Proiect Retele de calculatoare

Nafornita Ana
Virlan Francisc-Gabriel

Ce este Protocolul CoAP?

CoAP este un protocol IoT care se traduce prin "Constrained Application Protocol" și este definit în RFC 7252. Acesta este un protocol simplu cu o structură de date mică, special conceput pentru dispozitive cu resurse limitate, cum ar fi microcontrolerele și pentru rețele cu restricții. Protocolul CoAP este folosit în schimbul de date M2M și este similar cu HTTP în anumite aspecte, deși există diferențe semnificative.

Principalele caracteristici ale Protocolului CoAP includ:

- Protocol web utilizat în comunicațiile M2M cu cerințe restrânse
- Schimb de mesaje asincrone
- O structură de date compactă și ușor de parsat
- Suport pentru URI-uri și tipuri de conținut
- Capacitate de proxy și cache

Așadar, Protocolul CoAP a fost optimizat pentru a răspunde nevoilor dispozitivelor IoT cu resurse limitate.

Modelul de Mesaje CoAP

Protocolul CoAP are două straturi principale: Mesaje și Cerere/Răspuns. Straturile se ocupă de schimbul de mesaje între dispozitive și gestionează interacțiunea cerere/răspuns pe baza mesajelor de cerere/răspuns.

CoAP acceptă patru tipuri diferite de mesaje:

1. Mesaj Confirmabil
2. Mesaj Non-confirmabil
3. Mesaj de Acknowledgment (Recunoaștere)
4. Mesaj de Reset

Fiecare mesaj CoAP are un identificator unic, care ajută la detectarea duplicatelor. Mesajul CoAP este format dintr-un antet binar, opțiuni compacte și conținutul propriu-zis. Protocolul folosește două tipuri de mesaje principale: cele confirmabile și cele non-confirmabile.

- Mesajul confirmabil (CON) este un mesaj fiabil care este retrimis până când destinatarul trimite un mesaj de recunoaștere (ACK) pentru a confirma primirea acestuia.
- Mesajul non-confirmabil (NON) este un mesaj care nu necesită un ACK și este utilizat pentru transmiterea de date care nu sunt critice sau nu necesită confirmare.

Modelul de Cerere/Răspuns CoAP

Al doilea strat al protocolului CoAP gestionează cererile și răspunsurile. O cerere poate fi trimisă fie utilizând un mesaj confirmabil (CON) fie un mesaj non-confirmabil (NON). Răspunsul poate fi trimis imediat, folosind un mesaj de ACK, sau în mod asincron, folosind un mesaj confirmabil (CON).

Piggybacked

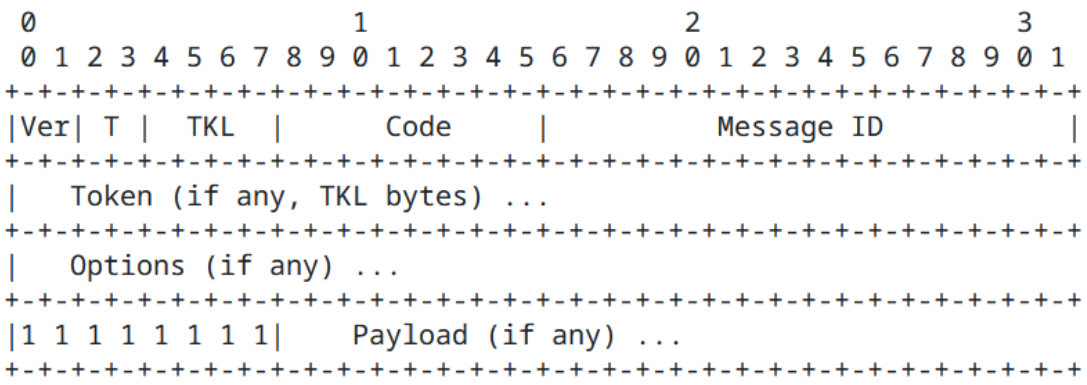
În cel mai simplu scenariu, răspunsul la o cerere este inclus direct în mesajul de Acknowledgement care confirmă primirea cererii (aceasta presupune că cererea a fost trimisă într-un mesaj Confirmable). Acest tip de transmitere se numește "Piggybacked Response"

Separate

Uneori, serverul nu poate include răspunsul direct în mesajul de Acknowledgement din diverse motive. De exemplu, dacă serverul are nevoie de timp mai mult pentru a obține informațiile cerute decât poate aștepta pentru a trimite înapoi Acknowledgement-ul, există riscul ca clientul să retrimită cererea. În acest caz, răspunsul este trimis ulterior într-un mesaj separat. Serverul poate să trimită imediat un Acknowledgement dacă știe în avans că nu va exista un răspuns atașat. Acknowledgement-ul servește ca o promisiune că cererea va fi acționată ulterior. Când serverul obține în cele din urmă reprezentarea resursei, trimite răspunsul.

