



# Aplicație de tip browser FS. Client CoAP

Grecu Cristian, 1305A

Rusu Iulian, 1305A

Facultatea de Automatică și Calculatoare, Universitatea “Gh. Asachi” Iași

# Cuprins

1. Introducere .....	3
1.1 Constrained Application Protocol (CoAP) .....	3
1.2 Concepte de bază .....	3
1.2 User Datagram Protocol (UDP) .....	5
2. Implementare .....	7
2.1 Cerințe .....	7
2.2 Structura generală a aplicației Client .....	7
2.3 Implementare pachetelor CoAP .....	8
2.4 Implementarea comenzilor .....	9
2.5 Implementarea sistemului de fișiere .....	10
2.6 Implementarea clientului .....	11
2.7 Testarea aplicației .....	11
2.8 Diagrama UML a aplicației .....	12
2.9 Poze din interfață .....	13
2.10 Capturi Wireshark .....	14
2.11 Logarea la nivelul aplicației .....	14

# 1. Introducere

## 1.1 Constrained Application Protocol (CoAP)

Constrained Application Protocol (CoAP) este un protocol specializat de transfer pentru utilizare cu noduri și rețele constrânse (de exemplu, rețele cu consum redus de energie, cu pierderi). Protocolul este conceput pentru aplicații de tipul mașina la mașină (M2M), cum ar fi energia inteligentă și automatizarea.

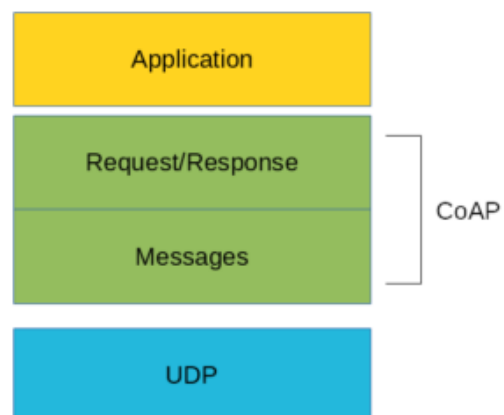
CoAP oferă un model de interacțiune cerere/răspuns între punctele finale ale aplicației, acceptă descoperirea serviciilor și resurselor și include concepte cheie ale web-ului, cum ar fi URI-uri și Internet Media. CoAP este conceput pentru a interfața cu ușurință cu HTTP pentru integrarea cu Web în timp ce îndeplinește cerințe specializate cum ar fi suport multicast, cheltuieli reduse și simplitate pentru medii constrânse.

Modelul de interacțiune al CoAP este similar cu cel al modelului client/server al HTTP-ului. Totuși interacțiunile M2M în general sunt într-o implementare CoAP atât pentru client, cât și pentru server. O cerere CoAP este echivalentă cu cea HTTP și este trimisă de client pentru a cere o acțiune pe o resursă (identificată prin URI) pe un server. Apoi serverul trimite un răspuns cu un cod de răspuns.

## 1.2 Concepte de bază

Principalele caracteristici ale protocolului CoAP sunt:

- este conceput pentru aplicații de tipul mașina la mașina cum ar fi energia inteligentă și automatizarea
- schimb de mesaje asincron
- are un overhead mic și parsarea este simplă
- URI și content-type suport
- Are capacități pentru proxy și caching



Protocolul CoAP definește 4 tipuri diferite de mesaje:

1. Confirmabile
2. Non-confirmabile
3. Acknowledge
4. Reset

Pentru tipul de cerere confirmabilă, comunicația are loc după modelul de mai jos.

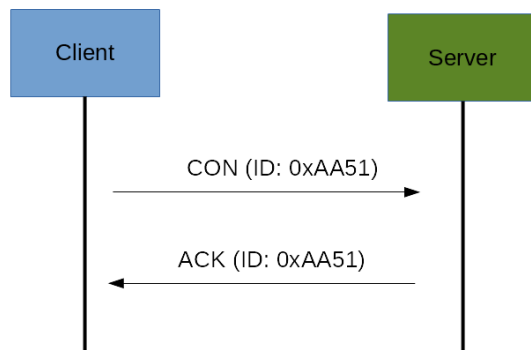


Figura 2

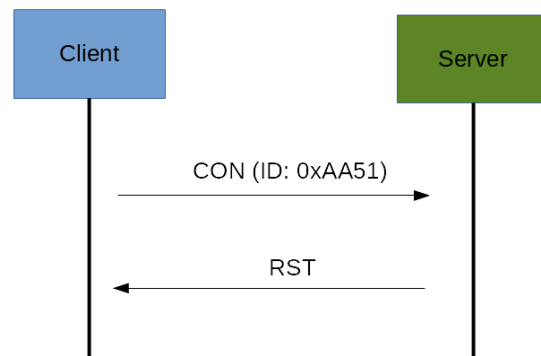


Figura 3

În figura 2 este reprezentată situația transmiterii unui mesaj care primește Acknowledge pe când în figura 3 este cazul în care serverul are probleme în a procesa mesajul și astfel în loc de Acknowledge se transmite mesajul de Reset.

Pentru tipul non-confirmabil este afișată figura următoare. Pentru acest tip de mesaj nu se returnează Acknowledge din partea server-ului.

