Implementare Server DHCP Tolerant la Erori

Onofrei Sonia Cringasu Andreea

***Abstract*—Acest document tehnic detaliază ciclul complet de dezvoltare a unui sistem client-server care implementează protocolul DHCP (Dynamic Host Configuration Protocol), cu un accent special pe conceptele avansate de Programare a Sistemelor de Operare (PSO). Proiectul propune o soluție robustă la problemele fundamentale ale sistemelor distribuite: concurența, sincronizarea și consistența datelor. Pentru componenta Server, s-a adoptat o arhitectură multi-thread (Thread Pool hibrid) controlată prin semafoare POSIX și mutex-uri, asigurând o scalabilitate liniară și protecție împotriva condițiilor de cursă (Race Conditions). Componenta Client inovează printr-o arhitectură multi-proces decuplată, utilizând Cozi de Mesaje (Message Queues) pentru logare asincronă, Memorie Partajată (Shared Memory) pentru persistența stării și Pipe-uri pentru coordonarea inter-proces. Sistemul include, de asemenea, o componentă dual de testare: `simulate_clients.sh` pentru testarea robusteței (Fault Injection) și `test_clienti.sh` pentru validarea fluxului ideal (fără erori).**

***Index Terms*—DHCP, Distributed Systems, Concurrency Control, POSIX Subsystem, System V IPC, Threading Models, Synchronization, Fault Tolerance, Socket Programming**

I. Introducere

În era rețelelor definite prin software (SDN) și a infrastructurilor dinamice, protocolul DHCP rămâne un pilon fundamental, automatizând procesul de alocare a adreselor IP și prevenind conflictele de configurare manuală. Deși specificat formal în RFC 2131, implementarea sa într-un mediu real necesită o înțelegere profundă a interacțiunilor dintre sistemul de operare și aplicație.

Acest proiect nu urmărește doar replicarea funcțională a schimbului de mesaje DHCP, ci construirea unui sistem care respectă principiile ingineriei software moderne:

- **High Availability:** Capacitatea serverului de a răspunde chiar și sub presiunea unui număr mare de cereri (”Flood”).
- **Resilience:** Capacitatea clientului de a funcționa corect într-un mediu de rețea degradat (pierderi de pachete, latență).
- **Observability:** Mecanisme detaliate de logging și introspecție a stării interne.

Structura lucrării este următoarea: Secțiunea II prezintă fundamentele teoretice; Secțiunea III detaliază protocolul; Secțiunile IV și V aprofundează implementarea Serverului și a Clientului; Secțiunea VI prezintă manualul de utilizare și Secțiunea VII analizează rezultatele.

II. Fundamente Teoretice și Concepte PSO

A. Procese vs. Fire de Execuție (Threads)

O decizie arhitecturală majoră a fost alegerea între modelul bazat pe procese și cel bazat pe fire de execuție.

- **Procesele** oferă izolare completă a memoriei. O eroare într-un proces copil (ex: Segmentation Fault) nu afectează procesul părinte. Acesta este motivul pentru care Partea de Client folosește `fork()` pentru monitorul de lease – dacă acesta crapă, clientul principal poate încerca să îl repornească sau să continue (Graceful Degradation).
- **Thread-urile** partajează spațiul de adrese (Heap, Globals), permițând o comunicare extrem de rapidă fără primitive IPC complexe. Serverul folosește acest model deoarece necesită acces partajat frecvent la `ip_pool`.

B. Primitive de Sincronizare

Într-un mediu concurent, accesul la datele partajate trebuie serializat.

- **Mutex (Mutual Exclusion):** Folosit pentru a proteja secțiuni critice scurte (ex: marcarea unui IP ca ”OCUPAT”). Este un mecanism de tip ”locking”.
- **Semafor (Counting Semaphore):** Folosit pentru gestionarea resurselor numerabile (ex: numărul maxim de sloturi de procesare disponibile). Este un mecanism de tip ”signaling”.

C. Inter-Process Communication (IPC)

Comunicarea între procese separate necesită intervenția kernel-ului.

- **Pipe:** Un canal unidirecțional de byte-stream.
- **Message Queue:** O listă linkată de mesaje gestionată de kernel, care permite filtrarea după tip (priority).
- **Shared Memory:** Cea mai rapidă formă de IPC, unde o zonă de RAM este mapată în spațiul virtual al ambelor procese. Necesită sincronizare externă (semafoare) pentru a evita coruperea datelor.

III. Definiția Protocolului și Structura Datelor

Protocolul implementat este o versiune fidelă a DHCP, utilizând transport **UDP/IP (INET Sockets)** pe porturile standard 9000 (Server) și Ephemeral (Client).

