contribuie la unirea părților de mesaje: recepționarea $\{X,Y\}_K$ nu este întotdeauna acelaşi lucru cu recepționarea $\{X\}_K$ şi $\{Y\}_K$. Dacă criptarea este folosită doar pentru unirea părților de mesaje este suficientă semnătura digitala. Modul de unire a mesajelor este dependent de protocol şi este de multe ori subtil.
- generarea de numere aleatoare cu ajutorul funcțiilor „one-way".

Principiul 4 este legat de folosirea criptării: trebuie să se specifice clar de ce se foloseşte criptarea. Criptarea nu este sinonimă cu securitatea, şi folosirea ei improprie poate conduce la erori. Criptarea nu este ieftină, şi dacă nu se ştie de ce este folosită poate conduce la redundanță.

Principiul 5 este legat de semnarea datelor criptate: când un interlocutor semnează un material care a fost deja criptat nu trebuie presupus că acesta cunoaşte conținutul mesajului. Pe de altă parte, este corect să presupui că interlocutorul care semnează un mesaj şi apoi îl criptează pentru secretizare cunoaşte conținutul mesajului.

O parte importantă în sensul mesajului este formată din informațiile temporale. O precondiție comună pentru a prelucra un mesaj este că există un motiv care te face să crezi că mesajul este recent, şi nu un răspuns al unui mesaj mai vechi. Această noutate a mesajului trebuie să fie indusă de o componentă a mesajului. Această componentă trebuie să fie legată de restul mesajului pentru a nu putea fi ataşată unui mesaj mai vechi.

Principiul 6: Scopul folosirii nonce-urilor (număr generat aleatoriu) trebuie bine precizat, fie pentru asociere, fie pentru succesiunea temporală. Ceea ce poate asigura succesiunea temporală poate afecta asigurarea asocierii, şi poate asocierea este mai bine stabilită prin alte mijloace.

Principiul 7: valorile (nonce-urile) predictibile (cum este valoarea unui contor) pot fi folosite pentru garantarea noutății în cadrul unui protocol întrebare-răspuns („challenge-response"), dar pentru ca această cantitate predictibilă să fie eficientă trebuie protejată pentru ca un intrus să nu poată simula o întrebare şi mai târziu să trimită un răspuns.

Principiul 8: Dacă timestamps (marca de timp) sunt folosite pentru a garanta noutatea prin referire la timpul absolut, atunci diferența dintre ora locală a diferitelor maşini trebuie să fie mult mai mică decât durata de viață permisă a unui mesaj pentru a fi considerat valid.

Principiul 9: o cheie se poate să fi fost folosită recent, pentru criptarea unui nonce de exemplu, şi totuşi să fie destul de veche şi posibil compromisă. Folosirea recentă nu face ca cheia să pară mai sigură decât nefolosirea ei recent.

Principiul 10: dacă o codare este folosită pentru a prezenta sensul unui mesaj, atunci ar trebui să fie posibil să spui ce codare este folosită. În cazurile comune unde codarea este dependentă de protocol, ar trebui să fie posibil să se deducă că mesajul aparține acelui protocol, şi mai exact aparține unei rulări particulare a protocolului, şi să se afle numărul mesajului în acest protocol.

Principiul 11: proiectantul protocolului ar trebui să ştie care sunt relațiile de încredere de care depinde protocolul său şi de ce sunt necesare aceste dependențe. Motivele pentru care o relație particulară de încredere să fie acceptabilă ar trebui explicit bazate pe judecată şi politici decât pe logică.

2.2.2 Proiectare via teste de autentificare

Proiectarea protocoalelor de securitate, descrisă în [17] urmăreşte următoarele obiective, între doi participanți P şi Q:

Confidențialitate: se defineşte ca fiind protecția datelor trimise în fata persoanelor neautorizate

Autenticitatea I: două entități aflate într-un schimb de mesaje să se poată identifica una pe cealaltă. În prima fază, la inițierea conexiunii, acest serviciu asigură ca cele două entități sunt autentice. În al doilea rând, autenticitatea presupune ca transferul de date între cele două entități să nu fie interferat astfel încât o a treia entitate să se legitimeze ca fiind una din ele. Fiecare participant P trebuie să aibă garanție că fiecare partener Q a primit şi a acceptat datele lui P.

Non-repudierea: previne situația în care o entitate refuza să recunoască acțiuni anterioare. Când un mesaj este trimis, destinatarul poate demonstra ca mesajul primit este cel trimis de emițător. Fiecare participant P trebuie să-şi dovedească autenticitatea I, garantat de o terță parte.

Autenticitatea II: Fiecare participant Q trebuie să aibă garanție ca un mesaj având ca expeditor P, a fost creat într-adevăr de partenerul P, într-o rulare recentă a acestui protocol.

Să presupunem o entitate într-un protocol criptografic ca fiind creatorul şi transmițătorul unui mesaj care conține o valoare nouă *v,* iar mai târziu primind un mesaj

conținând valoarea *v*, dar într-un alt context criptografic. De aici poate conclude că o altă entitate, având în posesie cheia relevantă *K* a recepționat şi a transformat mesajul care conținea valoarea *v*. Dacă cheia *K* este secretă, această entitate nu poate fi penetratorul, deci trebuie să fie un participant reglementar. Testele de autentificare [19] oferă suficiente condiții ca să putem dovedi că aceste schimbări a formei criptografice sunt acțiunile unui participant regular. Putem deosebi două tipuri de teste de autentificare:

- ***Teste de ieşire***: o valoare unică *a* poate fi transmisă numai în formă criptată $\{|\ldots a\ldots|\}_K$, unde cheia K^{-1} este secretă. Dacă mai târziu un mesaj primit conține valoarea *a* în afara contextului $\{|\ldots a\ldots|\}_K$, înseamnă că un participant regular a fost responsabil de transformarea $\{|\ldots a\ldots|\}_K \rightarrow \ldots a\ldots$. Este un test de ieşire pentru că elementul criptat iasă.
- ***Teste de intrare***: dacă, în schimb, *a* este primit într-o formă criptată $\{|\ldots a\ldots|\}_K$ dar nu a fost trimis în acest context, şi cheia *K* este secretă, atunci putem afirma din nou că transformarea nu a fost efectuată de un penetrator. Testul care conține transformarea $\ldots a\ldots \rightarrow \{|\ldots a\ldots|\}_K$ este o un test de intrare pentru că elementul criptat intră.
- Dacă o valoare unică *a* este transmisa intr-o formă criptată $\{|h|\}_K$, şi primită înapoi într-o altă formă criptată $\{|h'|\}_{K'}$, iar cheile K^{-1} şi *K'* sunt secrete, atunci este un test şi de intrare, şi şi de ieşire.

2.2.3 Independența protocoalelor de securitate

Dacă este de aşteptat ca două sau mai multe protocoale de securitate să ruleze în paralel trebuie să se determine dacă un protocol afectează securitatea celorlalte protocoale. Acest lucru se realizează analizând independența protocoalelor implicate. Când mai multe protocoale de securitate rulează combinate pentru intruşii apar noi oportunități pentru a obține mesajele. Această combinare de protocoale s-a dovedit o cauză semnificativă a eşuării protocoalelor şi face mult mai dificilă analiza protocoalelor. A fost dovedită o problemă în aplicarea metodelor formale protocoalelor de securitate.

Un protocol, numit protocol primar, este independent de alte protocoale, numite protocoale secundare, dacă protocolul primar îşi îndeplineşte scopul de securitate fără a depinde dacă un protocol secundar rulează în acelaşi timp (1). Pentru dovedirea independenței protocoalelor au fost folosite spațiile strand. Un strand este o secvență de mesaje transmise şi

recepționate. Protocoalele de securitate sunt modelate cu ajutorul acestor spații strand. În [20] este propusă reprezentarea protocoalelor de securitate folosind un model canonic care permite realizarea unei analize sintactice pentru determinarea independenței protocoalelor. Construirea mesajelor după modelul propus este realizată prin înlocuirea fiecărei componente a mesajului din specificarea obişnuită cu un tip. În modelul canonic propus, mesajele schimbate de participanți sunt reprezentate sub forma unor construcții sintactice, evidențiind structura mesajelor care influențează direct independența protocoalelor. Bazându-se pe cunoştințele fiecărui participant se modelează viziunea fiecărui participant asupra aceluiaşi mesaj. Primul pas este îmbunătățirea specificațiilor obişnuite folosind cunoştințele participanților pentru a se observa clar care sunt componentele mesajului care pot fi validate de participanți. Pornind de la această reprezentare este construit un model canonic, numit model de tipuri, care înlocuieşte componentele mesajelor cu tipul lor corespunzător, reprezentându-se astfel explicit structura mesajului. Această reprezentare permite realizarea analizei sintactice pentru determinarea independenței protocoalelor.

Tipurile componentelor mesajelor sunt:

- K_t : cheie
- r: rol
- n: nonce
- i: index, de exemplu o secvență de numere sau o marcă de timp
- u: tip necunoscut
- m_t : numele protocolului de securitate

2.3 Infrastructura Cheilor Publice (PKI)

Infrastructurile cu chei publice (Public Key Infrastructure / PKI) nu este o tehnologie relativ nouă, ea se bazează pe criptografia asimetrică şi oferă diverse servicii în încercarea de a rezolva diverse probleme de securitate. PKI este o combinație de produse hardware şi software, politici şi proceduri care asigură securitatea de bază necesară astfel încât doi utilizatori, care nu se cunosc sau se află în puncte diferite de pe glob, să poată comunica în siguranță. La baza PKI se află certificatele digitale, un fel de paşapoarte electronice ce mapează semnătura digitală a utilizatorului la cheia publică a acestuia. O infrastructură cu chei publice reprezintă cadrul şi serviciile ce pun la dispoziția utilizatorului metode pentru a genera, distribui, controla, contoriza şi revoca certificate cu chei publice. O structură PKI se

constituie, de obicei, din una sau mai multe autorități de certificare (CA), un container cu certificate, şi documentația ce include politica de certificare. Într-un sens mai larg, se poate spune că PKI integrează certificatele digitale, criptografia cu cheie publică şi noțiunea de autoritate de certificare într-o arhitectură de securitate a rețelei.

2.3.1 Certificatele X.509

Certificatele identifică persoana specificată în certificat şi asociază acelei persoane o anumită pereche de chei publică/privată. Certificatul X.509 [21] a fost folosit prima dată în 1988, şi este un standard ITU-T (International Telecommunication Union) pentru Infrastructura Cheilor Publice (Public Key Infrastructure / PKI). Certificatele de obicei sunt emise de autorități de certificate (certificate authorities / CA), şi conțin următoarele informații: versiunea certificatului, un număr de serie, identificator de algoritm, numele emitentului, perioada de valabilitate a certificatului, numele posesorului, algoritmul cheii publice, cheia publică şi câteva câmpuri opționale. În afară de aceste informații predefinite, certificatele X.509 v3 suportă şi date personalizate, unde voi stoca drepturile de acces pentru fiecare utilizator separat. Certificatele sunt relativ uşor de generat, iar datorită dimensiunii mici, şi fiind stocate in format .PEM (Privacy Enhanced Mail), codificat BASE64, transmisia lor se realizează fără probleme. Un dezavantaj constituie faptul că suportă un singur algoritm de criptare la un moment dat.

Un exemplu de certificat X.509 v3, emis de către Thawte, pentru Google Mail se poate vedea în Fig. 2.

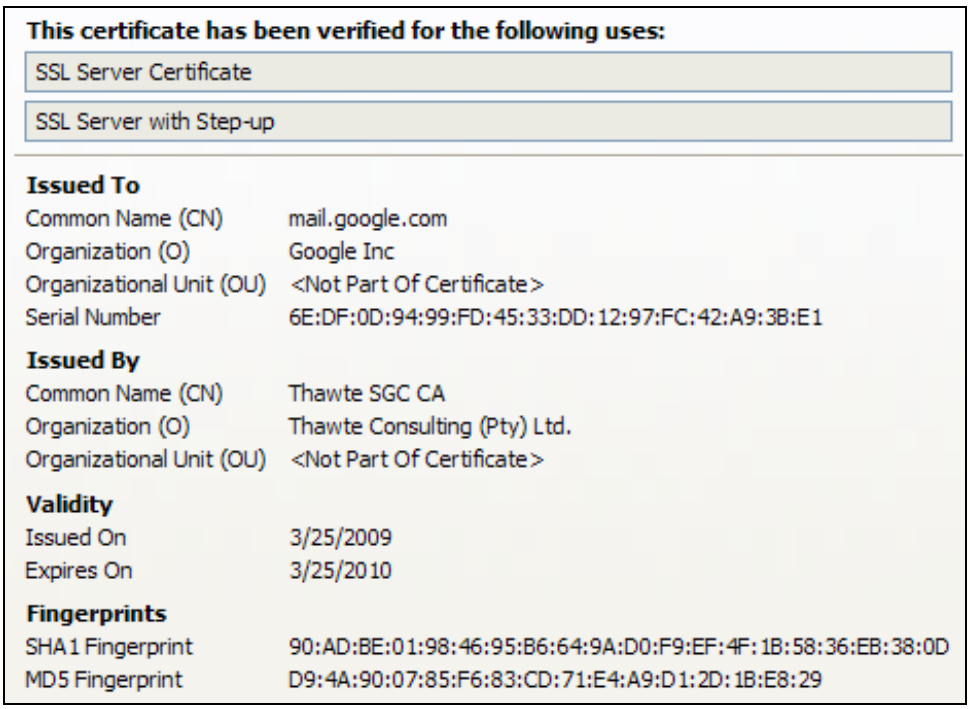

Fig. 2 – Certificat X.509

2.4 Mecanismul Single Sign-On (SSO)

Utilizatorii de rețea folosesc un set de informații pentru identificare, de obicei numele de utilizator şi parola, pentru fiecare furnizor de servicii (Service Provider / SP) la care sunt înregistrați. În cadrul acestei lucrări, un furnizor de servicii este o entitate care oferă anumite resurse sau informații utilizatorilor, de exemplu servicii de web, servicii de mesaje, site-uri web/FTP sau orice fel de format de flux. Numărul acestor furnizori de servicii a crescut enorm, şi a ajuns la un punct unde utilizatorii reţin cu greu informațiile necesare pentru identificare. Cea mai simplă şi răspândită soluție este de a folosi aceeaşi parolă la fiecare furnizor la care sunt înregistrați – un compromis între securitate şi uşurința de folosire. O soluție la această problemă de securitate ar fi implementarea unui mecanism Single Sign-On (SSO). SSO este o proprietate de control de acces a sistemelor software multiple, legate, dar independente. Cu această proprietate, utilizatorul se autentifică doar o singură dată, iar identificarea se face de către sistem, de câte ori va fi nevoie, la fiecare furnizor din rețea. Această autentificare este automată, nu necesită intervenția utilizatorului [22]. Beneficiile unui sistem SSO includ:

- Scade numărul parolelor folosite, îmbunătățind semnificativ securitatea sistemelor
- Se reduce din timpul folosit reintroducând parole pentru aceeaşi identitate
- Securitate la toate nivelele de intrare/ieşire/accesare a sistemelor

Există mai multe modalități de a crea un sistem SSO, de exemplu: protocolul de autentificare Kerberos interoghează utilizatorul pentru datele de identificare, şi emite un tichet (Ticket granting gicket / TGT), care va fi folosit pentru autentificare în continuare. Câteva dezavantaje a acestui sistem includ arhitectura centralizată, care constă in serverul principal care trebuie să fie funcțional pentru a permite autentificarea utilizatorilor. Acest server central stochează datele tuturor utilizatorilor, fiind astfel ținta eventualelor atacuri la adresa sistemului. Kerberos necesită sincronizarea ceasurilor pe diferite servere, tichetele emise de server au o perioadă de valabilitate, de obicei 10 minute, iar dacă ceasul furnizorului nu este sincronizat cu cel al serverului care a emis tichetul, autentificarea eşuează. Un alt sistem SSO foloseşte smart-carduri pentru a stoca informații despre posesor, şi pentru autentificarea utilizatorilor. Acest card, fabricat de obicei din plastic, conține un circuit integrat, capabil de a prelucra anumite date. Smart cardurile sunt răspândite, şi foarte utile în anumite domenii, dar necesitatea acestui hardware îl face greu de utilizat în cazul nostru. Încă câteva sisteme SSO de menționat ar fi cel bazat pe parole de unică folosință (One-time passwords / OTP) sau Integrated Windows Authentication (IWA), introdus cu Windows 2000. Pentru modelul meu am ales să folosesc autentificarea pe baza certificatelor client X.509, descris în secțiunea 2.3.1.

2.5 Ingineria programării bazată pe componente (CBSE)

Componentele software sunt unități binare produse, dezvoltate şi achiziționate independent care interacționează pentru a forma un sistem funcțional [23]. Este esențial să fie dezvoltate independent şi sub formă binară pentru a permite integrarea lor într-un produs indiferent de identitatea producătorilor şi pentru a putea fi integrate robust în sistem. Construirea unor aplicații noi prin combinarea componentelor achiziționate din exterior şi a celor dezvoltate în interior îmbunătățeşte calitatea produsului şi scurtează perioada de dezvoltare, produsul ajungând pe piață mult mai repede. În acelaşi timp adaptarea aplicației la noi cerințe poate fi făcută doar asupra componentelor care necesită acest lucru fără a fi nevoie să se lanseze o noua versiune a aplicației.

2.5.1 Application Programming Interface (API)

O interfață API este un set de funcții, structuri de date, clase de obiecte şi/sau protocoale accesibile din librării, sau oferite de serviciile sistemului de operare, pentru a suporta crearea aplicațiilor. Un program care oferă funcționalitatea descrisă de interfața API este implementarea interfeței API. Interfața API în sine este abstractă, în sensul că specifică instanța dar nu se implică în detalii de implementare. Un API poate fi dependent de un limbaj, dar şi independent, adică poate fi folosit din mai multe limbaje de programare. Standardul POSIX (Portable Operating System Interface for Unix) defineşte un API care permite scrierea unor game mari de funcții de bază în aşa fel încât să fie compatibile cu mai multe sisteme de operare. Câteva API-uri populare includ: ASPI pentru interfața SCSI, DirectX pentru Microsoft Windows, OpenGL cross-platform 3D, Carbon şi Cocoa pentru Macintosh, etc.

Câteva principii de bază în proiectarea unui API: [24] să facă un singur lucru, dar să-l facă bine – funcționalitatea să fie uşor de explicat. Să fie cât se poate de mic, dar să-şi satisfacă cerințele – poți să adăugi mai târziu, dar e mai greu să scoți. Implementarea nu ar trebui să influențeze API-ul. Un aspect important, şi un scop important al fiecărui API este să ascundă informațiile irelevante, clasele şi membrii ar trebui să fie privați, cât posibil. Numele membrilor publici să fie sugestive, dar API-ul să aibă o documentație detaliată. În general să fie uşor de învățat, uşor de folosit chiar şi fără documentație, greu de folosit incorect.

2.6 Platforma Mozilla

Mozilla este o platformă portabilă, gratuită, open-source creată în totalitate cu ajutorul componentelor software. Mozilla a luat naştere în 1998 prin decizia companiei Netscape de a publica sursele browserului Netscape şi declararea lor ca open-source permițând contribuția programatorilor din toată lumea pentru a îmbunătății browserul. Cu timpul platforma Mozilla a devenit dintr-o aplicație browser foarte mare, o platformă mare portabilă pe care un set de aplicații mai mici sunt construite şi rulează. Ea oferă fundația pe care programatorii pot să dezvolte uşor aplicații de nivel înalt fără a mai irosi timp prețios implementând funcționalități de bază, oferite acum de platformă.

Cel mai important avantaj al aplicațiilor dezvoltate pe platforma Mozilla este portabilitatea pe majoritatea sistemelor de operare, programele vor avea aceeaşi structură indiferent dacă vor rula pe Windows, Unix sau Mac. Acest lucru e posibil pentru că Mozilla se comportă ca un nivel de interpretare între aplicație si sistem de operare. Portabilitatea este

realizată cu ajutorul XPFE (Extreme Portability Front End), o tehnologie dezvoltată pentru a economisi timp datorită dezvoltării open-source şi care s-a dovedit o inovație.

XPFE foloseşte o serie de standarde web existente cum ar fi Cascading Style Sheets (CSS) (un limbaj cu ajutorul căruia se creează aspectul grafic al aplicației), XUL (un dialect al limbajului XML utilizat pentru descrierea interfețelor grafice), JavaScript (un limbaj pentru script-uri cu o sintaxă asemănătoare cu cea a limbajului C), RDF (un dialect al XML utilizat pentru salvarea datelor) şi XPCOM (un sistem pentru descoperirea şi administrarea obiectelor care permite JavaScript sau oricărui limbaj pentru script-uri să acceseze librăriile C şi C++). Dezvoltarea de aplicații se realizează folosind aceste tehnologii combinate cu limbaje de programare cum ar fi C, C++, Python şi Interface Definition Language (IDL).

2.6.1 Arhitectura Mozilla

În Fig. 3 este prezentată arhitectura simplificată a platformei Mozilla. Se observă o aşezare pe nivele semi-independente. Legătura dintre sistemul de operare şi platformă se realizează la nivelurile inferioare prin intermediul unei interfețe, accesul la serviciile oferite de acesta făcându-se în trei moduri: printr-un API (Application Programming Interface) portabil numit NSPR (Netscape Portable Runtime), prin cod Java interpretat de JVM (Java Virtual Machine) sau prin implementarea unor plugin-uri. Pentru o flexibilitate şi o deschidere mai mare a sistemului funcționalitatea nivelelor inferioare este înglobată în mai multe componente independente, administrarea lor fiind realizată de un sistem de mediere portabil numit XPCOM (Cross Platform Component Object Model). Componente XPCOM pot fi dezvoltate în C, C++, JavaScript, Python sau alte limbaje pentru care exista legături speciale create. Portabilitatea tehnologiilor folosite în aceste nivele inferioare conduce la o portabilitate crescută a întregii platforme şi implicit a tuturor aplicațiilor dezvoltate pe ea. În momentul de față Mozilla este disponibilă pe principalele trei sisteme de operare existente pe piață: Microsoft Windows, Linux şi Mac OS.

Aplicațiile care vor sa obțină acces la componentele XPCOM folosesc tehnologia XPConnect care asigură accesul limbajelor pentru script-uri (cum este JavaScript) la interfețele componentelor de la nivelele inferioare. Legătura funcționează şi în sens invers permițând interacțiunea unor obiecte implementate în JavaScript cu alte obiecte implementate de obicei în limbaje de programare clasice cum ar fi C sau C++.

Platforma Mozilla, pentru a oferii portabilitate, descrie interfețele componentelor utilizând un limbaj independent de platformă numit XPIDL (Cross Platform Interface

Definition Language, o variantă a limbajului IDL utilizat de sistemul de administrare a obiectelor CORBA). Deasemenea se pot accesa servicii cum este NSS (Netscape Secure Services) un serviciu de securitate care printre alte facilitați oferă posibilatea de utilizare a certificatelor digitale.

Pentru a obține acces la interfața componentelor se foloseşte JavaScript care utilizează tehnologia XPConnect, menționat anterior. Interfața cu utilizatorul este implementată sub forma unor documente scrise în XUL (limbajul derivat din XML specific platformei) sau mult mai cunoscutul limbaj HTML, acestea se pot combina cu CSS (Cascade Sheet Style), eXtensible Binding Language (XBL), DTD (Document Type Definition) pentru a oferi un caracter internațional.

Platforma pune la dispoziția programatorilor un format portabil pentru salvarea datelor numit RDF (Resource Description Framework) care este o particularizare a limbajului XML.