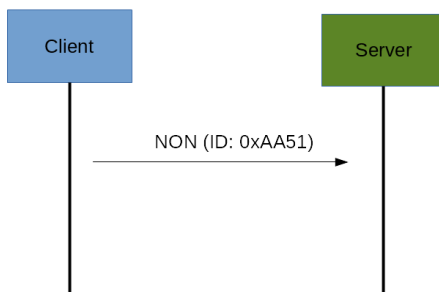


Figura 4

Modelul de cerere/răspuns în CoAP

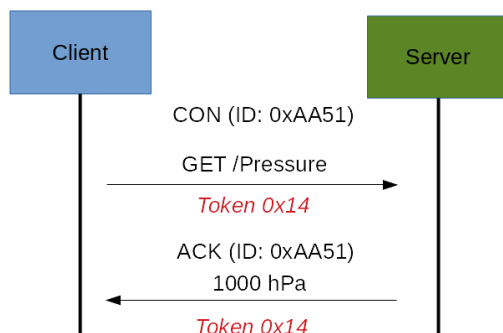


Figura 5

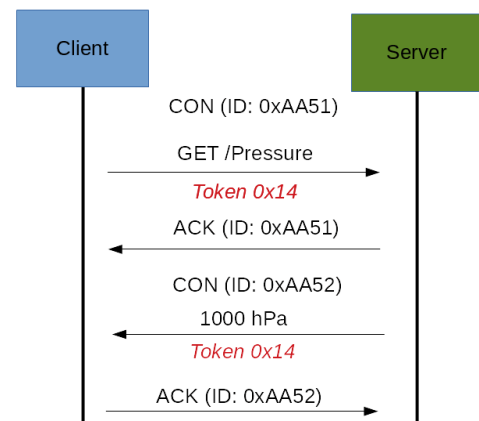


Figura 6

În figura 5 este ilustrată transmiterea unui mesaj și returnarea răspunsului sau a codului de eroare. De observat este faptul că token-ul este același pentru perechea cerere-răspuns. În figura 6, serverul nu poate răspunde imediat clientului și atunci se transmite în prima fază un pachet gol. Când răspunsul pentru client este finalizat, se transmite un mesaj confirmabil cu acesta iar clientul va trimite un Acknowledge către server pentru a-l înștiința că a primit răspunsul.

Structura unui mesaj CoAP este prezentată în figura de mai jos.

CoAP Header																																	
Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
4	32	VER		Type		Token Length				Request/Response Code								Message ID															
8	64	Token (0 - 8 bytes)																															
12	96																																
16	128	Options (If Available)																															
20	160	1	1	1	1	1	1	1	1	Payload (If Available)																							

Figura 7

## 1.2 User Datagram Protocol (UDP)

UDP este un protocol de comunicație pentru calculatoare ce aparține nivelului Transport (nivelul 4) al modelului standard OSI. CoAP este bazat pe schimbul de mesaje prin UDP între punctele finale.

Împreună cu Internet Protocol (IP), acesta face posibilă livrarea mesajelor într-o rețea. Spre deosebire de protocolul TCP, UDP constituie modul de comunicație fără conexiune. Este similar cu sistemul poștal, în sensul că pachetele de informații (corespondența) sunt trimise în general fără confirmare de primire, în speranța că ele vor ajunge, fără a exista o legătură efectivă între expeditor și destinatar. Practic, UDP este un protocol ce nu oferă siguranța sosirii datelor la destinație (nu dispune de mecanisme de confirmare); totodată nu dispune nici de mecanisme de verificare a ordinii de sosire a datagramelor sau a datagramelor duplicate. UDP dispune, totuși, în formatul datagramelor, de sume de control pentru verificarea integrității datelor sau de informații privind numărul portului pentru adresarea diferitelor funcții la sursa/destinație.

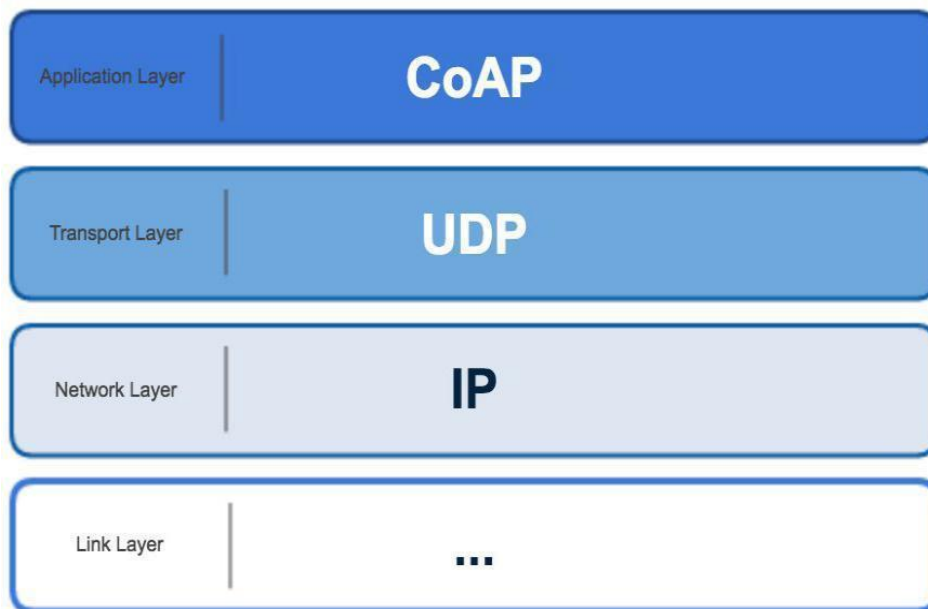


Figura 8

Caracteristicile de baza ale UDP îl fac util pentru diferite aplicații:

- orientat către tranzații - util în aplicații simple de tip întrebare-răspuns cum ar fi DNS
- este simplu foarte util în aplicații de configurări, precum DHCP sau TFTP (Trivial FTP)
- lipsa întârzierilor de retransmisie îl pretează pentru aplicații în timp real ca VoIP, jocuri online
- lucrează excelent în medii de comunicații unidirecționale precum furnizarea de informații broadcast, în servicii de descoperire (discovery services), sau în partajarea de informații către alte noduri (RIP)

