Slovenská Technická Univerzita v Bratislave
Fakulta informatiky a informačných technológií

Návrh a implementácia efektívneho komunikačného protokolu pre IoT meteorologické zariadenia

Počítačové a Komunikačné siete

Peter Uhrin
AIS ID: 133003

Vedúci cvičenia: M. Janeba
Čas cvičenia: Štvrtok 12:00 – 14:00

Obsah:

Vývojové diagramy

Tester.py (client)

Po nastavení IPs a portov pre klient aj server, je možné zvoliť možnosť auto_message_generation. Pri tejto voľbe sa vytvoria 2 nové threads; auto_send_data a receive_data.

auto_send_data(client):

Pre odosielanie som zvolil 10 sekúnd time.sleep(). Thread skontroluje či má senzor pridelený token (či prebehla registrácia) a či je aktívny. Ak sú podmienky splnené, server pre každý senzor vygeneruje JSON správu v rozsahu preddefinovaných hodnôt. Správu taktiež uloží do pamäte senzora (nutné kvôli akcept. Test 3 – zavedenie chyby a znovuodoslanie správy). Nakoniec správu odošle.

Receive_data(client):

Prijaté správy sa spracujú vo funkcii process_message(msg) a následne sa podľa msg_type z headeru rozhodne čo sa s nimi stane. Ak je prijatá správa typu:

1. Error: prejde systém všetky odoslané správy pre daný senzor a nájde tú, ktorá má totožný time_stamp. Túto správu opätovne spracuje a odošle na server
2. Activity: ak obdrží správu o neaktivite senzora, tak po 3. správe (akcept. Test 4) sa senzor zapne a odošle správa o zapnutí senzora na server.

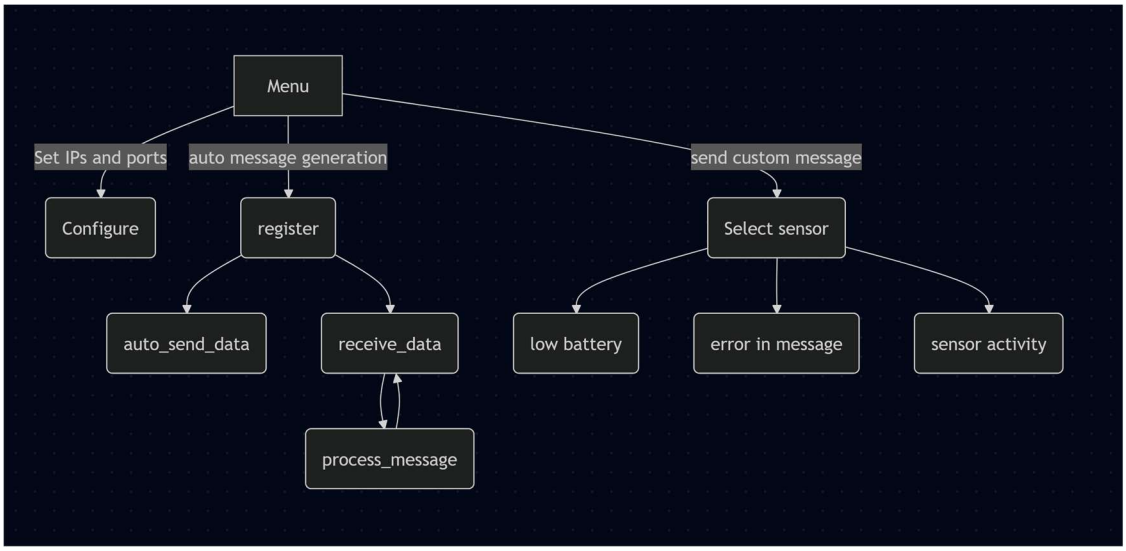

Figure 1: tester.py diagram

Server.py

Po nastavení IP a portu servera sa spustia 3 thready: listen(server), logger() a activity_check(server).

listen (server):

Toto je hlavný thread pre príjmanie správ. Správu prijme a odošle na spracovanie funkcii process_message(data,server). Tá zo správy vyberie header a podľa message type ďalej volá funkcie:

1. Handle_corrupted(dictionary, server, error)

   Flag error = 0 znamená, že vypočítané CRC nesedelo s tým ktoré prišlo v správe. Od senzora sa opätovne vyžiada táto správa.

   Error = 1 znamená, že prišla odozva od senzora so správou, ktorá bola predtým poškodená.
2. Handle_registration(dictionary, server)

   Senzoru priradí token, uloží si ho do databázy a odošle správu s vygenerovaným tokenom klientovi.

