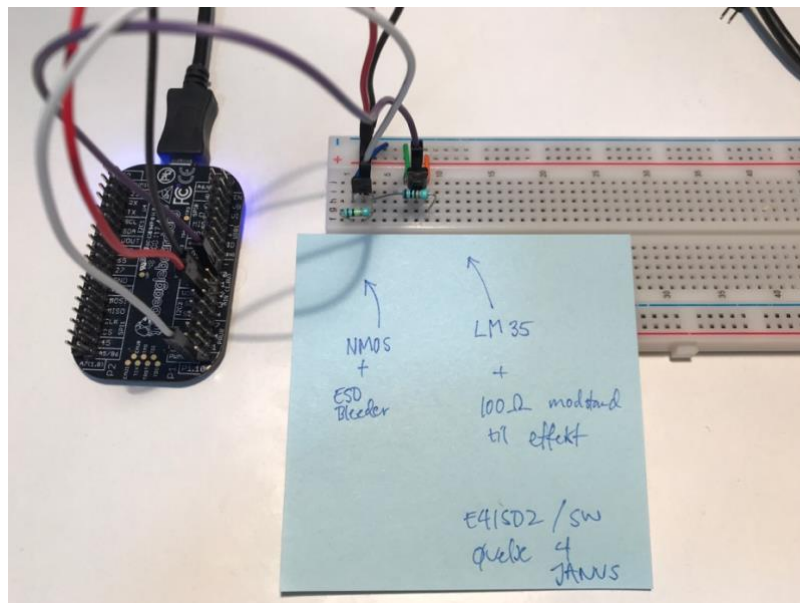


Temperaturservice via RPC på PocketBeagle

E4ISD2/SW – Øvelse 5



Rapportskriver:	Janus Bo Andersen
Studienummer:	JA67494
Afleveringsdato:	12. april 2020

Indholdsfortegnelse

1	Indledning og formål	4
2	Problemstilling / kravsspecifikation	4
3	Systemdesign.....	5
4	Hardware	6
5	Softwareimplementering	6
5.1	Inkludering af JSON-bibliotek.....	6
5.2	Datatyper.....	6
5.3	Validering af besked som valid JSON og JSON-RPC ver. 2.0.....	7
5.4	Afkodning af specifikt metodekald og svar med temperaturværdi.....	9
5.5	Aflæsning af parameterliste.....	10
5.6	Ukendt metodekald.....	11
6	Test	11
6.1	Valide RPC-kald med korrekt svar.....	11
6.2	Fejlkoder.....	12
6.3	Resultater	12
6.4	Diskussion, forbedringsmuligheder	13
7	Samlet konklusion	13
8	Referencer	14
9	Bilag	15
9.1	Pin-outs på Pocketbeagle.....	15

Forkortelser og definitioner

Forkortelse, begreb	Definition
ADC	Analog to Digital Converter
JSON	JavaScript Object Notation
PWM	Pulse-Width Modulation
Regex	Regular Expressions
RPC	Remote Procedure Call
SIGHUP	Hangup Signal
TCP	Transmission Control Protocol

1 Indledning og formål

Denne rapport besvarer øvelse 5, som er den tredje af de obligatoriske afleveringsopgaver i software-delen af kurset E4ISD2. Rapporten er udarbejdet individuelt.

Formålet med opgaven er at udvide det distribuerede system¹ fra øvelse 4. Kommunikationen skal struktureres vha. en RPC-protokol, så kommandoer, parametre og svar overføres på en struktureret og standardiseret måde. Til RPC-protokollen benyttes JSON-RPC version 2.0 [4]. Til JSON-implementering bruges biblioteket "JSON for Modern C++" [3], i nyeste version per d.d. (ver. 3.7.3). Det adskiller sig fra det foreslåede RapidJSON ift. API, men ikke i kernefunktionalitet (sammenlign [3] med [6-7])².

Løsningen til PocketBeagle er udviklet i C++ ved brug af cross-compiling og remote debugging i Eclipse, med toolchain fra Linaro (arm-linux-gnueabi ver. 7.4.1 2019-02). C++ standarden er C++14 (c++1y).

Løsningen til Linux på desktop er udviklet med CLion. Her er C++ standarden også sat til C++14.

