Proiectarea și Implementarea unui Server DNS folosind Programare de Sistem

Tuță Dumitru Adi Gabriel
Facultatea de Sisteme Informatice si Securitate Cibernetică
Proiectarea Sistemelor de Operare - Proiect
Academia Tehnica Militara Ferdinand I, România

***Rezumat*—Lucrarea de față descrie procesul de dezvoltare a unui server DNS capabil să gestioneze rezolvări locale de zone, mecanisme de forwarding către rezolvitori externi și politici de caching. Implementarea pune accent pe utilizarea mecanismelor low-level ale sistemului de operare Linux, precum gestiunea proceselor prin apeluri de sistem (fork), memoria partajată pentru IPC și sincronizarea prin semafoare, oferind o soluție eficientă și scalabilă.**

***Index Terms*—DNS, Programare de Sistem, Linux, fork(), mmap(), Semafoare, IPC, Caching.**

I. Introducere

Sistemul de Nume de Domeniu (Domain Name System - DNS) reprezintă una dintre pietrele de temelie ale infrastructurii internetului modern, îndeplinind rolul critic de serviciu de director distribuit. Funcția principală a acestui sistem este translatarea numelor de domenii din format human-readable (ex: www.google.com) — FQDN (Fully Qualified Domain Names) — în adrese numerice IP (Internet Protocol). Acest procedeu este esențial pentru funcționarea serviciilor web, a poștei electronice și a majorității protocoalelor din stiva TCP/IP. Deși la nivel conceptual DNS pare un serviciu simplu de interogare-răspuns, implementarea unui server DNS performant vine cu cerințe stricte în ceea ce privește scalabilitatea, latența scăzută și gestionarea eficientă a resurselor hardware și software.

În cadrul acestui proiect, ne vom apleca asupra detaliilor de proiectare a sistemelor de operare și a programării de sistem în UNIX/Linux. Așadar, în proiectarea unui astfel de server, este necesar să cunoaștem detalii privind gestionarea comunicației utilizând socket-uri, gestionarea atentă a memoriei și a accesului concurent la resurse, sincronizarea accesului la resursele partajate și utilizarea mecanismelor avansate de comunicare inter-proces (IPC – Inter-Process Communication). Un server DNS eficient trebuie să poată deservi simultan un volum ridicat de cereri fără a compromite integritatea datelor sau viteza de răspuns, ceea ce impune utilizarea tehnicilor de paralelizare a proceselor și a firelor de execuție (threads).

Obiectivul principal al acestui proiect nu este doar recrearea funcționalității protocolului DNS definit în RFC 1035, ci și demonstrarea modului în care apelurile de sistem fundamentale pot fi orchestrate pentru a crea un serviciu de rețea performant. Serverul implementat oferă o soluție completă: rezolvarea autoritară a zonelor locale configurate manual (authoritative DNS server), un mecanism de tip „forwarding” către rezolvitori externi (precum Google Public DNS) și o strategie de cache dinamică pentru optimizarea traficului de rețea. Prin această abordare, în proiect se evidențiază simbioza dintre protocoalele de comunicație de nivel înalt și mecanismele interne de gestiune a proceselor și memoriei din nucleul sistemului de operare.

II. Analiza protocolului DNS (RFC 1035)

Pentru a implementa un server DNS, este necesară înțelegerea formatului pachetelor de date și a structurii binare a mesajelor, conform specificațiilor standardelor RFC 1033, 1034 și 1035 [1]. DNS este un protocol de nivel aplicație care, în majoritatea cazurilor, utilizează UDP ca protocol de transport pe portul 53, datorită necesității unei latențe minime.

A. Structura pachetului DNS

Orice mesaj DNS este compus din 5 secțiuni principale, tratate ca un flux continuu de octeți:

1) **Header:** Conține informații de control (ID, flag-uri, numărul de intrări în celelalte secțiuni).
2) **Question:** Interogarea propriu-zisă trimisă de client (numele domeniului și tipul resursei).
3) **Answer:** Răspunsul care conține înregistrările de resurse (Resource Records - RR).
4) **Authority:** Informații despre serverele autoritare pentru domeniul respectiv.
5) **Additional:** Informații suplimentare care ar putea ajuta rezolvitorul.