3. Handle_data(dictionary)

   Hlavná funkcia pre spracovanie dát. Pridá prijaté dáta do databázy, pozrie či je batéria na high a nastaví senzor ako aktívny (ak predtým nebol)

4. Handle_reconnect(dictionary)

   Ak bola prijatá správa od senzora znamená to že je aktívny, teda dáta sa pridajú do zoznamu a vypíše sa správa o znovupripojení.

logger (client):

Thread programu pre vypisovanie prijatých správ. Každých 10 sekúnd prejde prijaté správy a zistí či flag Log je False, teda správa ešte nebola vypísaná. Ak nebola vypísaná tak ju vypíše a flag nastaví ako true.

activity_check((server):

Tento thread kontroluje či sú senzory aktívne. Ak je posledná správa v zozname staršia ako 15 sekúnd tak spustí nový thread reconnect(sensor_id, token, battery, 0, server). Ten ak do sekundy neobdrží správu od senzora tak mu pošle správu, potom takúto správu posiela každých 5 sekúnd ale po dobu maximálne 50 sekúnd resp 10x alebo pokiaľ od senzora neprijde správa – ktorá sa pridá cez handle_data, activity sa nastaví na 1 a teda while cyklus a tým aj thread skončí.

Pozn.: Funkcia „Z“ v diagrame je ack_data(client), ktorá mala pomocou samostatného threadu riešiť akcept. Test č. 5, avšak pri kontrole akc jednotlivých správ mi program viacnásobne posielal tú istú správu, čo spôsobovalo neprehľadnosť vo výstupe. Preto som ju zakomentoval (#) a budem ju musieť ešte opraviť.

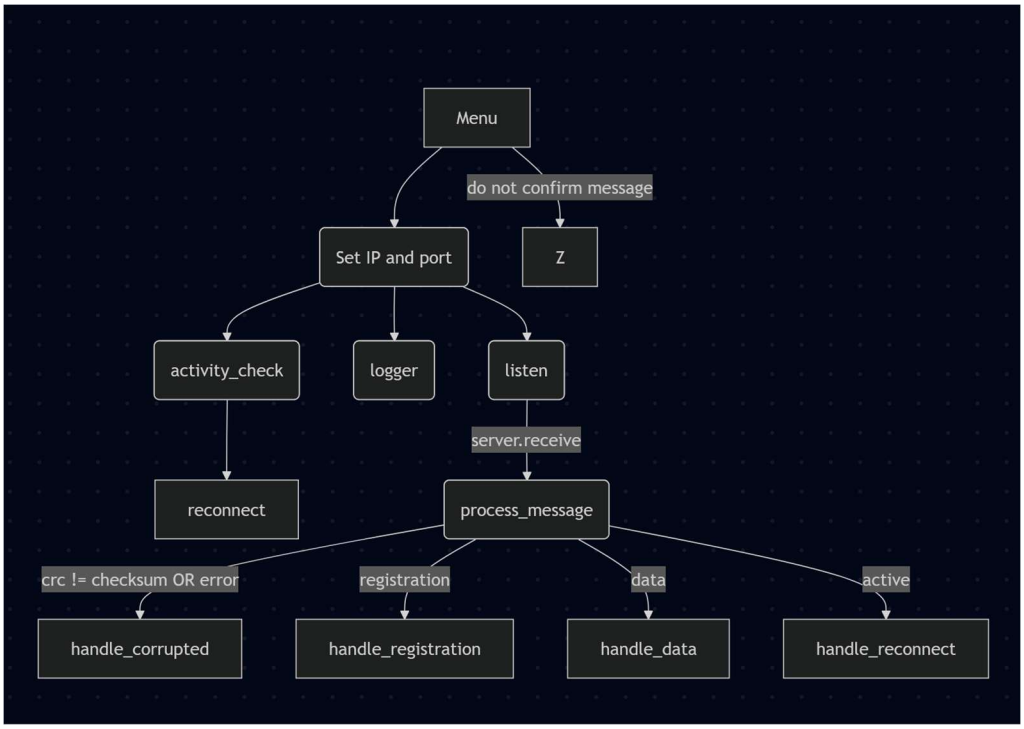

Figure 2: server.py diagram

Json správy

Každá správa odoslaná alebo prijatá je vo formáte:

"header": {

"msg_type": msg_type,

"time_stamp": time_stamp,

"sensor_id": sensor_id,

"battery": battery,

"token": token,

"crc": "",

"data": data }

Msg_type sa v programe rozlišujú: registration, data, error, active:

```
message = {
    "header": {
    "msg_type": "registration",
    "time_stamp": getTime(),
    "sensor_id": sensor_id,
    "battery": "high",
    "token": token,
    "crc": "",
    "data": {"registration": 1}
    }
}
```

Figure 3: registration

```
message = {
        "header": {
        "msg_type": "data",
        "time_stamp": timestamp,
        "sensor_id": "WindSense",
        "battery": battery,
        "token": token,
        "crc": "",
        "data": { "wind_speed": round(random.uniform(0, 50), 1),
            "wind_gust": round(random.uniform(0, 70), 1),
            "wind_direction": random.randint(0,359),
            "turbulence": round(random.uniform(0, 1), 1)}
        }
    }
```

Figure 4: WindSense data

```
message = {
        "header": {
        "msg_type": "data",
        "time_stamp": timestamp,
        "sensor_id": "ThermoNode",
        "battery": battery,
        "token": token,
        "crc": "",
        "data": { "temperature": round(random.uniform(-50.0, 60.0), 1),
        "humidity": round(random.uniform(0.0, 100.0), 1),
        "dew_point": round(random.uniform(-50.0, 60.0), 1),
        "pressure": round(random.uniform(800.00, 1100.00), 2)}
        }
    }
```

Figure 4: ThermoNode data

```
message = {
        "header": {
        "msg_type": "data",
        "time_stamp": timestamp,
        "sensor_id": "RainDetect",
        "battery": battery,
        "token": token,
        "crc": "",
        "data": { "rainfall": round(random.uniform(0, 500), 1),
        "soil_moisture": round(random.uniform(0, 100), 1),
        "flood_risk": random.randint(0,3),
        "rain_duration": random.randint(0,60)}
        }
    }
```

Figure 5: RainDetect data

```
message = {
        "header": {
        "msg_type": "data",
        "time_stamp": timestamp,
        "sensor_id": "AirQualityBox",
        "battery": battery,
        "token": token,
        "crc": "",
        "data": { "co2": random.randint(300,5000),
        "ozone": round(random.uniform(0,500), 1),
        "air_quality_index": random.randint(0, 500)}
        }
    }
```

Figure 6: AirQualityBox data

```
message = {
        "header": {
        "msg_type": "error",
        "time_stamp": timestamp,
        "sensor_id": sensor_id,
        "battery": battery,
        "token": token,
        "crc": "",
        "data": { "error: 1"}
        }
    }
```

Figure 7: error

```
message = {
    "header": {
    "msg_type": "active",
    "time_stamp": timestamp,
    "sensor_id": sensor_id,
    "battery": battery,
    "token": token,
    "crc": "",
    "data": { "active: 1"}
    }
}
```

