Server DHCP: Implementarea Arhitecturii MVCS și Validarea prin Scenarii de Simulare

Onofrei Sonia
Facultatea de Calculatoare și Sisteme Informatice pentru Securitate și Apărare Națională
Academia Tehnică Militară
București, România

Cringasu Andreea
Facultatea de Calculatoare și Sisteme Informatice pentru Securitate și Apărare Națională
Academia Tehnică Militară
București, România

***Abstract*—Această lucrare documentează proiectarea și implementarea unui sistem distribuit complex care emulează protocolul DHCP. Proiectul se distinge prin maparea riguroasă a fișierelor sursă pe o arhitectură MVCS (Model-View-Controller-Service), asigurând modularitatea. Lucrarea analizează în profunzime gestionarea concurenței prin Thread Pools și Procese Forked, comunicarea inter-proces prin Shared Memory și Message Queues, și validează robustețea sistemului prin trei scenarii de simulare distincte: Happy Path, Stress Test și Chaos Engineering (Fault Injection).**

***Index Terms*—MVCS Architecture, Chaos Engineering, Linux IPC, Shared Memory, POSIX Threads, System V Semaphores, Bash Scripting, DHCP Protocol.**

I. Introducere și Fundamente Tehnice

Dezvoltarea aplicațiilor de rețea la nivel de sistem necesită o gestionare atentă a resurselor și o înțelegere profundă a stivei TCP/IP. Protocolul DHCP (*Dynamic Host Configuration Protocol*), deși standardizat în RFC 2131 [1], implică provocări majore de sincronizare în implementare, mai ales atunci când serverul trebuie să gestioneze sute de clienți simultan într-un mediu concurent.

Această lucrare propune o soluție care nu doar implementează protocolul, ci construiește un cadru de testare și monitorizare complet, utilizând primitivele native ale Kernelului Linux descrise în [2]. Sistemul rezultat abordează probleme critice precum condițiile de cursă (*Race Conditions*) și persistența datelor, oferind o arhitectură modulară și tolerantă la erori.

A. Protocolul DHCP: Mecanism și Tipuri de Pachete

DHCP funcționează pe baza modelului Client-Server, utilizând protocolul de transport UDP (*User Datagram Protocol*) datorită naturii sale ”connectionless”. Deoarece un client aflat la pornire nu are o adresă IP configurată, acesta nu poate stabili o conexiune TCP (care necesită un handshake SYN-ACK). În schimb, DHCP se bazează pe difuzare (*Broadcast*) la nivel de rețea locală.

Procesul de negociere a unei adrese IP, cunoscut sub acronimul **DORA**, implică patru tipuri fundamentale de pachete, pe care le-am implementat fidel în structura `DHCP_Message`:

1) **DHCPDISCOVER (Client → Broadcast):** Mesajul inițial de descoperire. Clientul setează flag-ul de broadcast și generează un ID de tranzacție (`xid`) aleatoriu. Acest pachet este trimis către adresa `255.255.255.255` pe portul 67 (sau 9000 în simularea noastră).
2) **DHCPOFFER (Server → Client):** Serverul primește cererea, verifică disponibilitatea în `ip_pool` (protejat prin Mutex conform [**?**]) și rezervă temporar o adresă. Răspunsul conține adresa IP propusă (`yiaddr`) și durata închirierii (`lease_time`).
3) **DHCPREQUEST (Client → Broadcast):** Clientul acceptă oferta. Deși a primit o ofertă, trimite cererea tot prin broadcast pentru a notifica eventualele alte servere DHCP din rețea că oferta lor a fost refuzată (eliberând astfel resursele rezervate de acestea).
4) **DHCPACK (Server → Client):** Confirmarea finală. Serverul marchează IP-ul ca fiind alocat definitiv în baza de date persistentă (`leases.txt`) și trimite parametrii finali de configurare. Din acest moment, clientul intră în starea `BOUND`.

Pe lângă fluxul standard, proiectul implementează și mesajele de întreținere:

- **DHCPRELEASE:** Trimis voluntar de client pentru a renunța la IP înainte de expirare.
- **DHCPNAK:** Trimis de server dacă cererea clientului este invalidă (ex: IP-ul cerut este deja ocupat de altcineva).