În proiectul implementat, am acordat o atenție deosebită secțiunii de *Header*. Aceasta are o dimensiune de 12 octeți. Pentru a manipula eficient biții de control (QR, OpCode, AA, TC, RD, RA), am utilizat structuri de tip *bit-fields* în C. Această tehnică permite maparea directă a variabilelor pe poziții bit-cu-bit din pachetul primit din rețea, eliminând operațiile complexe de *shift* și *mask* care s-ar executa în mod normal pe variabile de tip char.

```
struct DNS_HEADER {
    unsigned short id;
    //ordinea bitilor este standard
    unsigned char rd : 1;
    unsigned char tc : 1;
    unsigned char aa : 1;
    unsigned char opcode : 4;
    unsigned char qr : 1;

    unsigned char rcode : 4;
```

```
    unsigned char cd : 1;
    unsigned char ad : 1;
    unsigned char z : 1;
    unsigned char ra : 1;

    unsigned short q_count;
    unsigned short ans_count;
    unsigned short auth_count;
    unsigned short add_count;
}__attribute__((packed));
```

B. Mecanismul de compresie a numelor (Pointeri)

O provocare tehnică majoră în parsarea DNS este gestionarea mecanismului de compresie. Numele de domenii (ex: www.google.com) pot fi repetitive într-un singur pachet. Pentru economie de spațiu și eliminarea duplicatelor, în protocolul DNS se utilizează pointeri (offset-uri de 2 octeți care încep cu biții 11). În programul implementat, este inclusă o funcție iterativă (`read_dns_name`) capabilă să urmărească acești pointeri în bufferul de memorie pentru a reconstrui complet numele domeniului, gestionând corect lungimile variabile ale label-urilor.

C. Tipuri de înregistrări și semantica TTL

Proiectul suportă trei tipuri fundamentale de înregistrări, fiecare cu o semantică diferită în procesul de rezolvare:

- **A (Address):** Maparea unui nume de domeniu la o adresă IPv4 de 32 de biți.
- **CNAME (Canonical Name):** Crearea unui alias pentru un alt nume de domeniu.
- **MX (Mail Exchanger):** Identificarea serverelor de mail, incluzând un câmp suplimentar pentru prioritate.

Fiecare înregistrare este însoțită de un parametru **TTL (Time To Live)**. Din perspectiva sistemului de operare, acest parametru determină durata de viață a unei resurse în cache-ul partajat. Gestionarea corectă a acestuia este crucială: un TTL prea scurt generează trafic excesiv către exterior, în timp ce un TTL prea lung poate duce la utilizarea unor date învechite (*stale data*).

D. Considerații privind proiectarea sistemului

Proiectarea unui astfel de server necesită o abordare multidisciplinară care depășește simpla manipulare a socket-urilor de rețea. Aceasta implică gestionarea riguroasă a concurenței, sincronizarea accesului la resurse partajate și utilizarea unor mecanisme avansate de comunicare între procese (IPC - Inter-Process Communication). Un server DNS eficient trebuie să poată deservi simultan un volum ridicat de cereri fără a compromite integritatea datelor sau viteza de răspuns, fapt ce impune utilizarea unor tehnici precum paralelizarea prin procese multiple sau fire de execuție (*threads*).

III. Programare de Sistem - Concepte Utilizate

Pe lângă funcționalitatea de rețea, acest proiect este gândit ca un exercițiu de utilizare eficientă a resurselor pe care kernel-ul Linux le pune la dispoziția programatorului [2]. Un server DNS este departe de a fi un program secvențial; design-ul său trebuie să asigure că alți clienți nu sunt nevoiți să aștepte până când cererea unui alt utilizator este rezolvată. În acest scop, am utilizat mecanisme de sistem avansate, precum programarea multiproces și utilizarea firelor de execuție (threads) pentru deservirea paralelă a cererilor.