2 Problemstilling / kravsspecifikation

Systemet er en videreudvikling af **temperaturservicen**, som blev kravspec'et, udviklet og dokumenteret i øvelse 4. I øvelse 5 skal systemet ændres således:

- Kommunikation foregår vha. RPC:
 - Kald: Kommando til server om at sende temperatur.
 - Svar: Den målte temperatur returneres til klient.
- Både klient og server benytter RPC-kald til at indpakke hhv. anmodninger (requests) og svar (responses).
- Valide kald og svar:
 - Kald: Metode "GETTEMP"
 - Svar: <værdi>, seneste aktuelle temperaturmåling.

Teknologi:

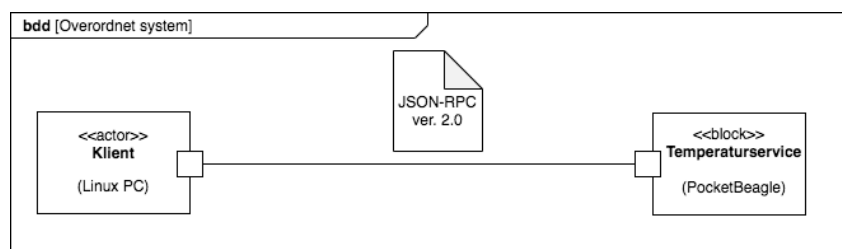
- RPC-protokollen skal være JSON-RPC [4].
 - I denne opgave vælges nyeste specifikation, version 2.0.
- JSON implementeres fx vha. RapidJSON [6], [7]. Andre biblioteker er også OK.
- Ingen krav om performance for servicen.

¹ "A distributed computing system is one where the resources used by the applications are spread across numerous computers which are connected by a network." [5, s. 9]

² JSON-implementationen af Niels Lohmann [3] er valgt frem for RapidJSON, fordi API'et er enklere og mere i tråd med moderne C++. JSON-objekter er fx struktureret som moderne STL-containerer, og mange algoritmer fra STL kan så benyttes direkte. Det er template-baseret og meget fleksibelt ift. datatyper. Der er desuden stream-implementering samt nem casting mellem JSON og C++-datatyper fra STL. API'et minder desuden en hel del om det Python-JSON-bibliotek, som vi i Team 2 har benyttet til PRO4.

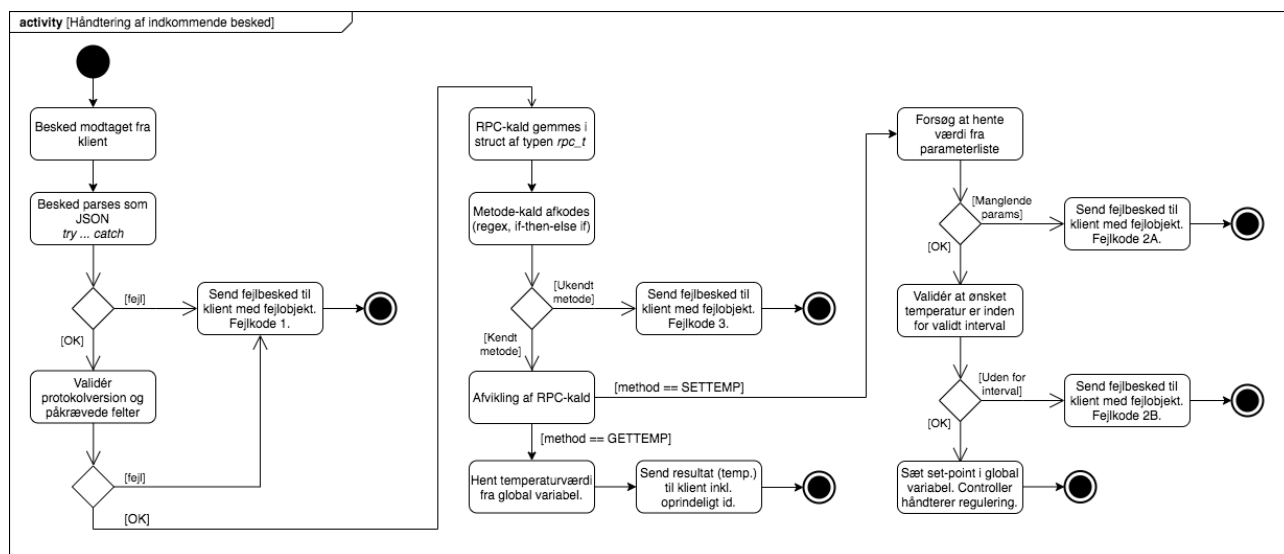
3 Systemdesign

Det overordnede systemdesign er, som vist nedenfor, at en klient og en server kommunikerer vha. RPC-kald, indeholdt i beskeder/dokumenter med specifikationen JSON-RPC ver. 2.0. Kommunikationen foregår stadig via stream sockets (TCP).