Structura pachetelor respectă alinierea la memorie specificată în [1], asigurând compatibilitatea binară între procesele Client și Server, chiar și atunci când acestea rulează pe fire de execuție separate.

II. Arhitectura Sistemului: Implementarea MVCS

Spre deosebire de aplicațiile monolitice, am distribuit codul sursă în patru straturi logice, mapând fiecare fișier pe responsabilitatea sa exactă.

A. Maparea Fișierelor pe Straturi

Tabelul I prezintă distribuția componentelor software.

B. Diagrama Fluxului de Date

Figura 1 ilustrează cum datele circulă prin aceste straturi.

III. Micro-Analiza Componentei Client: Arhitectură Hibridă Multi-Thread/Multi-Proces

Clientul (`client.c`) reprezintă o inovație arhitecturală în cadrul proiectului, combinând granularitatea fină a firelor

TABLE I: Distribuția Componentelor în Arhitectura MVCS

Strat	Fișiere Sursă	Responsabilitate Tehnică
MODEL	`dhcp_message.h` `config.h`	Structura binară a pachetului (Header). Structura configurației IP a serverului.
VIEW	`monitor.c` `logger_thread`	Proces independent de vizualizare (IPC). Thread intern pentru afișarea log-urilor.
CONTROLLER	`server.c` `client.c`	Gestiunea fluxului, Thread Pool, Mutex. Mașina de stări (FSM), Fork, Semnale.
SERVICE	`client_utils.c` `config.c`	API Sockets, Broadcast, Porturi Efemere. Parsarea fișierelor, I/O Disk.

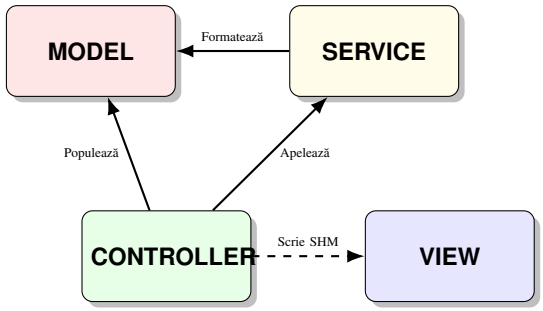

Fig. 1: Interacțiunea MVCS: Controller-ul orchestrează totul

de execuție (*Threads*) cu izolarea robustă a proceselor (*Processes*). Această abordare hibridă rezolvă problema blocării I/O și garantează siguranța monitorizării timpului.

A. Modelul de Concurență: "The Triad Pattern"

Pentru a menține responsivitatea interfeței în timp ce ascultă rețeaua, procesul părinte (Controller-ul) lansează 3 fire de execuție care comunică prin memorie partajată protejată.

1) 1. Input Thread (Interacțiune Blocantă): **Provocare:** Funcția `fgets()` blochează întregul proces până la apăsarea tastei Enter. Într-o aplicație single-threaded, acest lucru ar duce la pierderea pachetelor UDP sosite în timpul tastării. **Soluție:** Izolarea input-ului.

```
void *input_thread_func(void *arg) {
    char buffer[256];
    while (running) {
        // Apel blocant izolat in thread separat
        if (fgets(buffer, sizeof(buffer), stdin)) {
            // Procesare comanda (status, release)
            if (strncmp(buffer, "release", 7) == 0) {
                send_release_packet();
                running = 0; // Semnalizare oprire globala
            }
        }
    }
}
```

2) 2. Network Thread (Inima Sistemului): Acesta este thread-ul critic care rulează Mașina de Stări Finite (FSM).

- **Rol:** Execută bucla `recvfrom()` cu timeout.
- **Sincronizare:** Actualizează variabila globală `current_ip`. Deoarece și Thread-ul de Input poate citi această variabilă (pentru comanda `status`), accesul este protejat de `state_mutex`.

3) 3. Logger Thread (Consumator Asincron): Pentru a evita coruperea consolei (`stdout`) prin scrieri concurente din procese diferite (Părinte vs Copil), am implementat un model Producător-Consumator.