Formatul mesajelor

Mesajul nu trebuie sa fie mai mare de 1152 bytes (payload de 1024 bytes)



Codificare

-[Ver] Pentru primii 2 bit folosim 01, adica versiunea 1 de protocol CoAP, iar conform IETF, celelalte valori trebuie pastrate pentru versiuni viitoare.

-[T] Tipul mesajului, format de 2 biti si pastram standardul in care 00 este confirmable, 01 non-confirmable, 10 acknoledgement si 11 reset.

-[TKL] Token length reprezentat pe 4 biti, indica lungimea valorii Token, iar pentru utilizarea aplicatiei nu este nevoie de o lungime prea mare pentru ca pe partea clientului va fi un utilizator care trimite cereri asa ca 0010 ca valoarea permanenta pentru TKL e suficienta

-[Code] 8 biti pe care ii folosim pentru a reprezenta metodele principale (01 GET, 02 POST, 03 PUT, 04 DELETE)

-[Message ID] Cod de 16 biti folosit pentru a identificarea unui mesaj. Pentru crearea codului generam un seed pseudo-random si il incrementam pentru fiecare mesaj trimis (si cererea si raspunsul contin acelasi Message ID).

-[Token] In cazul nostru tot un cod pe 16 biti generat la fel ca Message ID doar ca nu folosim acelasi pseudo-random seed. In felul asta sansele sunt extrem de scazute de a confunda mesaje si in cazul trimiterii asincrone a mesajelor (si cererea si raspunsul contin acelasi Token).

-[Options] Conform protocolului CoAP se folosesc 4 biti pentru Option Delta si 4 biti pentru Option Length. Option Length specifica lungimea campului de biti succesiv, Option Value, care poate contine informatii ajutatoare pentru metodele principale. Alegerea optiunii se face in felul urmator, se aduna Option Delta la numarul optiunii alese anterior(in cazul in care aceasta este prima optiune, se foloseste Option Delta = 0). Totusi noi am optat pentru neutilizarea acestui camp.

-[Payload] In functie de optiuni aici se mai pot adauga informatii pe numar variabil de biti.

Transmiterea mesajelor

Parametri:

name	default value
ACK_TIMEOUT	2 seconds
ACK_RANDOM_FACTOR	1.5
MAX_RETRANSMIT	4
NSTART	1
DEFAULT_LEISURE	5 seconds
PROBING_RATE	1 byte/second

Metode

GET intoarce o reprezentare a informatiei care corespunde resursei identificata de URI-ul cererii.
POST are scopul de a procesa resursa care este ingolobata in request
PUT are scopul de a actualiza sau crea
DELETE sterge resursa indicata de URI-ul cdererii

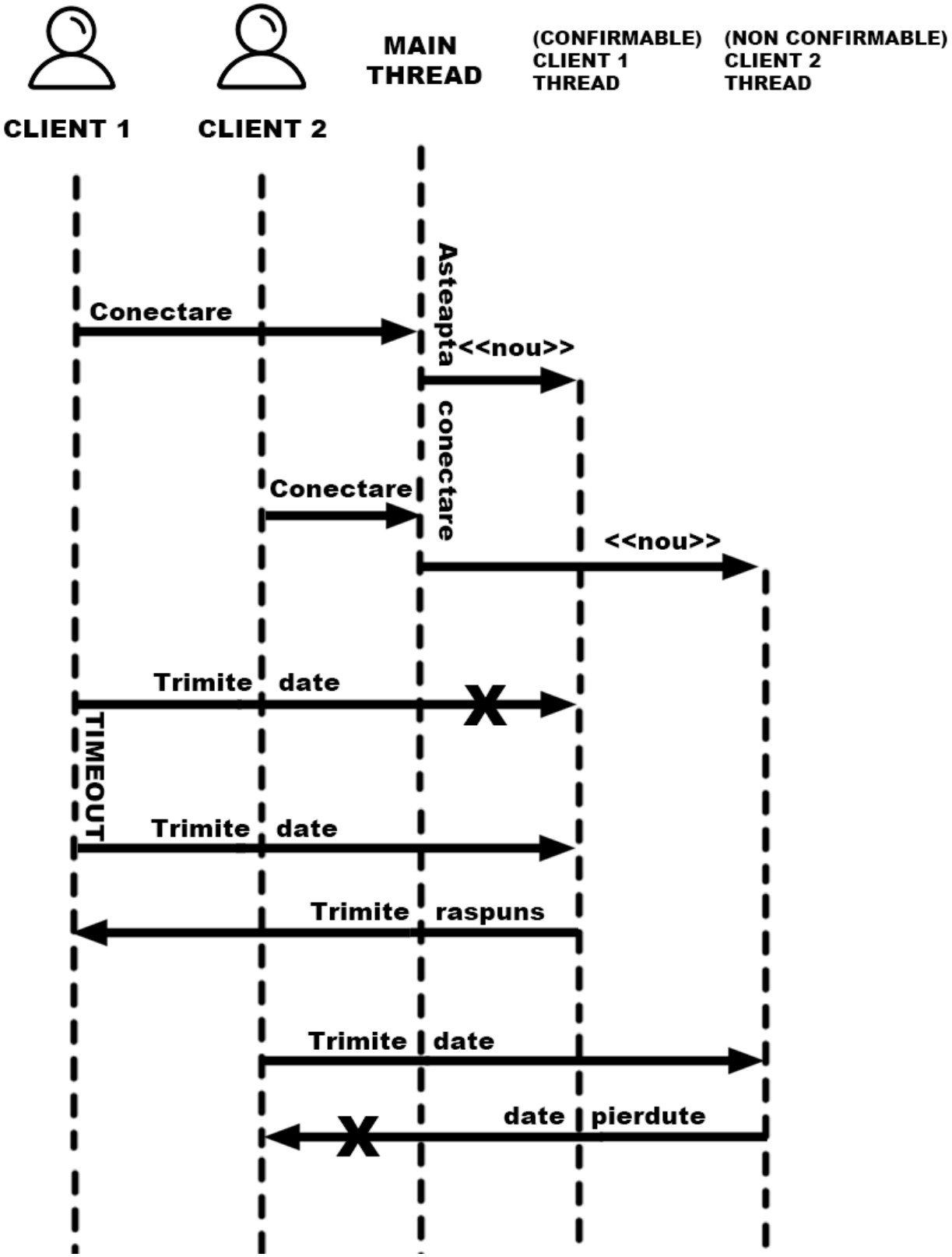
Code	Name
0.01	GET
0.02	POST
0.03	PUT
0.04	DELETE