Figur 1: Overordnet blik på kommunikation i det distribuerede system

Der er behov for væsentlig mere validering, da protokollen nu bl.a. har flere krævede felter og flere forskellige fejltyper. Nedenstående aktivitetsdiagram viser håndtering af en indkommende besked.



Figur 2: Aktivitetsdiagram over håndtering af en indkommende besked

Temperaturservicen er en server med "worker" tråde. Der er hhv. en tråd til at håndtere forbindelse (*listen*), en tråd til at aflæse temperatur (*temperature*), og en tråd til at regulere temperatur op imod set-point (*controller*) samt en main-tråd, der opretter de andre tråde, og definerer globale variable. Denne sammenhæng er uændret fra øvelse 4, og derfor ikke gentaget her.

4 Hardware

Forhold omkring LM-35 sensoren [2], pinouts og elektriske begrænsninger i PocketBeagle (se bilag), kredsløb til variabel effektaflejring og samlet opstilling (se forside), er alle uændrede fra opgave 4, og derfor ikke gentaget her.

5 Softwareimplementering

Synkronisering af tråde, koordineret nedlukning, implementering af temperaturlæsning via ADC, variabel effektaflejring via PWM og closed-loop PD-regulator er uændrede fra opgave 4, og er ikke gentaget her.

Implementering af JSON-RPC-specifikationen [4] og benyttelse af det valgte JSON-bibliotek til C++ [3], er illustreret i afsnittene nedenfor, og følger generelt aktivitetsdiagrammet ovenfor. Alle ændringer er foretaget i *listen.cpp*, som implementerer trådfunktionen, der håndterer indkommende beskeder (iterativ server).

5.1 Inkludering af JSON-bibliotek

JSON-biblioteket er header-only (1 fil) og nyeste version er hentet fra Github [3]. Det er derefter symlinket til `/usr/include/nlohmann`. Der sat et include-flag til C++ cross-compileren, der ellers kun benytter biblioteker fra egen tool-chain. Nedenfor er også vist, at der benyttes et forkortet namespace, og at resten af implementeringen kræver version 2.0 af JSON-RPC.

```
19 // nlohmann/json
20 #include <json.hpp>
21
22 // for convenience
23 using json = nlohmann::json;
24
25 const std::string REQUIRED_JSONRPC_VER("2.0");
26
```

Figur 3: Inkludering af JSON-biblioteket, namespacing og definition af RPC-standard

5.2 Datatyper

Der er et par centrale datatyper til behandlingen af RPC-kald. De er vist i følgende figurer.

```
42 // struct to hold the deserialized RPC call
43 typedef struct {
44     std::string protocol_ver;
45     std::string method;
46     std::string id;
47     std::string params;
48 } rpc_t;
49
```

Figur 4: Hjemmerullet typedef'et struct til at holde data for et RPC-kald efter afkodning

Den viste struct benyttes efter beskeden er afkodet. Det er ikke en essentiel mekanisme, men er implementeret for nemheds skyld.

Følgende definitioner er foretaget i tråd-funktionen, der håndterer indkommende beskeder.

```
53     float set_point_unsafe;  
54     std::string rec;  
55     json j_req;  
56     bool valid_rpc;  
57     rpc_t rpc_call;  
58
```

Figur 5: Variabeldefinitioner i tråd-funktionen

De viste variabler bliver behandlet i de følgende afsnit. Væsentligst er at bemærk linje 55, typen `json`, som kan holde et enkelt JSON-objekt, dvs. fra et variabelnavn/værdi til et helt dokument/DOM. Type-definitionen kommer fra det benyttede bibliotek.

5.3 Validering af besked som valid JSON og JSON-RPC ver. 2.0

For at afkode en besked efter JSON-RPC-specifikationen, skal beskeden først valideres som valid JSON. Der er to metoder; enten at validere op imod et foruddefineret schema (så benyttes `json::accept()` → `bool`), eller simpelthen bare at forsøge at parse beskeden, og så fange evt. exceptions.