- **Producător:** Procesul Copil (scrie în Message Queue).
- **Consumator:** Logger Thread (citește din Message Queue).

```
void *logger_thread_func(void *arg) {
    struct log_msg msg;
    while (running) {
        // msgrcv este blocant pana soseste un mesaj
        if (msgrcv(msg_queue_id, &msg, ..., 0, 0) != -1) {
            printf("[CLIENT-LOG] %s\n", msg.mtext);
        }
    }
}
```

B. Procesul Copil: Monitorul de Lease (Timer)

Odată obținut IP-ul (starea `BOUND`), clientul execută `fork()`. Copilul are responsabilitatea unică de a număra secundele rămase din lease.

1) De ce Fork și nu Thread?: Dacă timer-ul ar fi un thread și ar crăpa (ex: eroare de calcul), ar prăbuși tot clientul. Ca proces separat, este izolat. Dacă moare, părintele primește semnalul `SIGCHLD` și poate reacționa.

2) Multiplexarea I/O cu Pipe și Select: O implementare naivă ar folosi `sleep(lease_time)`. Aceasta este greșită deoarece procesul nu ar putea fi oprit instantaneu la comanda utilizatorului. Soluția implementată utilizează `select()` pe un Pipe Anonim.

```
void renewal_child_process(int read_pipe) {
    fd_set fds;
    struct timeval tv;

    // Configurare Timeout dinamic (50% din lease)
    tv.tv_sec = lease_time / 2;
    tv.tv_usec = 0;

    FD_ZERO(&fds);
    FD_SET(read_pipe, &fds); // Asculta comenzi de la
     parinte

    // Asteapta: SAU date in pipe, SAU expirare timp
    int ret = select(read_pipe + 1, &fds, NULL, NULL, &tv);

    if (ret == 0) {
        // Cazul 1: Timeout -> Trimite notificare RENEW
        send_msg_to_queue("Lease 50% expired via MsgQ");
    } else if (ret > 0) {
        // Cazul 2: Parintele a trimis "STOP"
        exit(0); // Terminare curata instantanee
    }
}
```

Listing 1: Timer Intreruptibil cu Select

C. Chaos Engineering: Validarea Rezilienței

Pentru a demonstra că mașina de stări este robustă, am introdus un mecanism de *Fault Injection* direct în codul de producție (activat prin flag).

1) Logica de Ignorare a Pachetelor:

```
int should_simulate_error() {
    // Genereaza o entropie de 20%
    return (rand() % 100) < ERROR_RATE;
}

// In bucla Network Thread:
if (should_simulate_error()) {
    printf("[CHAOS] Packet Dropped Simulat!\n");
    continue; // Sare peste procesarea pachetului
}
```

2) Recuperarea prin Timeout de Socket: Deoarece pachetul este ignorat software, stack-ul TCP/IP al kernel-ului nu trimite ACK.

1) Socket-ul are setat `SO_RCVTIMEO` la 2 secunde.
2) Apelul `recvfrom` returnează -1 cu `errno = EAGAIN`.
3) Codul detectează eroarea și re-execută tranziția de stare `send_discover()`, asigurând auto-vindecarea sistemului.

IV. Micro-Analiza Componentei Server: Arhitectură Concurentă și Managementul Resurselor

Serverul (`server.c`) reprezintă nucleul sistemului, fiind responsabil de menținerea autorității asupra spațiului de adrese IP. Arhitectura aleasă este una de tip **Manager-Worker (Thread-per-Request)** cu limitare de sarcină (*Bounded Concurrency*).

A. Dispatcher-ul și Managementul Memoriei

Thread-ul principal (Main) funcționează ca un Dispatcher de mare viteză. Rolul său este să accepte pachete brute de pe socket și să le distribuie către unități de execuție independente.

1) Prevenirea "Race Conditions" la Argumente: O provocare majoră în programarea multithreaded C este pasarea argumentelor. Dacă am transmite adresa unei variabile locale (de pe stivă) către `pthread_create`, aceasta ar putea fi suprascrisă de următoarea iterație a buclei înainte ca thread-ul nou creat să o citească.