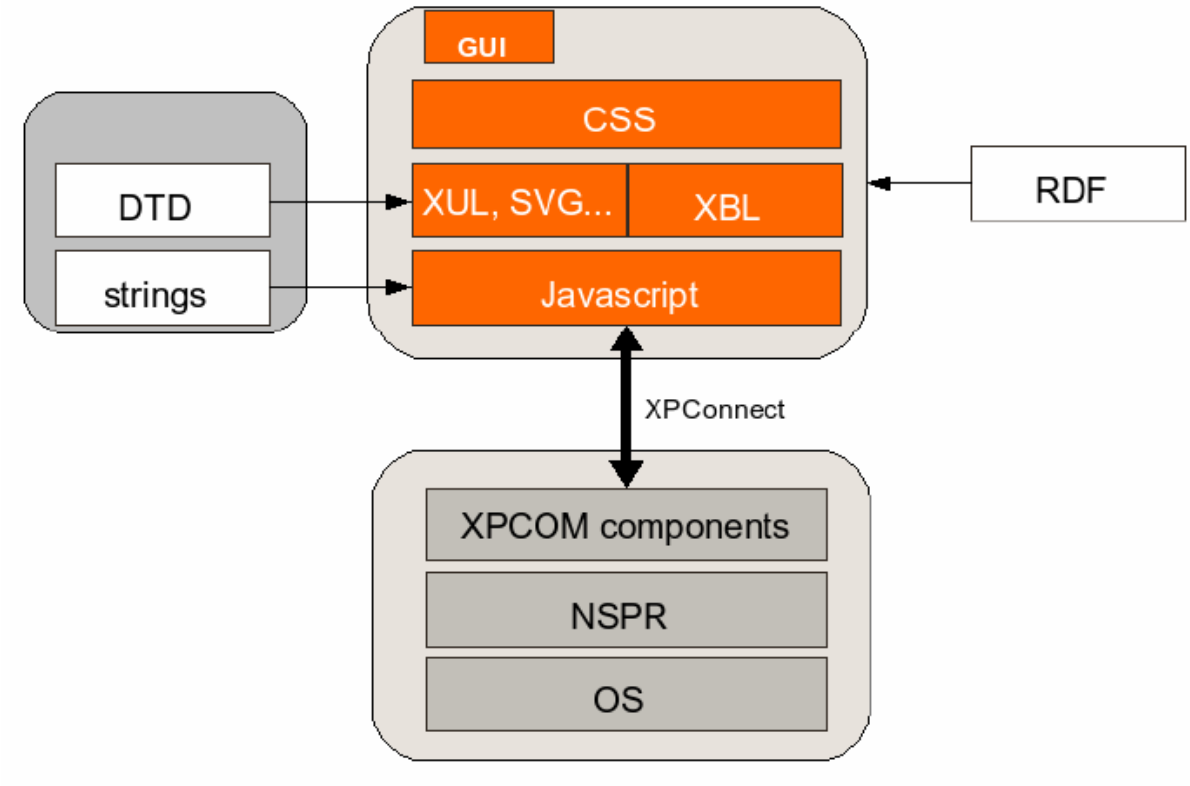

Fig. 3 – Arhitectura simplificată a platformei Mozilla

2.6.2 XPCOM

XPCOM (Cross Platform Component Object Model) reprezintă principala inovație oferită de platforma Mozilla. Aceasta este principalul motiv pentru care Mozilla este cu adevărat o platforma de dezvoltare. XPCOM este tehnologia prin care platforma Mozilla îşi

administrează propriile componente sau pe cele implementate de alte aplicații, deasemenea se ocupa de regăsirea şi instanțierea în timpul rulării a diverselor componente.

Flexibilitatea ridicată a platformei a fost obținută prin organizarea codului sub forma componentelor XPCOM independente accesibile printr-un set de interfețe pe care le implementează. Această abordare facilitează adăugarea de funcționalități noi la momente ulterioare cu eforturi şi costuri minime cu condiția ca toate componentele să implementeze un set de interfețe standard care să poată face posibilă comunicarea dintre ele.

Atât componentele cât şi interfețele sunt identificate printr-un UUID (Universally Unique Identifier, un număr pe 128 de biți) sau printr-un identificator specific platformei numit ContractID, o secvență de caractere într-o formă uşor de reținut de utilizatori. Majoritatea componentelor XPCOM sunt scrise în C sau C++ cu NSPR la bază, dar dacă interfețele lor utilizează exclusiv tipuri de date definite de XPIDL ele pot fi accesate uşor de limbaje ca JavaScript prin intermediul tehnologiei XPConnect. JavaScript este de obicei limbajul care asigură legătura dintre interfața cu utilizatorul şi componentele de la baza platformei.

Capitolul 3 – Proiectarea sistemului

3.1 Cerințe

Se consideră un sistem format din servere (Service Provider / SP) care găzduiesc diferite servicii în mod *request-response* (cerere-răspuns). Pentru a avea acces la resursele disponibile pe aceste servere, clienții trebuie să se autentifice. În Fig. 4 este prezentată arhitectura unui astfel de sistem. Fiecare dintre servere găzduieşte un serviciu de autentificare, nu se bazează pe un punct central fix de autentificare.

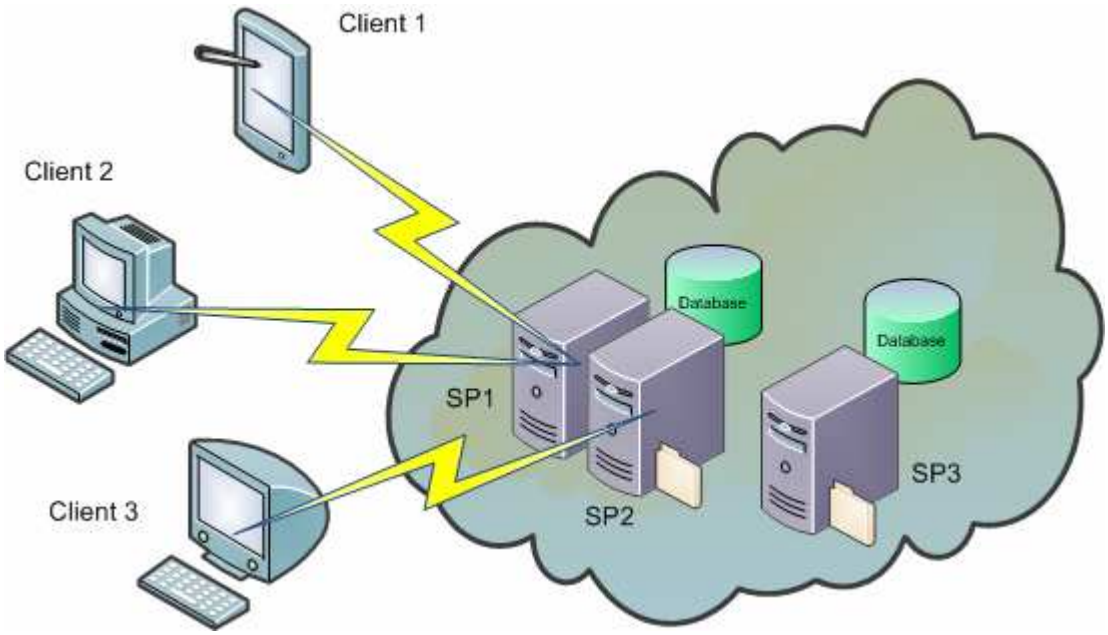

Fig. 4 – Arhitectura sistemului

Autentificarea clienților va fi asigurată de un mecanism Single Sign-On, care se va ocupa de identificarea clienților fără a-i interoga la fiecare conectare pentru informațiile personale. Pentru acesta se va folosi de certificatele generate la comandă de oricare dintre serverele din rețea. Rețeaua poate fi publică, deci informațiile trebuie protejate. Pentru acesta toate comunicațiile şi datele vor fi criptate, şi coordonate de protocoale de securitate sigure.

Componentele trebuie să fie compatibile cu mai multe platforme, şi aceste componente vor fi distribuite intr-un API. Acesta pune la dispoziție o interfață simplă care permite crearea unui astfel de sistem propus cu uşurință, permițând dezvoltatorului să se concentreze pe celelalte aspecte ale proiectului.

3.2 Arhitectura sistemului

3.2.1 Proiectarea mecanismului Single Sign-on

Single Sign-on este o proprietate de control de acces a sistemelor software multiple, legate, dar independente. Cu această proprietate, utilizatorul se autentifică doar o singură dată, iar identificarea se face de către sistem, de câte ori va fi nevoie, la fiecare furnizor din rețea. Aşa cum s-a mai menționat, sistemul propus va fi alcătuit din „noduri”, independente unul de altul, dar fiecare ştie de celălalt. Aceste „noduri” care în cazul nostru vor fi servere, şi mai concret modulul XServer, vor găzdui două tipuri de servicii: unul pentru autentificarea , controlul accesului şi autorizarea clienților, şi unul pentru distribuirea resurselor. Fiecare server are capabilitatea de a efectua aceste procese, nu depind de un alt nod central, astfel adăugarea nodurilor noi se face fără probleme.

Clienții sunt înregistrați la unul dintre aceste servere, fiecare client are un *home server*. Acest home server se diferențiază de celelalte, doar din punctul de vedere al acelui client, care este înregistrat la el, şi doar din punctul de vedere al autentificării, restul facilităților sunt disponibile în aceeaşi măsură. Altfel spus, fiecare server este un home server pentru toți clienții înregistrați la el. Înregistrarea se poate face în orice mod, fie prin e-mail, o pagină web, manual de către administratorul serverului, important e ca după înregistrare clientul să aibă acreditări, de orice tip (de obicei numele de utilizator şi parola) stocate şi protejate de server. Aceste informații personale vor fi păstrate secrete, oricine poate lua rolul unui client dacă are în posesie aceste date.

Un server poate să furnizeze resurse clienților, fie acestea fişiere locale, de orice tip, fie date de flux. În cadrul lucrării o să numim aceste servere, servere de resurse, *resource server*, sau *service provider* (SP). Toate datele trimise către şi de serviciul de resurse vor fi criptate, cu chei generate la autentificare, explicat în următoarele paragrafe. Dacă un server nu rulează serviciul de resurse, atunci va avea rolul doar de a autentifica utilizatori, şi de a trimite lista cu celelalte servere la cererea acestora. Serverele nu sunt limitate doar la aceste două servicii, ci se pot implementa numeroase şi diferite tipuri de servicii, de exemplu serviciu de nume, webmail, FTP, etc. Un model al unui sistem descris se poate vedea în Fig. 4, unde 3 clienți sunt conectați la mai multe servere de resurse ale rețelei.

Pentru a se folosi de serviciile serverelor de resurse, un utilizator trebuie să fie autentificat şi să aibă drepturi de acces suficiente pentru această acțiune. Autentificarea este

un proces prin care o entitate verifică dacă o altă entitate este cine sau ce pretinde a fi. Această entitate poate fi un utilizator, un cod executabil sau un calculator. Verificarea identității unui proces de la distanță este dificilă şi necesită protocoale complexe de securitate bazate pe criptografie. Autentificarea în modelul meu se face pe baza numelui de utilizator şi parolă. Clientul se va conecta la home serverul lui, adică unde a fost înregistrat, iar serverul verifică dacă este salvat vreun utilizator cu acel nume la el. Dacă numele clientului se regăseşte în baza de date la server, se începe protocolul de autentificare care va asigura transmiterea datelor clientului în siguranță. (pasul 1 din Fig. 5) Protocolul se foloseşte de infrastructura cheilor publice, şi va cripta mesajul clientului cu cheia publică a serverului. Dacă vreun client nu are cheia publică a unui server la care vrea să se conecteze, sau certificatul nu este valid, înainte de conectare va cere certificatul clientului de la serverul respectiv, care conține cheia publică.

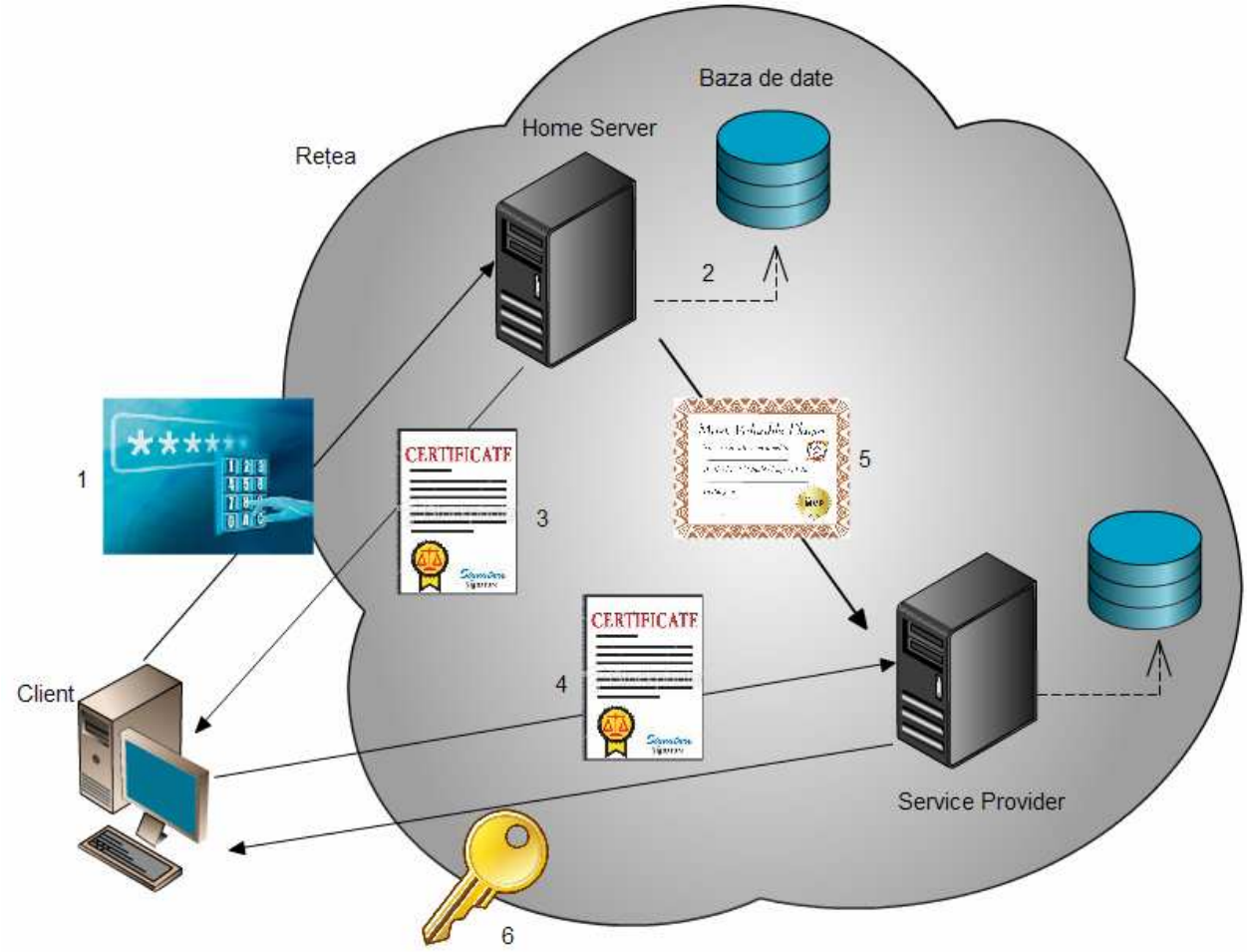

Fig. 5 – Paşii de execuție pentru mecanismul Single Sign-on

După recepționarea datelor personale de la client, home serverul va verifica dacă numele de utilizator şi parola furnizată se potrivesc cu cele din baza de date, ceea ce se poate

vedea în Fig. 5, pasul 2. Aceste informații secrete a clientului vor fi trimise doar o singură dată , asta fiind important nu numai din punctul de vedere al confortului, ci şi al securității. După autentificare, serverul va genera perechea de chei RSA şi certificatul clientului. Aceste certificate vor fi folosite de acum încolo pentru autorizare în sistem, deci trebuie să fie create şi folosite în aşa fel încât să asigure un nivel securitate ridicat. Acest certificat va conține câteva informații despre utilizator (numele, locația, organizația, adresa e-mail, etc), despre certificat (algoritmul de criptare, rezultate hash, data expirării), dar şi date despre emitent, pentru verificarea autenticității. Certificatul este semnat digital cu cheia secretă a serverului, protejând-ul astfel de orice modificare a conținutului acestuia. Certificatele folosite de mine, X.509v3, permit adăugarea unor extensii la setul de câmpuri predefinite, unde se pot înscrie comentarii sau date alese de utilizator. Aceste câmpuri vor conține permisiunile clienților care vor servi la controlul accesului în rețea, după un model bazat pe roluri (role-based access control / RBAC). Controlul accesului se referă la acordarea sau refuzul de privilegii la o entitate după autentificare. Un privilegiu este un drept pe care îl are un utilizator. Unele operații sunt considerate privilegiate şi trebuie atribuite doar persoanelor de încredere. Utilizatorilor nu se vor atribui direct permisiunile, ci aceştia le vor dobândi prin rolurile pe care le iau în cadrul rețelei. Astfel gestionarea permisiunilor utilizatorilor, sau adăugarea permisiunilor utilizatorilor noi se face uşor. Rolurile sunt salvate în aceeaşi bază de date ca şi numele de utilizator şi parola. În pasul 3, Certificatul şi cheia secretă vor fi trimise clientului via canale securizate. Clientul are în posesie un certificat valid, care poate fi folosit ca un „paşaport” pentru a se mişca liber şi a primi acces la oricare dintre servere din rețea unde are permisiune. Certificatul şi cheia secretă vor fi salvate pentru alte dăți, având o valabilitate îndelungată, atâta timp cât acesta nu a expirat, poate fi folosit pentru a se conecta la alte servere. Dacă a expirat, se contactează home serverul, şi se repetă paşii 1-3.

Pasul 4 din Fig. 5 reprezintă conectarea la unul dintre serverele de resurse, folosind certificatul generat anterior. Serverul verifică certificatul, adică numele să fie identic cu cel al clientului, să nu fie expirat, să fie emis de un server cunoscut din rețea şi semnătura digitală a emitentului. Pentru verificarea semnăturii se foloseşte cheia publică a serverului care a generat certificatul. De regulă fiecare server are o copie după certificatele ale celorlalte servere din rețea, dar în caz contrar se poate face rost de el cu uşurință, cunoscând adresa serverului (pasul 5). Dacă verificarea a fost efectuată cu succes, şi clientul are drepturi de acces la acel server, se va genera o cheie secretă. Această cheie se va folosi pentru criptarea datelor atât de la client la server cât şi vice-versa. Cheia va fi folosită de algoritmul de criptare simetric, AES, şi este validă doar pentru sesiunea curentă. Dacă cheia a expirat, clientul va

trimite încă odată certificatul pentru a solicita o altă cheie. Durata de viață scurtă a cheilor asigură o protecție ridicată împotriva atentatelor asupra rețelei. Cererile către serviciul de resurse şi datele trimise ca răspuns de către server vor fi criptate cu această cheie.

3.2.2 Componentele sistemului

În Fig. 6 este prezentată arhitectura propusa a sistemului pentru care s-au formulat cerințele. Sunt prezentate componentele din care este format şi legăturile dintre acestea.

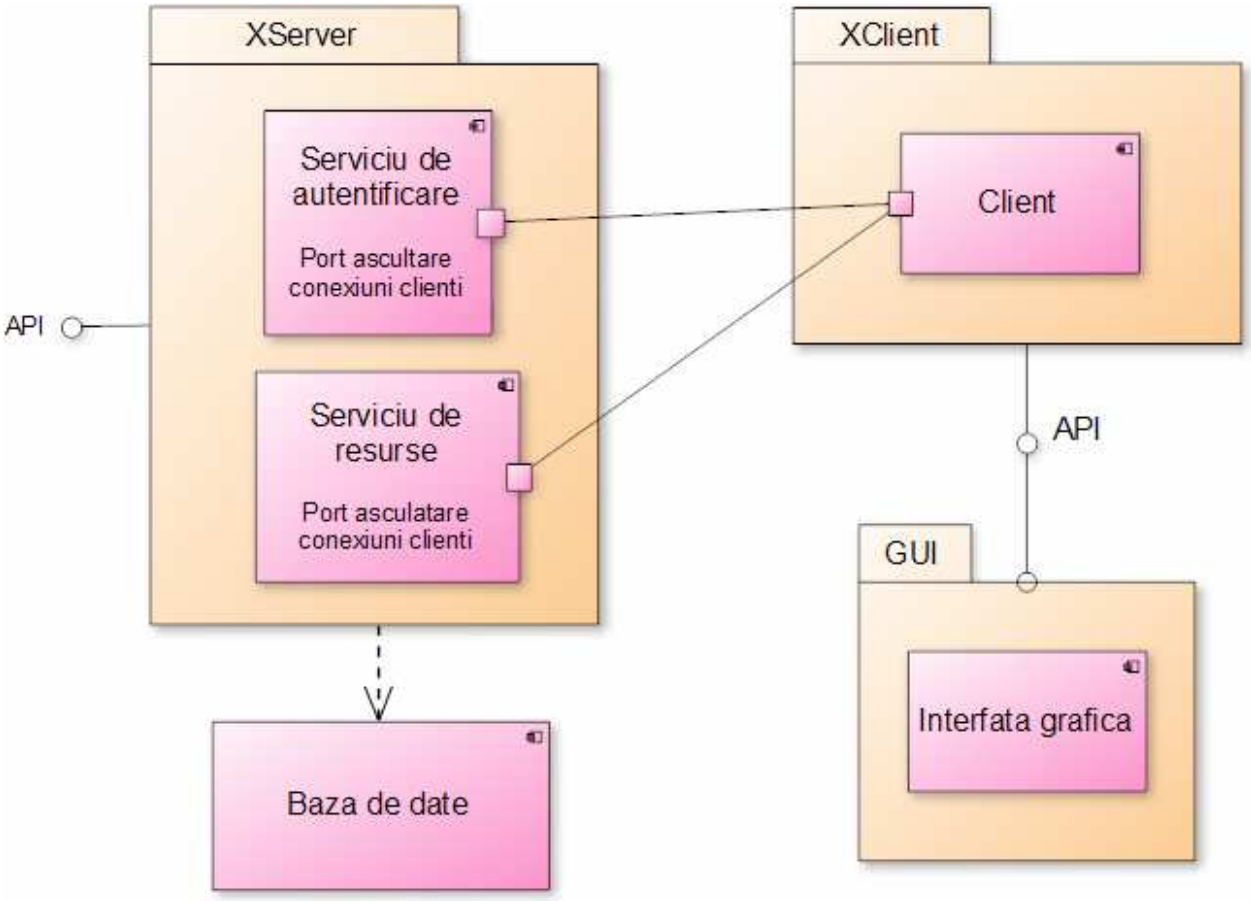

Fig. 6 – Componentele sistemului

Sistemul este format din următoarele componente:

- ***XServer***: găzduieşte diferite servicii, cu diverse funcționalități. Numărul acestor servicii nu este limitat, dar în modelul meu folosesc doar un serviciu de autentificare şi unul de resurse. Serviciul de autentificare este responsabil de identificarea clienților, prin conectarea la baza de date, generarea certificatelor şi a cheilor pentru aceştia, şi trimiterea listei cu serveri. Serviciul de resurse

încarcă fişierele cerute, îi criptează şi transmite datele înapoi la client, bineînțeles doar după autentificare.

- ***XClient***: componenta oferă o interfață pentru conectare la servere, autentificare, cerere lista de serveri, lista de fişiere, cerere de fişiere şi scrierea acestora pe o unitate fizică. Conectarea se face automat, fără intervenția utilizatorului, aşa cum şi decriptarea fişierelor primite.
- ***Baza de date****:* fiecare server se poate conecta la o bază de date pentru a stoca utilizatorii înregistrați. Aceste date se accesează de către serviciul de autentificare doar o singură dată pentru fiecare client. Baza de date mai conține o listă cu fiecare server disponibil din rețea.