Sidstnævnte metode er valgt, fordi der så ikke særskilt skulle opstilles et JSON-schema. Det er vist nedenfor.

```
71     /* Init */  
72     valid_rpc = false;  
73  
74     /* Attempt to receive a message from the client */  
75     rec = server.receive(BUF_SIZE);  
76  
77     /* Attempt to de-serialize the json-rpc request */  
78     std::cout << rec << std::endl;  
79     try  
80     {  
81         j_req = json::parse(rec);  
82         valid_rpc = true;  
83     }  
84     catch (json::parse_error& e) {  
85         // output exception information  
86         std::cout << "message: " << e.what() << '\n'  
87             << "exception id: " << e.id << '\n'  
88             << "byte position of error: " << e.byte << std::endl;  
89         valid_rpc = false;  
90     }
```

Figur 6: Parsing/validering af indkommende besked som JSON

Parsing forsøges med en try-catch-blok i linjerne 79-90. I linje 81 foretages den egentlige parsing af beskeden modtaget fra klient. Hvis beskeden kan parses, dvs. de-serialiseres, gemmes den i `j_req` (json-rpc-request), og `valid_rpc` er sand – indtil videre. Hvis parsing fejler, fanges og rapporteres fejlen og beskeden er *invalid*.

Antaget at beskeden var JSON, kan den nu valideres op mod JSON-RPC-specifikationen. Der benyttes to metoder fra biblioteket.

Metoden `json::find -> iterator` søger efter en *key*, og returnerer en iterator (ligesom på STL-containerne). Hvis iteratoren *ikke* peger på slutningen af JSON-objektet, så er *key* fundet. Iteratoren kan så dereferences som en pointer for at få værdien. Det gøres fx i linje 112.

Metoden `json::contains -> bool` søger også efter *key* i JSON-objektet, men returnerer en boolsk værdi.

```

92  /* Continue validation
93  * Now check for malformed JSON-RPC call
94  * Must at least contain:
95  * {
96  *   "jsonrpc": "2.0",
97  *   "method": "somemethod",
98  *   "id": "someid"
99  * }
100  */
101  if (valid_rpc) {
102      // Iterator location, can check against j_req.end() or deref'ed with *
103      auto protocol_def = j_req.find("jsonrpc");
104
105      // Boolean values t/f if keys are contained
106      auto method_def = j_req.contains("method");
107      auto id_def = j_req.contains("id");
108
109      // validate "jsonrpc":"2.0"
110      if (protocol_def != j_req.end() ) {
111          // Check that protocol version is 2.0
112          std::string version_txt = (*protocol_def).dump();
113          valid_rpc = ( REQUIRED_JSONRPC_VER.compare(version_txt) );
114      }
115
116      // validate that method and id are defined too
117      if ( !(method_def && id_def) ) {
118          valid_rpc = false;
119      }
120  }

```

Figur 7: Validering af indhold som JSON-RPC

Som beskrevet i kommentaren: Hvis beskeden indeholder visse felter, og protokolversionen er 2.0, så erklæres den som et (indtil videre) validt JSON-RPC-kald. Dette testes i hhv. linjerne 110-113 og 117-118.

Hvis valideringen er OK, så kan RPC-kaldet gemmes i den hjemmerullede datatype. Hvis valideringen ikke er OK, så sendes en fejlbesked (fejlkode 1) til klienten. Det er vist nedenfor.

```

122  // If all validation OK
123  if (valid_rpc) {
124      // Insert values into rpc struct (implicit cast to std::string)
125      rpc_call.protocol_ver = j_req["jsonrpc"];
126      rpc_call.method = j_req["method"];
127      rpc_call.id = j_req["id"];
128
129  } else {
130      // Tell the client there is an error
131      json invalid_resp;
132      invalid_resp["jsonrpc"] = REQUIRED_JSONRPC_VER;
133      invalid_resp["error"]["code"] = "1";
134      invalid_resp["error"]["message"] = "Parse error, or malformed use of protocol.";
135      invalid_resp["error"]["data"] = rec; //return the sent RPC call string
136      invalid_resp["id"] = "0";
137
138      // Send error message
139      server.send(invalid_resp.dump() + "\n");
140
141      // Restart loop to wait for new message
142      continue;
143  }

```

Figur 8: Lagring af RPC-kald eller fejlbesked til klient

Først bør bemærkes i linjerne 125-127, at biblioteket giver mulighed for implicit cast fra `json`-type til `std::string`. Desuden bør bemærkes ved dannelse af fejlbeskeden fra linje 131, at:

- Fejlbeskeden overholder JSON-RPC-specifikationen for fejlbeskeder [4].
- Indlejrede/nestede objekter dannes ved en kæde af associative arrays / hash-maps.
- Et `json`-objekt serialiseres til en UTF-8-streng med metoden `json::dump()`, og kan da sendes.
- Hvis valideringen er fejlet, startes en uendelig while-løkke forfra, og venter på en ny besked fra klient.