## TCP vs UDP Communication

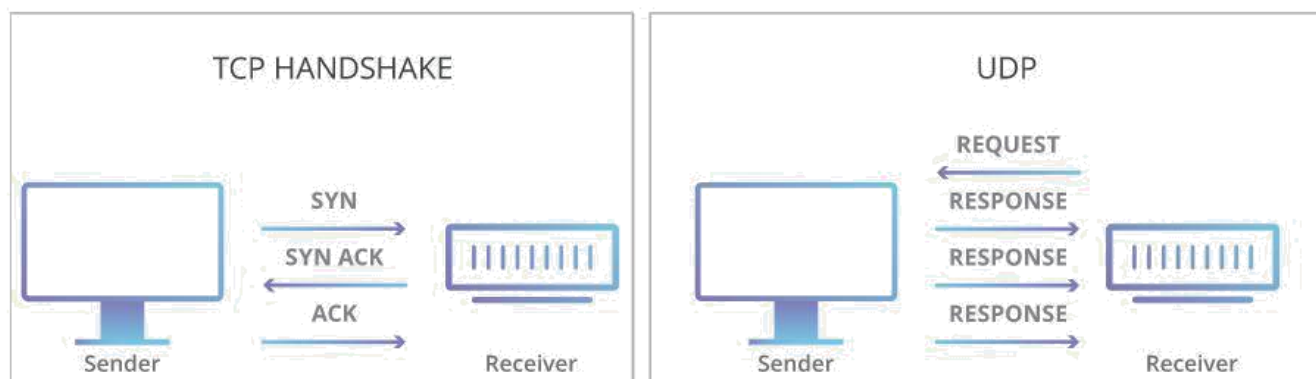


Figura 9

# 2. Implementare

## 2.1 Cerințe

- Cele două echipe (server și client) trebuie să colaboreze în vederea implementării unei soluții de accesare a unui sistem de fișiere (FS) la distanță
- Server-ul va pune la dispoziție resursele FS
- Clientul va fi capabil să execute toată gama de operații standard (acces, creare, modificare, ștergere) pentru foldere și fișiere
- Cele două aplicații trebuie să suporte un număr de coduri (vezi formatul mesajului) care să includă - codul 0.00 (mesaj fără conținut), metodele GET, POST (vezi Method Codes) și o metodă nouă propusă de echipe în contextul temei (fiți inventivi!), codurile de răspuns relevante pentru aplicație
- Aplicațiile trebuie să suporte mecanismul de comunicație cu confirmare, cât și mesaje fără confirmare (selectabil din GUI)

## 2.2 Structura generală a aplicației Client

Aplicația operează pe două threaduri - unul pentru interfața grafică (main thread), și altul pentru client (client thread).

Interfața grafică are 2 pagini:

- pagina de conectare la server, unde utilizatorul va specifica adresa IPv4 și portul serverului
- pagina de browser, unde vor apărea fișierele și butoanele de interacțiune cu acestea. Această pagină afișează lista de componente ale sistemului de fișiere sub formă tabelară și permite introducerea căii spre un director pentru a-l deschide.

Interacțiunea cu aplicația se bazează pe comunicarea între main thread și client thread printr-o coadă de mesaje, în care se pun comenzile generate prin apăsarea butoanelor. Aceste comenzi sunt transformate în cereri CoAP și trimise spre server. În caz de răspuns cu un cod de succes, comanda se execută, apelându-se o funcție de callback ce actualizează datele din GUI.

Modelul de interacțiune între utilizator și aplicație e prezentat pe scurt în schema de mai jos.