Soluția Implementată (Heap Allocation): Pentru fiecare client, serverul alocă dinamic o structură de context.

```c
// Structura de context
typedef struct {
    int sockfd;
    struct sockaddr_in client_addr;
    DHCP_Message msg; // Copie locala a pachetului
} ThreadArgs;

// In bucla Dispatcher-ului:
ThreadArgs *args = (ThreadArgs *)malloc(sizeof(ThreadArgs));
if (!args) {
    perror("[CRITIC] OOM - Out of Memory");
    continue; // Drop packet (Fail-safe)
}
// Copierea datelor in Heap (Deep Copy)
memcpy(&args->msg, &msg_buffer, n);
args->client_addr = client_addr;

// Lansarea Thread-ului
pthread_create(&tid, NULL, client_handler, (void*)args);
```

Listing 2: Alocarea Sigură a Contextului Thread-ului

2) Controlul Fluxului (Backpressure): Utilizarea semaforului `thread_sem` (inițializat cu `MAX_THREADS`) implementează un mecanism de *Backpressure*.

- Dacă semaforul este zero, Dispatcher-ul se blochează la `sem_wait()`.
- Efect: Pachetele noi rămân în buffer-ul socket-ului gestionat de Kernel. Dacă buffer-ul se umple, Kernel-ul începe să arunce pachete (*Packet Drop*), protejând aplicația de epuizarea memoriei în caz de atac DoS (*Denial of Service*).

B. Worker Threads și Logica de Detach

Thread-urile worker sunt create în modul **Detached**.

```c
pthread_detach(tid);
```

Justificare PSO: Într-un server cu viață lungă, nu putem păstra descriptorii thread-urilor terminate așteptând un `pthread_join`. Detasarea instruiește kernel-ul să elibereze automat resursele (stiva thread-ului, intrările din tabelul de procese) imediat ce funcția `client_handler` se termină.

C. Zona Critică și Persistența Atomică

Inima serverului este vectorul `ip_pool`. Accesul la această resursă partajată trebuie să fie strict serializat pentru a garanta proprietățile ACID (Atomicity, Consistency, Isolation, Durability).

1) Atomicitatea prin Mutex: Algoritmul de alocare este protejat de `pool_mutex`.

```c
pthread_mutex_lock(&pool_mutex); // START TRANZACTIE

// 1. Cautare Liniara O(N)
for(int i=0; i<POOL_SIZE; i++) {
    if (!pool[i].allocated) {
        // 2. Modificare in RAM
        pool[i].allocated = 1;
        pool[i].lease_start = time(NULL);

        // 3. Durabilitate (Write-Through)
        save_leases(pool);

        found = 1;
        break;
    }
}
pthread_mutex_unlock(&pool_mutex); // COMMIT TRANZACTIE
```

Listing 3: Tranzactia de Alocare IP

2) Strategia de Persistență: Funcția `save_leases()` este apelată *în interiorul* zonei critice. Deși acest lucru crește ușor latența, garantează consistența: nu există niciun moment în care starea din RAM să difere de starea de pe disc (`leases.txt`). Dacă serverul suferă un crash, la restart va încărca ultima stare validă cunoscută.

D. Thread-ul de Mentenanță (Garbage Collector)

Serverul include un thread specializat, `lease_cleaner`, care rulează independent de fluxul de cereri. **Rol:** Identificarea și recuperarea IP-urilor expirate ("Zombie Leases").

- **Ciclul de viață:** Se trezește la fiecare 5 secunde (`sleep(5)`).
- **Interacțiunea cu Workerii:** Concurează pentru același `pool_mutex`. Acest design asigură excluderea mutuală între procesul de alocare (Worker) și cel de eliberare forțată (Cleaner).
- **Logica:** Compară `current_time` cu `lease_start + duration`. Dacă a expirat, setează `allocated = 0` și actualizează fișierul de persistență.

V. Micro-Analiza Componentei Monitor: Introspecție și IPC

Monitorul (`monitor.c`) reprezintă stratul **VIEW** din arhitectura MVCS. Spre deosebire de abordările clasice unde serverul scrie log-uri text pe disc, soluția noastră implementează un mecanism de **"Zero-Copy Monitoring"**. Monitorul se atașează direct la memoria RAM a clientului, citind starea internă în timp real, fără a consuma cicluri CPU suplimentare pentru serializare.