A. Gestiunea concurenței prin procese (`fork()`)

În implementarea serverului, am utilizat apelul de sistem `fork()` pentru fiecare cerere DNS primită, asigurând astfel izolarea și simplitatea execuției. În momentul în care serverul părinte primește un pachet prin `recvfrom`, acesta nu îl procesează în aceeași buclă, ci creează un nou context prin apelul `fork()`, dând naștere unui proces fiu care are acces la aceleași resurse.

Avantajele unei astfel de abordări sunt:

- **Izolarea:** Dacă un proces copil întâmpină o eroare (de exemplu, un pachet corupt care cauzează un crash), serverul principal rămâne intact și poate deservi în continuare alți clienți.
- **Partajarea resurselor:** Copilul moștenește automat descriptorul de fișier al socket-ului, putând trimite răspunsul înapoi către client direct, fără a mai comunica cu părintele.

Utilizarea proceselor în detrimentul thread-urilor este justificată de siguranță: o eroare fatală într-un thread ar genera un semnal ce ar putea opri întregul proces părinte (`exit()`), anulând astfel toate cererile aflate în curs de prelucrare.

Totuși, procesele implică gestionarea proceselor ”zombie” (procese care și-au terminat execuția, dar încă ocupă un loc în tabela de procese deoarece părintele nu le-a colectat starea). Pentru a rezolva acest lucru, am implementat un handler de semnal: `signal(SIGCHLD, handle_sigchild)`. Astfel, de fiecare dată când un proces copil se termină, nucleul notifică părintele, care apelează `waitpid` în mod non-blocant (`WNOHANG`), curățând resursele instantaneu.

B. Memoria partajată (`mmap`) și comunicarea inter-procese (IPC)

O provocare majoră în modelul cu procese multiple este izolarea spațiului de adrese. Dacă un proces copil primește un răspuns de la un server extern (ex: Google DNS, 8.8.8.8) și salvează datele în cache, această informație s-ar pierde odată cu terminarea procesului respectiv, nefiind disponibilă celorlalte procese.

Pentru a depăși această barieră, am utilizat un mecanism de Comunicare între Procese (IPC) numit **memorie partajată**. Folosind apelul `mmap` [3] cu flag-urile `MAP_SHARED` și `MAP_ANONYMOUS`, am rezervat o porțiune de memorie comună gestionată de kernel. În acest segment am stocat structura `SharedMemory`, care conține tabela de cache. Aceasta este zona în care procesele ”comunică”: dacă un proces a aflat IP-ul pentru un domeniu, următoarele procese îl vor citi direct din această memorie comună.

```
SharedMemory* init_shared_memory() {
    SharedMemory* shm = mmap(NULL, sizeof(
        SharedMemory), PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

```
    if (shm == MAP_FAILED) {
        perror("Eroare alocare memorie cu mmap.\n");
        exit(1);
    }
    if (sem_init(&shm->mutex, 1, 1) < 0) {
        perror("Initializare semafor esuata\n");
        exit(1);
    }
    shm->total_requests = 0;
    for (int i = 0; i < MAX_CACHE_SIZE; i++) {
        shm->cache[i].is_valid = 0;
    }
    return shm;
}
```

Listing 2. Functie pentru alocarea de memorie partajata

C. Sincronizarea prin semafoare POSIX

Centrul de greutate al proiectului constă în mecanismul de gestiune a accesului concurent. Memoria partajată este rapidă, dar susceptibilă la *Race Conditions*. Dacă două procese încearcă să scrie în același slot din cache simultan, datele ar fi corupte.

Pentru a contracara acest risc, am introdus un semafor de tip Mutex (`sem_t`), mecanism detaliat în literatura de specialitate [2], plasat în memoria partajată. Un proces care dorește să acceseze cache-ul trebuie să obțină semaforul prin `sem_wait`. Dacă semaforul este ocupat, procesul așteaptă la rând. După finalizarea operațiunii, eliberează semaforul prin `sem_post`. Această disciplină garantează consistența tabelei de cache indiferent de volumul de cereri simultane.