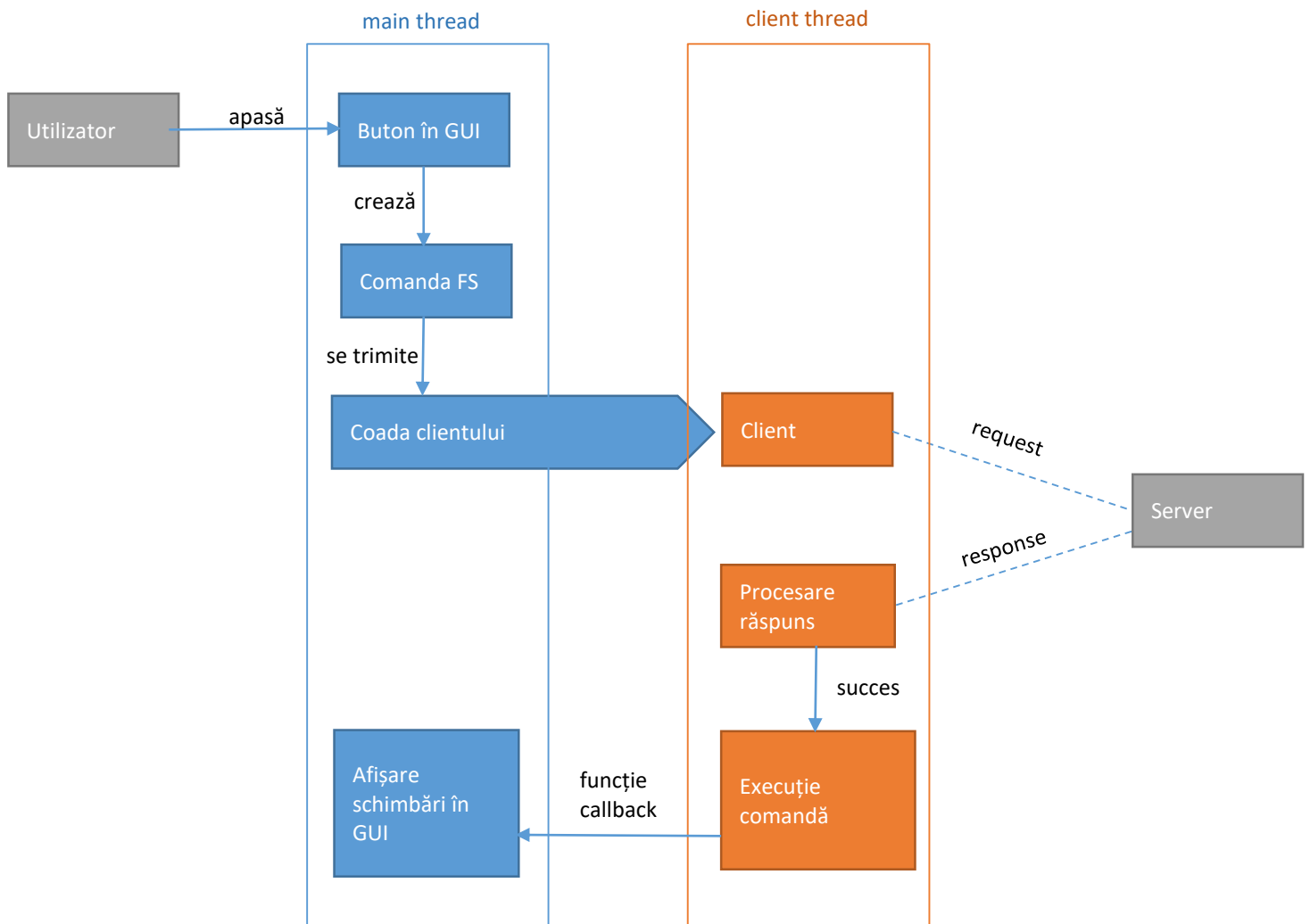


Figura 10

## 2.3 Implementare pachetelor CoAP

La comunicarea prin protocolul CoAP s-au utilizat două clase de bază – CoAP și CoAPMessage.

Clasa **CoAPMessage** reprezintă o abstractizare a unui pachet CoAP, pentru o operabilitate ușoară în interiorul aplicației. Clasa conține membri ce reprezintă câmpuri ale header-ului CoAP standard. Clasa dispune de un constructor ce primește ca parametri o secvență de biți și returnează un obiect **CoAPMessage**.

```
@classmethod
def from_bytes(cls, data_bytes: bytes) -> 'CoAPMessage':
```



În procesul parsării secvenței de biți, se fac anumite verificări legate de sintaxa protocolului, aruncând în caz de eroare excepția `InvalidFormat`.

Se verifică:

- versiunea header-ului, care trebuie să fie 1
- lungimea token-ului, care trebuie să fie în limita 0-8 octeți
- formatul corect al mesajelor de tip special (EMPTY, RESET)

Clasa `CoAP` definește modalitatea de implementare a protocolului în conformitate cu standardul RFC-7252.

În această clasă sunt definite:

- tipurile de mesaj și valorile numerice asociate acestora
- clasele de mesaje valide
- clasele de mesaje rezervate
- codurile de răspuns și de cerere
- codul pentru mesaje EMPTY
- secvența de delimitare a payload-ului (0xFF)
- versiunea header-ului
- lungimea minimă a unui header
- semnificația codurilor de răspuns recunoscute

Cele 2 metode statice definite în această clasă sunt:

- `@staticmethod`  
`def wrap(msg: CoAPMessage) -> bytes` - primește ca parametru o instanță a unui `CoAPMessage` și o transformă în biți conform protocolului CoAP.
- `@staticmethod`  
`def build_header(msg: CoAPMessage) -> bytes` - construiește header-ul unui mesaj CoAP, este apelată în prima metodă.

## 2.4 Implementarea comenzilor

Fiecare comandă are la bază clasa abstractă `FSCommand`, care oferă o interfață pentru obținerea informațiilor relevante despre tipul comenzii.

Metodele oferite de această clasă sunt:

```
def get_coap_class() -> int
def get_coap_code() -> int
def server_data_required() -> bool
def confirmation_required() -> bool
def coap_payload(self) -> str
def exec(self, response_data: str)
```

Metoda `exec()` este apelată la primirea unui răspuns cu cod de succes de la server. Această metodă poate necesita date de la server în cazul unor comenzi care corespund cererilor GET.

Pentru codificarea comenzilor s-a introdus un octet ce reprezintă codul metodei:

```
# Command headers
CMD_PING = ''
CMD_BACK = '\x01'
CMD_OPEN = '\x02'
CMD_SAVE = '\x03'
CMD_NEWF = '\x04'
CMD_NEWD = '\x05'
CMD_DEL = '\x06'
```

De menționat ca primul cod corespunde unei comenzi de ping, care e transmisă ca un mesaj EMPTY, deci nu are octet de codificare.

Pe lângă comanda de ping, sunt implementate comenzi de deschidere a unui component (metoda GET), de întoarcere la directorul părinte, de salvare a unui fișier ce a fost editat (metoda POST), de creare a unui fișier/director și de ștergere (metoda DELETE). Toate aceste comenzi moștenesc clasa `FSCommand` și suprascriu metodele necesare ca să returneze date relevante codificării lor în CoAP.

## 2.5 Implementarea sistemului de fișiere

Aplicația modelează sistemul de fișiere recepționat de la server prin obiecte abstracte, implementate prin clase, cu utilizarea modelului de proiectare *Composite*. Există clasa de bază `FSComponent`, care reprezintă orice component generic al sistemului de fișiere.

Diagrama UML a modului ce implementează sistemul de fișiere e prezentată mai jos.