3.2.3 Proiectarea componentei XServer

Componenta este responsabilă de rularea serviciilor, care la rândul lor sunt şi ei componente, permițând crearea, înlocuirea sau adăugarea acestora la fiecare server. După cum s-a menționat anterior, în modelul meu voi folosi două servicii: unul pentru autentificare şi unul pentru gestionarea resurselor.

Serviciul de autentificare este cea mai complexă componentă din model. Ea se ocupă în primul rând de actualizarea listei cu serveri, şi informarea clienților de această listă, la comandă. Clienții înregistrați la un server sunt gestionate de acest serviciu, identificarea lor se face pe baza numelui de utilizator şi a parolei, şi pe baza acestor informații serviciul generează certificatul şi cheile RSA pentru fiecare client. Autentificarea cu certificat este procesat tot de această componentă, astfel toate protocoalele de securitate sunt rulate de acest serviciu. La sfârşitul autentificării se generează o cheie secretă pentru sesiunea curentă, şi se comunică această cheie cu serviciul de resurse. Componentele majore al serviciului de autentificare şi a relațiile dintre ele se pot vedea în Fig. 7.

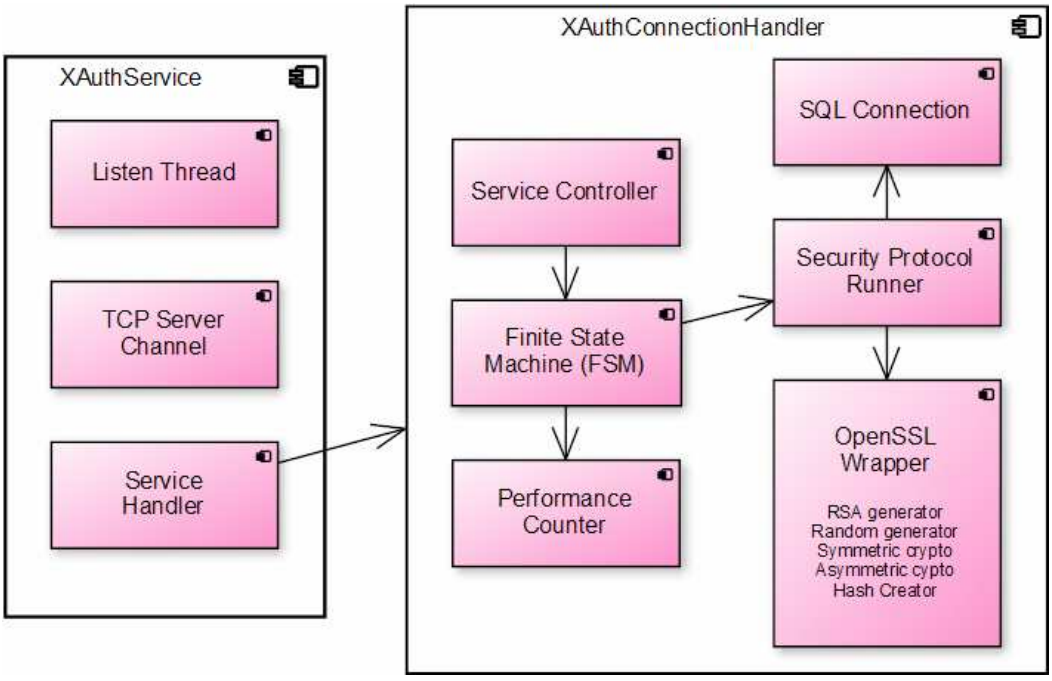

Fig. 7 – Structura internă al serviciului de autentificare

Principalele module interne ale componentei de autentificare sunt:

- ***XAuthService***: reprezintă modulul principal care se ocupă de acceptarea conexiunilor clienților. La conexiune acesta creează o instanță a modulului XAuthConnectionHandler. Totodată se ocupă de eliberarea memoriei, distrugerea modulelor inactive, notificarea serviciului de resurse de utilizatorilor noi şi preluarea mesajelor de la alți participanți.
  - ***Listen Thread***: firul de execuție generează evenimente care vor rula anumite funcții ale modulului. Aici se creează canalele de comunicație, se verifică starea fiecărui modul activ şi preluarea mesajelor.
  - ***TCP Server Channel***: implementează soclurile (socket) care acceptă conexiuni şi date. Se ocupă de distribuirea mesajelor între componente.
  - ***Service Handler***: gestionează serviciile create pentru fiecare conexiune. Creează şi distruge canalele de comunicație folosite pentru toate transmisiunile de date.
- ***XAuthConnectionHandler***: modulul este creat de către XAuthService la fiecare conexiune nouă. Această componentă se ocupă de tot procesul de autentificare şi generare de certificate şi chei.
  - ***Service Controller***: inițializează şi controlează unele funcționalități al modulului, comunică cu XAuthService.

- ***Final State Machine***: automatul finit controlează cursul protocoalelor de securitate, setând stările potrivite pentru o bună funcționare a acestora.
- ***Security Protocol Runner****:* aici sunt implementate majoritatea protocoalele de securitate. Accesează baza de date, controlează componenta OpenSSLWrapper, generează răspunsurile pentru clienți şi verifică datele din mesaje la fiecare pas.
- ***OpenSSLWrapper****:* încapsulează toate funcțiile OpenSSL folosite în proiect şi inițializează librăriile OpenSSL. Conține mai multe clase care pot fi folosite separat pentru diferite procese. Modulul XAuthConnectionHandler foloseşte majoritatea acestor clase: wrapperul pentru criptare simetrică şi asimetrică, diferite funcții legate de crearea, folosirea, încărcarea, scrierea şi verificarea certificatelor; clase pentru crearea şi verificarea semnăturilor digitale, generatoare de randomuri.
- ***Performance Counter***: măsoară timpul de execuție a anumitelor procese în timpul rulării. Unele informații sunt salvate într-un fişier jurnal.

3.2.3.1 Inițializarea componentei XServer

Componenta XServer este pornită automat cu apelarea funcției *StartServices()* la nivelul cel mai de sus, din API-ul public. Înainte de pornire se pot seta anumite proprietăți al serverului, de exemplu directorul care conține resursele. În Fig. 8 se poate vedea secvența de inițializare mult simplificată a modulului XServer.

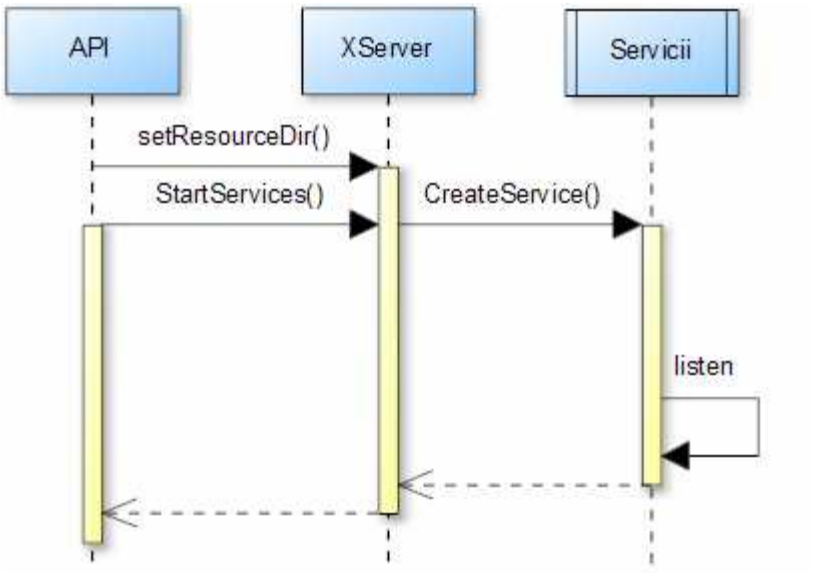

Fig. 8 – Inițializarea modulului XServer

După pornirea serviciilor, modulul XServer creează cele două servicii, care la rândul lor sunt inițiate, şi se trec în stare de aşteptare de conexiuni noi de la clienți.

3.2.3.2 Autentificarea unui client

În Fig. 9 este prezentată diagrama de secvență a autentificării unui client nou la componenta de autentificare. Pentru a porni procesul de autentificare, clientul emite o cerere de conectare. Dacă clientul nu are încă un certificat valid, atunci se va conecta prima dată la serverul gazdă (home server), unde va fi verificat în baza de date dacă este înregistrat sau nu. Dacă informațiile de identificare au fost corecte, home serverul va genera un certificat cu datele personale şi permisiunile clientului, cât şi perechea de chei RSA şi va transmite aceste date clientului. Dacă clientul are deja în posesie un certificat valid, poate să sară peste paşii de la serverul gazdă, şi va iniția o conexiune la serverul de resurse. Acest server poate să fie chiar şi home serverul. La conectare la serverul de resurse, clientul va trimite certificatul, iar serverul va verifica validitatea acestui folosind cheia publică a serverului care a emis acest certificat. Dacă nu are în posesie această cheie, va emite o cerere către acest server, şi va solicita certificatul acestuia. Se verifică mai multe informații din certificat: validitatea semnăturii, data expirării, numele înscris în certificat trebuie să coincidă cu numele clientului, emitentul certificatului trebuie să aibă proprietatea de Certificate Authority (CA), etc. Dacă toate testele sunt trecute cu succes, serverul generează o cheie privată, valabilă doar pentru sesiunea curentă, şi o va comunica clientului.

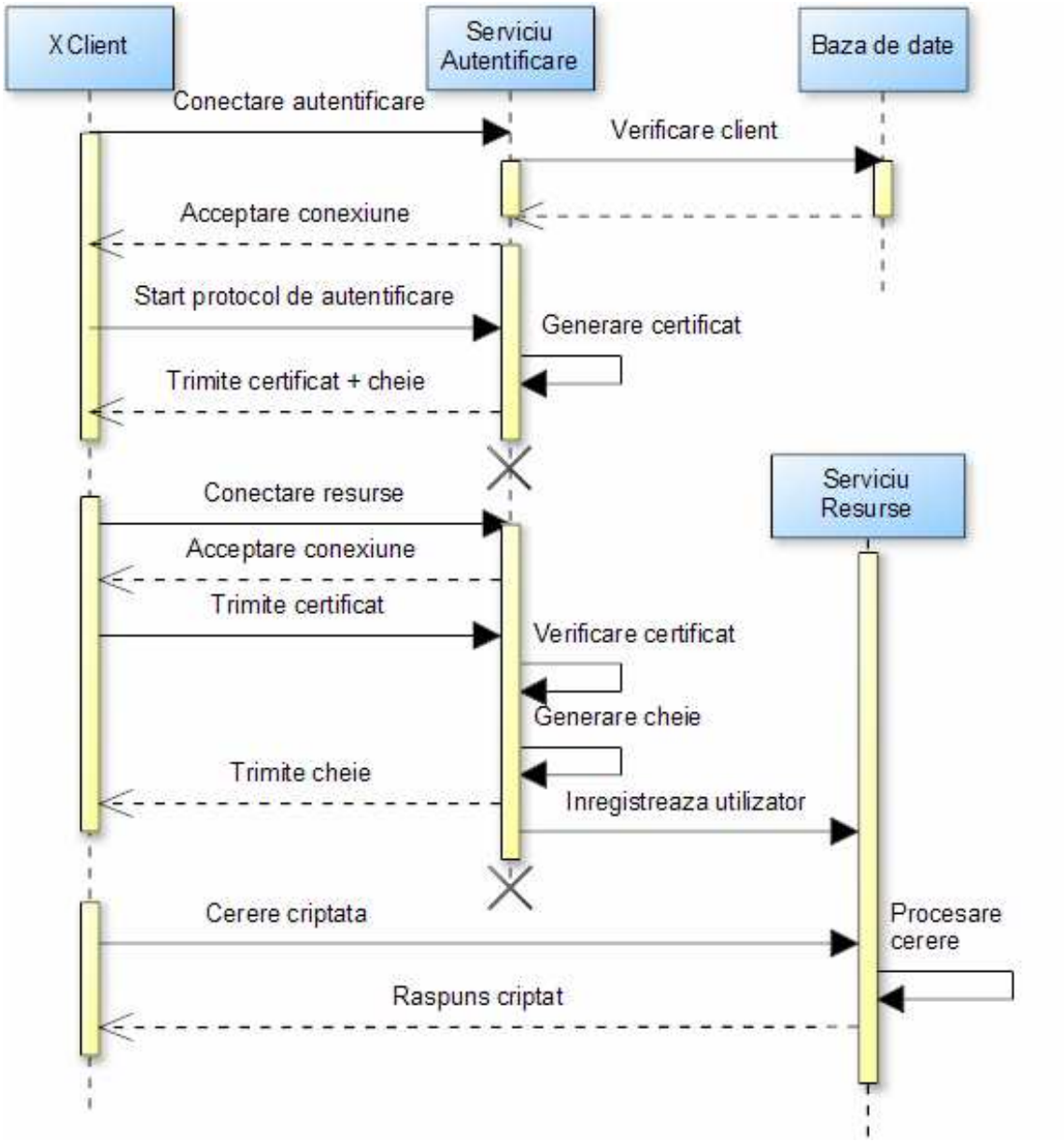

Fig. 9 – Diagrama de secvență a autentificării unui client

3.2.3.3 Serviciul de resurse

Serviciul de resurse pune la dispoziția utilizatorului fișiere găzduite pe serverul respectiv. La fiecare conectare cu succes la modulul de autentificare, numele utilizatorului și cheia acestuia vor fi înscrie într-o listă, care poate fi accesată de serviciul de resurse. La conectarea clienților la acest serviciu, se va căuta numele în acea listă. Dacă numele nu a fost găsit, fie încă nu s-a autentificat, fie a expirat cheia. Dacă numele a fost găsit în listă, se decodează cheia, și se folosește pentru decriptarea cererii clientului. Dacă decriptarea eșuează,

înseamnă că cheia nu a fost corectă. În acest caz, clientul este deconectat. În caz contrar, se procesează mesajul clientului, si în funcție de natura acestuia se va genera răspunsul.

Momentan acest serviciu poate procesa două tipuri de cereri: să trimită lista cu fişiere disponibile, şi să trimită unul dintre aceste fişiere înapoi la client. În primul caz se parcurge recursiv lista directoarelor care au fost adăugate ca resurse, şi se construieşte o listă cu toate fişierele, calea acestora şi mărimea fişierului. Serviciul, şi API-ul suportă filtrarea tipurilor de fişiere, pentru rezultate mai precise. Această listă cu fişiere va fi criptată cu cheia secretă, folosind algoritmul de criptare simetric AES, şi trimis clientului. La recepționare se decriptează mesajul, se verifică integritatea listei, şi se copiază in locație de memorie specificată de utilizator. În cazul celuilalt tip de mesaj, clientul solicită un fişier de la serviciu. Se decriptează cererea, se verifică dacă fişierul există şi dacă se află în directorul de resurse. În caz afirmativ, se încarcă fişierul într-un buffer, şi se criptează cu aceeaşi algoritm. La recepționarea fişierului de către client, ca şi în cazul anterior, se decriptează şi se copiază într-un buffer specificat de utilizator.

3.2.4 Proiectarea componentei XClient

Modulul XClient încapsulează toate procedurile pe partea de client necesare pentru a implementa mecanismul SSO descris în secțiunea 3.2.1. Interfața componentei ascunde marea parte a autentificării, totul se face automat, uşurând munca utilizatorului. Pentru a se conecta la orice server, se creează o instanță a componentei, şi se apelează funcția pentru conectare. Se verifică dacă clientul posedă un certificat valid pentru a primi acces în sistem. Dacă acesta nu există, se inițiază primul protocol pentru a solicita şi primi un certificat şi cheile respective. Aceste protocoale includ câteva subprotocoale care vor fi descrise în secțiunea următoare. Clientul se ocupă de salvarea şi încărcarea acestor certificate şi chei pentru fiecare utilizator în parte. După ce certificatul a fost verificat pentru autenticitate, se va trimite serverului care a fost selectat de utilizator inițial. Acest server va verifica certificatul încă odată, de data asta va fi verificat şi semnătura digitală cu cheia publică a emitentului. De obicei aceste certificate care conțin cheile publice a serverilor sunt salvate la fiecare participant, fiindcă se schimbă destul de rar, iar în caz contrar, fiecare server suportă un protocol simple pentru a trimite certificatul propriu solicitanților. Dacă certificatul clientului a fost găsit autentic, se trimite cheia de sesiune via al doilea protocol de securitate. Clientul va folosi această cheia până la deconectare de la acest nod, sau până expiră pe partea serverului. Generarea unei chei noi şi protocolul de transport se vor efectua la fiecare conectare, acesta având nevoie, în condiții

optime, de 100 de milisecunde timp de rulare, ceea ce practic nu este detectabil de utilizator. După conectare, sunt disponibile două funcții pentru a interacționa cu server: unul pentru a cere lista de fișiere găzduite aici, și altul pentru a solicita un fișier din această listă. Filtrarea listei în funcție de extensia fișierelor se poate face pe partea de client, pentru a găsi obiectul căutat mai uşor. Momentan nu există un indicator de progres pentru transferul fișierului, dar se poate adăuga mai târziu. Interfața mai oferă câteva funcții, ca de exemplu: interogarea componentei despre erori, solicitarea listei cu serveri din rețea, simularea unui pachet ping, pentru a măsura timpul de răspuns al unui server, scrierea fișierelor pe un dispozitiv de stocare fizic, schimbarea utilizatorului, etc. Toate aceste procese se fac automat și ascuns, dar totuși tot traficul între participanți este confidențial, datorită protocoalelor de securitate și criptografiei performante.

Componentele majore și relațiile între ele sunt prezentate în Fig. 10.

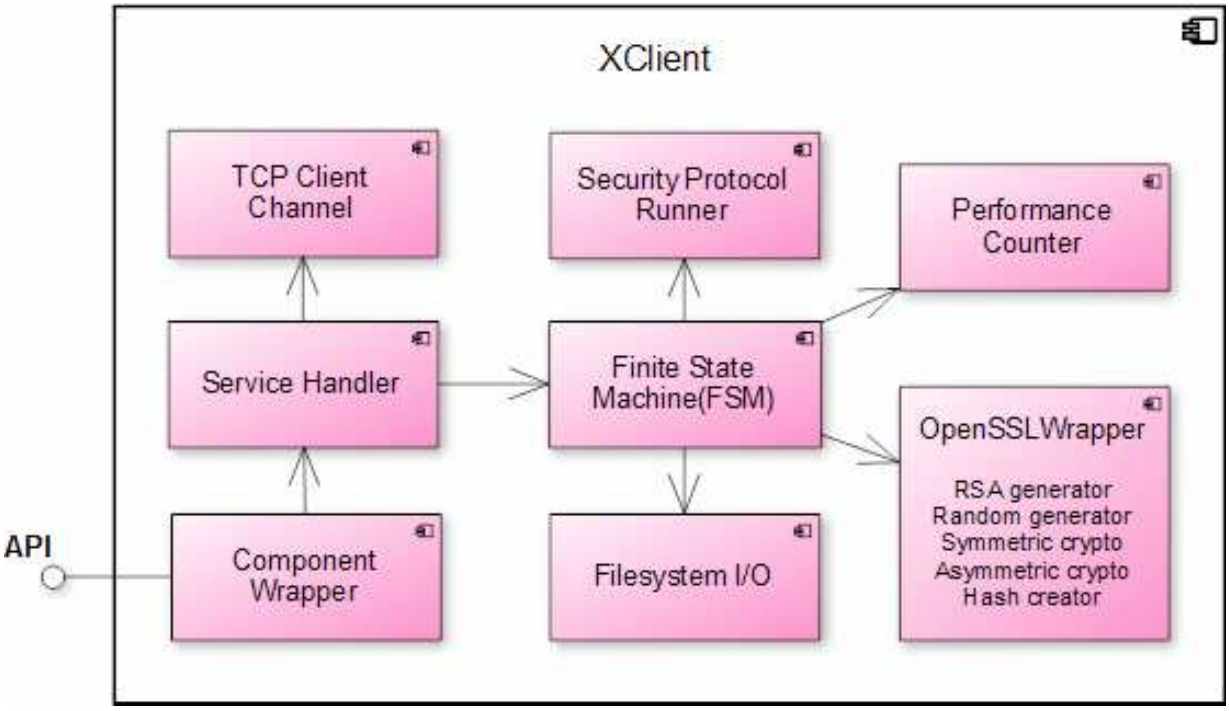

Fig. 10 – Structura internă a componentei XClient

Principalele module interne ale componentei XClient sunt:

- ***XClient***: reprezintă modulul principal care leagă componentele interioare între ele, şi face legătura cu API-ul.
    - ***TCP Client Channel***: implementează soclurile (socket) care comunică cu servere, prin protocoale TCP/IP. Se ocupă de distribuirea mesajelor între componente.
    - ***Service Handler***: gestionează serviciile create pentru fiecare conexiune. Creează şi distruge canalele de comunicație folosite pentru toate transmisiunile de date.
    - ***Component Wrapper***: Reprezintă interfața dintre utilizator şi componentele interne.
    - ***Final State Machine***: automatul finit controlează cursul protocoalelor de securitate, setând stările potrivite pentru o bună funcționare a acestora.
    - ***Security Protocol Runner****:* aici sunt implementate majoritatea protocoalele de securitate. Accesează baza de date, controlează componenta OpenSSLWrapper, generează răspunsurile pentru clienți şi verifică datele din mesaje la fiecare pas.
    - ***OpenSSLWrapper****:* încapsulează toate funcțiile OpenSSL folosite în proiect şi inițializează librăriile OpenSSL. Conține mai multe clase care pot fi folosite separat pentru diferite procese. Modulul XAuthConnectionHandler foloseşte majoritatea acestor clase: wrapperul pentru criptare simetrică şi asimetrică, diferite funcții legate de crearea, folosirea, încărcarea, scrierea şi verificarea certificatelor; clase pentru crearea şi verificarea semnăturilor digitale, generatoare de randomuri.
    - ***FileSystem IO***: reprezintă setul de funcții care accesează fişierele necesare pentru funcționare.

3.2.5 Proiectarea bazei de date

Baza de date conține date legate de localizarea serviciilor în cadrul rețelei, cât şi informații despre utilizatorii înregistrați la un server. În mod normal, fiecare server are o bază de date cu aceeaşi structură. În Fig. 11 este prezentată structura bazei de date.

Tabelul *servers* conține informații despre fiecare server în rețea. Aceste date sunt folosite pentru a forma lista serverilor la solicitarea clienților. Datele se citesc de fiecare

cerere. Fiecare server are asociat un nume, o adresă şi o opțional o descriere a serverului sau a resurselor găzduite. Tabelul *users* stochează datele despre utilizatorii înregistrați la serverul respectiv. Fiecare utilizator înregistrat are asociat un nume de utilizator, o parolă şi un rol în cadrul rețelei. Aceste date se verifică la autentificare, şi se generează certificatul pe baza lor.

Fig. 11 – Structura bazei de date

3.3 Proiectarea interfeței de programare (API)

Componenta XClient asigură o interfață minimală, dar suficientă pentru implementare. Majoritatea funcțiilor sunt cu blocare, şi au nume sugestive. Componenta oferă rapoarte detaliate în caz de eroare.

