IoT Simulator: Raport tehnic

Onofrei Tudor-Cristian

11.12.2020

1 Introducere

Acest raport tehnic are rolul de a prezenta aplicația "IoT Simulator" alături de procesul de dezvoltare ce a stat la baza ei; pentru început, vom afla ce funcționalități trebuie să respecte această aplicație, ce tehnologii au fost utilizate în crearea ei, ce presupune un astfel de remote și, la final, de ce este importantă o astfel de aplicație de tip remote.

Aplicația "IoT Simulator" oferă utilizatorului posibilitatea de a alege, controla și schimba anumiți parametri ai unor dispozitive cu care acesta alege să se conecteze; dispozitivele funcționează fiecare în parte ca un server concurent, iar utilizatorul joacă rolul de client al acestor servere, fiecare schimbare de parametru fiind o cerere pe care serverul este menit să o proceseze și să-i ofere un răspuns.

Astfel, aplicația constă din mai multe servere concurente ce pot fi accesate de un client pentru a modifica stările acestora; dacă mai mulți clienți sunt conectați simultan la același dispozitiv, se va reține ultima stare modificată.

Fiecare server are la dispoziție un fișier utilizat pentru configurarea sa atât la pornire, cât și la oprire pentru a reține ultima configurație de stare rezultată în urma cererilor trimise de clienți.

Fiecare client, odată conectat, poate trimite oricât de multe cereri de modificare a stării dispozitivului ales, deconectarea de server realizându-se prin trimiterea unei anumite comenzi serverului.

În continuare, vom afla ce tehnologii au respectat standardele de dezvoltare a unei astfel de aplicații.

2 Tehnologiile utilizate

Toate sursele proiectului au fost scrise folosind limbajul de programare C; dezvoltarea ulterioară a acestei aplicații va implica utilizarea anumitor noțiuni din limbajul C++ - folosirea paradigmei de programare orientată-obiect pentru crearea unei interfețe grafice.

Pentru a realiza conexiunea dintre client și server, am utilizat modelul concurent client-server TCP/IP; acest model are la bază o conexiune la internet ce asigură siguranța transmiterii datelor de la client la server și de la server la client, necesară pentru a modifica în mod sigur starea unui dispozitiv ce se află sau nu în câmpul de proximitate al clientului.

Am evitat utilizarea modelului UDP deoarece acesta nu asigura siguranța și consistența transmiterii pachetelor de date, constituind un factor de risc în integritatea transmiterii acestora și putând pune clientul în postura de a nu se mai putea deconecta de server.

Pentru a asigura concurența serverului, am ales să servesc fiecare client prin utilizarea mai multor fire de execuție - multithreading. Am optat pentru utilizarea mai multor fire de execuție în defavoarea creării proceselor-fii în server pentru servirea clienților dintr-o serie de motive:

- **RECEPTIVITATE**: Într-un proces divizat în mai multe fire de execuție, rezultatul obținut de un fir ce și-a terminat execuția poate fi returnat în mod imediat.
- **COMUTATOR DE CONTEXT MAI RAPID**: Timpul de comutare a contextului dintre firele de execuție în comparație cu timpul necesar schimbării de context dintre procese - ultimul mod de schimbare necesitând și cheltuieli suplimentare a unității centrale de procesare.
- **UTILIZARE EFICIENTĂ A RESURSELOR**: Dacă există mai multe fire de execuție într-un singur proces, atunci putem planifica mai multe fire de execuție pe un procesor multiplu, ceea ce va face execuția procesului mai rapidă.
- **PARTAJAREA RESURSELOR**: În afara regiștrilor și a cozii procesului respectiv, firele de execuție au partajate în cadrul unui proces toate resursele de tip cod, valori de date și fișiere.
- **COMUNICARE**: Datorită motivului anterior, comunicarea între mai multe fire de execuție este mai ușoară; utilizarea de procese-fii pentru

asigurarea concurenței presupune crearea unor canale de comunicație auxiliare pentru actualizarea datelor.

Pentru configurarea stării dispozitivului-server la pornirea acestuia am utilizat un fișier de configurare în care a fost salvată ultima configurație de stare rezultată în urma cererilor trimise de clienți.