Ierarhia cuprinde clasele:

- `FileContent` – reprezintă conținutul unui fișier
- `FSNamedComponent` – reprezintă o componentă a sistemului de fișiere care are nume și poate fi reprezentată în GUI prin numele său
- `File` – reprezintă un fișier
- `Directory` – reprezintă un director, are o listă de copii de tipul `FSNamedComponent`

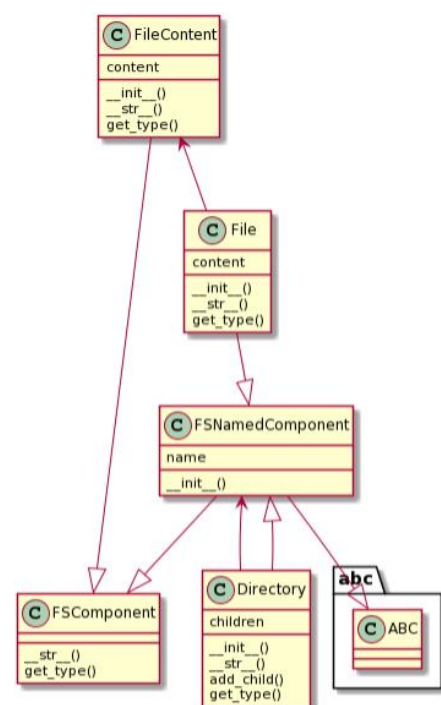


Figura 11

## 2.6 Implementarea clientului

Clasa principală a proiectului este `Client`. Această clasă realizează comunicarea cu serverul și asigură ca se îndeplinesc cerințele specificate de protocol.

Metodele principale ale acestei clase sunt:

- `def run(self)` – lansează în execuție clientul. Această metodă preia din coada de mesaje câte o comandă și o transmite la server. Dacă comanda nu e de tip confirmabil și nu necesită date de răspuns, ea va fi executată imediat.
- `def command_to_coap(self, cmd: FSCommand) -> CoAPMessage` – primește un obiect `FSCommand` și returnează codificarea sa în CoAP.
- `def send_and_receive(self, coap_msg: CoAPMessage) -> Optional[CoAPMessage]` – transmite un mesaj confirmabil sau care necesită un răspuns și așteaptă până când serverul va trimite un răspuns cu token valid și Acknowledge în cazul în care este necesar. Dacă serverul nu răspunde, se generează o excepție de tip `socket.timeout` și retransmisia se abandonează. Dacă serverul răspunde cu date invalide sau cu un Acknowledge care nu are același ID de mesaj, se face retransmisia cererii până la un număr maxim predefinit de retransmisii, după care se loghează un mesaj de eroare și se abandonează retransmisia.
- `def process_response(self, coap_response: CoAPMessage, cmd: FSCommand)` – procesează un răspuns valid primit de la server. Se ia în considerație tipul răspunsului. Dacă serverul răspunde cu Reset, ultima comandă va fi pusă din nou în coada de așteptare pentru a fi retransmisă. Dacă s-a răspuns cu un cod de succes, se execută comanda, altfel se loghează eroarea și se trimite un mesaj de eroare la interfață. Pentru răspunsurile de tip Confirmabil se trimite și un Acknowledge.

## 2.7 Testarea aplicației

În scopul testării tuturor funcționalităților, am implementat un server de test care e programat să răspundă cu o serie de mesaje predefinite conform unui scenariu. Acest server emulează comportamentul unui server real și poate fi utilizat atât pentru a trimite răspunsuri valide, cât și pentru a testa reacția clientului la răspunsuri neașteptate. Dacă clientul nu primește datele așteptate, acesta afișează un mesaj informativ în interfață și trece la următoarea comandă din coadă. În caz de răspuns cu cod de eroare, descrierea erori le salvează într-un fișier de log și de asemenea apare în interfață.

Serverul e programat să poată primi mesaje de Acknowledge la răspunsuri confirmabile și să răspundă cu același token și ID mesaj pentru a face Acknowledge cererilor.

În procesul testării, s-a utilizat mecanismului de *Piggybacking* pentru transmiterea răspunsului, însă clientul este gata să recepționeze și Acknowledge separat de răspuns.

La recepția unui mesaj, serverul generează un răspuns cu metoda de mai jos.

```
@staticmethod
def gen_responses() -> Iterator[CoAPMessage]:
    # Scenario
    response_list = [
        # Acknowledge to ping
        CoAPMessage(payload='',
                    msg_type=CoAP.TYPE_ACK,
                    msg_class=CoAP.CLASS_SUCCESS,
                    msg_code=3,
                    msg_id=0),

        # More responses here ...
    ]
    for response in response_list:
        yield response
```