- ***Constructorul***: ia ca parametru numele de utilizator şi parola, care vor fi folosite pe toată durata instanței. Aceste date se pot schimba mai târziu.

```
XClient( const std::string UserName, const std::string Password );
```

- ***Connect***: conectare la un server din rețea, adresa acestui server fiind precizat prin parametru. Clientul se conectează la *home server* şi va solicita un certificat, în cazul în care nu are deja unul valid, iar după aceea la serverul precizat. Conectarea la cele două servere se face automat şi cu blocare până terminare. Un al doilea parametru va returna numărul de milisecunde scurse până la obținerea certificatului respectiv a cheii pentru această sesiune. Funcția returnează o expresie booleană, semnificând reuşita operației.

```
bool Connect( std::string ResourceServerAddress, double &TimeElapsed );
```

- ***getLastErrorMessage***: în caz de eroare, se apelează această funcție pentru a afla mai multe detalii legate de problemă, în formă de text.

```
std::string getLastErrorMessage();
```

- ***getServerList***: cere lista de serveri disponibili în rețea de la *home server*. Lista include adresa fiecărui server, cât şi descrierea acestora. Funcția se poate apela oricând, nu necesită autentificare. Lista va fi stocată intr-un vector dat ca parametru. Funcția blochează firul de execuție în care se rulează până primeşte un răspuns de la server, sau expiră time-out-ul. Funcția returnează o expresie booleană, semnificând reuşita operației.

```
bool getServerList( std::vector< ServerInfo* > &SList );
```

- ***getFileList***: se solicită lista de fişiere disponibile pe serverul curent. Funcția se apelează doar după o conectare cu succes la unul dintre serverele din rețea. Datele trimise şi recepționate vor fi criptate cu o cheie secretă prestabilită la conectare. Lista va fi stocată intr-un vector dat ca parametru. Lista va include numele, calea şi mărimea în octeți pentru fiecare fişier. Funcția blochează firul de execuție în care se rulează până primeşte un răspuns de la server, sau expiră time-out-ul. Funcția returnează o expresie booleană, semnificând reuşita operației.

```
bool getFileList( std::vector< FileInfo* > &FList, std::string
Mask );
```

- ***RequestFile***: se solicită un fişier de la serverul curent. Funcția se apelează doar după conectare cu succes la unul dintre serverele din rețea. Datele trimise şi recepționate vor fi criptate cu o cheie secretă prestabilită la conectare. Calea către fişierul dorit, cât şi un buffer pentru stocarea acestuia vor fi date ca parametru. Funcția blochează firul de execuție în care se rulează până primeşte un răspuns de la server, sau expiră time-out-ul. Funcția returnează o expresie booleană, semnificând reuşita operației.

```
bool RequestFile( std::string FileName, std::vector< char >
&FileBuffer );
```

- ***PingServer***: simulează un *ping* către un server din rețea. Singurul parametru este adresa serverului ales. Funcția este cu blocare şi returnează timpul trecut în milisecunde până la recepționarea răspunsului de la server.

```
int PingServer( std::string HostName );
```

- ***WriteFile***: scrie un fişier pe o unitate fizică pentru stocare. Primul parametru specifică calea către noul fişier, iar al doilea va fi bufferul care conține fişierul primit de la server. Funcția returnează o expresie booleană, semnificând reuşita operației.

```
bool WriteFile( std::string FileName, std::vector< char > FileBuffer );
```

- ***getCertificate***: funcția returnează un certificat într-un format decriptat, doar pentru afişare. Singurul parametru precizează certificatul ales, 1 pentru a returna certificatul clientului, 2 al *home serverului* şi 3 al serverului de resurse.

```
std::string getCertificate(int Select);
```

- ***SwitchUser***: permite schimbarea utilizatorului, fără a crea o instanță nouă XClient. Noul utilizator va trebui să se reconecteze la unul dintre serverele disponibile. Parametrii vor conține numele de utilizator, adresa home serverului, şi parola utilizatorului. Funcția nu returnează o valoare.

```
void SwitchUser( std::string UserName, std::string Password );
```

Componenta XServer pune la dispoziție câteva funcții care permit crearea unui server de autentificare şi de resurse funcțional.

- ***Constructorul***: are ca parametru numele serverului

```
XAuthServer( const tstring ServerName );
```

- ***setResourceDir***: permite selectarea dosarelor care vor conține fişiere resurse. Se pot preciza mai multe surse, separându-le cu virgulă. Aceste dosare vor fi parcurse recursiv de la fiecare cerere de listă de fişiere. Dacă serverul nu va găzdui resurse, se poate ignora această funcție. Funcția se apelează doar înainte de pornirea serverului.

```
void setResourceDir( tstring Directories );
```

- ***startServices***: porneşte serverele de autentificare şi de resurse în mod de ascultare. Conexiunile vor fi procesate automat. Ca parametru se pot specifica porturile de ascultare pentru cele două servere, sau 0 pentru a folosi porturile predefinite. Funcția blochează curentul fir de execuție până la oprirea serverului.

```
void startServices ( const int AuthServicePort, const int ResourceServicePort );
```

- ***stopServices***: opreşte cele două servicii, clienții vor fi deconectați.

```
void stopServices();
```

3.4 Proiectarea protocoalelor de securitate

Comunicarea între componentele sistemului de autentificare se realizează prin schimb de mesaje pe soclu. Mesajele sunt parte a unor protocoale de securitate care au rolul de a păstra confidențialitatea datelor, identificarea şi autentificarea participanților. Protocoalele diferă în funcție de componentele între care se desfăşoară, dar şi în funcție de scopul fiecărui protocol. Proiectarea protocoalelor s-a realizat pe baza principiilor prezentate în capitolul 2.2.1. Protocoalele de securitate existente ca SSL (Secure Socket Layers) sau TLS (Transport Layer Security) sunt complexe, şi conțin multe informații care nu ne folosesc în modelul meu. Fiindcă noi vrem să navigăm cu uşurință şi rapiditate între serverele din rețea nu ne putem permite să avem date inutile în mesaje, de aceea am proiectat un set nou de protocoale, care conțin datele strict necesare pentru a atinge scopurile propuse. Protocolul complet este descris în Fig. 12, descrierea lui în secțiunea următoare.

Componentele mesajelor şi ce anume semnifică acestea:

A – Home server (serverul unde este înregistrat clientul B), respectiv în mesaj nume server

B – Client, respectiv în mesaj nume client

C – Serverul de resurse ales pentru conectare, respectiv în mesaj nume server

N – Nonce, un număr aleator

(X)h – O funcție hash aplicat pe X

M – Cuvinte cheie folosite pentru identificarea tipul mesajului

K_{XY} – Cheie simetrică cunoscută doar de X şi Y

sk_X – Cheie secretă asimetrică a lui X, se decriptează cu pk_X, folosit pentru semnături digitale

pk_X – Cheie publică asimetrică a lui X, se decriptează cu sk_X

$\{|X|\}_Y$ – X criptat cu cheia Y

$Cert_X$ – Certificatul lui X, care conține şi cheia publică

Sig – Semnătură digitală

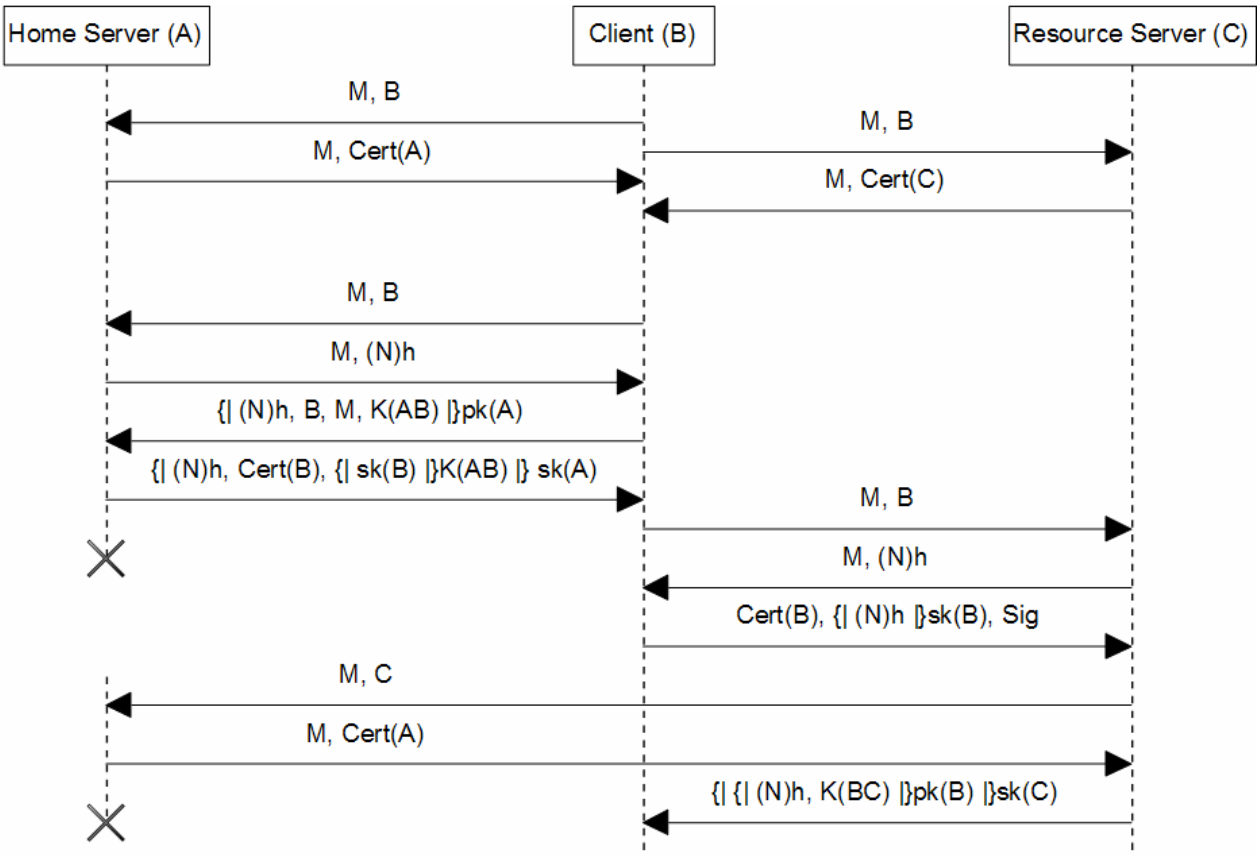

Fig. 12 – Protocolul complet de autentificare

3.4.1 Protocolul de autentificare la home server

Pentru a primi acces la serviciul de resurse, un utilizator trebuie să aibă un certificat valid. Serverul generează aceste certificate la cerere, pe baza numelui de utilizator si a parolei. Aceste informații sunt stocate într-o bază de date sigură, pe partea serverului, dar transmiterea lor nu se poate face direct, pentru că oricine poate asculta comunicația în rețea. Acest protocol de securitate va garanta transmisia lor în secret. Protocolul se efectuează ori de câte ori este nevoie de un nou certificat, dar pentru că certificatul este salvat, asta se întâmplă în mod normal doar după expirarea certificatului. Pașii protocolului de autentificare:

1. $B \rightarrow A: M, B$
2. $A \rightarrow B: M, (N)h$
3. $B \rightarrow A: \{|(N)h, B, M, K_{AB}|\}_{pk(A)}$
4. $A \rightarrow B: \{|(N)h, Cert_B\{|sk_B|\}_{K(AB)}|\}_{sk(A)}$

Primul pas reprezintă inițierea conexiunii de către client (B). Mesajul conține tipul mesajului, acesta fiind solicitare de autentificare pe baza parolei, şi numele clientului. Serverul (B) decide dacă poate accepta o nouă conexiune, şi verifică dacă numele clientului se găseşte la el în baza de date. În caz afirmativ, se generează un nonce, se aplică funcția hash, şi se trimite înapoi clientului. Mesajul serverului va conține fie acceptarea conexiunii, fie un mesaj de eroare cu motivul eşuării. Dacă conexiunea este acceptată, clientul va aplica aceeaşi funcție hash pe hashul primit de la server, şi va genera o cheie simetrică care va fi folosită pentru criptarea cheii secrete din certificat. Clientul în acest moment are în posesie certificatul serverului, fie avându-l salvat, fie cerându-l înainte de inițierea conexiunii. Acest certificat conține cheia publică a serverului (pk_A), şi va fi folosit pentru criptarea părților secrete a mesajului, adică nonce-ul, numele de utilizator, parola şi cheia generată. Datele binare sunt codate cu Base64 înainte de trimitere. Acest text cifra poate fi decriptat doar cu cheia secretă a serverului (sk_A), iar după decriptare, se verifică prima dată nonce-ul. Acesta se face prin aplicarea funcției hash pe hash-ul salvat în pasul anterior, şi compararea rezultatului cu cel din mesajul trimis de client. Dacă cele două şiruri nu coincid, se trimite un mesaj de eroare, şi se termină conexiunea. Acesta este un Dacă verificarea a fost efectuată cu succes, semnificând că mesajul este recent, se caută numele clientului în baza de date şi se compară parola trimisă de client cu cea stocată în baza de date. După acesta se generează certificatul pentru client, care va conține numele şi permisiunile solicitantului, numele emitentului, perioada de validitate, cheia publică generată şi alte informații necesare pentru verificarea certificatului. Certificatul este semnat digital cu cheia secretă a serverului. Perechea cheii din certificat este criptat cu cheia generată de client. Hash-ul nonce-ului este procesat similar ca în pasul precedent, şi împreună cu cheia criptată şi certificatul generat vor fi criptate cu cheia secretă a serverului, adică semnat digital. Acesta este un test de autentificare de intrare, pentru că nonce-ul trimis va fi reprimit de către client într-o formă criptată cu o cheie care este cunoscută numai de către server. Mesajul este autentic, dar pentru a se asigura că este şi recent, se verifică nonce-ul. Cheia secretă este decriptată cu cheia generată anterior, şi va fi salvat împreună cu certificatul generat.

3.4.2 Protocolul de autentificare la serverul de resurse

După ce clientul are un certificat valid, se poate folosi de el pentru a primi acces la anumite servere din rețea, în funcție de permisiunile fiecăruia. Acest certificat ar fi de ajuns pentru a se autentifica şi pentru a transmite date criptate, dar algoritmul de criptare RSA este

mai încet decât algoritmii de criptare cu chei simetrice. De aceea, cu ajutorul cheilor din certificat se negociază o cheie secretă simetrică, care va fi folosit pentru criptarea datelor între client şi serverul de resurse ales. Pentru că cheile simetrice pot fi descoperite mai uşor decât cele asimetrice, o cheie nouă se va genera la fiecare conectare, şi va fi valabilă doar pe durata sesiunii curente. Această cheie trebuie să rămână secretă, dar fiindcă este generat de către server, transmisia ei necesită „protecție", oferită de acest protocol de securitate. Protocolul este inițiat de client la fiecare conectare la orice server de resurse, şi are următorii paşi:

1. $B \rightarrow C: M, B$
2. $C \rightarrow B: M, (N)h$
3. $B \rightarrow C: Cert_B, \{|(N)h|\}_{sk(B)}, Sig$
4. $C \rightarrow B: \{|\{|(N)h, K_{BC}|\}_{pk(B)}|\}_{sk(C)}$

În primul pas clientul (B) inițiază conexiunea, trimițând numele şi tipul mesajului: autentificare cu certificat. Serverul răspunde fie cu acceptare, fie cu un mesaj de eroare şi motivul acestuia. Mesajul va conține şi un nonce generat de acest server. Clientul aplică funcția hash asupra nonce-ului, şi îl semnează cu cheia lui secretă. Această semnătură şi certificatul va fi trimis înapoi serverului. Nonce-ul semnat poate fi verificat cu cheia publică din certificat, dar înainte de acesta, trebuie verificat şi autenticitatea certificatului. Se verifică hash-urile din certificat, perioada de valabilitate, anumite flaguri care specifică dacă emitentul este sau nu un Certificate Authority (CA), şi bineînțeles semnătura din certificat. Această semnătură a fost făcută cu cheia secretă a emitentului, deci se verifică cu cheia publică pereche. Dacă acest server nu are certificatul serverului emitent, îl solicită şi extrage cheia publică pentru a verifica semnătura din certificat. Dacă certificatul nu poate fi validat, autentificarea va eşua. În caz contrar, se extrage cheia publică din certificat şi se verifică semnătura clientului. Şi aici întâlnim un test de autentificare de intrare, pentru că elementul criptat intră: …*N*… → {|…*N*…|}. Acest test asigură că mesajul este recent şi că este trimis de posesorul certificatului. Opțional, se verifică permisiunile înscrise în certificat, şi în funcție de regulile serverului anumite clase de utilizatori sunt filtrate. Se generează o cheie privată, şi împreună cu nonce-ul hash-uit, este criptat cu cheia publică a clientului, extras din certificat. Tot mesajul este semnat digital de către server şi transmis clientului. La recepționare se decriptează mesajul, şi se verifică nonce-ul, comparându-l cu cel salvat anterior. Se verifică şi semnătura digitală a serverului, şi se salvează cheia de sesiune. Această cheie va fi folosit doar o singură sesiune, şi doar cu acest server. Toate cererile de resurse şi răspunsurile sunt

criptate cu această cheie. La reconectare se rulează acest protocol încă odată pentru a genera o cheie nouă.

3.4.3 Protocolul de solicitare certificat

Sistemul Single Sign-on este construit pe o infrastructură cu chei publice, unde certificatele sunt elementele de bază. De aceea fiecare protocol de autentificare are nevoie de certificatele serverelor participanți. Pentru asta am creat un alt protocol simplu care solicită certificatul serverului precizat şi îl salvează local. Pentru că în cazul meu cel mai important aspect după securitate este simplitatea, acest protocol conține doar doi paşi:

1. $X \rightarrow Y: M, X$
2. $Y \rightarrow X: M, Cert_Y$

Primul pas specifică tipul mesajului şi numele solicitantului, iar al doilea pas conține rezultatul cererii, şi certificatul solicitat. Fiindcă certificatul conține toate elementele de securitate pentru a verifica autenticitatea lui, nu este nevoie de un protocol complex pentru a asigura transmisia acestuia.

Capitolul 4 – Implementarea sistemului

4.1 Arhitectura internă a claselor

La baza sistemului stau clasele construite cu librăriile NSPR, şi care ofer funcționalități simple, de exemplu: fire de execuție, socluri, parsere, containere şi elemente de legătura cu clasele la nivele mai înalte.

O altă grupă de clase este cea de securitate, asta include clasele care se conțin funcționalități pentru crearea, încărcarea, verificarea certificatelor, criptografie, funcții hash, semnături digitale, generatoare de nonce-uri, etc. Aceste clase au fost construite folosind librăriile OpenSSL.

Canalele de comunicație sunt moştenesc soclurile şi firele de execuție, pentru a oferi un control mai avansat al fluxului de mesaje. Acesta se realizează apelând funcții callback. Gestionarea canalelor se face cu ajutorul clasei XChannelHandler, care creează şi şterge canalele.

Peste aceste canale sunt construite serviciile, care deja au funcționalități complexe. Sunt două tipuri de servicii, client şi server, fiecare are canale de comunicație, care vor transmite mesajele. Gestionarea serviciilor se face de clasa XServiceHandler, cu ajutorul căreia se pot crea şi şterge serviciile.

Fig. 13 – Clasa XThread

4.1.1 Clasele de bază

În această secțiune sunt prezentate clasele mai importante care sunt folosite pentru a construi componentele principale, amintite anterior.

Clasa XThread (Fig. 13) implementează datele necesare şi funcționalitatea unui fir de execuție aşa cum este el definit de NSPR. Printre datele membru se regăseşte *m_Thread*, descriptorul firului de execuție şi două variabile de tip boolean *m_bRunning* şi *m_bExited* utilizate pentru a semnala starea firului. Funcțiile membru *start(), stop()* sunt folosite pentru a porni, respectiv a opri

firul de execuție. Pentru a lansa un fir de execuție trebuie derivată o clasă din XThread şi suprascrisă metoda *Run()*, instanțiată clasa, apoi se va apela metoda *start()*. Firul de execuție va rula până când metoda *Run()* îşi va încheia execuția sau este apelată metoda *stop()*. Metodele *isRunning()* şi *isExited()* returnează starea firului de execuție.

Clasa XTwinThread încapsulează două fire de execuție, unul prin moştenirea clasei XThread, iar al doilea printr-un membru de tipul clasei XThreadRunner care moşteneşte la rândul ei clasa XThread. Membrul *m_thread* este un pointer către instanța clasei XThreadRunner iar metodele *tstart()*, *tstop()*, *tisRunning()* şi *tisExited()* se folosesc pentru a porni, opri, respectiv interoga starea celui de-al doilea fir de execuție.

Clasa XTCPSocket (Fig. 14) încapsulează funcționalitatea unui soclu utilizat pentru a stabili conexiuni TCP/IP. Printre datele membru este *m_Socket*, descriptorul obiectului soclu. Funcțiile *getData(char* buffer,int bufSize)* şi *sendData(char* buffer,int bufSize)* sunt folosite pentru a receptiona date pe soclu, respectiv pentru a trimite date.

Clasa XTCPServerSocket moşteneşte în mod direct clasa XSocket şi implementează metode pentru a pune un soclu în starea de "ascultare" şi pentru a accepta cereri de conexiune. Aceasta oferă în plus fața de clasa XSocket două metode: *Listen()* şi *Accept()* folosite pentru a pune soclul în starea de ascultare, respectiv pentru a accepta o conexiune. Mai conține şi un membru *m_port* folosit pentru a specifica pe ce port să se pornească ascultarea.

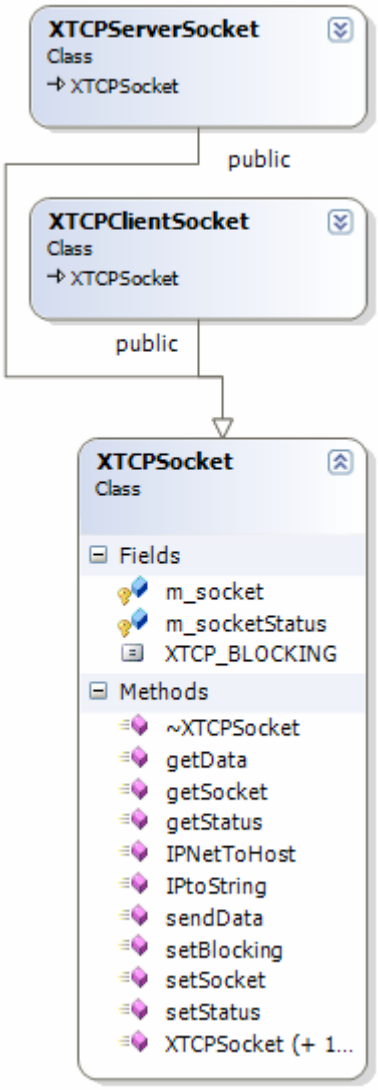

Fig. 14 – Clasa XTCPSocket

Clasa XTCPClientSocket moşteneşte în mod direct clasa XSocket şi implementează metode pentru a realiza o conexiune cu un server. Aceasta oferă în plus față de clasa XSocket câteva metode: *Connect()* pentru a iniția o cerere de conexiune şi *isConnected()* prin care se poate interoga starea soclului (dacă este sau nu conectat). Printre membri acestei clase se pot aminti următorii: *pAddr* şi *m_iPort* prin care se specifică adresa şi portul pe care să se facă conectarea şi *m_bConnected* utilizat pentru stabilirea stării soclului, valoarea sa putând fi setată prin intermediul metodei *setConnectionStatus()* şi obținută prin intermediul metodei isConnected().

Clasa TCPServer încapsulează clasa XServerSocket, şi XTwinThread, şi oferă funcționalitatea unui server, adică ascultă pentru conexiuni noi, atât timp cât membrul privat *m_bShouldRun* este true. Tot aici se vor apela funcțiile callback care vor fi suprascrise în clasele care vor moşteni clasa TCPServer. Oferă trei funcții pentru a interacționa cu serverul: *createServer()*, *runServer()* şi *stopServer()*, rolul cărora sunt evidente.