```
// Cautare in cache
sem_wait(&shm->mutex);
shm->total_requests++;
    for (int i = 0; i < MAX_CACHE_SIZE; i++) {
        if (shm->cache[i].is_valid && strcmp(shm->
            cache[i].domain, (char*)domain_name) ==
             0) {
            cache_slot = i;
            break;
        }
    }
sem_post(&shm->mutex);
```

Listing 3. Mecanism semafoare (mutex) pentru acces in cache

D. Monitorizarea asincronă prin fire de execuție (Pthreads)

Pe lângă procesele efemere, serverul necesită un sistem de mentenanță continuă pentru verificarea timpului de viață (TTL) al înregistrărilor. Un proces obișnuit nu este adecvat pentru această sarcină deoarece se termină după trimiterea răspunsului.

Serverul creează un thread în procesul părinte la pornire, folosind `pthread_create`. Acest thread rulează în fundal pe toată durata de viață a serverului. Din 5 în 5 secunde, acesta scanează memoria partajată și verifică dacă vreo intrare din cache a expirat (comparând timpul curent cu *timestamp-ul* creării). Dacă o intrare este depășită, thread-ul o marchează ca invalidă. Utilizarea thread-urilor este ideală aici, fiind un mecanism mai „ușor” decât un proces, având acces implicit la memoria procesului părinte și facilitând monitorizarea fără consum excesiv de resurse.

```
void* monitor_ttl_thread(void* arg) {
    SharedMemory* shm = (SharedMemory*)arg;
    printf("[THREAD] Monitor TTL pornit.\n");
    while (1) {
        sleep(5);       //verificare din 5 in 5
            secunde
        time_t now = time(NULL);
        int expired_count = 0;

        sem_wait(&shm->mutex);
        for (int i = 0; i < MAX_CACHE_SIZE; i++) {
            if (shm->cache[i].is_valid) {
                double elapsed = difftime(now, shm->
                    cache[i].created_at);
                if (elapsed > shm->cache[i].ttl) {
                    printf("[THREAD] Domeniul %s a
                        expirat. Eliminare din cache
                        .\n", shm->cache[i].domain);
                    shm->cache[i].is_valid = 0;
                    expired_count++;
                }
            }
        }
        sem_post(&shm->mutex);

        //Afisaz cand am facut curatenie
        if (expired_count > 0)
            printf("[THREAD] Curatenie: %d intrari
                sterse.", expired_count);