## 2.8 Diagrama UML a aplicației

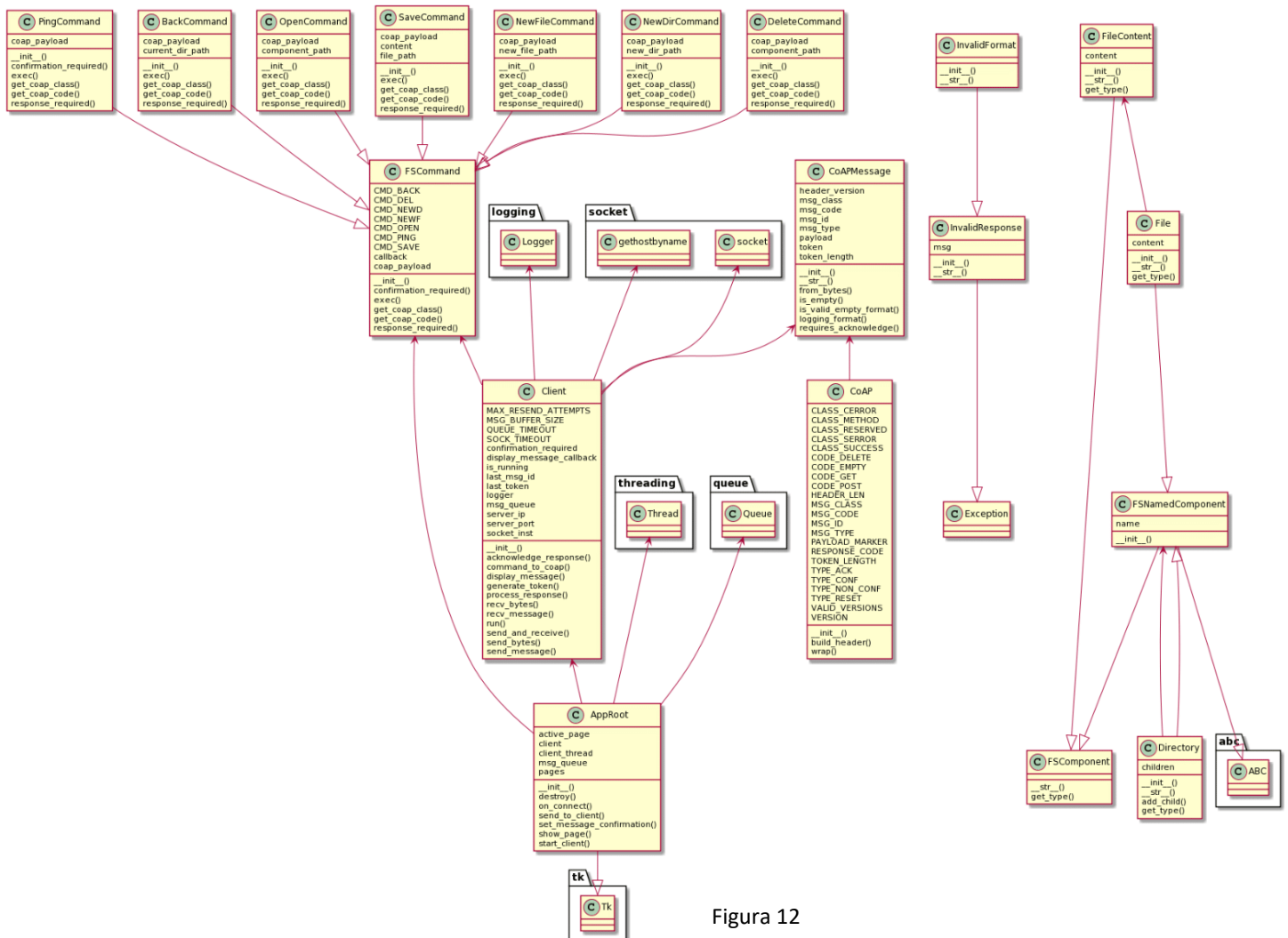


Figura 12

## 2.9 Poze din interfață

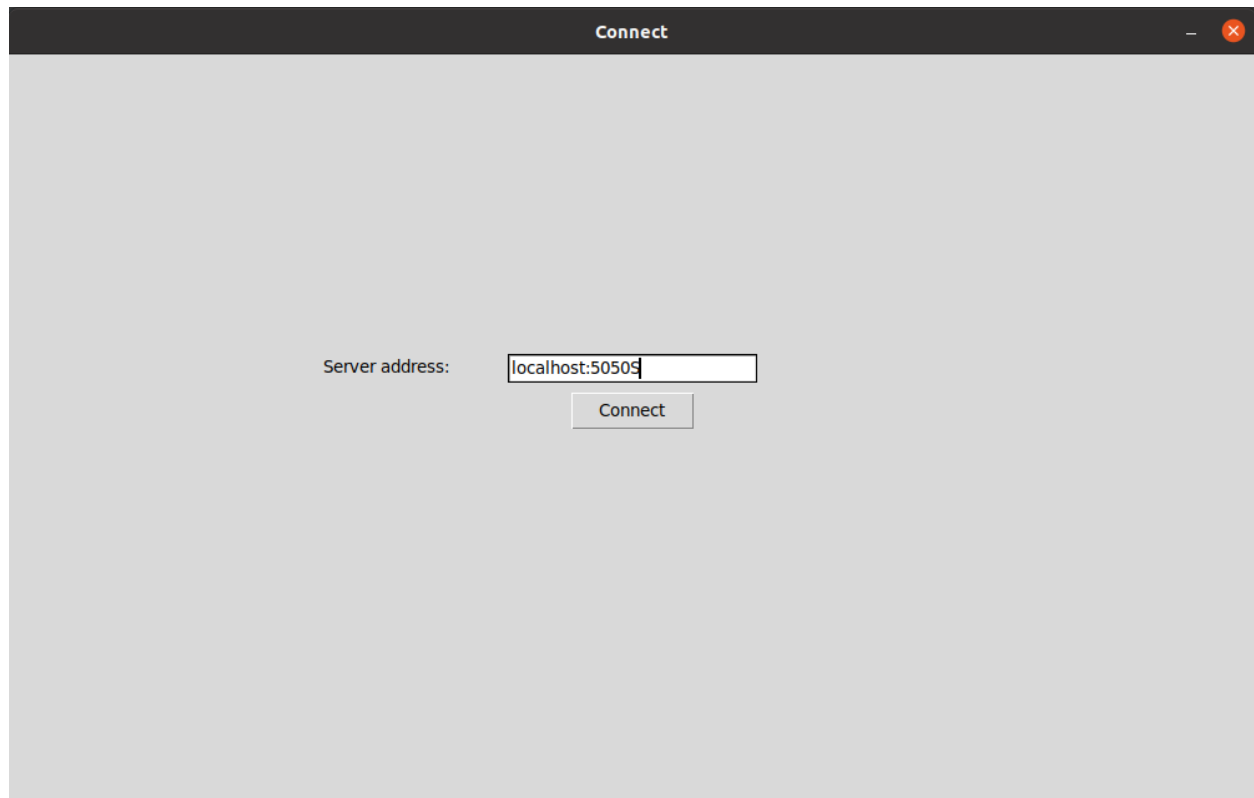


Figura 13 – Pagina de conectare la server

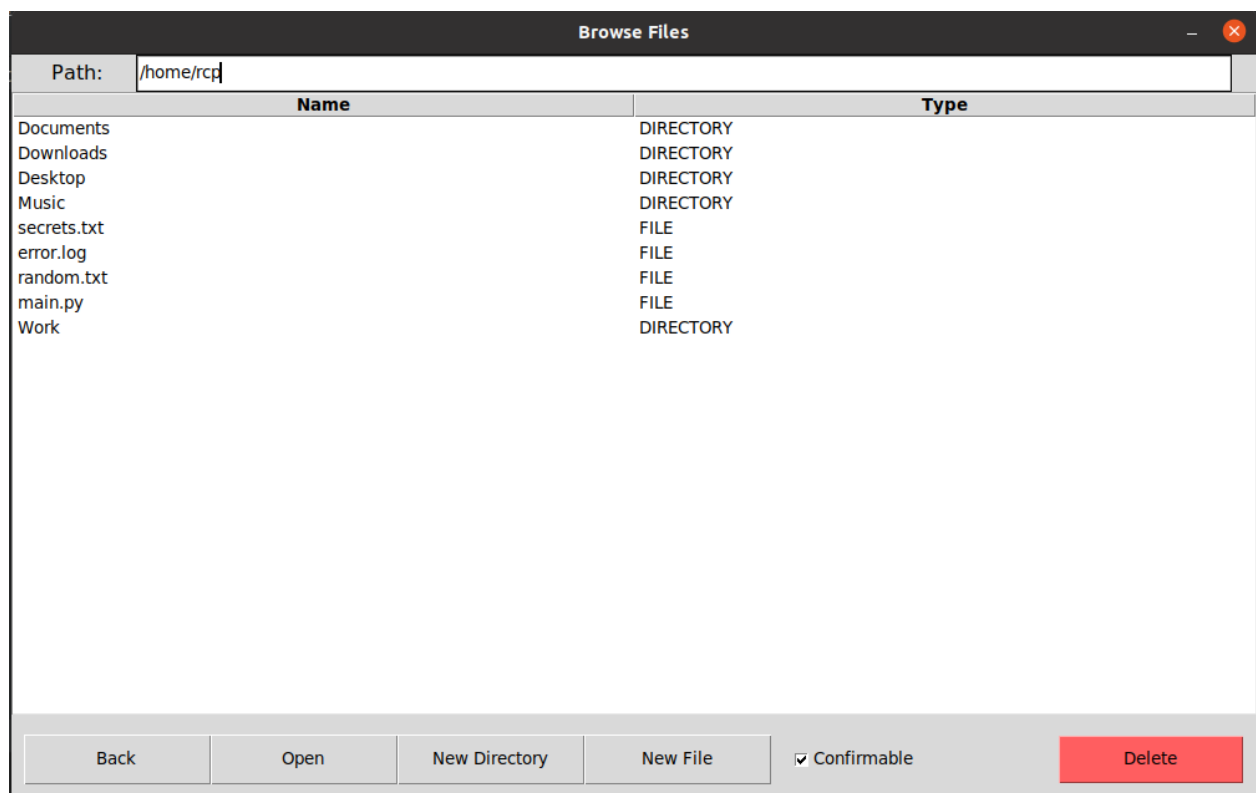


Figura 14 – Pagina de browser

## 2.10 Capturi Wireshark

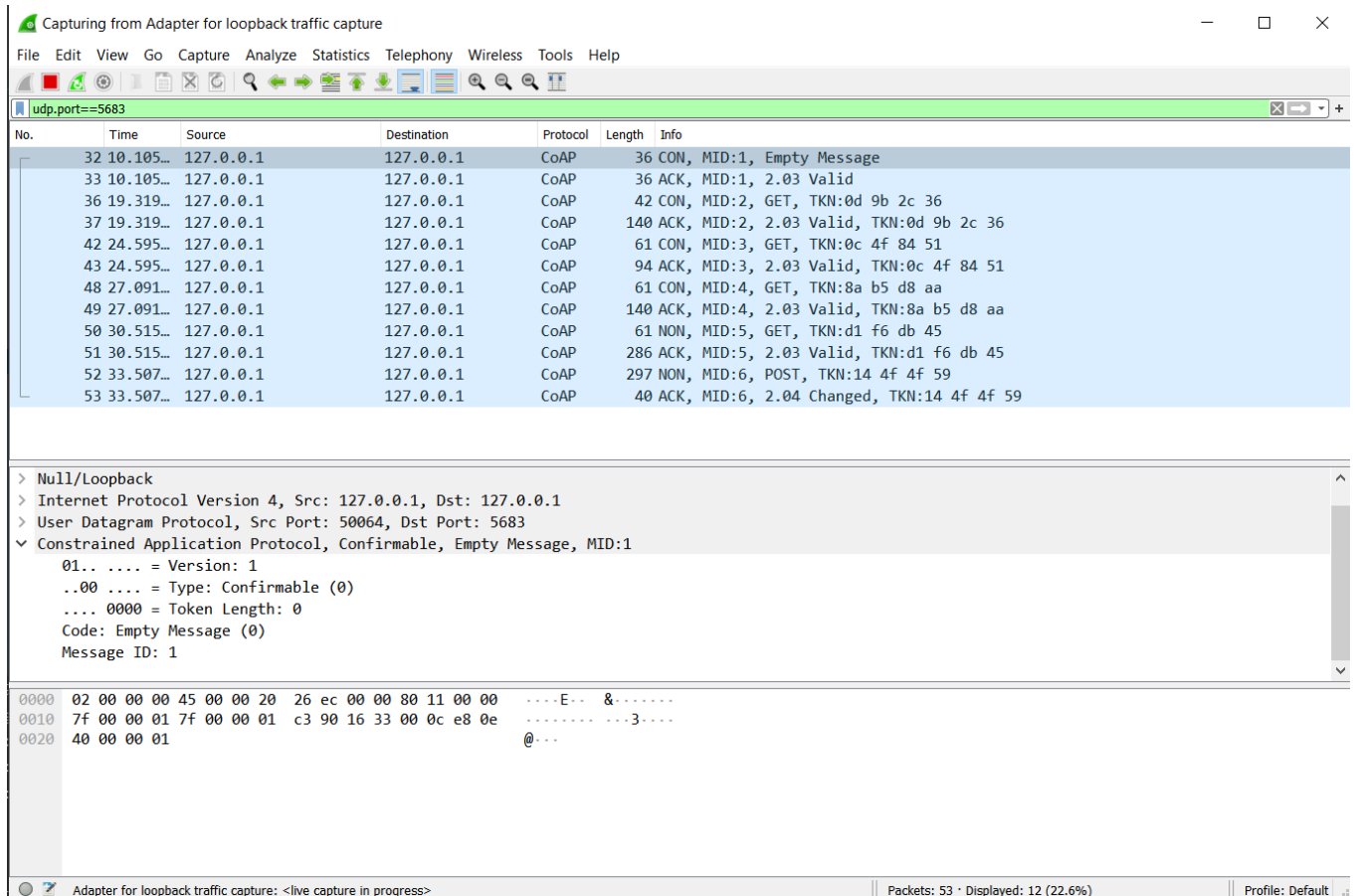


Figure 15 shows a Wireshark capture of CoAP traffic on a loopback adapter. The packet list displays a sequence of CoAP messages (CON, ACK, NON) between 127.0.0.1. The packet details pane shows the structure of a CoAP message (Version 1, Confirmable, Empty Message). The packet bytes pane shows the raw data.

No.	Time	Source	Destination	Protocol	Length	Info
32	10.105...	127.0.0.1	127.0.0.1	CoAP	36	CON, MID:1, Empty Message
33	10.105...	127.0.0.1	127.0.0.1	CoAP	36	ACK, MID:1, 2.03 Valid
36	19.319...	127.0.0.1	127.0.0.1	CoAP	42	CON, MID:2, GET, TKN:0d 9b 2c 36
37	19.319...	127.0.0.1	127.0.0.1	CoAP	140	ACK, MID:2, 2.03 Valid, TKN:0d 9b 2c 36
42	24.595...	127.0.0.1	127.0.0.1	CoAP	61	CON, MID:3, GET, TKN:0c 4f 84 51