5.4 Afkodning af specifikt metodekald og svar med temperaturværdi

Ligesom i opgave 4 benyttes regex til at afkode, hvilken kommando der er sendt over RPC. Det er overkill med kun GETTEMP og SETTEMP, men det er nemmere at bibeholde denne implementering end at lave en ny "simplere" løsning. Desuden giver den mulighed for avanceret mønstergenkendelse, hvis grammatikken i RPC skal udvikles væsentligt.

Der medsendes parameterlister, og metodenavn er kun ét ord, så regex-mønstrene er simplere end i opg. 4.

```
32 /* Match GETTEMP followed by anything else*/
33 std::regex GET_TEMP("GETTEMP(.*)");
34
35 /* Matching of SETTEMP is now a METHOD and a PARAM
36 * Match e.g. SETTEMP 37.1, with values possible like 37, 37., 37.1, 37.11
37 * But there can only be one decimal point ( [\.\.]? )
38 * And then anything else can follow .* */
39 //std::regex SET_TEMP("SETTEMP ([0-9]+[\.\.]?[0-9]*).*");
40 std::regex SET_TEMP("SETTEMP(.*)");
```

Figur 9: Regex-mønstre til matching af metodekald

I kodestumpen nedenfor testes, om RPC-kaldet er til metoden "GETTEMP". I så fald, som vist i aktivitetsdiagrammet, hentes den seneste temperaturmåling (`cur_temp`). Denne indsættes i et JSON-svar, hvor også det oprindelige besked-ID indsættes, således at klienten ville kunne matche RPC-kald og RPC-svar.

```
145 /* Create a regex match object to extract match groups */
146 std::smatch s_match;
147
148 /* GET TEMP */
149 if ( std::regex_search(rpc_call.method, s_match, GET_TEMP) && s_match.size() > 0 ) {
150     // Build message to send
151     json j_resp;
152     j_resp["jsonrpc"] = REQUIRED_JSONRPC_VER;
153     j_resp["result"] = cur_temp;
154     j_resp["id"] = rpc_call.id;
155
156     // Send temperature result
157     server.send(j_resp.dump() + "\n");
158 }
```

Figur 10: Hentning af seneste temperaturværdi og afsendelse af resultat / response

Således genereres et svar, der er i overensstemmelse med JSON-RPC-specifikationen.

5.5 Aflæsning af parameterliste

Hvis metodekaldet er til "SETTEMP", afvikles nedenstående kode.

Det tjekkes nu, *just-in-time*, om RPC-kaldet indeholder et *params*-felt. I så fald hentes første værdi fra parameterlisten, som så castes til et float. Bemærk at værdien endnu ikke er valideret ift. validt temperaturinterval, og derfor er *_unsafe*.

```

161     else if ( std::regex_search(rpc_call.method, s_match, SET_TEMP) && s_match.size() > 0 ) {
162
163         // If params included
164         if ( j_req.contains("params") ) {
165             /* try to extract the matched set point value
166              * It is still unsafe, not sanitized */
167             set_point_unsafe = j_req["params"][0];
168
169         } else {
170
171             std::string errmsg = "Please include params object, like \"params\": [40.4]";
172             json invalid_resp;
173             invalid_resp["jsonrpc"] = REQUIRED_JSONRPC_VER;
174             invalid_resp["error"]["code"] = "2A";
175             invalid_resp["error"]["message"] = errmsg;
176             invalid_resp["error"]["data"] = j_req;
177             invalid_resp["id"] = rpc_call.id;
178
179             // Send error message
180             server.send(invalid_resp.dump() + "\n");
181
182             // Go back and wait for next message
183             continue;
184         }

```

Figur 11: Værdi udhentes fra JSON-parameterliste

Hvis der ikke er en parameterliste i RPC-kaldet, så kan SETTEMP ikke udføres, og der sendes en fejlbesked retur til klienten. Fejlkode er 2A, som vist i aktivitetsdiagrammet. Der inkluderes en beskrivende fejltekst.