    }
    return NULL;
}
```

Listing 4. Functie monitorizare TTL inregistrari in cache

IV. Arhitectura domeniilor și gestiunea zonelor DNS

Sistemul de Nume de Domeniu (Domain Name System) nu este doar o bază de date centralizată, ci un sistem distribuit, structurat sub forma unui arbore inversat. Înțelegerea modului în care proiectul interacționează cu această ierarhie este esențială pentru a evidenția valoarea implementării de față ca un server hibrid: autoritar pentru propriile zone și recursiv prin delegare (*forwarding*) pentru restul internetului.

A. Ierarhia domeniilor și a subdomeniilor

În vârful ierarhiei DNS se află *Root Zone* (reprezentată printr-un punct în literatura de specialitate). Sub aceasta se află domeniile de nivel superior (TLD – *Top Level Domains*), precum .com, .org sau .ro. Subdomeniile sunt extensii ale acestor domenii (ex: google.com), iar acestea, la rândul lor, pot avea mai multe subdomenii (ex: mail.google.com).

În proiectul realizat, serverul tratează numele de domenii ca pe un șir de caractere structurat. O provocare majoră în programarea de sistem a fost conversia acestui format „uman” în formatul binar DNS. În pachetele de rețea, lungimea fiecărui segment (*label*) precede segmentul respectiv (ex: `3www6google3com0`). Funcția `dn_format`, implementată în codul sursă, realizează exact această transformare critică, asigurându-se că răspunsurile trimise către clienți sunt conforme cu standardele internaționale.

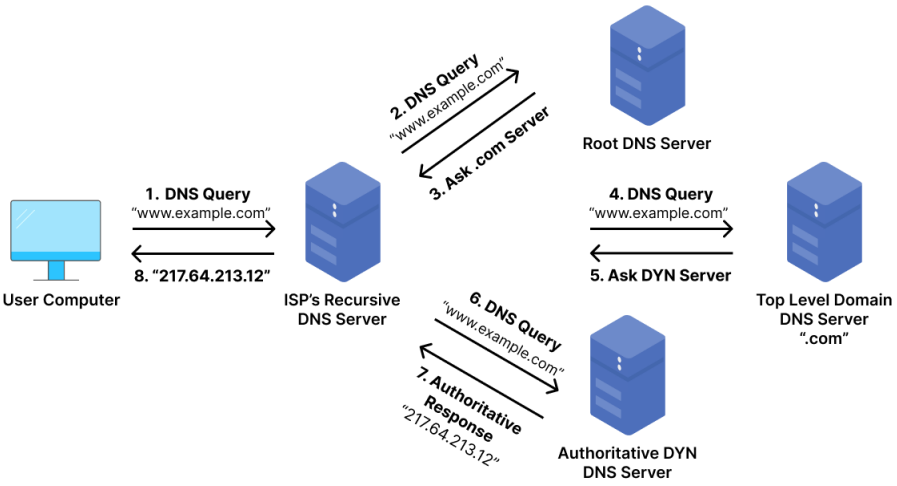

Figura 1. Fluxul standard de rezolvare a unei interogări DNS într-o ierarhie distribuită.

B. Conceptul de zonă DNS (Generalități și implementare locală)

O zonă DNS reprezintă o porțiune administrativă a arborelui de domenii pentru care un server este „autoritar" (deține informația certă, nu doar o copie temporară).

În cadrul proiectului, gestionarea zonelor este realizată prin funcția `load_zones()`. La pornire, serverul citește un fișier de configurare extern (`zones.txt`), care acționează ca bază de date locală.

- **La nivel general:** O zonă poate conține sute de înregistrări și delegări către alte sub-zone.
- **În proiect:** Am implementat o structură de date `DNSRecord` care stochează atributele esențiale: domeniul, tipul (A, CNAME, MX), valoarea (IP sau alias), prioritatea și TTL-ul. Această structură este încărcată într-un tablou static (`local_zones`), oferind o viteză de căutare extrem de mare pentru domeniile gestionate local.

```
typedef struct {
    char domain[256];
    char type[10];              // A, CNAME, MX
    char value[256];            //IP sau numele
        domeniului tinta
    int priority;
    time_t created_at;
    int ttl;
    int is_valid;
}DNSRecord;
```

Listing 5. Structura de date DNSRecord

C. Fluxul de rezolvare și logica de decizie

Când serverul primește o interogare, acesta parcurge o ierarhie de decizie logică, implementată riguros în procesul copil creat prin `fork()`:

1) **Verificarea cache-ului (Memoria partajată):** Înainte de orice altceva, procesul accesează segmentul de memorie partajată protejat de semafor. Dacă domeniul a fost solicitat recent și încă se află în fereastra de validitate (TTL), răspunsul este extras direct. Aceasta simulează comportamentul unui server DNS de tip *Caching-only*.
2) **Căutarea în zonele locale (Autoritar):** Dacă datele nu există în cache, serverul caută în `local_zones`. Dacă găsește domeniul, setează flag-ul **AA (Authoritative Answer)** în header-ul DNS, marcând faptul că serverul este sursa oficială a informației.
3) **Mecanismul de forwarding (Delegarea):** Dacă domeniul este necunoscut (ex: yahoo.com), serverul acționează ca un rezolvitor. Acesta creează un nou socket UDP și interoghează un server de nivel superior (ex: Google DNS la 8.8.8.8). După primirea răspunsului extern, serverul îl transmite clientului original și îl salvează simultan în memoria partajată.