Figure 8: active

Sekvenčný diagram pre komunikáciu pre UAT1 a UAT2

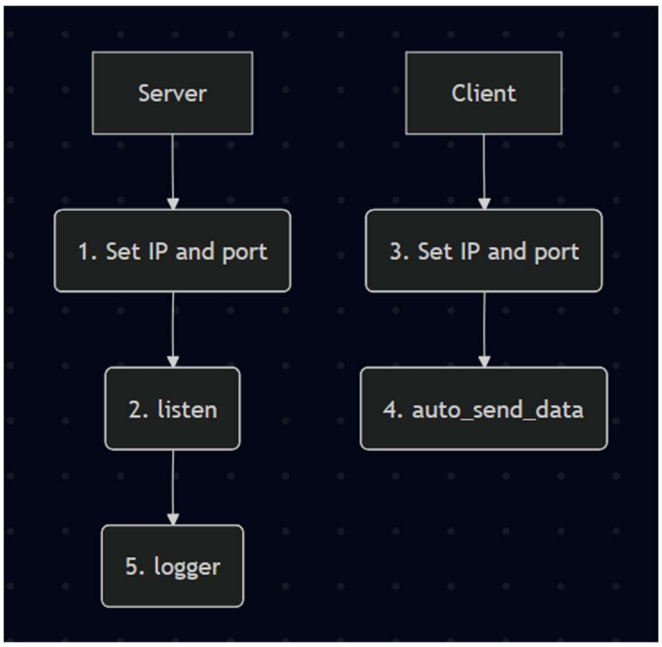

Figure 9: UAT1

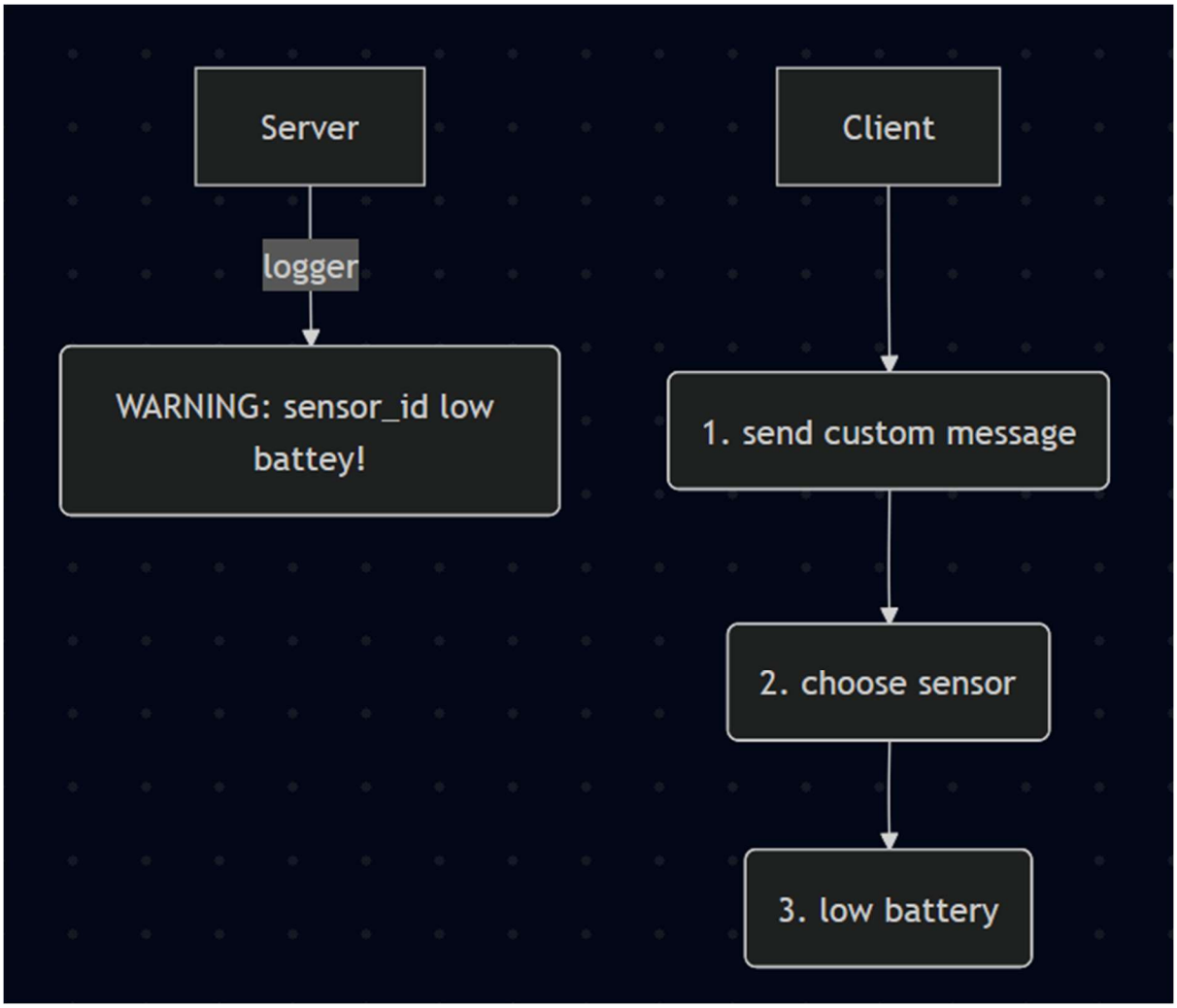

Figure 10: UAT2

Zoznam použitých knižníc

Json, socket, crc, threading, time, random a messages.py

Návrh merania efektivity prenášaných dát

Meranie efektivity môže prebiehať podľa rôznych parametrov:

1. payload_size = veľkost prenášaných dát: ethernet header + UDP + IP + JSON
2. Počet odoslaných vs prijatých správ
3. Počet správ so správnym/nesprávnym CRC
4. Latencia vďaka sledovaniu time_stamp v jednotlivých správach

Môžeme ju vypočítať: Efficiency% = (payload_size / payload_total) * 100

Alebo: Latency = time_received – time_sent

Alebo: message_good_crc_total – message_bad_crc_total

Alebo: messages_sent – messages_received