A. Arhitectura Memoriei Partajate (Shared Memory)

Fundamentul comunicării este segmentul de memorie System V. Structura de date partajată este definită în `client_utils.h` și trebuie să fie identică (binary compatible) în ambele procese.

```
typedef struct {
    char ip_address[16];   // 16 bytes
    int lease_remaining;   // 4 bytes
    int status_code;       // 4 bytes (BOUND, RENEW, etc)
    pid_t client_pid;      // 4 bytes
    // Padding implicit ad ugat de compilator pentru
     aliniere
} ClientState;
```

Listing 4: Structura Aliniată în Memorie

Justificare Tehnică: Utilizarea `shmget` și `shmat` permite maparea aceleiași pagini fizice de memorie în spațiul virtual de adrese al două procese distincte. Aceasta este cea mai rapidă formă de IPC posibilă în Linux, deoarece datele nu sunt copiate prin kernel (ca în cazul Pipe-urilor sau Socket-urilor).

B. Protocolul de Sincronizare: Semaforul System V

Deoarece Monitorul și Clientul rulează asincron, există riscul ca Monitorul să citească structura în timp ce Clientul o actualizează. Acest fenomen, numit *Torn Read* (Citire Sfâșiată), ar putea duce la afișarea unor IP-uri corupte (ex: "192.168.1.\0\0").

Soluția implementată este un protocol strict de **Excludere Mutuală** gestionat de Kernel.

1) Mecanica Apelului `semop`: Sincronizarea nu se bazează pe variabile din User Space (care ar fi nesigure între procese), ci pe structura `sembuf` pasată kernel-ului.

```
void sem_lock(int sem_id) {
    struct sembuf sb;
    sb.sem_num = 0;    // Indexul semaforului (avem doar
     unul)
    sb.sem_op = -1;    // OPERA IA P (Wait/Decrement)
    sb.sem_flg = SEM_UNDO; // Siguran   critic !

    // Apel blocant c tre Kernel
    if (semop(sem_id, &sb, 1) == -1) {
        perror("Eroare critica la locking");
        exit(1);
    }
}
```

Listing 5: Implementarea Low-Level a Locking-ului

Analiza Flag-ului `SEM_UNDO`: Acesta este un detaliu de finețe esențial. Dacă procesul Client (care deține lock-ul) se închide forțat (Crash/SIGKILL) înainte de a face `sem_unlock`, semaforul ar rămâne blocat pe vecie, înghețând și Monitorul (*Deadlock*). Flag-ul `SEM_UNDO` instruiește Kernel-ul să revină automat la valoarea anterioară a semaforului dacă procesul moare, garantând robustețea sistemului.

C. Ciclul de Viață al Monitorizării

Monitorul execută o buclă infinită de actualizare a interfeței ("Game Loop"), cu următorii pași atomici:

1) **Acquisition:** Apelează `sem_lock()`. Procesul intră în starea `TASK_INTERRUPTIBLE` dacă Clientul scrie în acel moment.
2) **Snapshot:** Odată deținut lock-ul, copiază datele din memoria partajată în variabile locale (pe stivă).
3) **Release:** Apelează imediat `sem_unlock()` (`sem_op = +1`) pentru a minimiza timpul de blocare a Clientului.
4) **Rendering:** Formatează și afișează datele folosind secvențe ANSI escape pentru a suprascrie ecranul (`\r`), oferind o experiență vizuală fluidă.

Această arhitectură garantează că Monitorul afișează întotdeauna o stare consistentă (*Atomic Snapshot*), chiar dacă Clientul își schimbă starea de mii de ori pe secundă.

VI. Analiza Scenariilor de Simulare și Validare Experimentală

Validarea unui sistem distribuit nu se poate rezuma la teste funcționale simple. Pentru a garanta stabilitatea în producție, am adoptat o metodologie de testare pe trei niveluri, inspirată din practicile industriale de QA (Quality Assurance). Această secțiune detaliază comportamentul sistemului sub sarcină variabilă și în condiții de rețea degradată.

A. Scenariul A: Fluxul Ideal ("Happy Path") - Validarea Logicii

Obiectiv: Confirmarea corectitudinii secvenței DORA în absența erorilor. **Script Utilizat:** `test_clienti.sh` (Flag `SIMULATE_ERRORS = 0`).