```
 //Facem cautarea
int found_idx = -1;      // index pentru local zones
int cache_slot = -1;     // index in cache

// Cautare in cache
sem_wait(&shm->mutex);
shm->total_requests++;
    for (int i = 0; i < MAX_CACHE_SIZE; i++) {
        if (shm->cache[i].is_valid && strcmp(shm->
            cache[i].domain, (char*)domain_name) ==
             0) {
            cache_slot = i;
            break;
        }
    }
sem_post(&shm->mutex);
```

Listing 6. Verificare cache si gasirea intrarii corespunzatoare

```
  //2. Cautare in zonele locale
if (cache_slot == -1) {
    for (int i = 0; i < zones_count; i++) {
        if (strcmp(local_zones[i].domain, (char*)
            domain_name) == 0) {
            found_idx = i;
            break;
```

```
        }
    }
}
```

Listing 7. Cautare in zonele locale

```
 printf("[FORWARD] %s -> 8.8.8.8\n", domain_name);
struct sockaddr_in dest;
memset(&dest, 0, sizeof(dest));
dest.sin_family = AF_INET;
dest.sin_port = htons(53);
inet_aton("8.8.8.8", &dest.sin_addr);

int fs = socket(AF_INET, SOCK_DGRAM, 0);
struct timeval tv = { 4,0 };
setsockopt(fs, SOL_SOCKET, SO_RCVTIMEO, (char*)&tv,
     sizeof tv);

sendto(fs, buffer, n, 0, (struct sockaddr*)&dest,
    sizeof(dest));


char gbuf[BUFF_SIZE];
socklen_t dlen = sizeof(dest);
int rlen = recvfrom(fs, gbuf, BUFF_SIZE, 0, (struct
     sockaddr*)&dest, &dlen);
close(fs);
......
```

Listing 8. Forwarding

D. Suport pentru tipuri de resurse (Resource Records)

Proiectul implementează trei dintre cele mai importante tipuri de înregistrări:

- **Tipul A (Address):** Returnează adresa IPv4 asociată numelui.
- **Tipul CNAME (Canonical Name):** Returnează un „alias”. Implementarea necesită utilizarea funcției `dn_format` pentru a coda numele destinație în pachetul de răspuns.
- **Tipul MX (Mail Exchange):** Utilizat pentru rutarea e-mail-urilor. Am integrat câmpul de prioritate specific, demonstrând o înțelegere avansată a structurii pachetelor DNS.

V. Detalii de implementare la nivel de octet

A. Gestionarea ordinii octeților (Endianness)

Deoarece protocolul DNS este utilizat într-o rețea eterogenă, ordinea octeților (Network Byte Order - Big Endian) diferă de ordinea utilizată de procesorul sistemului gazdă (Host Byte Order - de regulă Little Endian pe arhitecturi x86). În implementare, am utilizat funcțiile din familia `htons()`, `ntohs()`, `htonl()` și `ntohl()` pentru a asigura conversia corectă a câmpurilor de tip *short* (16 biți) și *long* (32 biți) din header-ul DNS și din secțiunile de răspuns.

B. Alinierea structurilor (Packed Structures)

În mod implicit, compilatorul C introduce octeți de „padding” pentru a alinia membrii unei structuri la adrese de memorie pare. Totuși, la parsarea pachetelor de rețea, avem nevoie ca structura să fie o reprezentare exactă, bit cu bit, a fluxului de date. Utilizarea directivei `__attribute__((packed))` este esențială pentru a preveni interpretarea eronată a câmpurilor precum `type` sau `class` din structura `R_DATA`.

VI. Analiza Implementării Tehnice

Pentru a asigura conformitatea cu RFC 1035 [1], am utilizat directiva $_{a}ttribute_{((}packed))$ pentru structurile de date ce mapează pachetele binare.