43	24.595...	127.0.0.1	127.0.0.1	CoAP	94	ACK, MID:3, 2.03 Valid, TKN:0c 4f 84 51
48	27.091...	127.0.0.1	127.0.0.1	CoAP	61	CON, MID:4, GET, TKN:8a b5 d8 aa
49	27.091...	127.0.0.1	127.0.0.1	CoAP	140	ACK, MID:4, 2.03 Valid, TKN:8a b5 d8 aa
50	30.515...	127.0.0.1	127.0.0.1	CoAP	61	NON, MID:5, GET, TKN:d1 f6 db 45
51	30.515...	127.0.0.1	127.0.0.1	CoAP	286	ACK, MID:5, 2.03 Valid, TKN:d1 f6 db 45
52	33.507...	127.0.0.1	127.0.0.1	CoAP	297	NON, MID:6, POST, TKN:14 4f 4f 59
53	33.507...	127.0.0.1	127.0.0.1	CoAP	40	ACK, MID:6, 2.04 Changed, TKN:14 4f 4f 59

Packet details for packet 32 (CoAP):

- Null/Loopback
- Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
- User Datagram Protocol, Src Port: 50064, Dst Port: 5683
- Constrained Application Protocol, Confirmable, Empty Message, MID:1
  - 01.. .... = Version: 1
  - ..00 .... = Type: Confirmable (0)
  - .... 0000 = Token Length: 0
  - Code: Empty Message (0)
  - Message ID: 1

Packet bytes:

```
0000 02 00 00 00 45 00 00 20 26 ec 00 00 80 11 00 00  ....E...&.....
