

PSO-RAT: Implementarea unui Remote Administration Tool pentru Proiectarea Sistemelor de Operare

Romaş Ştefan-Sebastian

Facultatea de Calculatoare şi Sisteme Informatic de Securitate şi Apărare Naţională

Academia Tehnică Militară „Ferdinand I”

Bucureşti, România

stefan-sebastian.romas@stud.mta.ro

Rezumat—Acest document prezintă implementarea unui Remote Administration Tool (RAT) educaţional dezvoltat în cadrul cursului de Proiectarea Sistemelor de Operare la Academia Tehnică Militară. Proiectul demonstrează aplicarea practică a conceptelor fundamentale de sisteme de operare: socket-uri TCP/UDP pentru comunicare în reţea, thread-uri pentru execuţie concurentă, mecanisme de sincronizare prin mutex, gestionarea memoriei şi proceselor prin fork/exec şi utilizarea pipe-urilor pentru comunicare inter-proces. Arhitectura client-server implementată utilizează comunicare JSON şi permite controlul remote al staţiilor client prin comenzi administrative. Implementarea în C++17 foloseşte smart pointers pentru gestionarea automată şi sigură a memoriei.

Index Terms—sisteme de operare, socket-uri, thread-uri, mutex, pipe-uri, gestionarea memoriei, procese client-server, RAT, C++17, POSIX

I. INTRODUCERE

Remote Administration Tools (RAT) sunt aplicaţii software care permit controlul şi administrarea la distanţă a sistemelor de calcul. Deşi astfel de instrumente pot fi utilizate în contexte legitime (administrare IT, suport tehnic), acest proiect îşi propune să demonstreze conceptele fundamentale ale sistemelor de operare într-un context educaţional strict controlat.

Proiectul PSO-RAT a fost dezvoltat în cadrul cursului de Proiectarea Sistemelor de Operare la Academia Tehnică Militară şi implementează o arhitectură client-server complexă care demonstrează:

- **Gestionarea memoriei** — Smart pointers (unique_ptr, shared_ptr) pentru management automat al resurselor
- **Pipe-uri** — Utilizarea popen() pentru execuţie comenzi shell şi capturare output
- **Procese** — fork(), exec(), wait() pentru crearea şi gestionarea proceselor copil
- **Thread-uri** — Arhitectură multi-threading pentru execuţie concurentă şi non-blocking I/O
- **Mutex** — Sincronizare thread-safe prin std::mutex şi std::lock_guard
- **Socket-uri TCP/UDP** — Comunicare client-server prin SFML Network wrappers

Obiectivul principal este aplicarea practică a conceptelor teoretice învăţate în cadrul cursului PSO, cu accent pe corecti-

tudinea implementării şi respectarea principiilor de programare sistem moderne.

II. ARHITECTURA SISTEMULUI

A. Prezentare Generală

Sistemul este organizat într-o arhitectură client-server clasică, cu un server central care gestionează conexiuni multiple de la clienţi şi oferă o interfaţă administrativă pentru controlul acestora. Comunicarea se realizează prin socket-uri TCP pentru transfer de date şi UDP pentru mecanismul de keep-alive.

B. Componente Server

Serverul este structurat în mai multe module specializate:

ServerManager — Singleton care coordonează întregul server. Gestionează listener-ul TCP pentru acceptarea clienţilor noi şi menţine referinţe către toate controller-ele active. Implementează send şi receive, utilităţi pentru controllere şi gestionarea comunicării între server şi clienţi.

ClientManagement — Modul centralizat responsabil cu gestionarea thread-safe a clienţilor conectaţi. Foloseşte std::mutex pentru protejarea accesului concurrent la harta de clienţi şi implementează generarea automată de nume unice pentru fiecare client conectat (ex: arch_0, ubuntu_1).