```
struct DNS_HEADER {
    unsigned short id;
    unsigned char rd : 1;
    unsigned char tc : 1;
    unsigned char aa : 1;
    unsigned char opcode : 4;
    unsigned char qr : 1;
    // ...
} __attribute__((packed));
```

Listing 9. Implementarea Header-ului DNS

VII. Rezultate Experimentale și Testare

Pentru validarea serverului, am utilizat utilitarul profesional dig (Domain Information Groper).

A. Testarea Rezolvării Locale (Zone Hit)

Prin interogarea unui domeniu definit în zones.txt, serverul a returnat răspunsul marcat cu flag-ul *Authoritative Answer (aa)*. Timpul de răspuns a fost sub 1ms, demonstrând eficiența căutării în tabelul static.

B. Testarea Mecanismului de Caching

Am efectuat două interogări succesive pentru domeniul google.com:

- **Prima cerere (Forwarding):** Serverul a redirecționat cererea către 8.8.8.8. Timp de răspuns: $\approx$ 30-50ms.
- **A doua cerere (Cache Hit):** Serverul a servit datele din memoria partajată. Timp de răspuns: $<$ 2ms.

Acest experiment confirmă funcționarea corectă a semafoarelor și a segmentului de memorie mmap.

VIII. Concluzii

Proiectul demonstrează utilizarea eficientă a apelurilor de sistem UNIX pentru a crea un serviciu de rețea stabil. Integrarea memoriei partajate și a sincronizării între procese independente a permis crearea unui sistem de caching performant, similar cu cel utilizat în serverele DNS profesionale.

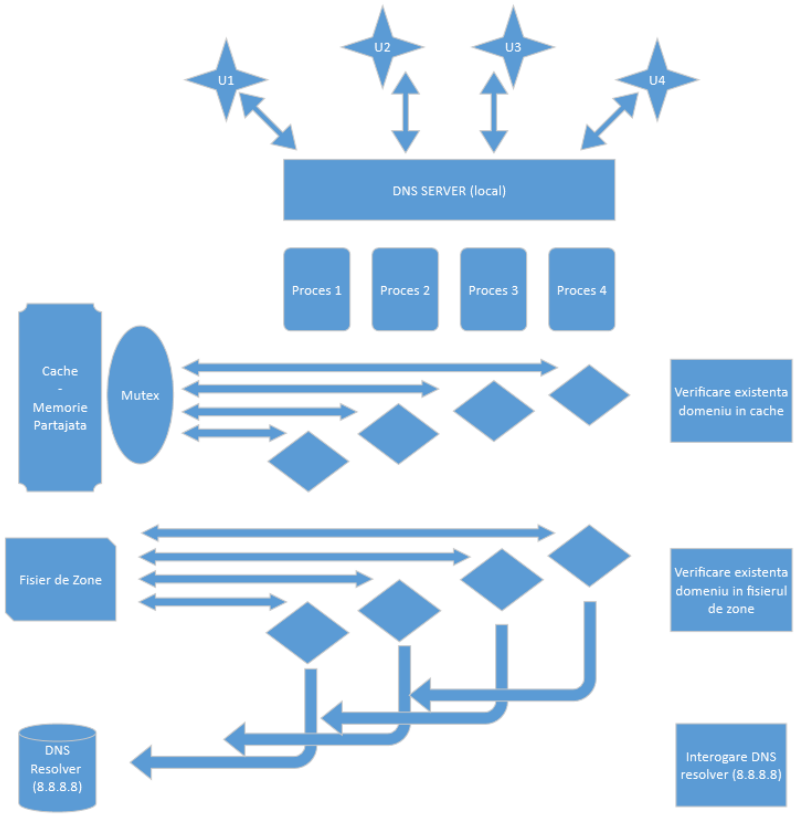

Figura 2. Arhitectura sistemului implementat: Interacțiunea dintre Procese, Memoria Partajată (Cache) și Logica de Decizie.

BIBLIOGRAFIE

[1] P. Mockapetris, ``Domain Names - Implementation and Specification,'' RFC 1035, November 1987.
[2] W. Richard Stevens, ``UNIX Network Programming, Volume 1: The Sockets Networking API,'' 3rd ed. Addison-Wesley, 2003.
[3] Linux Programmer's Manual, ``mmap(2) - map files or devices into memory''.