A. Structura Pachetelor (DHCP Message)

Toate mesajele schimbate respectă structura `DHCP_Message`, definită în header-ul comun:

```
typedef struct {
    struct {
        uint8_t op;    // 1=Request, 2=Reply
        uint8_t htype; // Hardware Type
        uint8_t hlen;  // Hardware Length
        uint8_t hops;  // Relay hops
        uint32_t xid;  // Transaction ID
         /* ... flags, secs ... */
    } header;
    uint32_t msg_type; // DISCOVER, OFFER...
```

```
    char offered_ip[16];
    char router[16];
    char dns[16];
    uint32_t lease_time;
    /* ... padding ... */
} DHCP_Message;
```

Câmpul `xid` este crucial. Clientul generează un număr aleator la pornire și ignoră orice pachet primit care nu are același `xid`, prevenind atacurile de tip ”Spoofing” sau confuzia în cazul în care mai mulți clienți rulează pe aceeași mașină.

IV. Componenta Server: Arhitectură și Implementare

Dezvoltată de Cringasu Andreea, componenta server acționează ca autoritate centrală pentru alocarea adreselor.

A. Initializare și Configurare

La pornire, serverul execută rutina `load_config()`, care parsează fișierul `ipconfig.txt`. Aceasta permite administratorului sa definească:

- Plaja de IP-uri (Range Start - Range End).
- Adresa Gateway-ului și a serverului DNS.
- Timpul implicit de Lease.

Datele sunt încărcate în structuri rezidente în memorie pentru acces rapid.

B. Gestionarea Concurenței (Thread Pool Dinamic)

Bucla principală a serverului implementează un model ”Producer-Consumer” simplificat:

1) **Acceptare:** Socket-ul principal (UDP Broadcast) primește o datagramă (`recvfrom`).
2) **Admitere:** Se execută `sem_wait(&thread_sem)`. Dacă sunt deja 5 thread-uri active, serverul blochează aici, protejând resursele sistemului.
3) **Delegare:** Se alocă o structură `ThreadArgs` cu detaliile pachetului și se lansează un thread worker (`pthread_create`).
4) **Detasare:** Thread-ul este detașat (`pthread_detach`) pentru a elibera resursele automat la terminare.

C. Logica de Alocare (Sincronizare)

În interiorul funcției `client_handler`, thread-ul procesează cererea. Pentru a evita conflictele, se folosește `pool_mutex`:

```
pthread_mutex_lock(&pool_mutex);
// CRITICAL SECTION START
for(int i=0; i<POOL_SIZE; i++) {
   if (!ip_pool[i].allocated) {
       ip_pool[i].allocated = 1;
       strcpy(response.ip, ip_pool[i].ip);
       break;
   }
}
// CRITICAL SECTION END
pthread_mutex_unlock(&pool_mutex);
```

Fără acest mutex, două thread-uri ar putea găsi același index i liber simultan, alocând același IP (Catastrophic Failure).

V. Componenta Client: Inovație prin Multi-Processing

Dezvoltată de Onofrei Sonia, componenta client este un exemplu de arhitectură software defensivă.

A. Mașina de Stări Finită (FSM)

Clientul nu execută liniar, ci tranzitează între stări:

- **INIT:** Nu are IP. Trimite `DISCOVER`.
- **WAIT_OFFER:** Așteaptă oferte. Dacă expiră timer-ul socket-ului (2s), revine la INIT.
- **WAIT_ACK:** A trimis `REQUEST`, așteaptă confirmarea finală.
- **BOUND:** Are IP valid. Porneste procesul de monitorizare.

B. Sistemul de Monitorizare a Lease-ului

Odată obținut IP-ul, clientul trebuie să contorizeze timpul. Pentru a nu bloca firul principal (care trebuie să răspundă la comenzi de consolă), această logică este delegată unui proces copil creat prin `fork()`.

1) Comunicarea Părinte-Copil:

1) **Pipe Anonim:** Părintele deține capătul de scriere (`pipe_fd[1]`). La comanda ”exit” sau ”release”, scrie un token. Copilul monitorizează capătul de citire cu `select()`. Dacă apare data, termină execuția imediat.
2) **Message Queue:** Copilul nu are acces la consola părintelui în mod sincronizat. Astfel, el trimite mesaje de status (ex: ”Lease la 50%”) într-o coadă System V. Un thread dedicat din părinte (`Logger Thread`) citește din coadă și afișează mesajele.

C. Persistența Stării (Shared Memory)

Starea curentă (IP, Lease Time, Status) este duplicată într-un segment de memorie partajată (`shmget`). Acest design permite ca, într-o versiune viitoare, un alt proces (ex: un GUI de administrare) să citească starea clientului fără a interoga procesul principal. Accesul la memoria partajată este protejat de un semafor System V (`semget`).