Codul 0.00 indica un mesaj gol.
Coduri pentru posibile raspunsuri

Raspuns

Code	Description
2.01	Created
2.02	Deleted
2.03	Valid
2.04	Changed
2.05	Content
4.00	Bad Request
4.01	Unauthorized
4.02	Bad Option
4.03	Forbidden
4.04	Not Found
4.05	Method Not Allowed
4.06	Not Acceptable
4.12	Precondition Failed
4.13	Request Entity Too Large
4.15	Unsupported Content-Format
5.00	Internal Server Error
5.01	Not Implemented
5.02	Bad Gateway
5.03	Service Unavailable
5.04	Gateway Timeout
5.05	Proxying Not Supported

Model de functionare



Despachetarea mesajelor:

```
def unpack_data(self,data):
    try:
        message = Message()
        if len(data) < 96:
            raise ValueError("Data packet too short for header")

        header = data[:16]
        if len(header) != 16:
            raise ValueError("Invalid header length")

        message.version, message.message_type, message.token_length, message.code = struct.unpack('!HHLQ', header)
        message.message_id = struct.unpack('!16s', data[16:32])[0]

        token_end = 32 + message.token_length
        message.token = struct.unpack(f'!{message.token_length}s', data[32:token_end])[0]

        options_end = 64 + 32
        message.options = struct.unpack('!32s', data[64:options_end])[0]

        message.payload = data[options_end:]

        return message
    except struct.error as e:
        print(f"Error unpacking data: {e}")
        return None
```

Functionalitate:

- Verifica si extrage parti specifice din pachetul de date, cum ar fi versiunea mesajului, tipul, lungimea token-ului, codul, ID-ul mesajului, token-ul, optiunile si payload-ul.
- Intoarce un obiect Message cu datele extrase sau None in caz de eroare.

Procesarea mesajului:

(payload a fost procesat drept un sir de caractere de forma action;folder_path;content)

```python
def handle_request(self, data, client_address):
    message = self.unpack_data(data)
    if message:
        try:
            parts = message.payload.decode('utf-8').split(';')
            action = parts[0]
            folder_path = parts[1]
            file_name = parts[2]
            content = parts[3] if len(parts) > 3 else None

            if action == "create":
                success = create_empty_file(folder_path, file_name)
            elif action == "delete":
                success = delete_file(folder_path, file_name)
            elif action == "update":
                success = update_file(folder_path, file_name, content)
            elif action == "read":
                file_content = read_file_content(folder_path, file_name)
                success = file_content is not None

            response_payload = "Success" if success else "Failure"
            if action == "read" and success:
                response_payload = file_content

            response = struct.pack(f'!BBBHQ{len(message.token)}s{len(response_payload)}s',
                                   message.version, 2, 0, 205, message.message_id, message.token,
                                   response_payload.encode('utf-8'))
            self.server_socket.sendto(response, client_address)
```

Functionalitate:

- Dezambaleaza datele primite.
- In functie de actiunea specificata in payload (creare, stergere, actualizare, citire), executa operatiunea corespunzatoare asupra fisierelor.
- Trimite un raspuns inapoi la client cu rezultatul operatiunii.