Controller-e Specializate:

- **ServerCommandController** — Interfaţă CLI administrativă care rulează într-un thread dedicat pentru citirea comenzilor din stdin
- **ServerFileController** — Gestionează transferul bidirecţional de fişiere cu encoding Base64
- **ServerPingController** — Implementează mecanismul de keep-alive prin UDP pentru detectarea clienţilor deconectaţi
- **ServerLogController** — Centralizează logging-ul thread-safe în /tmp/rat_server.log şi gestionează vizualizarea log-urilor prin spawn-area terminalelor

C. Componente Client

ClientManager — Orchestratorul principal care gestionează conexiunea la server şi rotează mesaje primite către controller-ele corespunzătoare.

Controller-e Client:

- *ClientBashController* — Execută comenzi shell folosind `popen()` și returnează `stdout/stderr`
- *ClientFileController* — Implementează operațiile de upload/download cu Base64 encoding
- *ClientScreenshotController* — Capturează screenshot-uri folosind utilitare sistem (`scrot`, `import`, `gnome-screenshot`)
- *ClientPingController* — Răspunde la request-urile de keep-alive UDP
- *ClientKillController* — Gestionează deconectarea gracefully

D. Utilități Comune

Modulul `Utils` conține wrapper-e peste API-ul SFML Network:

- `TCPSocket` — Wrapper pentru socket-uri TCP cu API simplificat
- `UDPSocket` — Wrapper pentru socket-uri UDP
- `Base64` — Librărie header-only pentru encoding/decoding date binare

III. CONCEPTE PSO IMPLEMENTATE

A. Gestionarea Memoriei

Proiectul demonstrează management modern al memoriei în C++17, eliminând complet necesitatea `malloc/free` și `new/delete` manual:

Smart Pointers:

- `std::unique_ptr` — Pentru ownership exclusiv (socket-uri, controller-e). La distrugerea obiectului, resursele sunt automat eliberate, prevenind memory leaks.
- `std::shared_ptr` — Pentru `ServerManager` (singleton) și resurse partajate între thread-uri cu reference counting automat.

RAII Pattern — Toate clasele implementează constructori/destructori care alocă/eliberează resurse automat. Exemplu în `ServerManager`:

```
ServerManager::~ServerManager() {
    stop(); // oprește thread-uri
    std::lock_guard<std::mutex> lock(mtx);
    cleanupResources(); // închide socket-uri
}
```

La ieșirea din scope, destructorul este apelat automat, chiar și în cazul excepțiilor, asigurând cleanup complet al resurselor (socket-uri închise, thread-uri oprite cu `join()`).

B. Pipe-uri pentru Execuție Comenzi

`ClientBashController` demonstrează utilizarea pipe-urilor pentru execuția comenzilor shell și capturarea output-ului:

```
FILE* pipe = popen(cmd.c_str(), "r");
if (!pipe) {
    return json>{"error", "popen() failed"};
}

std::string output;
```

```
char buffer[256];
while (fgets(buffer, sizeof(buffer),
             pipe) != nullptr) {
    output += buffer;
}

int status = pclose(pipe);
int exitCode = WEXITSTATUS(status);

return json{
    {"output", output},
    {"exitcode", exitCode}
};
```

Funcția `popen()` creează un proces copil prin `fork()` intern, execută comanda într-un shell și returnează un file descriptor conectat la `stdout`-ul procesului. Citirea se face printr-un pipe implicit creat de sistem, demonstrând comunicarea inter-proces unidirecțională.

C. Gestionarea Proceselor

`ServerLogController::spawnTerminal()` demonstrează utilizarea `fork()` și `exec()` pentru lansarea terminalelor externe pentru vizualizarea log-urilor:

```
pid_t pid = fork();
if (pid == 0) { // proces copil
    setsid(); // detașare de terminal

    int devnull = open("/dev/null", O_WRONLY);
    dup2(devnull, STDOUT_FILENO);
    dup2(devnull, STDERR_FILENO);
    close(devnull);

    execlp("qterminal", "qterminal", "-e",
           "bash", "-c", cmd.c_str(), NULL);
    _exit(127); // fallback dacă exec eșuează
}
// proces părinte continuă imediat
return pid;
```

Pași cheie:

- 1) `fork()` creează proces copil identic
- 2) `setsid()` detașează copilul de terminalul părinte (daemonizare)
- 3) `dup2()` redirectionează `stdout/stderr` către `/dev/null`
- 4) `execlp()` înlocuiește imaginea procesului cu terminalul dorit
- 5) Procesul părinte returnează imediat, permițând execuție asincronă

D. Thread-uri și Programare Concurentă

Arhitectura multi-threading este esențială pentru funcționarea asincronă:

- **Thread principal** — Rulează loop-ul de acceptare clienți, blocând în `accept()` până la conectarea unui client nou

- **Thread CLI** — `ServerCommandController` rulează un thread dedicat care citește comenzi din `stdin` prin `std::getline()`, permițând interacțiuni administrativă simultană
- **Thread-uri keep-alive** — `ServerPingController` poate avea thread-uri de ping periodic pentru monitorizarea conexiunii clienților

Crearea thread-urilor:

```
std::thread stdinThread = std::make_unique<std::thread>(listener->setBlocking(false); // non-blocking
    &ServerCommandController::stdinLoop, this
);

// Server - bind și listen
listener->bind(5555);

// Loop accept
auto client = listener->accept();
if (client) {
    client->setBlocking(true);
    // blocking cu timeout
    handleClient(std::move(client));
}

void ServerCommandController::stop() {
    running = false;
    if (stdinThread && stdinThread->joinable()) {
        stdinThread->join();
    }
}
```

E. Sincronizare prin Mutex

Accesul concurrent la resurse partajate este protejat prin `std::mutex`, demonstrând necesitatea sincronizării în sisteme multi-threaded:

ClientManagement folosește un mutex pentru protejarea hărții `clients` care stochează socket-urile active:

```
bool ClientManagement::addClient(
    const std::string& name,
    std::unique_ptr<TCPSocket> socket) {
    std::lock_guard<std::mutex> lock(mtx);
    clients[name] = std::move(socket);
    return true;
}
```

ServerLogController protejează queue-ul de log-uri `logs` cu un mutex dedicat `logMtx`:

```
void ServerLogController::pushLog(
    const std::string& msg) {
    std::lock_guard<std::mutex> lock(logMtx);
    logs.push(msg);

    std::ofstream ofs(getLogPath(),
        std::ios::app);
    if (ofs) ofs << msg << std::endl;
}
```

Pattern-ul folosit este `std::lock_guard` pentru **RAII** — mutexul este automat eliberat la ieșirea din scope, chiar și în cazul excepțiilor, prevenind deadlock-urile.

F. Socket-uri TCP și UDP

Comunicarea în rețea este realizată prin `SFML Network`, folosind două tipuri de socket-uri:

Socket-uri TCP — Folosite pentru comunicarea principală client-server. Serverul deschide un listener pe portul 5555 configurat în mod non-blocking pentru a accepta conexiuni noi fără a bloca execuția. Socket-urile client sunt configurate în mod blocking cu timeout de 5 secunde pentru a preveni deadlock-urile:

```
// Server - bind și listen
listener->bind(5555);

// Loop accept
auto client = listener->accept();
if (client) {
    client->setBlocking(true);
    // blocking cu timeout
    handleClient(std::move(client));
}
```

Socket-uri UDP — Implementează mecanismul de keep-alive (ping/pong) pentru detectarea clienților deconectați. Alegerea UDP este justificată de natura stateless și overhead-ul redus pentru pachete mici periodice.

Implementarea wrapper-elor `TCPSocket` și `UDPSocket` demonstrează încapsularea API-ului de nivel jos `SFML` și oferă interfețe simplificate pentru operațiile de `bind()`, `listen()`, `accept()`, `connect()`, `send()` și `receive()`.