Clasa TCPTransport încapsulează clasa XClientSocket, dar este folosit atât pe partea clientului cât şi pe partea serverului, ocupându-se de transportul datelor. Apelează funcțiile callback, la orice eveniment de receive sau send, atât timp cât membrul privat *m_bShouldRun* este true.

Un obiect care stă la baza transportului de date este clasa XMessage (Fig. 15), care conține toate setările necesare ca datele să ajungă la destinație, cât şi datele în sine. Clasa oferă stocarea datelor atât în format raw, neformatat, cât şi într-un format delimitat, având funcționalitatea unui parser.

Clasa XSimpleParser (Fig. 15) este folosit foarte des, deoarece se ocupă de prelucrarea mesajelor, şi a datelor în general. Prelucrarea în cazul acestui parser înseamnă despărțirea întregului şir în mai multe părți (tokenuri), iar delimitatorul este specificat de utilizator la instanțierea obiectului. Metodele publice includ funcții pentru a returna numărul acestor tokenuri, verificarea dacă conține un anumit subşir, returnarea fiecărui token separat, etc.

Clasa XSimpleList (Fig. 15) încapsulează lista din librăriile standard [25], adăugând câteva funcții pentru compatibilitate cu celelalte clase din proiect.

Clasa XBase64Coder (Fig. 15) oferă codare şi decodare în popularul format standardizat Base64 [26]. Acest format permite trimiterea datelor binare între participanți fără a pierde integritatea datelor. Clasa stochează rezultatul şi mărimea acestuia în membrii privați *m_pResult* respectiv *m_nResultSz*. Funcțiile publice *encode()* şi *decode()* sunt folosite pentru codarea şi decodarea datelor, iar rezultatul şi mărimea rezultatului sunt returnate de funcțiile *getResult()* respectiv *getResultSize()*.

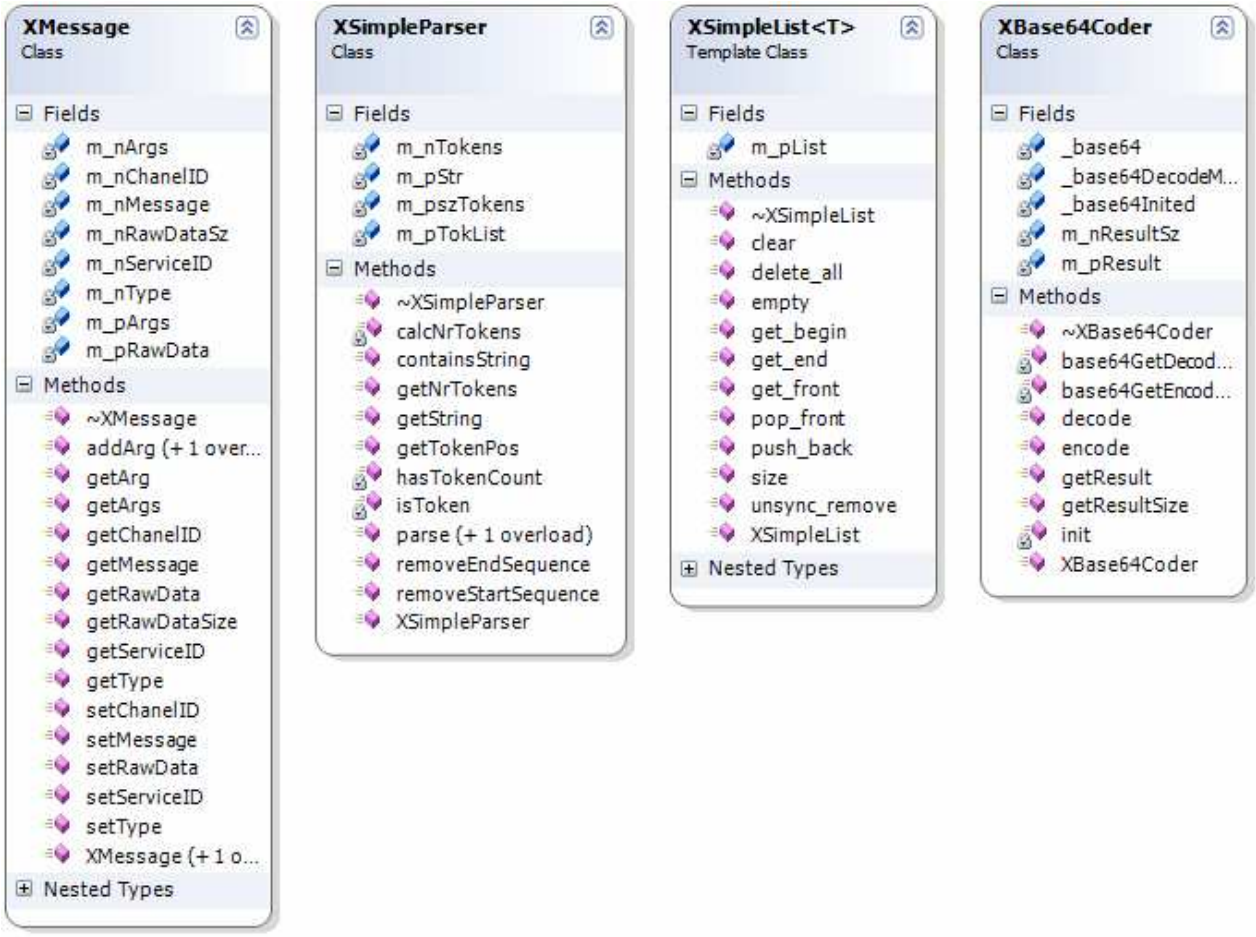

Fig. 15 – Diagrama de clase pentru clasele: XMessage, XSimpleParser, XSimpleList şi XBase64Coder

4.1.2 Clasele de securitate

Clasele OpenSSL sunt construite pe librăriile OpenSSL, şi se folosesc pentru a implementa facilitățile de securitate. Diagrama de clase este prezentată în Fig. 16.

Majoritatea claselor au un membru privat care conține datele procesate, şi funcții de tip „getter / setter” pentru accesarea lor.

Clasa DigitalSignatureCreator crează semnături digitale pe baza datelor de intrare şi a unei chei transmise ca parametrii în funcția *signData()*. Semnătura şi hash-ul acesteia pot fi accesate prin funcțiile respective. Această clasă oferă si verificare prin funcția *verifySignature()*, care ia ca parametru datele semnate digital şi o cheie.

Clasa HashCreator, cum spune şi numele, aplică o funcție hash pe datele introduse. Funcția *hashData()* ia ca parametru datele de intrare şi tipul algoritmului hash. Algoritmii suportați sunt MD5 şi SHA1. Rezultatul se returnează cu funcțiile *getHash()* şi *getHashSize()*.

Clasele CertificateCreator, CertificateChecker şi CertificateLoader conțin proceduri pentru gestionarea certificatelor. Cum spune şi numele, CertificateCreator generează

certificatele pe baza datelor trimise ca argumente funcției *CreateCertificate()*. Informațiile necesare pentru generarea certificatului sunt prezentate în secțiunea 2.4. Tot aici se generează şi perechea de chei RSA. Clasa CertificateChecker verifică certificatele, după anumite criterii specificate ca parametru pentru funcția *VerifyCertificate()*. Funcția *getPermission()* citeşte permisiunile clienților pentru controlul accesului în rețea. Clasa CertificateLoader are rolul de a încărca certificatele stocate pe un dispozitiv fizic. Funcțiile *getPrivateKey()* şi *getPublicKey()* extrag cheile specificate din certificat, care a fost încărcat folosind funcția *loadCertificate()* respectiv din fişierul cu cheia privată, folosind funcția *loadPrivateKey()*.

Clasa RandomCreator generează numere aleatorii, care sunt folosite în orice protocol de securitate. Un număr 100% aleatoriu este imposibil de generat folosind numai calculatorul, de aceea aceste numere sunt doar pseudorandom-uri. Totuşi se încearcă să fie influențat de cât mai multe surse întâmplătoare posibil. Acesta foloseşte timpul calculatorului combinat cu valori din stivă. Funcția *generateRandom()* creează un număr de mărime specificată ca parametru. Rezultatul este returnat de funcția *getRandData()*.

Clasa AszmmEncCreator încapsulează funcțiile de criptografie simetrică din librăriile OpenSSL. Suportă toate algoritmele disponibile în aceste librării. Pentru selectarea algoritmului şi specificarea cheii folosite se apelează funcția *initEncryptionCtx()*. După inițializare se pot apela funcțiile *encryptData()* şi *decryptData()* pentru criptarea respectiv decriptarea datelor. Rezultatul se obține apelând funcția *getData()* iar mărimea rezultatului prin funcția *getDataSize()*.

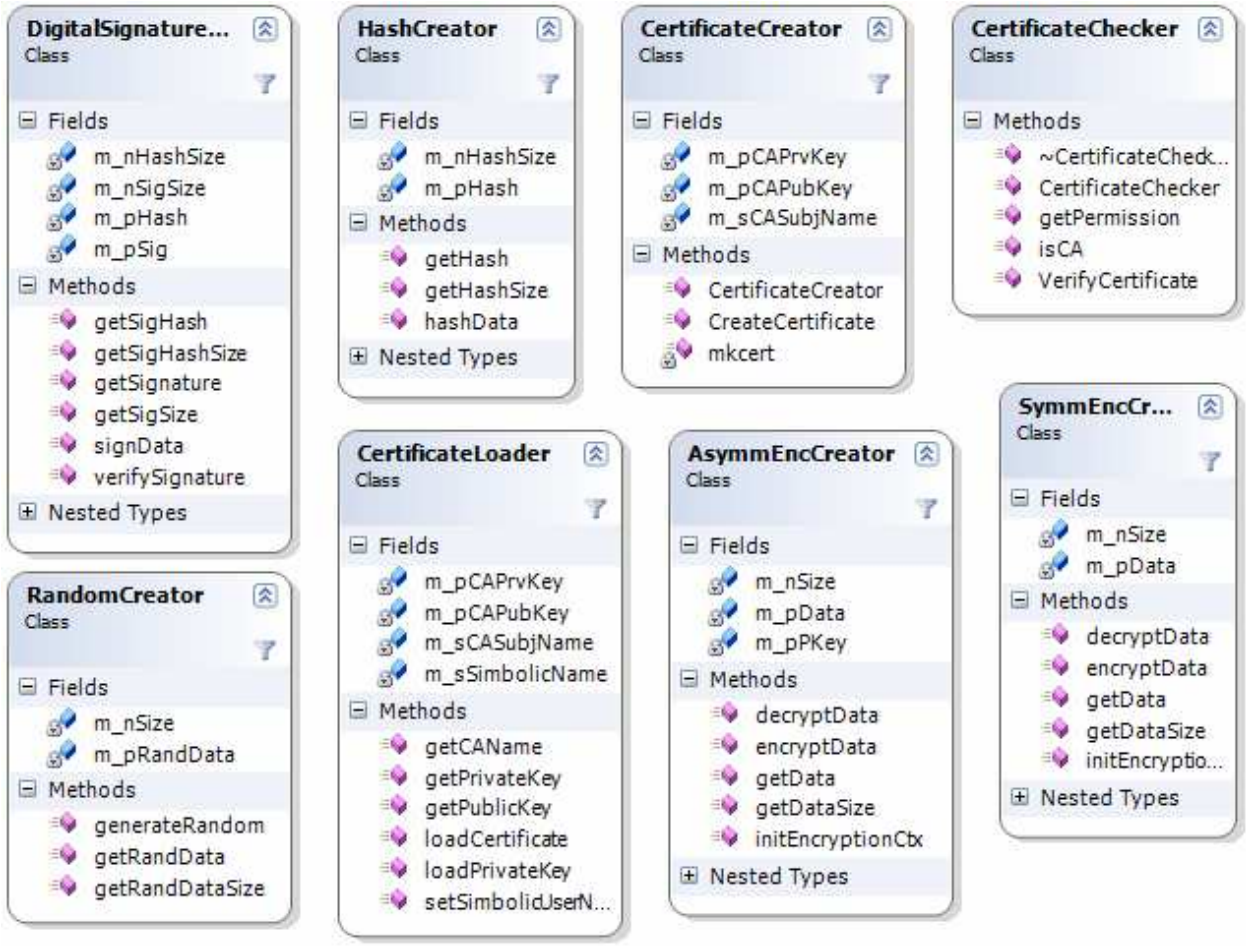

Fig. 16 – Diagrama de clase – OpenSSL

4.1.3 Canalele de comunicație

Prin folosirea funcțiilor callback a claselor TCPServer şi TCPTransport se creează o listă care va conține obiecte de tip XMessage, cu ajutorul cărora se va efectua transmisia datelor. Canalele sunt reprezentate de clasa XIChannel, care gestionează această listă cu mesaje, *m_msgList*. Identificatorul canalului este stocat în membrul *m_nChannelID*, iar de funcțiile callback se ocupă *m_pchCallback*. Aceste variabile sunt accesabile prin metodele „getter” şi „setter”: *getChannelID(), setChannelID(), addMessage(), getChMessage()*.

Clasa XHandlerImplementation moşteneşte clasele XIChannelCallback şi XThread, şi se ocupă de gestionarea canalelor şi redirecționarea mesajelor la destinația potrivită. Firul de execuție, pornit cu *run()*, citeşte datele recepționate, şi decide canalul pentru care este destinat acesta. Canalele sunt create şi distruse de acest obiect, prin funcțiile *createChannel()* şi *destroyChannel()*. Membrul *m_channelMap* stochează toate canalele create. Mesajele sunt trimise canalelor prin funcțiile *sendToChannel()* şi *getMessage()*, iar mesajele interne sunt

trimise funcției *handleInternalChMessage()* pentru procesare locală. Clasa XChannelHandler este un wrapper pentru clasa XHandlerImplementation.

Clasele XTCPServerChannel şi XTCPClientChannel moştenesc unele clase menționate anterior, şi adaugă anumite funcționalități specifice canalelor server / client, şi se ocupă de crearea şi verificarea header-urilor mesajelor.

Diagrama de clase a canalelor de comunicație este prezentată în Fig. 17.

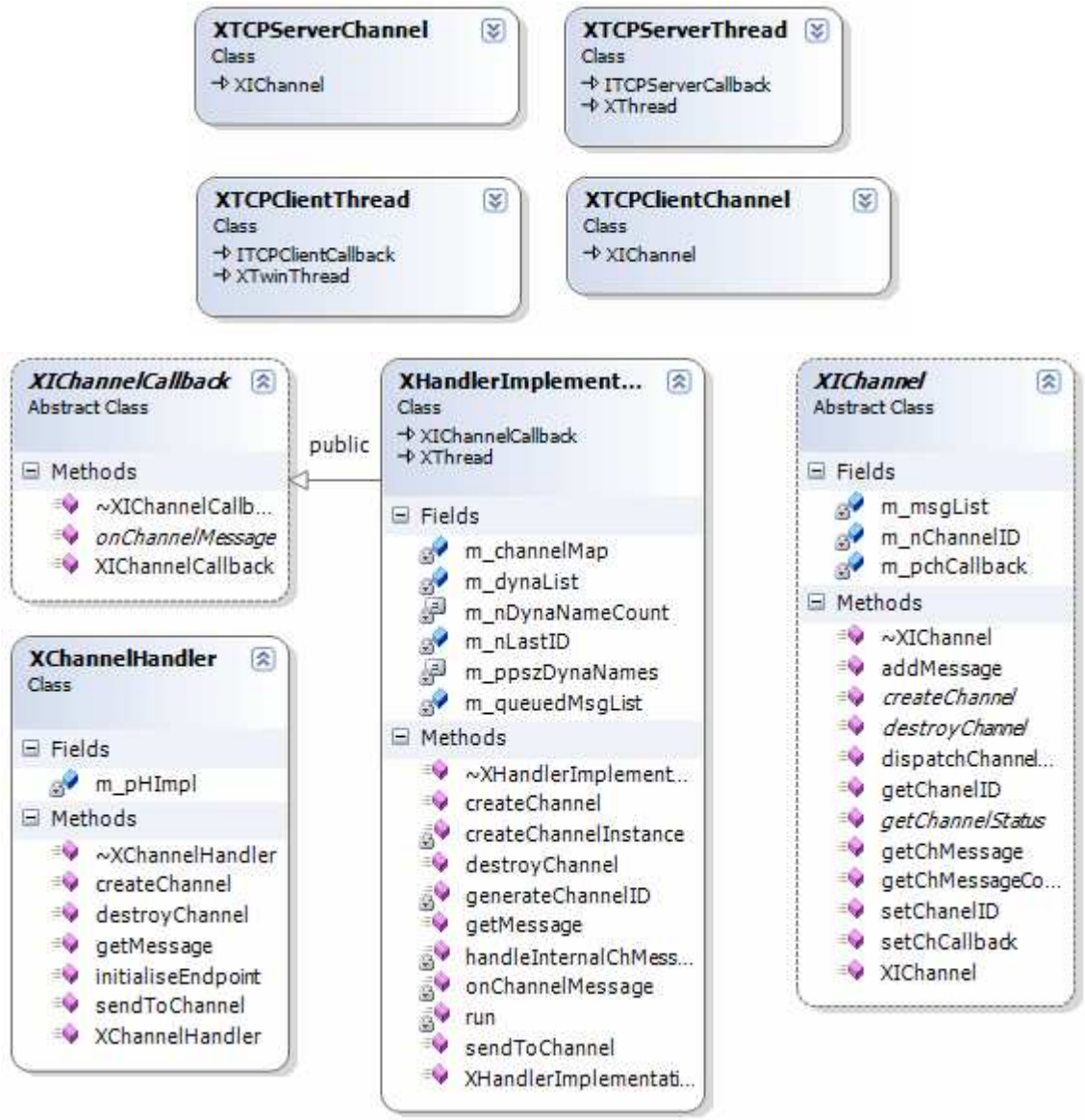

Fig. 17 – Diagrama de clase a canalelor de comunicație

4.1.4 Servicii

Serviciile încapsulează canalele de comunicații şi au o structură similară cu acestea: funcțiile callback generează evenimentele când se recepționează un mesaj. Aceste mesaje sunt trimise canalelor, iar de la canale la servicii. Gestionarea serviciilor se face de către un obiect global, ca şi XChannelHandler-ul. La crearea serviciilor, se setează proprietățile fiecăruia în parte. Acest lucru se face cu ajutorul structurii *service_info*, care conține tipul noului serviciu, portul de ascultare, şi configurări opționale.

Elementul de bază este XIService care la rândul lui moşteneşte clasa XThread. Această clasă este abstractă, şi defineşte structura clasei care o va încapsula. Acesta va fi XBaseService care are deja funcționalitățile unui serviciu. Membrii privați stochează ultimele mesaje (*m_lstMessages*), o instanță a XChannelHandler-ului, prin care se va face transmisia datelor (*m_pChannelHandler*), proprietățile serviciului (*m_svInfo*), şi identificatorului canalului principal (*m_nMainChannelID*). Majoritatea funcțiilor membri a clasei sunt virtuale, şi folosite ca funcții callback pentru clasa care va moşteni acest obiect. Aceste funcții sunt apelate la evenimentele generate de mesaje primite, care sunt verificate în funcția *run* a firului de execuție. Funcțiile includ notificări pentru crearea, distrugerea, configurearea canalelor, furnizează informații despre starea canalelor, şi trimit notificări în cazul în care s-au recepționat mesaje pe canal.

Clasa XServiceDeployment gestionează serviciile, se ocupă de crearea acestora şi expedierea mesajelor. Toate serviciile active sunt stocate în variabila *m_serviceMap*. Distrugerea şi configurarea serviciilor, cât şi rutarea mesajelor se face la nivele mai joase. Clasa XServiceHandler este un „wrapper” pentru clasa XServiceDeployment. Diagrama acestor clase este prezentată în Fig. 18.

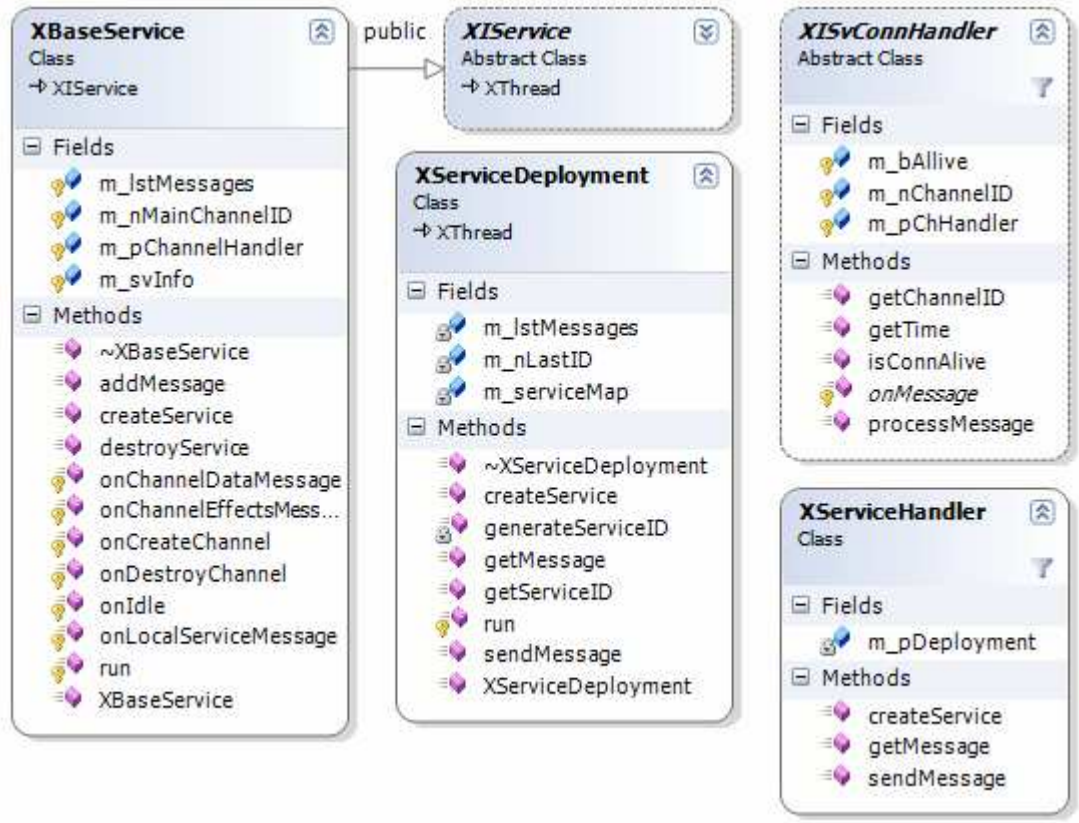

Fig. 18 – Diagrama de clase a serviciilor

Diagrama claselor de comunicație şi relațiile între ele sunt prezentate în Fig. 19.