3 Arhitectura aplicației

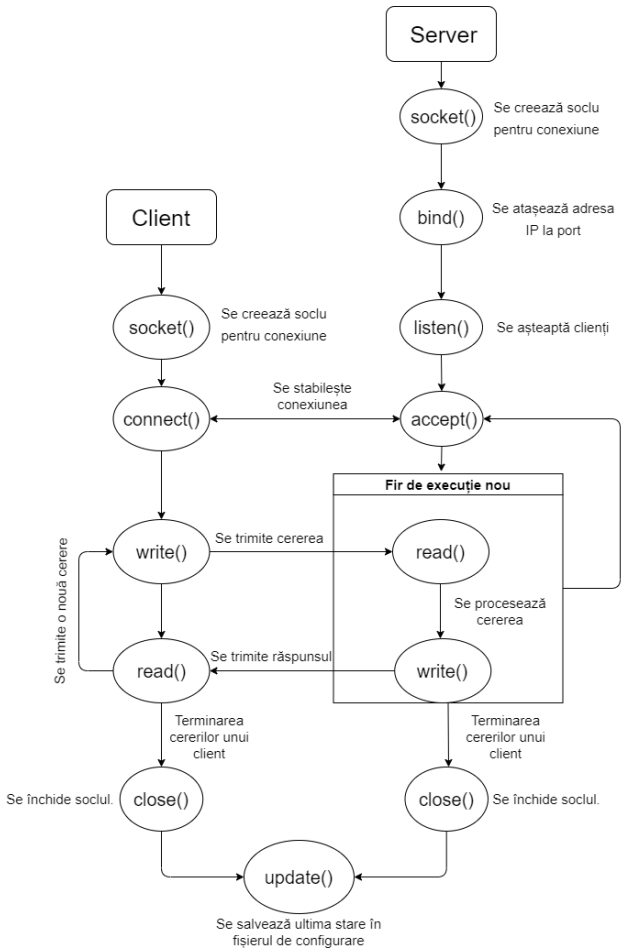


Figure 1: Arhitectura aplicației "IoT Simulator"

Serverul este constituit dintr-o singură componentă: în momentul în

care un client nou se conectează la server, este creat un nou fir de execuție ce are scopul de a răspunde cererilor trimise de clientul respectiv. Trimiterea unei cereri de părăsire a serverului constituie momentul în care firul de execuție este încheiat, moment în care serverul salvează în fișier configurația stării curente a serverului.

Altfel, dacă cererea nu este echivalentă cu cea de deconectare de la server, atunci firul de execuție va procesa cererea, actualizând starea curentă a serverului.

4 Detalii de implementare

4.1 Implementare Server

La baza implementării serverului stau 3 funcții principale:

- funcția principală main - cu rol de ilustrare a creării unui nou fir de execuție pentru rezolvarea cererilor
- funcția de tratare a unui client - creează un nou fir de execuție, fiind activ cât timp clientul trimite cereri; în cazul în care clientul a decis să se deconecteze - prin trimiterea mesajului "exit" la server, firul de execuție este detașat de proces, iar cel din urmă actualizează fișierul de stare al serverului cu starea curentă.
- funcția de trimitere a unui răspuns către client - ce are, în același timp, și rolul de a procesa cererea trimisă. Această funcție procesează cererea clientului de care firul de execuție este responsabil, apoi trimite înapoi clientului rezultatul cererii.