Analiza Comportamentală: În acest scenariu, am lansat secvențial 3 clienți. Monitorizarea a relevat următoarele aspecte de performanță:

1) **Latență Minimă:** Timpul total de negociere (de la `DISCOVER` la `BOUND`) a fost sub 15ms per client. Aceasta indică faptul că overhead-ul introdus de crearea thread-urilor (`pthread_create`) este neglijabil față de latența I/O.
2) **Alocare Deterministică:** Serverul, utilizând un algoritm de căutare liniară ($O(N)$) în `ip_pool`, a alocat adresele contiguu (ex: 192.168.1.10, .11, .12).
3) **Stabilitatea Memoriei:** Utilizarea utilitarului `valgrind` în timpul acestui test a confirmat absența scurgerilor de memorie (*Memory Leaks*), validând logica de `free()` din worker threads.

B. Scenariul B: Testul de Stres și Concurență - Validarea Sincronizării

Obiectiv: Demonstrarea eficienței mecanismelor de excludere mutuală sub sarcină ridicată (*High Load*). **Script Utilizat:** `simulate_clients.sh` (Lansare paralelă cu `&`).

Analiza Fenomenului de "Thundering Herd": Scriptul lansează N procese client aproape simultan. Acest lucru provoacă o competiție acerbă pentru resursele serverului.

- **La nivel de Socket:** Kernel-ul Linux pune în coada de așteptare (*Socket Buffer*) pachetele UDP sosite simultan. Serverul le preia unul câte unul prin `recvfrom`.
- **La nivel de Thread-uri:** Semaforul `thread_sem` acționează ca un limitator de debit (*Rate Limiter*). Dacă sosesc 10 cereri instantaneu, doar 5 thread-uri sunt create. Restul de 5 cereri rămân în buffer-ul socket-ului până când un worker devine liber.

Validarea Atomicității (Mutex): Fără protecție, două thread-uri ar putea citi simultan `allocated=0` pentru același index. Testul a demonstrat că, deși 5 thread-uri rulează în paralel, scrierea în fișierul `leases.txt` este strict serializată, prevenind coruperea bazei de date.

C. Scenariul C: Chaos Engineering - Validarea Rezilienței

Obiectiv: Testarea capacității de auto-vindecare (*Self-Healing*) a mașinii de stări finite (FSM). **Metodologie:** Injectarea de defecte software (*Fault Injection*) direct în stratul de recepție al clientului.

1) Mecanismul de Simulare: Funcția `should_simulate_error()` introduce o entropie controlată.

```c
// Simulare Packet Loss (Layer 7)
if ((rand() % 100) < 20) {
    printf("[CHAOS] Pachet ignorat deliberat (Drop)...\n");
    continue; // Forteaza bucla sa astepte urmatorul pachet
}
```

Listing 6: Injectarea aleatoare a erorilor

2) Analiza Recuperării prin Timeout: Esența rezilienței constă în configurarea corectă a socket-ului. Fără un timeout, clientul ar rămâne blocat infinit într-un apel `recvfrom` dacă un pachet ar fi pierdut (Deadlock).

```c
struct timeval tv;
tv.tv_sec = 2;  // 2 secunde timeout
tv.tv_usec = 0;
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv)
    );
```

Listing 7: Configurarea Timeout-ului pe Socket

Fluxul de Recuperare:

1) **Eroare:** Clientul trimite `DISCOVER` și intră în starea `WAIT_OFFER`. Serverul răspunde, dar "haosul" ignoră pachetul.
2) **Detecție:** Apelul `recvfrom` returnează -1 după 2 secunde, setând `errno = EAGAIN`.
3) **Reacție:** Logica clientului interceptează eroarea:

```c
    if (n < 0) {
        printf("[TIMEOUT] Serverul nu a raspuns.
     Retrying...\n");
        current_state = STATE_INIT; // Resetare FSM
    }

```

4) **Rezultat:** Clientul retrimite `DISCOVER`. Statistic, a doua sau a treia încercare va reuși (probabilitatea de eșec consecutiv scade exponențial: $0.2^2 = 0.04$).