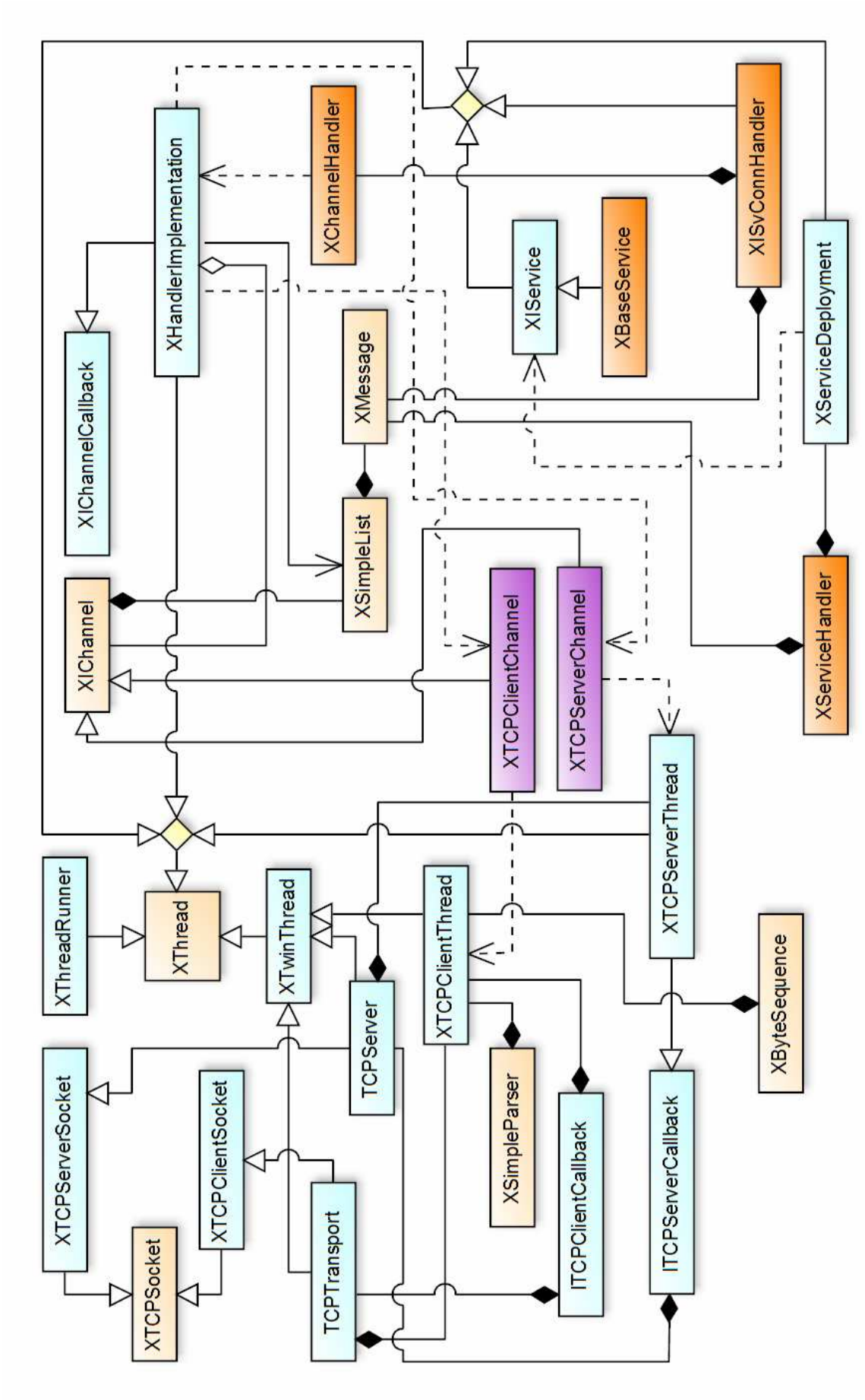

Fig. 19 – Clasele de comunicație şi relațiile între ele

4.2 XServer

Funcționarea componentei este descrisă în secțiunea 3.2.3. Componenta este o încapsulare a clasei XServiceHandler, cu ajutorul căreia sunt create serviciile de autentificare şi de resurse. După configurarea componentei, fluxul mesajelor şi generarea notificărilor se face „automat” de către clasele de comunicație descrie mai sus.

Un serviciu se creează prin moştenirea clasei XBaseService. Această clasă conține în mare parte funcții virtuale, care trebuie suprascrie de către noul serviciu creat. Un fir de execuție se ocupă de citirea mesajelor şi generarea notificărilor necesare. Funcțiile oferite de clasa XBaseService sunt vizibile în Fig. 18. Funcția *onCreateChannel()* este apelată de fiecare dată când o nouă conexiune este creată, şi odată cu acesta un nou obiect XChannelHandler, pentru a gestiona fluxul mesajelor. La conectarea unui client se creează un obiect de tip XAuthSvConnHandler pentru serviciul de autentificare, respectiv XResSvConnHandler pentru serviciul de resurse. Aceste două clase moştenesc clasa abstractă XISvConnHandler, care recepționează mesajele pentru acest canal, şi le procesează în funcție de tipul serviciului. Membrul privat *m_lstConnections* este o listă care conține aceste obiecte de tip XISvConnHandler, şi are rolul de a găsi destinatarii mesajelor la recepționare. De acesta se ocupă funcția *onChannelDataMessage()* care este apelată la fiecare mesaj recepționat. În acest moment lista cu conexiuni este parcursă şi se caută canalul care are identificatorul la fel ca şi mesajul primit. Dacă se găseşte canalul, se apelează funcția *processMessage()*, în caz contrar se generează o eroare şi se distruge canalul. Acest lucru declanşează o notificare care apelează funcția *onDestroyChannel()*. În această funcție se şterge obiectul care stoca conexiunea respectivă, şi se eliberează memoria. O altă notificare se declanşează când un canal este inactiv pentru un timp mai lung. În funcție de implementarea serviciului, aici se pot lua unele măsuri în legătură cu acest canal. În sistemul meu, conexiunile sunt eliberate când sunt inactive. Diagrama de stare a clasei este prezentată în Fig. 20.

Clasele XISvConnHandler suprascriu funcția *onMessage()* care va fi apelată de fiecare dată când este recepționat un mesaj care are ca destinație acest canal. În această funcție se procesează mesajul, în cazul acesta se verifică header-ul mesajelor,se extrage informația utilă din acestea, şi se trimite funcției *processRequest()*. Aici procesarea datelor este dependent de tipul serviciului. Diagrama de stare se poate vedea în Fig. 21.

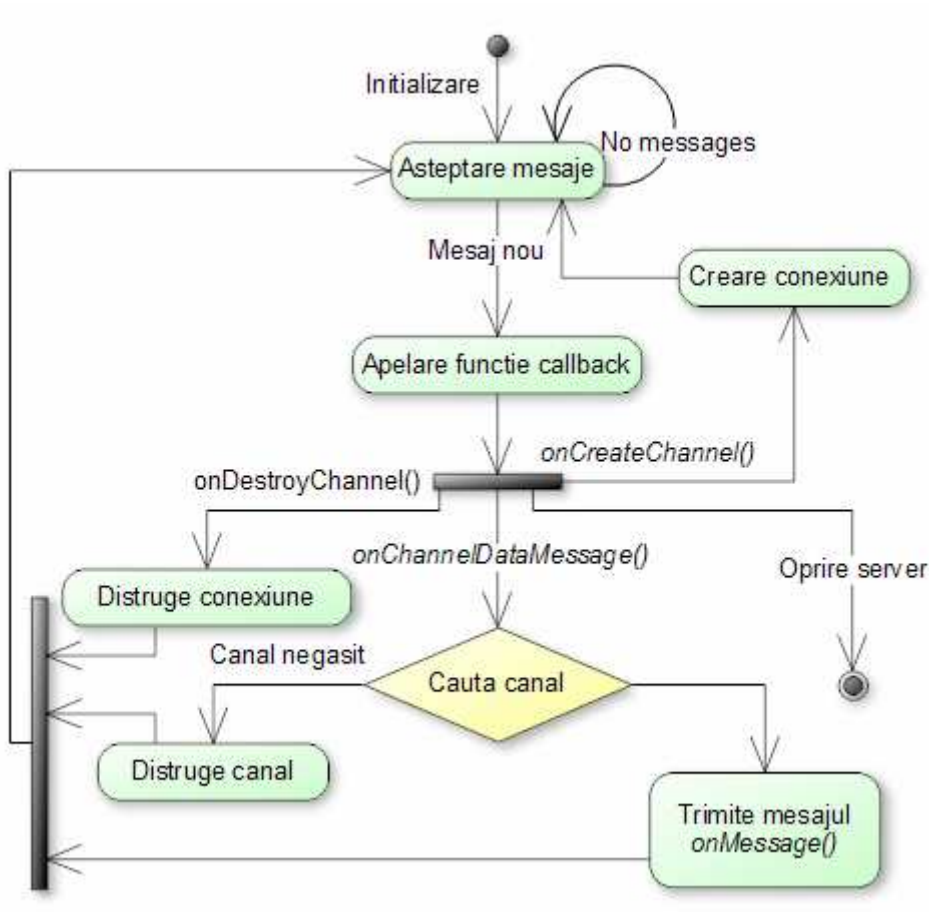

Fig. 20 – Diagrama de stare a sistemului de servicii

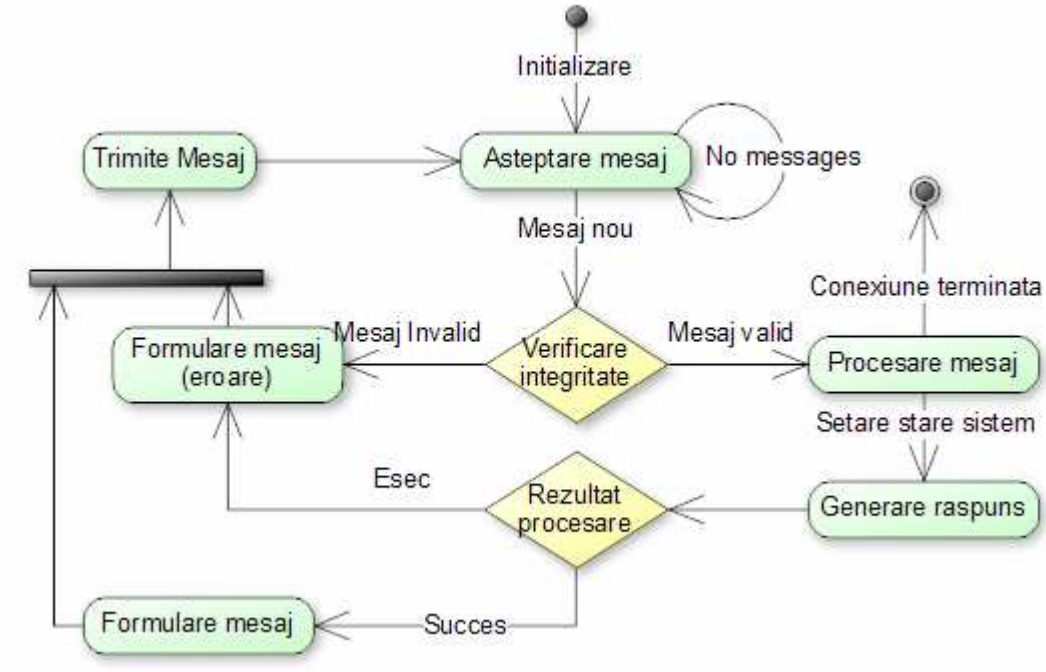

Fig. 21 – Diagrama de stare a sistemului de conexiuni

4.2.1 Serviciul de autentificare

Serviciul de autentificare are următoarele roluri:

- Autentifică utilizatorii pe baza numelui de utilizator şi a parolei
- Autentifică utilizatorii pe baza certificatului
- Trimite certificatul propriu la cerere
- Trimite lista de serveri disponibili din rețea la cerere
- Măsurarea performanței de autentificare a clienților

Fiecare server care rulează acest serviciu are aceste funcționalități, iar selectarea lor se face de către maşina de stare a sistemului. Stările sunt setate în funcție de conținutul mesajelor sau de pasul curent în procesul de autentificare. La acceptarea unei noi conexiuni în funcția *onCreateChannel()* se creează o nouă instanță a obiectului XAuthSvConnHandler, şi se adaugă în lista *m_lstConnections*. Periodic se verifică activitatea acestor conexiuni, şi dacă sunt inactive, sau şi-au terminat activitatea, sunt eliberate din memorie şi şterse din listă. Mesajele sunt „recepționate" prin funcția *onChannelDataMessage()* şi trimise ca parametru instanței XAuthSvConnHandler prin funcția *processMessage().* În cazul în care nu este găsit proprietarul mesajului, se şterge canalul de comunicație. La distrugerea conexiunilor se citeşte membrul *m_Time* a instanței XAuthSvConnHandler, prin funcția *getTime(),* care conține timpul de rulare în cazul autentificării. Aceste informații sunt salvate într-un fişier local.

La crearea unui obiect XAuthSvConnHandler sistemul este inițiat şi trece în stare de aşteptare mesaje. La recepționarea unui mesaj se verifică integritatea acestuia, şi se procesează. La procesare se setează starea sistemului în funcție de solicitarea clientului. Protocolul de autentificare are nevoie de mai mulți paşi pentru terminare, aşa că sistemul este configurat pentru a recepționa alte mesaje şi aşi continua autentificarea. Celelalte mesaje, şi ultimul pas din fiecare protocol, după generarea şi trimiterea datelor pun sistemul în starea finală, unde se eliberează memoria alocată în timpul rulării, şi se setează variabila *m_bAllive*, care indică starea conexiunii, false. Clasa XAuthService verifică această valoare periodic, şi la terminare, dacă e cazul citeşte performanța autentificării, distruge şi eliberează conexiunea. În Fig. 22 se vede diagrama de stare simplificată.

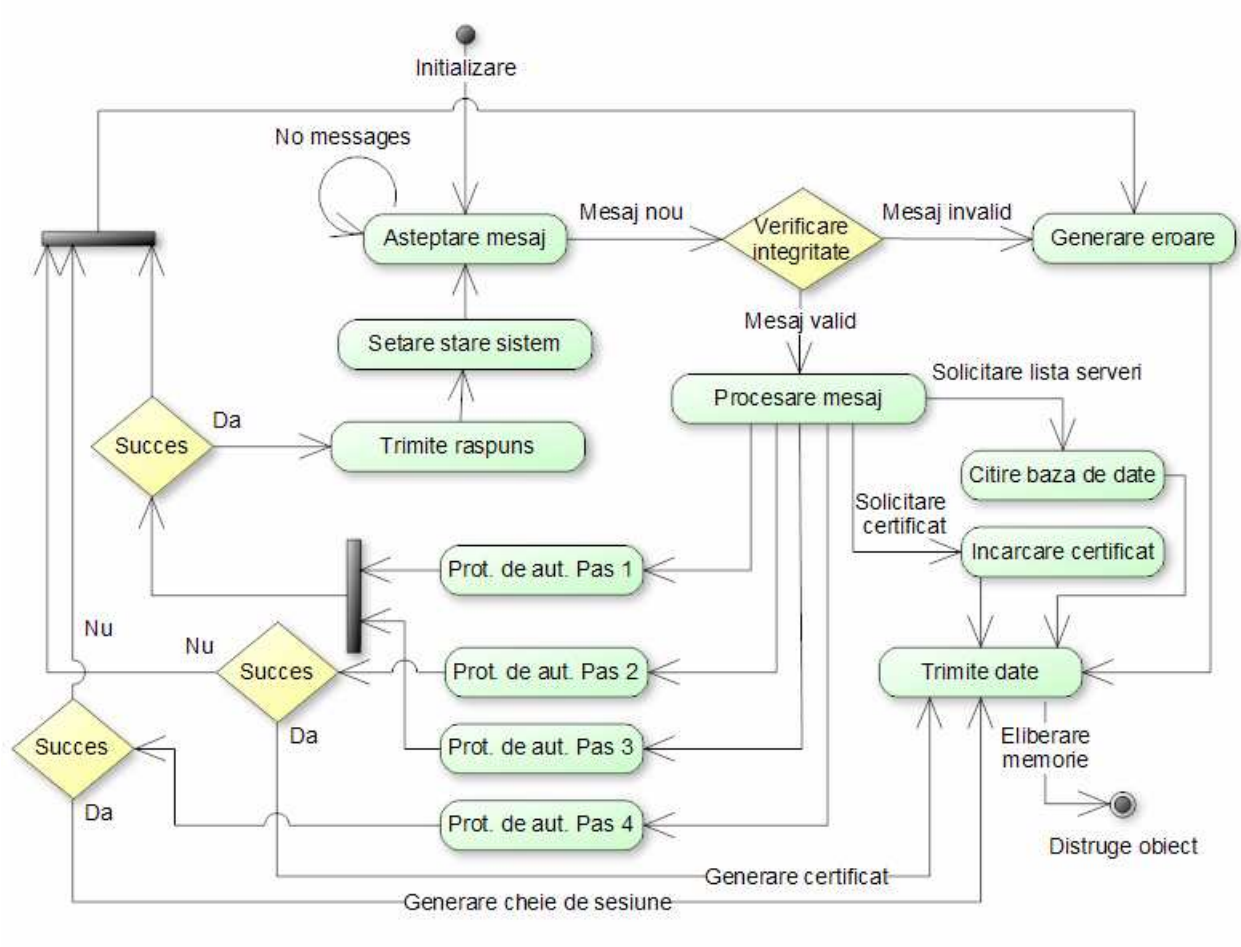

Fig. 22 – Diagrama de stare a execuției serviciului de autentificare

Funcționarea serviciului de autentificare mai detaliat, cu stările sistemului, sunt explicate în continuare. Sistemul inițial este în starea 0, unde se fac inițializările, şi se trece în starea 1. Aici se aşteaptă într-o buclă „infinită” până la recepționarea unui mesaj. Mesajul poate fi unul pentru configurarea sistemului, sau unul de la client, solicitând servicii.

La solicitarea listei cu serveri, se accesează baza de date pentru citirea datelor. Informațiile legate de server sunt formatate pentru trimitere, în formă de token-uri. Aceeaşi lucru se întâmplă şi în cazul solicitării a certificatului. La cerere se încarcă certificatul în memorie, şi se trimite clientului aşa cum este, fiind deja codat cu Base64, permite transportul fără pierderea integrității. În acest timp, serverul este în starea 2. În ambele cazuri, după terminarea procesării, sistemul trece în starea finală unde conexiunea este terminată iar obiectul XAuthSvConnHandler este distrus şi eliberat din memorie.

În cazul autentificării, clientul în primul pas trimite un mesaj exprimându-şi intențiile de autentificare. Serviciul verifică numele clientului, şi folosind *RandomGenerator*-ul, se generează un nonce. Acesta va fi salvat pentru pasul următor. Sistemul se trece iar în starea 1, aşteptând mesaje, dar se salvează progresul autentificării. La recepționarea unui mesaj valid

se trece în starea 4. Aici practic se desfăşoară protocolul de autentificare, paşii detaliați includ: se decodează mesajul, folosind o clasă XBase64Coder, declarat global. Se decriptează mesajul cu cheia privată a serverului, şi se verifică dacă sunt prezente toate token-urile necesare pentru acest pas. Se verifică nonce-ul trimis de client dacă coincide cu cel salvat în pasul anterior. Informațiile de identificare a clientului sunt verificate în baza de date, şi se începe generarea certificatului. Funcția *CertificateCreator::CreateCertificate()* ia toate informațiile necesare pentru a completa câmpurile din certificat. Cheia publică se află în certificat, iar perechea este criptată cu cheia trimisă de client, tot în acest mesaj. La orice eşuare a verificării unui element se termină procesul, şi se generează o eroare pentru client. Datele binare sunt codate cu Base64, şi împreună cu alte informații şi delimitatori, sunt stocate în variabile de tip *std::string*. Se adaugă o semnătură digitală, şi se trimit datele. Sistemul trece în starea finală, eliberând memoria.

Dacă se solicită o autentificare cu certificat, sistemul trece din starea 1 în starea 5. Se generează un nonce, şi se trimite clientului într-un mesaj de acceptare a conexiunii. Sistemul intră iar în starea 1, până la recepționarea unui alt mesaj, care îl pune în starea 6. Se decodează certificatul primit, datele binare sunt stocate în obiecte XByteSequence. Clientul a re-trimis nonce-ul semnat digital, iar pentru verificare se foloseşte cheia publică a clientului extrasă din certificat. Verificarea se face utilizând funcția *DigitalSignatureCreator::veritfySignature()*. Ca şi în pasul anterior, la orice eroare de orice fel, se termină procesul. Pentru verificarea certificatului primit se foloseşte certificatul emitentului. Aici se citesc şi permisiunile clientului, şi în cazul în care acesta nu are privilegii pentru a accesa resursele serverului, se generează un mesaj de eroare, şi se termină conexiunea. Dacă nonce-ul se potriveşte cu cel salvat anterior, se generează cheia de sesiune, se criptează cu cheia secretă a clientului şi este codificat cu Base64. Semnătura serverului este adăugat în mesaj, şi este trimis clientului. Un alt mesaj cu numele utilizatorului şi cheia generată se trimite serviciului de resurse. În acest pas, performanța de autentificare este citită şi înscrisă în variabila *m_Time*, pentru a fi citit la distrugerea obiectului. Sistemul trece în starea finală, şi se distruge obiectul.

4.2.2 Serviciul de resurse

Structura şi funcționarea acestui serviciu este foarte asemănătoare cu cel al serviciului de autentificare. Clasa XResService moşteneşte clasa XBaseService, iar la acceptarea unei noi conexiuni, se creează o instanță a clasei XResSvConnHandler, care moşteneşte clasa

XISvConnHandler. Notificările sunt aceeaşi, singurul lucru care diferă este procesarea mesajelor.

Serviciul se ocupă de distribuirea fişierelor în directoarele partajate. Aceste directoare pot fi setate prin intermediul funcției *setResourceDir()* la cel mai înalt nivel, din API. Acesta va seta membrul *m_resDir*, care va fi citit de fiecare solicitare a listei. Clientul poate cere lista cu fişiere, care va conține numele şi calea fişierelor, cât şi mărimea acestora.

Fiindcă aceste informații sunt trimise criptate, utilizatorii trebuie să se înregistreze înainte de a solicita orice date. Serviciul de autentificare trimite un mesaj intern, cu serviciul de resurse ca destinatar. Acest mesaj este trimis când un client se autentifică cu un certificat, şi se generează o cheie secretă de sesiune. Mesajul va conține numele utilizatorului, şi cheia generată, care odată recepționat de serviciul de resurse va fi stocat într-o structură de tip *std::map*. Orice cerere de informații va conține numele utilizatorului, astfel serviciul încarcă cheia şi o va folosi pentru sesiunea curentă. Dacă numele utilizatorului este incorect, sau cheia nu se potriveşte, mesajul nu va putea fi decriptat, şi se va răspunde cu un mesaj de eroare.

Când se primeşte un mesaj care solicită lista de fişiere, serviciul va parcurge directoarele recursiv, formând răspunsul cu toate datele necesare. Dacă se cere un fişier, se verifică dacă se află în aceste directoare, şi în caz afirmativ, se încarcă în memorie, într-un format *std::vector<char>*. În ambele cazuri datele sunt criptate şi trimise clientului.