Implementarea serverului, alături de celelalte funcții menționate, o puteți consulta în paginile următoare.

```
while(1)
{
  int client;
  threadDates* threadNou;
  int length = sizeof(from);

  printf("[server]Asteptam la portul %d...\n", PORT_CONECTARE);
  fflush(stdout);

  client = accept(socketDescriptor, (struct sockaddr *) &from, &length);                        //  acceptam un client nou

  if(client < 0)                                                                                // daca am esuat
    {
      perror("[server]Eroare la accept().\n");                                                   // afisam o eroare si continuam
      continue;
    }

    threadNou = (struct DateThread*)malloc(sizeof(struct DateThread));                  // am acceptat un client nou; pregatim un thread nou: ii alocam memorie
    threadNou->ID = i++;                                                              // asociem ID-ul threadului nou
    threadNou->DescriptorClient = client;                                             // salvam in thread descriptorul clientului

    pthread_create(&threads[i], NULL, &trateazaClientCuThread, threadNou);                                  // lansam in executie noul thread
  }
system("clear");                                                                  // la inchiderea serverului, afisam starea finala, o salvam in fisierul de configurare si inchidem
printf("STARE: %s\n", Stare.volum);

int fileConfig = open("_config_light.txt", O_WRONLY | O_TRUNC);
write(fileConfig, Stare.volum, sizeof(Stare.volum));
close(fileConfig);

return 0;
;
```

```
static void *trateazaClientCuThread(void * arg)
{
   char mesaj[255];

   while(1){
     system("clear");
     printf("Stare curenta volum: %s\n", Stare.volum);

     struct DateThread threadDeProcesat;
         threadDeProcesat = *((struct DateThread*)arg);
         printf("[thread]- %d - Asteptam mesajul...\n", threadDeProcesat.ID);
         fflush(stdout);

         trimiteRaspuns((struct DateThread*)arg, mesaj);                 // trimitem raspunsul clientului

     printf("THREAD: %s\n", mesaj);
     if(!strcmp(mesaj, "exit")){
         break;
     }
   }

   pthread_detach(pthread_self());

   int fileConfig = open("_config_light.txt", O_WRONLY | O_TRUNC);          // la terminarea threadului, il inchidem si salvam starea rezultata de modificarile aduse de client
   write(fileConfig, Stare.volum, sizeof(Stare.volum));
   close(fileConfig);

   close((intptr_t)arg);
   return(NULL);
};
```

```c
void trimiteRaspuns(void *arg, char* raspunsSalvat)
{
  char sir[255], nr, i=0;
    struct DateThread threadDeProcesat;
    threadDeProcesat = *((struct DateThread*)arg);

    if(read(threadDeProcesat.DescriptorClient, sir, sizeof(sir)) <= 0){       // daca am esuat cand am citit mesajul de la client in thread
         printf("[Thread %d]\n",threadDeProcesat.ID);

     int idThreadCurent = pthread_self();                                    // identificam threadul curent

     pthread_exit(&idThreadCurent);                                           // si ii oprim executia
    }

    printf("[Thread %d]Mesajul a fost receptionat...%s\n",threadDeProcesat.ID, sir);

  if(strcmp(sir, "exit")){
    strcpy(Stare.volum, sir);
  }

  bzero(raspunsSalvat, 255);                                                  // procesam mesajul primit de la client
  strcpy(raspunsSalvat, sir);

  printf("[Thread %d]Trimitem mesajul inapoi...%s\n",threadDeProcesat.ID, sir);

    if(write(threadDeProcesat.DescriptorClient, sir, sizeof(sir)) <= 0)
    {
        printf("[Thread %d] ",threadDeProcesat.ID);
        perror("[Thread]Eroare la write() catre client.\n");
    }
    else{
    system("clear");
    printf("Stare curenta volum: %s\n", Stare.volum);
  }
}
```

4.2 Implementare Client

Clientul a fost implementat utilizând o buclă posibil infinită care îi permite să trimită către server oricât de multe cereri în vederea schimbării stării serverului.