0010 7f 00 00 01 7f 00 00 01 c3 90 16 33 00 0c e8 0e  ....3....
0020 40 00 00 01  @...
```

Figura 15 – Exemple de pachete transmise

## 2.11 Logarea la nivelul aplicației

Aplicația utilizează modulul **logging** pentru a loga informații referitoare la pachetele transmise și la răspunsurile primite, dar și eventualele erori precum timeout-uri sau răspunsuri invalide, neașteptate sau cu coduri de eroare.

Mai jos este un exemplu extras din fișierul de log:

```
<CLIENT@2021-01-20 16:04:09,517>:[INFO] (REQUEST) 40 00 00 01
<CLIENT@2021-01-20 16:04:09,518>:[INFO] Request acknowledged
<CLIENT@2021-01-20 16:04:09,518>:[INFO] (RESPONSE) 203: Valid
<CLIENT@2021-01-20 16:04:55,657>:[INFO] (REQUEST) 44 04 00 0c ed a1 3d 8f ff b'\x06/home'
<CLIENT@2021-01-20 16:04:55,658>:[INFO] Request acknowledged
<CLIENT@2021-01-20 16:04:55,658>:[ERROR] (RESPONSE) 404: Not Found
<CLIENT@2021-01-20 16:05:04,585>:[INFO] (REQUEST) 44 04 00 0e 40 62 a7 fe ff b'\x06/home'
<CLIENT@2021-01-20 16:05:05,585>:[ERROR] (TIMEOUT) Server not responding
```