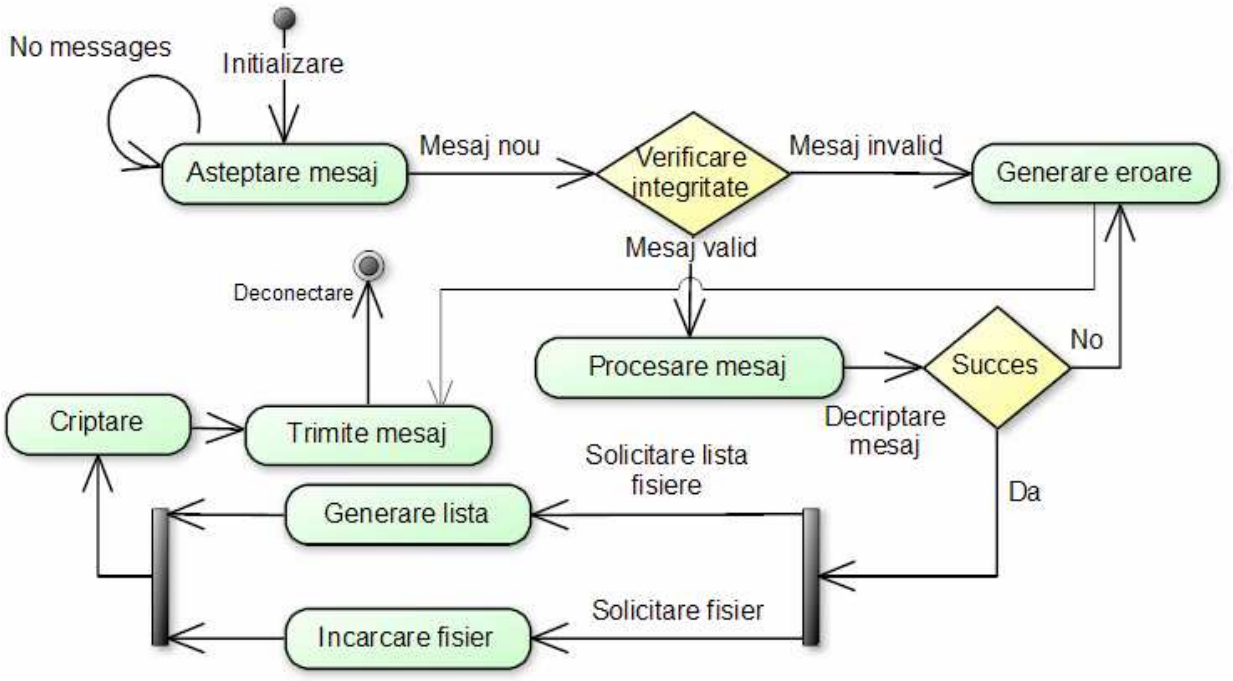

Fig. 23 – Diagrama de stare a execuției serviciului de resurse

4.3 XClient

Partea de client a unui serviciu se creează moştenind clasa XBaseService, care oferă funcțiile callback necesare pentru a primi şi trimite mesaje. Funcționarea şi structura sunt descrie în secțiunea anterioară. În Fig. 24 este prezentată diagrama de stare a clasei XClient.

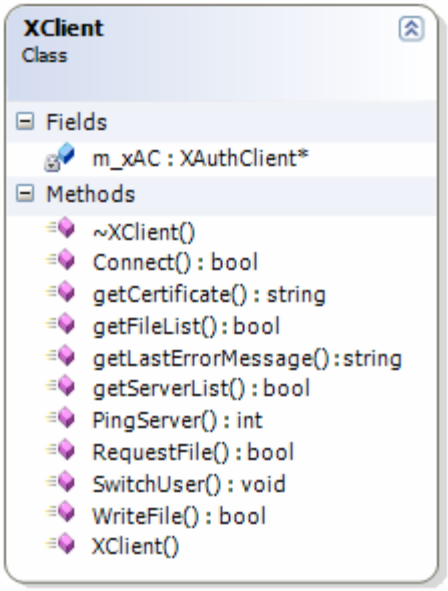

Fig. 24 – Diagram de stare: XClient

Constructorul componentei XClient are 2 parametrii, acestea fiind numele de utilizator şi parola. Aceste date se vor folosi numai pentru identificarea utilizatorului de către client, pentru a încărca certificatul potrivit, iar în cazul în care aceste nu se găseşte, informațiile vor fi folosite pentru a solicita un certificat. Datele private se pot schimba pe parcurs. XClient este controlat prin funcțiile din API-ul global. Majoritatea funcțiilor urmăresc stările prezentate în Fig. 25, adică se trimite solicitarea, se aşteaptă un răspuns, şi se generează rezultatul funcției, fie datele cerute, fie un mesaj de eroare. Unele proceduri se pot efectua fără conectarea la un server. Acestea sunt:

- ***getServerList()***: Se generează şi se trimite un mesaj către serviciul de autentificare a home serverului, care cere lista serverilor din rețea. După trimitere se aşteaptă un răspuns de la server. Aşteptarea răspunsului are un timeout de câteva secunde, dacă nu se primeşte se generează o eroare. Ultima eroare se salvează în variabila globală *m_errMsg*. La recepționarea unui mesaj, se creează un *std::vector* care conține structuri cu informațiile serverilor.
- ***switchUser()***: Se foloseşte în cazul în care se doreşte schimbarea utilizatorului. Dacă clientul a fost conectat la un server, acesta va fi deconectat. Practic se re-inițializează toate datele.
- ***writeFile()***: Funcția scrie un buffer din memorie pe un dispozitiv de stocare fizic folosind funcția *std::ofstream::write()*.
- ***pingServer()***: Simulează un ping, adică trimite un mesaj scurt serverului şi măsoară timpul între trimitere şi primire răspuns. Rezultatul va fi diferența între cele două timpuri.
- ***getLastErrorMessage()***: Returnează conținutul variabilei *m_errMsg*, care conține un mesaj care descrie ultima eroare raportată.

- ***Connect()***: Funcția ia ca parametru adresa serverului la care se dorește conexiunea, și o variabilă care va stoca timpul de conectare. Conectare în acest caz înseamnă autentificare, fie prin parolă, fie prin certificat. Se începe cu crearea directorului cu certificate și chei, dacă acesta nu există. Se verifică dacă există vreun certificat și cheie pe acest nume de utilizator. Dacă există, se începe verificarea validității cu ajutorul clasei *CertificateChecker::VerifyCertificate()*. În cazul în care certificatul sau cheia secretă nu sunt valide, se verifică certificatul home serverului. Dacă acesta nu este salvat local, se solicită de la server, și se așteaptă un răspuns. Dacă nu s-a primit nici un răspuns, se generează o eroare. Diagrama de stare este prezentată în Fig. 25. Odată având certificatul serverului în posesie, începe protocolul de autentificare. La sfârșitul cu succes al acestui protocol, clientul va avea un certificat valid. Acesta va fi stocat în variabila globală *cert_X509*, iar cheia secretă în variabila *cert_sk*. Aceste două obiecte vor fi scris în fișiere folosind funcțiile oferite de API-ul OpenSSL: *PEM_write_PrivateKey* și *PEM_write_PrivateKey*. Având certificatul se contactează serviciul de resurse, și se primește o cheie de sesiune, conform protocolului descris anterior. Cheia este verificată la fiecare solicitare de resurse, iar dacă nu este valid, se generează o eroare, notificând utilizatorul că o reconectare este necesară.

Următoarele funcții se pot apela după autentificare la serverul de resurse:

- ***getFileList()***: Prima dată se verifică dacă utilizatorul este autentificat, și dacă are o cheie validă. Se solicită lista de fișiere după modelul prezentat în Fig. 25. Datele recepționate sunt decriptate, se parcurge lista, și se adaugă fișierele unul câte unul în vectorul specificat de client.
- ***requestFile()***: După verificarea cheii de sesiune, se creează un serviciu pe partea clientului, care se va conecta la server. Se solicită fișierul și se trece în stare de așteptare. Dacă se primește răspuns, se decriptează și se copiază conținutul fișierului în bufferul specificat de utilizator de tip *std::vector<char>*.
- ***getCertificate()***: La conectare la unul dintre serverele din rețea, certificatele sunt stocate în memorie și în fișiere. Aceste certificate sunt folosite automat, dar pentru le a afișa, se poate apela această funcție, care returnează unul dintre trei certificate într-o formă decodată. Se folosește o funcție OpenSSL *X509_print()* pentru a copia conținutul într-un șir de caractere care va fi returnat.

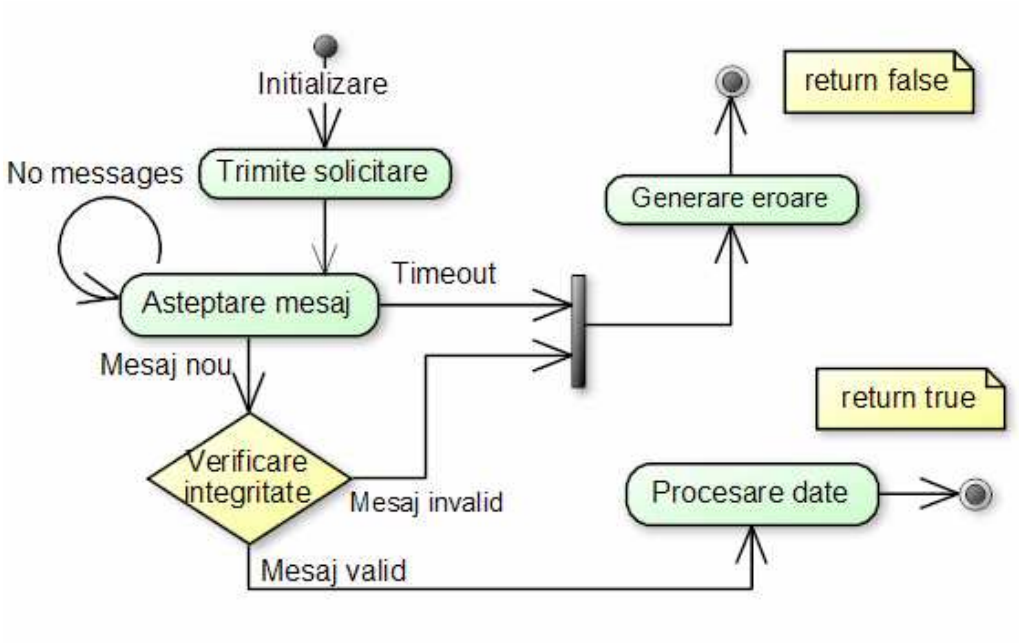

Fig. 25 – Diagrama de stare a execuției cererilor de tip request-response

4.4 Implementarea API-ului

Interfața de programare oferă proiectarea unui sistem descris în această lucrare cu ușurință, prin interfața descrisă în secțiunea 3.3. Interfața grafică a clientului a fost creată cu ajutorul framework-ului Qt [27], implementând funcțiile din interfață pentru a crea serviciile și a interacționa cu serverul. În Fig. 26 se poate vedea o captură de ecran a interfeței de utilizator a clientului, după pornire, afișând lista cu servere, controale pentru reîmprospătare listă, verificare disponibilitate server și conectare.

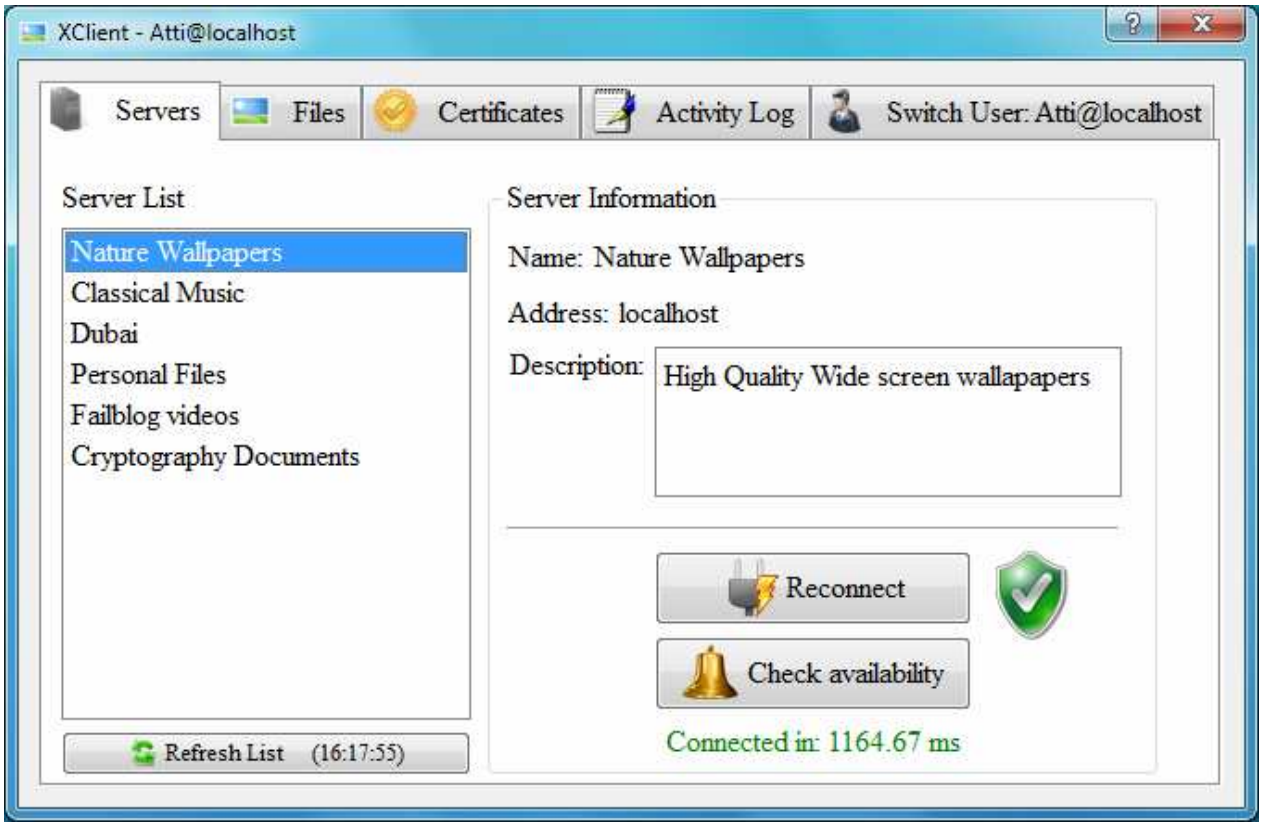

Fig. 26 – GUI client: Lista cu servere

După conectare se afişează lista cu fişiere disponibile şi detalii despre acestea. Clientul permite salvarea fişierelor unul câte unul, salvarea tuturor fişierelor, sau doar deschiderea fişierelor suportate. Qt suportă mai multe tipuri de imagini, iar modulul Phonon suportă şi clipuri video. Pentru a afişa certificatele folosite pentru această sesiune se navighează pe pagina ‘Certificates’, unde se pot solicita toate trei certificate. Pagina ‘Activity Log’ ține o evidență cu toate evenimentele din sesiunea curentă, iar navigând pe pagina ‘Switch User’, se poate schimba utilizatorul curent. Interfața grafică, cu lista de fişierele şi o imagine deschisă se poate vedea în Fig. 27.

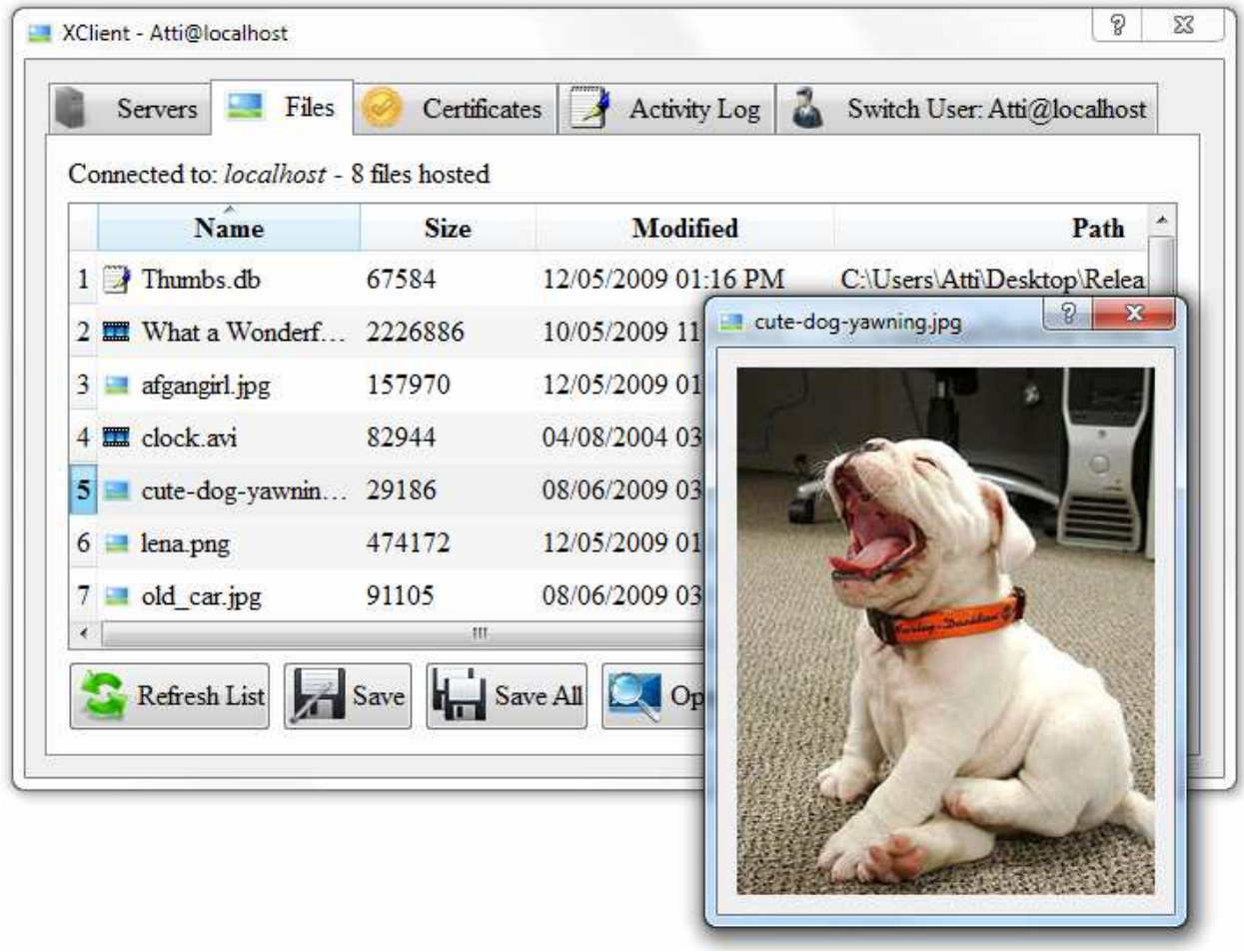

Fig. 27 – GUI client: lista cu fişiere

Serverul are doar interfață consolă, şi afişează evenimentele importante, care includ: conectarea unui client, solicitările de către alți participanți, deconectarea clienților, etc. Serverul ține o evidență a tuturor clienților conectați şi timpul de conectare a acestora într-un fişier local. Interfața serverului se poate vedea în Fig. 28.

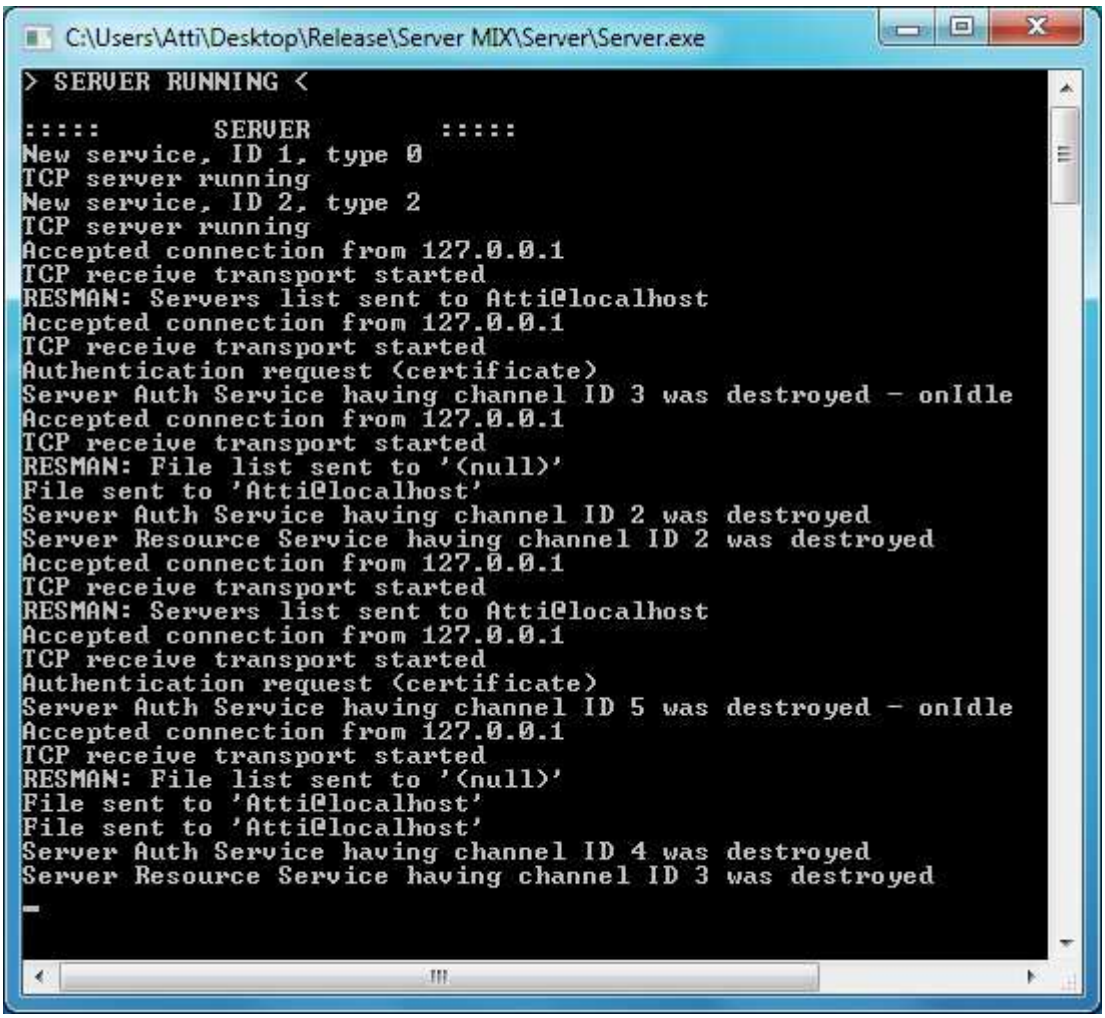

Fig. 28 – Interfața consolă a serverului

Rezultate experimentale

Testele au fost efectuate pe un calculator rulând sistemul de operare Windows XP SP3, procesor dual core cu frecvența 2800 MHz.

În Fig. 29 se văd timpii de autentificare pentru mai mulți clienți simultan. Timpul măsurat a fost intervalul dintre trimiterea cererii de autentificare până la recepționarea cheii de sesiune de la serverul de resurse. Se observă diferența între cele două autentificări, care este produsă de generarea certificatului şi paşii adiționali pentru trimiterea acestuia. Dacă conectarea se face la un server liber, performanțele sunt bune, ajungând la 130 ms pentru autentificare fără certificat şi 270 ms cu certificat. Performanța scade dacă mai mulți clienți se autentifică în acelaşi timp, ajungând la 607 / 1266 ms la 10 clienți simultan. Pentru a îmbunătății performanța unui server populat, se poate adăuga un alt server pentru a distribui autentificarea clienților.