IV. PROTOCOL DE COMUNICARE

A. Formatul Mesajelor

Comunicarea client-server folosește JSON (biblioteca `nlohmann::json`) pentru structurarea mesajelor. Alegerea JSON este motivată de:

- **Lizibilitate** — mesajele pot fi debug-ate ușor
- **Flexibilitate** — schema poate fi extinsă fără breaking changes
- **Suport nativ C++** — biblioteca `nlohmann` oferă interfață idiomatică

Structura generală a unui mesaj:

```
{
    "controller": "bash|file|screenshot|...",
    "action": "execute|download|upload|...",
    ...parametri specifici...
}
```

B. Pattern Request-Response Sincron

Serverul implementează comunicare sincronă prin două metode:

`sendRequest(clientName, json)` — Trimite cererea către client și returnează imediat status (success/failure pentru trimitere)

`receiveResponse(clientName, json&, timeout)` — Blochează până primește răspuns de la client sau expiră timeout-ul (5-30 secunde în funcție de operație)

Acest pattern simplifică logica și elimină necesitatea callback-urilor asincrone, potrivit pentru scopul educațional al proiectului.

V. FUNCȚIONALITĂȚI DEMONSTRATE

A. Execuție Comenzi Remote

Comanda bash demonstrează utilizarea pipe-urilor și fork/exec:

```
Admin: bash arch_0 ls -la /tmp
```

Server -> Client:

```
{"controller": "bash", "cmd": "ls -la /tmp"}
```

Client execută prin popen(), creează pipe intern

Client -> Server:

```
{
  "output": "total 48\ndrwxr-xr-x...",
  "exitcode": 0
}
```

B. Transfer Fișiere

Operațiile download și upload folosesc Base64 encoding pentru transfer binar în JSON:

Download:

```
Admin: download arch_0 /etc/hostname ./h.txt
```

Server -> Client:

```
{
  "controller": "file",
  "action": "download",
  "path": "/etc/hostname"
}
```

Client: citește fișier + Base64::encode()

Client -> Server:

```
{
  "success": true,
  "data": "YXJjaC1kZXNrdG9wCg==",
  "size": 13
}
```

Server: Base64::decode() + write to ./h.txt

Limitare: Fișiere max 50MB pentru a evita overflow-ul memoriei.

C. Screenshot Remote

Screenshot-ul demonstrează integrarea cu utilitare sistem și gestionarea proceselor externe:

Client detectează utilitar disponibil:

1. Verifică which scrot/import/gnome-screenshot
2. system("scrot /tmp/ss_<timestamp>.png")

3. Citește PNG ca binar

4. Base64::encode()

Client -> Server:

```
{
  "success": true,
  "data": "iVBORw0KGgoAAAANS...",
  "size": 145230
}
```

Server:

1. Base64::decode()
2. Write to /tmp/ss_arch_0_<time>.png
3. fork() + exec() pentru xdg-open

D. Vizualizare Log-uri

Comanda showlogs demonstrează fork/exec pentru spawnarea terminalelor:

```
Admin: showlogs
```

ServerLogController:

1. Detectează terminal disponibil (qterminal, xfce4-terminal, konsole, ...)
2. fork() proces copil
3. setsid() pentru detașare
4. dup2() pentru redirecționare stdout/stderr
5. execlp() cu "bash -c tail -F log"

Terminal se deschide independent, părinte continuă imediat

VI. TESTARE ȘI REZULTATE

A. Mediu de Testare

Sistemul a fost testat pe:

- **OS:** Kali Linux
- **Compiler:** GCC 13.2.1, C++17
- **Librării:** SFML 2.6.0, nlohmann-json 3.11.2

B. Scenarii de Testare

Test 1: Conectare multiplă și thread-safety

- Pornit 5 clienți simultan
- Verificat thread-safety în ClientManagement cu mutex
- **Rezultat:** Toți clienții înregistrați corect, fără race conditions

Test 2: Execuție comenzi concurente

- Trimis comenzi bash către 3 clienți
- Verificat că output-urile nu se amestecă
- **Rezultat:** Fiecare răspuns asociat corect cu clientul său, sincronizare corectă

Test 3: Transfer fișiere mari

- Upload fișier 45MB (sub limita de 50MB)
- Verificat integritate prin checksum MD5
- **Rezultat:** Transfer reușit, date integre, Base64 encoding/decoding corect