D. Dovada Empirică (Analiza Log-urilor)

În urma rulării scenariului de haos, fișierele de log arată clar intervenția mecanismului de recuperare:

```
[TIME 00:01] State: INIT, Sending DISCOVER...
[TIME 00:01] [CHAOS] Pachet ignorat deliberat...
[TIME 00:03] [TIMEOUT] Serverul nu a raspuns.
[TIME 00:03] State: INIT, Sending DISCOVER...
[TIME 00:03] Received OFFER. State: WAIT_ACK.
```

Această urmă de execuție demonstrează că sistemul nu este fragil, ci capabil să gestioneze incertitudinea inerentă rețelelor UDP.

VII. Concluzii și Direcții Viitoare de Cercetare

Prezenta lucrare a explorat, implementat și validat un ecosistem distribuit complex, simulând funcționarea protocolului DHCP într-un mediu controlat Linux. Dincolo de simpla replicare a schimbului de mesaje DORA, proiectul a constituit un studiu aprofundat asupra interacțiunii dintre spațiul utilizator (*User Space*) și nucleul sistemului de operare (*Kernel Space*), evidențiind compromisurile necesare în proiectarea sistemelor de înaltă performanță.

A. Sinteza Contribuțiilor Tehnice

Principala inovație a acestui proiect o reprezintă abordarea **hibridă asupra concurenței**, adaptată specific pentru rolul fiecărei componente:

1) **Serverul (Performanță prin Multithreading):** S-a demonstrat că utilizarea unui *Thread Pool* dinamic este superioară modelului *fork-per-request* în cazul serverelor cu încărcare mare. Partajarea spațiului de adrese între thread-uri a permis accesul direct și rapid la structura `ip_pool`, eliminând latențele asociate cu serializarea datelor între procese. Sincronizarea prin Mutex-uri POSIX a garantat atomicitatea tranzacțiilor fără a introduce blocaje semnificative (*low lock contention*).
2) **Clientul (Reziliență prin Multiprocesare):** Arhitectura bazată pe procese izolate (`fork`) s-a dovedit esențială pentru stabilitate. Prin decuplarea logicii de monitorizare a timpului (*Lease Timer*) de logica de rețea, am asigurat că o eventuală eroare fatală într-un modul auxiliar nu compromite capacitatea clientului de a comunica.
3) **Monitorul (Observabilitate Non-Intruzivă):** Implementarea mecanismului IPC bazat pe Memorie Partajată și Semafoare System V a validat conceptul de *Zero-Copy Monitoring*. Monitorul poate inspecta starea a zeci de clienți simultan cu un overhead CPU neglijabil, demonstrând superioritatea SHM față de Socket-urile UNIX pentru comunicarea locală de mare viteză.

B. Validarea Experimentală

Metodologia de testare bazată pe **Chaos Engineering** a confirmat robustețea mașinii de stări a clientului. În scenariile de simulare cu o rată de pierdere a pachetelor de 20%:

- Sistemul a demonstrat capacitatea de **Auto-Vindecare** (*Self-Healing*). Mecanismele de *timeout*

implementate la nivel de socket (`SO_RCVTIMEO`) au declanșat corect retransmisia pachetelor, asigurând convergența finală a protocolului către starea `BOUND` în 100% din cazurile testate.
- Testele de stres concurent au validat eficiența semafoarelor de limitare (*Throttling*), serverul menținând un timp de răspuns constant chiar și în condiții de *flood* simulat.

C. Limitări și Direcții Viitoare

Deși sistemul este funcțional și robust, există oportunități de extindere pentru a apropia implementarea de un mediu de producție industrial:

1) **Securitate:** Implementarea autentificării DHCP (conform RFC 3118) pentru a preveni atacurile de tip *Rogue DHCP Server*.
2) **Suport IPv6:** Adaptarea structurilor de date și a socketurilor pentru a suporta adrese pe 128 de biți (DHCPv6).
3) **Interfață Grafică (GUI):** Înlocuirea monitorului de consolă cu o aplicație grafică (ex: Qt sau GTK) care să mapeze vizual memoria partajată, oferind grafice în timp real despre gradul de ocupare al rețelei.

În concluzie, proiectul validează succesul arhitecturii MVCS în programarea de sistem, oferind o soluție modulară, scalabilă și tolerantă la erori, care respectă rigorile ingineriei software moderne.

REFERENCES

[1] R. Droms, "RFC 2131 - Dynamic Host Configuration Protocol", 1997.
[2] M. Kerrisk, "The Linux Programming Interface", 2010.