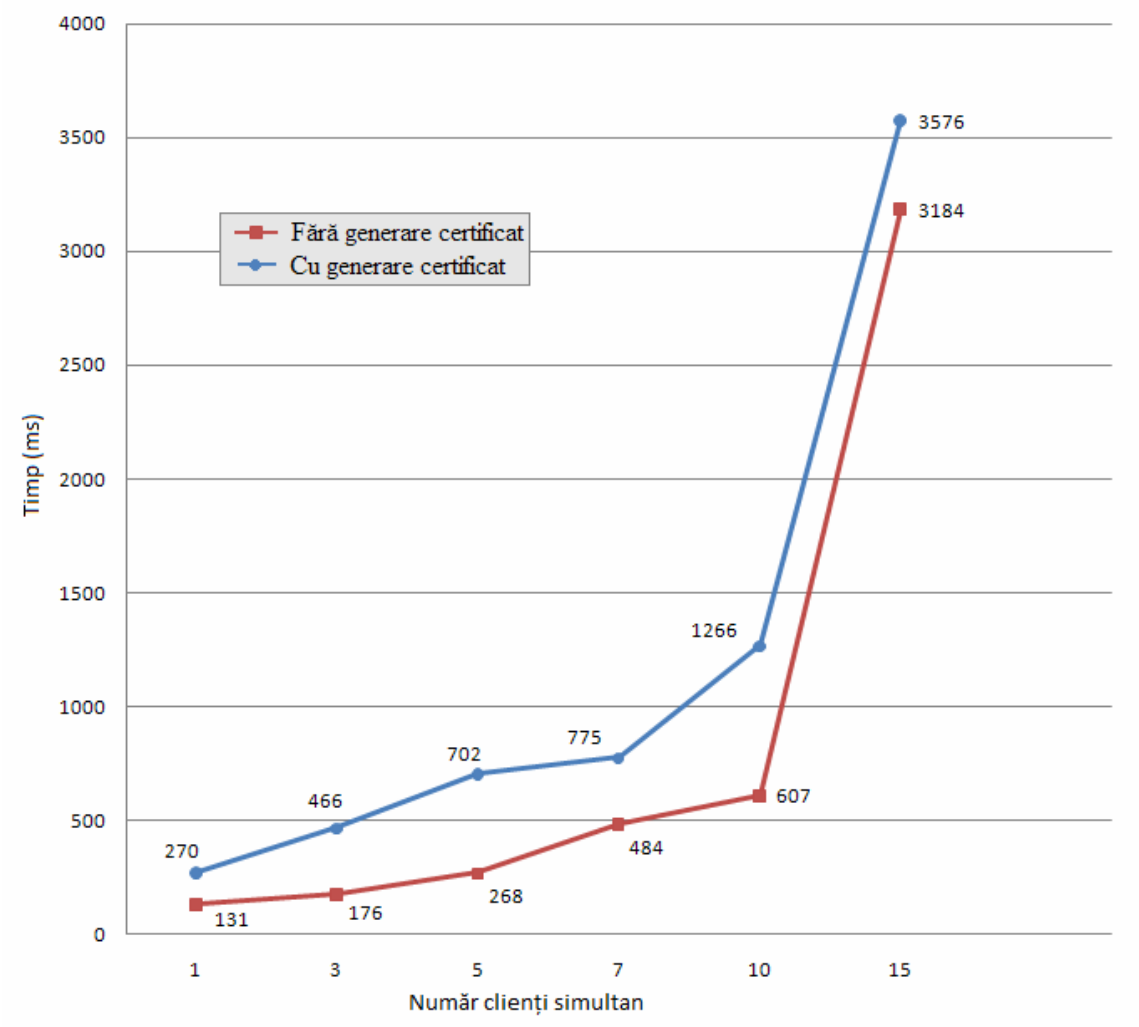

Fig. 29 – Timp de autentificare cu şi fără generare de certificat

În Fig. 30 se vede timpul de generare a cheilor RSA. Această operație necesită cele mai multe resurse ale procesorului. Generarea unei chei durează aproximativ 80 ms, ajungând la aproape 600 ms la 10 generări de chei în paralel. Fiindcă acest proces se bazează mai mult pe puterea de procesare a procesorului, cea mai evidentă metodă de a îmbunătății timpul de generare este de a folosi un procesor mai puternic. O alta metodă ar fi folosirea unui procesor dedicat doar pentru generare de chei, proiectat în special pentru această operație.

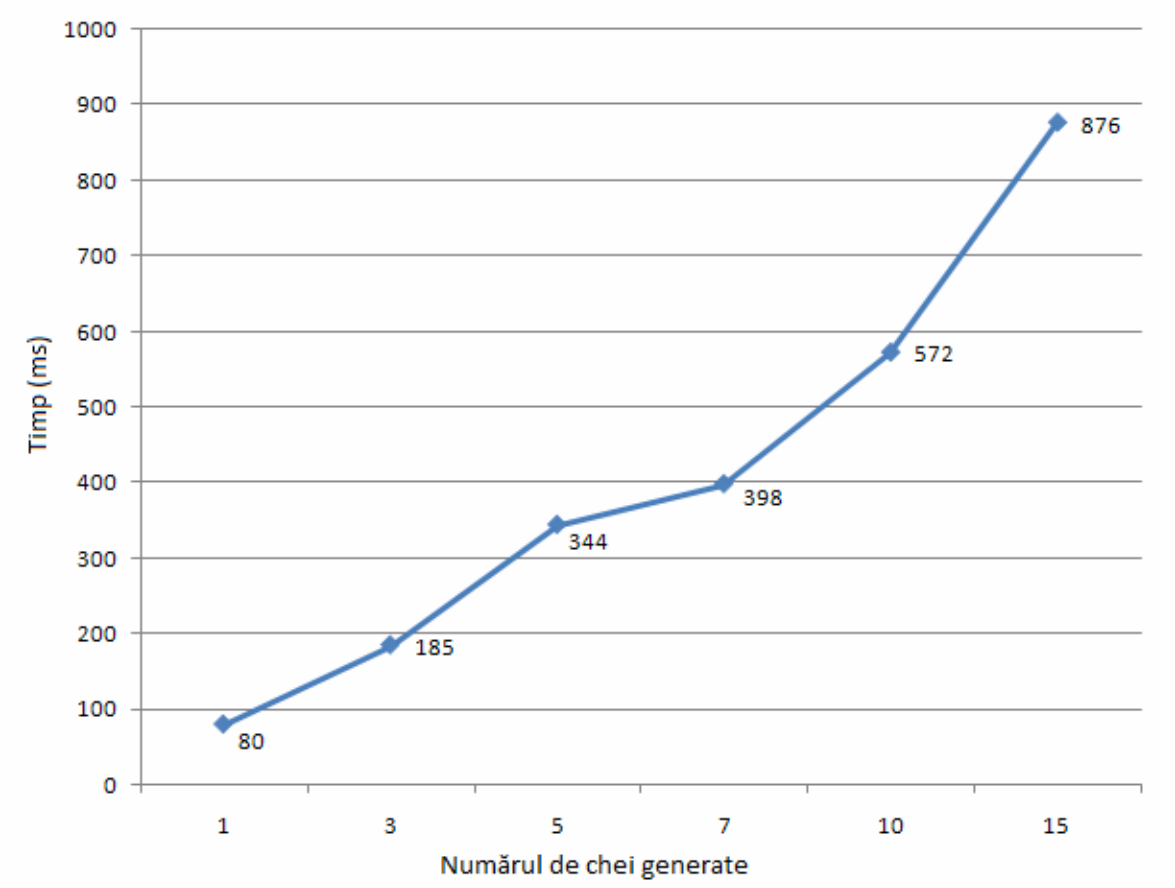

Fig. 30 – Timp de generare a cheilor RSA

Timpul de autentificare este afectat şi de starea serverului, adică dacă este inactiv, doar aşteptând conexiuni, sau dacă se află in cursul procesării datelor. În Fig. 31 se vede diferența dintre aceste două stări. Serverul la care s-a măsurat autentificarea, criptează şi transmite date la alți doi clienți deja conectați. Timpul de autentificarea în acest caz creşte cu 10% în cazul autentificării fără generare certificat, şi cu aproximativ 25% dacă se generează şi certificat pentru client.

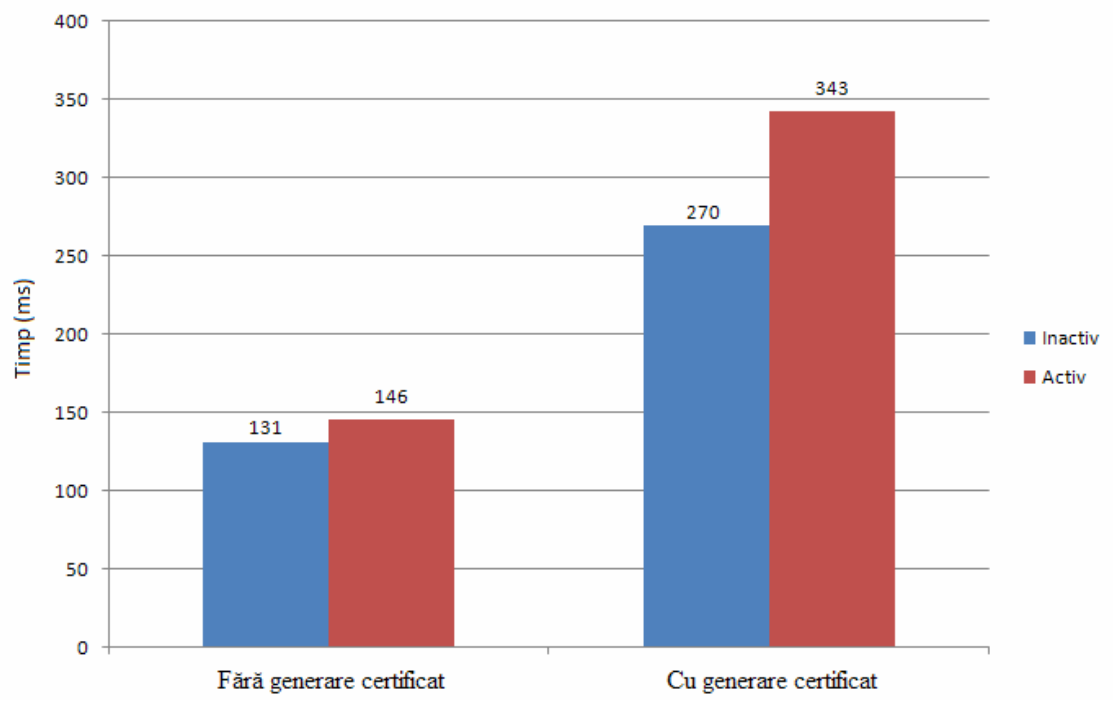

Fig. 31 – Timpul de autentificare la un server activ

Pentru a analiza performanța autentificării, am măsurat fiecare pas al protocolului separat. Pe partea serverului, cum era de aşteptat, generarea certificatului necesită cel mai mult timp de procesare, 45 ms. Cum s-a mai menționat acesta s-ar putea îmbunătății daca am folosi un dispozitiv dedicat pentru generarea cheilor. Generarea nonce-ului durează aproximativ 4 ms, ceea ce e acceptabil. Totuşi pentru a genera numere aleatorii cât mai sigure, se poate înlocui si acesta cu un dispozitiv dedicat care generează nonce-uri. În Fig. 32 se pot vedea durata de execuție a fiecărui pas din protocol.

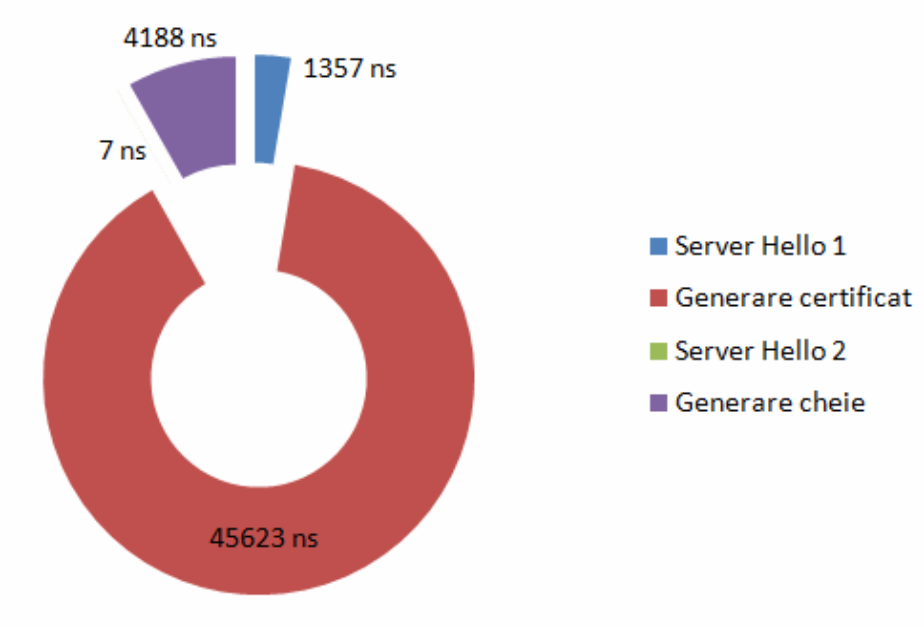

Fig. 32 – Timpul de execuție a protocoalelor pe partea serverului

Pe partea clientului, protocoalele au un pas în plus, adică încărcarea certificatelor şi a cheilor, ceea ce intră la categoria inițializare. Acesta necesită cel mai lung timp de completare, aproximativ 35 ms. Restul protocolului se desfăşoară relativ într-un timp scurt. Aceste măsurători nu includ transportul de date, doar procesarea datelor. Protocolul descompus şi durata de execuție a fiecărei pas se poate vedea în Fig. 33.

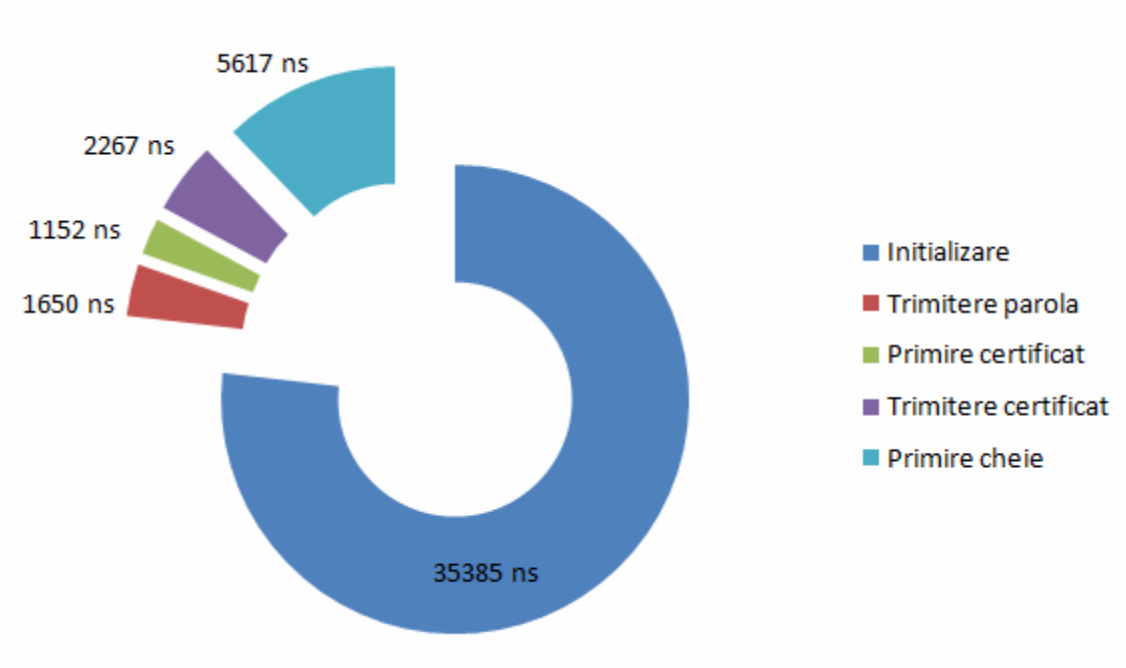

Fig. 33 – Timpul de execuție a protocoalelor pe partea clientului

În Tabelul 2 sunt măsurate autentificarea cu şi fără certificat, şi descărcarea datelor de la servere amplasate în diferite locații externe. Rezultatele, evident, nu arată performanța sistemului, fiindcă cel mai mare factor aici a avut lățimea de bandă şi distanța dintre client şi server. Totuşi au avut un rol important pentru a testa sistemul într-un mediu real, unde s-a comportat conform aşteptărilor. Cel mai important lucru, care depinde mai puțin de viteza de transfer, şi este efectuat mai des, este autentificarea. La autentificare fără certificat deja se observă influența distanței, dar la cel cu certificat se efectuează într-un timp acceptabil, permițând navigarea între servere chiar şi la distanțe de mii de kilometrii.

Tabel 2 – Performanța sistemului

Timp / Locație server	Autentificare fără certificat	Autentificare cu certificat	Descărcare date (50KB)	Descărcare date (500KB)	Descărcare date (6MB)
Local	270 ms	127 ms	61 ms	406 ms	11646 ms
LAN	392 ms	131 ms	73 ms	443 ms	11947 ms
Cluj Napoca, România	478 ms	158 ms	85 ms	475 ms	12005 ms
Ljubljana, Slovenia	914 ms	359 ms	415 ms	3901 ms	39123 ms
Aarhus, Danemarca	678 ms	310 ms	414 ms	1178 ms	13102 ms
Islamabad, Pakistan	3916 ms	1681 ms	8818 ms	87167 ms	-
Wichita, Kansas, SUA	1968 ms	814 ms	492 ms	4416 ms	36926 ms

Concluzii

Sistemul prezentat oferă dezvoltatorilor o interfață cu ajutorul cărei implementarea unei aplicații distribuite pe rețea se face cu uşurință. Sistemul a fost proiectat pentru a oferi siguranța necesară pentru transportul datelor confidențiale într-un mediu nesecurizat, suportând pierderea mesajelor, generarea cheilor şi a certificatelor. Canalele securizate sunt create pe linii de transmisiune normale, cu ajutorul celei mai avansate tehnici de criptografie, şi prin folosirea protocoalelor de securitate re-proiectate. Protocoalele necesită multă putere de procesare, dar arhitectura distribuită a serviciilor garantează acesta.

Arhitectura sistemului elimină serverul central de autentificare, înlocuindu-l cu serviciul de autentificare, care rulează pe fiecare server, distribuind astfel încărcarea. Înregistrarea utilizatorilor poate lua orice formă, oferind flexibilitate şi mai multă securitate. Permisiunile utilizatorilor sunt disponibile fiecărui server la care se conectează, acesta având controlul asupra accesului la diferite resurse. Navigarea între serverele rețelei se face cu uşurință, acesta fiind meritul mecanismului Single Sign-On, care elimină necesitatea re-autentificării la fiecare furnizor de servicii, şi a protocoalelor de securitate proiectate astfel încât aceste autentificări să se efectueze în cel mai scurt timp posibil, ajungând chiar şi la timpuri de 100 ms. Distribuirea serviciilor de autentificare şi de resurse permite scalarea sistemului cu uşurință, adaptându-se nevoilor utilizatorilor.

Platforma fiind dezvoltată după principiile ingineriei programării bazate pe componente este robustă, adaptabilă şi uşor de actualizat. Orice componentă poate fi foarte uşor înlocuită cu o alta cu condiția ca interfața să nu se modifice sau dacă trebuie aduse

modificări în funcționalitatea unei componente acestea se pot face cu un minim de efort. În plus componentele pot fi refolosite şi la dezvoltarea altor sisteme. Componentele fiind încapsulate, adică structura internă este ascunsă şi serviciile sunt oferite printr-o interfață bine definită, face ca programatorul să le poată integra mai uşor fără să ştie cum funcționează acestea.

Noutatea platformei mele constă în folosirea componentelor XPCOM, care permite dezvoltarea aplicațiilor portabile. Folosind platforma Mozilla şi API-ul NSPR pentru implementare, aplicațiile pot rula pe cele trei sisteme mari de operare existente: Microsoft Windows, Linux şi Mac OS, dar şi pe cele mobile, sau oricare unde Mozilla este disponibil.

Soluția propusă pentru sistemul de autentificare este distribuită, dar totuşi s-ar putea ivi probleme în cazul în care un număr mare de utilizatori vor să se conecteze în acelaşi timp. Pentru distribuirea autentificării se poate implementa în viitor un serviciu de balansare care să comunice cu fiecare server şi să repartizeze clienții la serverele mai puțin încărcate, sporind astfel performanța sistemului. Alte îmbunătățiri ar fi implementarea unui serviciu de nume mai performant.

Bibliografie

[1] Marian Nica, *Tehnici de autentificare*, http://itsecure.wordpress.com [Interactiv], 2007

[2] Butler Lampson, Martín Abadi, Michael Burrows, Edward Wobber. *Authentication in Distributed Systems: Theory and Practice.* s.l. : ACM Trans. Computer Systems 10, 1992

[3] Freier, A., O., Karlton, P., Kocher, P., C. *The SSL Protocol, Version 3.0, draft-ietf-tls-sslversion3-00.txt, Internet-Draft.* s.l., Transport Layer Security Working Group, noiembrie 1996

[4] Dierks T., Allen C., *The TLS Protocol, Version 1.0, Request for Comments: 2246.* s.l. : Network Working Group, ianuarie 1999

[5] Joshua D. Guttman, F. Javier Thayer Fabrega, *Authentication tests and the structure of bundles, Theoretical Computer Science, Vol. 283, No. 2, pages 333-380.*, iunie 2002

[6] Guttman, Joshua D., *Security Protocol Design Via Authentication Tests.* s.l. : In Proceedings of the 15th IEEE Computer Security Foundations Workshop, IEEE CS Press, iunie 2002

[7] Project, OpenSSL., available at http://www.openssl.org/ [Interactiv], 2008

[8] L. Hunter, *Active Directory User Guide, Springer-Verlag*, 2005

[9] R. Killpack**,** *eDirectory Field Guide, Springer-Verlag*, 2006

[10] OpenLDAP, versiunea 2.4.15., http://www.openldap.org/ [Interactiv], 2008

[11] Corporation, Mozilla., *XPCOM, Cross Platform Component Model, http://www.mozilla.org/projects/xpcom*, 2008

[12] *AES Algorithm (Rijndael) Information*, http://csrc.nist.gov/archive/aes/rijndael [Interactiv], 28 februarie, 2001

[13] Ronald L. Rivest, Adi Shamir, Leonard M. Adleman, *Communications of the ACM, 21(2):120-126.*, februarie 1978

[14] W. Stallings, *Cryptography and Network Security, 4th edition, Prentice Hall. ISBN 0-13-187319-3.*, 2005

[15] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, ISBN: 0-8493-8523-7, octombrie 1996

[16] M. Abadi, R. Needham, *Prudent Engineering Practice for Cryptographic Protocols, DEC SRC Research Report 125, pag 1 – 22.*, 1994

[17] J. D. Guttman, *Security Protocol Design via Authentication Tests.* 11 aprilie, 2002

[18] J. D. Guttman, *Security goals: Packet trajectories and strand spaces. In R. Gorrieri and R. Focardi, editors, Foundations of Security Analysis and Design, volume 2171 of LNCS.* . s.l. : Springer Verlag, 2001

[19] J.D. Guttman, F.J. Thayer Fabrega, *Protocol independence through disjoint encryption, Proceedings of the 13th IEEE Computer Security, pag. 24-34.* s.l. : Foundations Workshop, Cambridge, 2000

[20] B. Genge, I. Ignat, *Verifying the independence of security protocols, pag 155-163.* s.l. : , IEEE International Conference on Intelligent Computer Comumunication and Processings, 2007

[21] E. Gerck, *Overview of Certification Systems: X.509, PKIX, CA, PGP & SKIP.* s.l. : ISSN 1530-048X, 2000

[22] A. Pashalidis, C. J. Mitchell, *A Taxonomy of Single Sign-On Systems, Volume 2727/2003.* s.l. : Springer Berlin / Heidelberg, 2003

[23] C. Szyperski, *Component Software Beyond Object Oriented Programming, 2nd edition, Addison Wesley, pag. 3 – 47.*, 2002

[24] J. Bloch, *How to Design a Good API and Why it Matters*, noiembrie 2006

[25] Joint Technical Committee ISO/IEC JTCI; International Organization for Standardization; and International Electrotechnical Commission. Programming Languages — *C++*. s.l. : Geneva, Switzerland: ISO/IEC, 1998

[26] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, octombrie 2006

[27] Qt, versiunea 4.5, http://www.qtsoftware.com/products [Interactiv], 2009