Test 4: Deconectare bruscă și cleanup

- Închis forțat client (kill -9)
- Verificat cleanup resurse pe server prin valgrind
- **Rezultat:** Socket închis corect, RAII funcționează perfect, zero memory leaks

C. Metrici de Performanță

- **Latență medie:** 25-50ms pentru comenzi bash simple
- **Throughput transfer:** 8-10 MB/s pentru fișiere (limitat de overhead Base64 33%)
- **Memorie server:** 15MB RAM cu 5 clienți conectați
- **CPU idle:** 1% când nu procesează cereri active

VII. LIMITĂRI ȘI ÎMBUNĂTĂȚIRI

A. Limitări Actuale

Securitate: Comunicarea este în plain-text fără autentificare sau criptare. Potrivit doar pentru medii educaționale controlate în rețele locale de încredere.

Scalabilitate: Arhitectura sincronă nu suportă eficient sute de clienți simultan. Fiecare request blochează până la răspuns.

B. Îmbunătățiri Propuse

- **TLS/SSL:** Adăugare criptare prin OpenSSL
- **Asincron I/O:** Refactoring către epoll/io_uring pentru scalabilitate
- **Thread Pool:** Înlocuirea thread-urilor dedicate cu pool de workers

C. Module Propuse

- **KeyLogging:** Adăugare logging per client pentru tastele apasate
- **NetworkCapture:** Salvare PCAP per client cu tshark
- **LiveStream:** LiveStreaming service

VIII. CONCLUZII

Proiectul PSO-RAT demonstrează aplicarea practică a tuturor conceptelor fundamentale de sisteme de operare studiate în cadrul cursului la Academia Tehnică Militară:

- **Gestionarea memoriei** — Smart pointers și RAII elimină memory leaks
- **Pipe-uri** — popen() pentru execuție comenzi și capturare output
- **Procese** — fork(), exec(), setsid() pentru crearea proceselor independente
- **Thread-uri** — Arhitectură multi-threaded cu execuție concurentă
- **Mutex** — Sincronizare thread-safe cu std::lock_guard
- **Socket-uri** — TCP pentru comunicare, UDP pentru keep-alive

Implementarea în C++17 respectă principiile moderne de programare sistem, demonstrând înțelegerea profundă a modului în care conceptele teoretice se traduc în implementări practice funcționale.

Deși sistemul are limitări de securitate și scalabilitate (inente unui proiect educațional), arhitectura modulară permite extindere viitoare. Codul sursă este disponibil pe GitHub pentru studiu în medii controlate.

MULȚUMIRI

Acest proiect a fost realizat în cadrul cursului Proiectarea Sistemelor de Operare la Academia Tehnică Militară „Ferdinand I”, semestrul 2, anul universitar 2025-2026.

Mulțumiri cadrelor didactice pentru ghidarea conceptelor de sisteme de operare și comunității open-source pentru librăriile SFML și nlohmann::json care au facilitat implementarea.

BIBLIOGRAFIE

- [1] A. S. Tanenbaum și H. Bos, “Modern Operating Systems,” ediția a 4-a, Pearson, 2014.
- [2] A. Silberschatz, P. B. Galvin, și G. Gagne, “Operating System Concepts,” ediția a 10-a, Wiley, 2018.
- [3] W. R. Stevens și S. A. Rago, “Advanced Programming in the UNIX Environment,” ediția a 3-a, Addison-Wesley, 2013.
- [4] M. Kerrisk, “The Linux Programming Interface: A Linux and UNIX System Programming Handbook,” No Starch Press, 2010.
- [5] SFML Development Team, “SFML Network Module Documentation,” [Online]. Disponibil: <https://www.sfml-dev.org/>
- [6] N. Lohmann, “JSON for Modern C++,” [Online]. Disponibil: <https://json.nlohmann.me/>
- [7] IEEE și The Open Group, “POSIX Threads Programming,” [Online]. Disponibil: <https://pubs.opengroup.org/>
- [8] C++ Reference, “C++ Standard Library,” [Online]. Disponibil: <https://en.cppreference.com/>