D. Mecanismul de Fault Tolerance

Rețelele reale pierd pachete. Clientul simulează acest mediu ostil:

```
if ((rand() % 100) < 20) {
    printf("[SIMULARE] Pachet IGNORAT!\n");
    continue; // Drop packet
}
```

Datorită timeout-ului setat pe socket (`SO_RCVTIMEO`), funcția `recvfrom` nu blochează infinit. Când returnează eroare, bucla principală reia ultimul pas al handshake-ului, asigurând eventuala reușită a negocierii. De notat că, pentru a evita coliziunile pe localhost, clienții se leagă la porturi efemere (random), permițând Kernel-ului să routeze corect răspunsurile unicast.

VI. Manual de Utilizare și Compilare

A. Cerințe de Sistem

- Sistem de operare: Linux (kernel 2.6+)
- Compilator: GCC
- Biblioteci: pthread, rt (Realtime Extensions)

B. Compilare

Proiectul include un `Makefile` pentru automatizarea procesului de build:

```
$ cd DHCP_server2
$ make          # Compileaza server si client
$ make clean    # Sterge binarele
```

C. Rulare

1. Pornire Server: Serverul trebuie pornit primul. Acesta va crea socket-ul `server_sock` și va aștepta cereri.

```
$ ./server
Server DHCP pornit (INET Sockets)...
Ascult pe portul 9000
```

2. Pornire Client: Într-un terminal separat (sau mai multe, pentru testare concurentă):

```
$ ./client_app
[CLIENT] Pornire DHCP handshake...
[CLIENT] DISCOVER trimis
...
[STATUS] IP: 192.168.1.100 | Lease: 3600
```

3. Comenzi Interactive Client: Clientul acceptă comenzi la rulare:

- `status`: Afișează IP-ul curent și timpul rămas.
- `release`: Eliberează voluntar IP-ul (trimite DHCP RELEASE) și resetează starea.
- `exit`: Închide clientul (inclusiv procesele copil).

4. Monitor Process (Proces Copil): Acesta pornește automat odată cu intrarea în starea `BOUND`. Nu necesită intervenția utilizatorului.

- *Logare:* Mesajele generate de Monitor (”Half-time lease expired”) vor apărea asincron în consola principală a clientului, prefixate cu `[CLIENT-Logger]`.
- *Vizualizare:* Se poate verifica existența procesului cu:

```
$ ps -ef | grep client
user  1001  ... ./client (Parinte)
user  1002  ... ./client (Copil -
Monitor)
```

5. Testare Automată: Proiectul include două scripturi de testare în directorul `Controller`:

- **simulate_clients.sh:** Lansează clienți cu simularea erorilor activată (20% Packet Loss), pentru a testa revenirea din timeout și starea INIT.
- **test_clienti.sh:** Recompilează clienții cu flag-ul `-DSIMULATE_ERRORS=0` și lansează o simulare ”curată” (fără erori), ideală pentru demonstrații rapide.

```
$ ./Controller/test_clienti.sh
```

Script-ul va:

1) Compila clientul.
2) Lansa 3 (configurabil) instanțe în background.
3) Redirecționa output-ul în `client_1.log`, `client_2.log`, etc.
4) Aștepta apăsarea tastei ENTER pentru a trimite SIGINT tuturor proceselor (Shutdown curat).

VII. Analiză și Concluzii

A. Performanță

Testele de stres (scriptul `simulate_clients.sh` lansând 10 clienți) au arătat o utilizare eficientă a CPU. Timpul mediu de răspuns a fost sub 10ms (pe localhost). Limitarea la 5 thread-uri prin semafor a funcționat conform așteptărilor: al 6-lea client a trebuit să aștepte eliberarea unui slot, demonstrând eficacitatea controlului de flux.

B. Securitate

Deși este o simulare, proiectul implementează măsuri de securitate de bază:

- **Validare XID:** Previne injectarea de pachete false.
- **Resource Cap:** Previne epuizarea memoriei serverului prin limitarea thread-urilor.

C. Concluzii Finale

Sistemul dezvoltat îndeplinește toate obiectivele propuse, demonstrând o arhitectură matură. Îmbinarea mecanismelor de sincronizare POSIX (pentru performanță pe server) cu cele System V IPC (pentru robustețe pe client) oferă o perspectivă educațională valoroasă asupra paradigmelor de programare în Linux.

References

[1] R. Droms, ”Dynamic Host Configuration Protocol”, RFC 2131, 1997.
[2] A. Silberschatz, P. Galvin, G. Gagne, ”Operating System Concepts”, Wiley, 10th Edition, 2018.
[3] M. Kerrisk, ”The Linux Programming Interface”, No Starch Press, 2010.
[4] IEEE, ”POSIX System Interfaces”, IEEE Std 1003.1-2017.