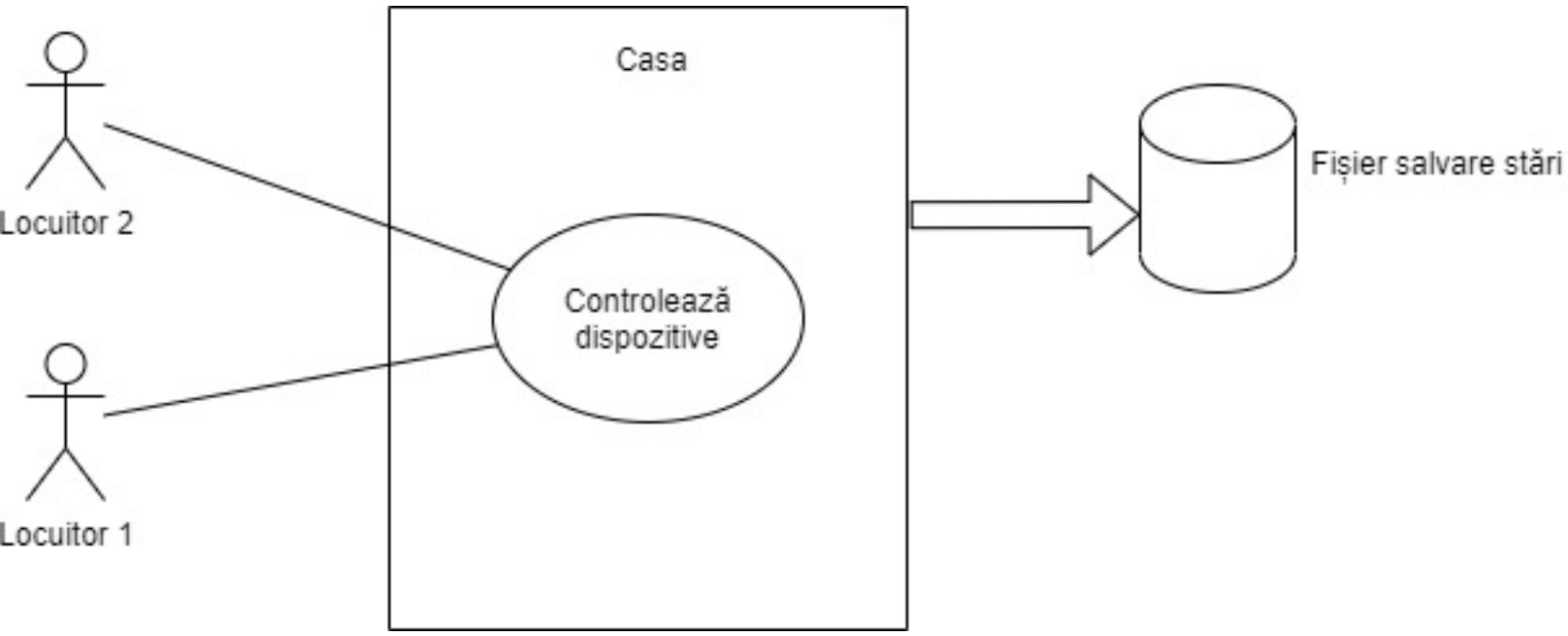
Implementarea clientului este disponibilă în paginile următoare.

```
int okConectat = 1;
while(okConectat){                                          // cat timp clientul decide sa trimita cereri
  system("clear");
  printf("[client]Introduceti un numar pentru volum: ");

  scanf("%s", bufferMesaj);                                 // se citeste cererea in buffer

  if(write(socketDescriptor, bufferMesaj, 255) <= 0)        // daca nu s-a reusit scrierea
  {
    perror("[client]Eroare la write() spre server.\n");     // afisam un mesaj de eroare si inchidem programul
    return errno;
  }

  if(read(socketDescriptor, bufferMesaj, 255) < 0)          // daca avem erori la citirea mesajului trimis de server
  {
    perror("[client]Eroare la read() de la server.\n");     // afisam un mesaj de eroare si inchidem programul
    return errno;
  }

  printf("[client]Mesajul primit este: %s\n", bufferMesaj); // afisam mesajul primit de la server
  if(!strcmp(bufferMesaj, "exit")){                             // daca clientul a decis sa se deconecteze
      okConectat = 0;                                           // iesim din bucla principala
  }
}
close(socketDescriptor);                                    // inchidem conexiunea

return 0;
}
```

4.3 Exemple de situații de utilizare

Această aplicație, cu câteva îmbunătățiri și optimizări, ar putea fi utilizată cu precădere în dezvoltarea ”smart-house”-urilor; cu un singur dispozitiv putem controla casa visurilor noastre, inclusiv de la distanță, evitându-se pierderea vreunui dispozitiv de control.

Astfel, prin crearea unei telecomenzi universale, putem controla orice dispozitiv din casă folosind o singură telecomandă.



5 Concluzii

Aplicația "IoT Simulator" nu este un simplu simulator; ea prezintă particularități importante asupra conexiunii dintre un client și un server, aducând o contribuție majoră asupra dezvoltării "smart-house"-urilor sau a dispozitivelor-smart.

Câteva îmbunătățiri care pot fi aduse aplicației sunt:

- Interfață grafică
- Permiterea adăugării de dispozitive noi în vederea controlării
- Restricționarea accesului asupra unui dispozitiv deja controlat de un proprietar
- Crearea unui sistem de login

References

[1] Pagina cursului de Rețele de Calculatoare:
`https://profs.info.uaic.ro/~computernetworks/`

[2] Pagina laboratorului de Rețele de Calculatoare:
https://sites.google.com/view/fii-rc/laboratoare?authuser=0

[3] Unealta utilizată pentru realizarea diagramelor:
https://app.diagrams.net/

[4] Despre fire de execuție: https://www.geeksforgeeks.org/thread-in-operating-system/

[5] Tutoriale LaTeX:
https://gitlab.com/eugennc/teaching/-/blob/master/GA/texample.tex
https://www.overleaf.com/learn/latex/Tutorials