Set-point til regulatoren sættes, som i opgave 4, hvis ønsket temperatur ligger i muligt interval. Ellers sendes en RPC-fejlbesked retur, denne gang med fejlkode 2B og en anden beskrivende fejltekst.

```

186         // Only change if the desired temp is inside valid range
187         if (set_point_unsafe >= TEMP_MIN && set_point_unsafe <= TEMP_MAX) {
188             set_point = set_point_unsafe;
189         } else {
190             std::stringstream message;
191             message << "Invalid temperature range, must be between "
192                 << TEMP_MIN << " and " << TEMP_MAX << ".\n";
193
194             json invalid_resp;
195             invalid_resp["jsonrpc"] = REQUIRED_JSONRPC_VER;
196             invalid_resp["error"]["code"] = "2B";
197             invalid_resp["error"]["message"] = message.str();
198             invalid_resp["error"]["data"] = j_req;
199             invalid_resp["id"] = rpc_call.id;
200
201             // Send error message
202             server.send(invalid_resp.dump() + "\n");
203         }

```

Figur 12: Validering af temperatur-set-point og fejlbesked, hvis uden for interval

5.6 Ukendt metodekald

Endelig behandles muligheden, at JSON-RPC-kaldet forsøger at kalde en metode, som ikke er kendt eller tilgængelig i API'et. Det udløser fejlkode 3, jf. aktivitetsdiagrammet, og en besked til klienten om at metoden ikke genkendes.

```
205     } else {
206         /* Valid RPC, but unrecognized method */
207         json invalid_resp;
208         invalid_resp["jsonrpc"] = REQUIRED_JSONRPC_VER;
209         invalid_resp["error"]["code"] = "3";
210         invalid_resp["error"]["message"] = "Method not recognized.";
211         invalid_resp["error"]["data"] = "Called method: " + rpc_call.method;
212         invalid_resp["id"] = rpc_call.id;
213
214         // Send error message
215         server.send(invalid_resp.dump() + "\n");
216     }
217 }
```

Figur 13: Fejlbesked ved kald af ukendt metode

6 Test

Formålet med testen er demonstrere, at:

- Serveren kan modtage og forstå RPC'kald
- Serveren kan besvare fejlagtige RPC-kald vha. fejlbeskeder fra JSON-RPC-standardens.
- Serveren reagerer som forventet på hhv. en OK RPC-beskeder og et par fejlagtige RPC-beskeder.

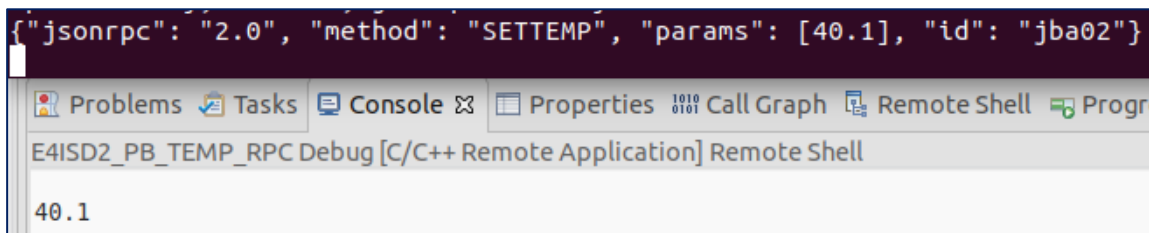
Det er ikke hovedformålet at monkey-teste valideringens robusthed (selvom det også ville være en god idé).

6.1 Valide RPC-kald med korrekt svar

Først afsendes en valid JSON-RPC-besked for at kalde GETTEMP (bemærk, LM35 var ikke tilkoblet under afvikling af test, så resultatet er højt grundet flydende ADC-input). Første linje er RPC-kald. Anden linje er serverens svar.

```
{"jsonrpc": "2.0", "method": "GETTEMP", "id": "jba01"}
{"id": "jba01", "jsonrpc": "2.0", "result": 157.763671875}
```

Det er som forventet, idet der svares med samme protokol, med det oprindelige ID, og der returneres en resultatværdi. Dernæst kaldes SETTEMP(40.1). Kommandoen sendes afsted, og der er ikke defineret nogen returbesked eller returværdi. Fra Eclipse ses, at den korrekte set-point-værdi er modtaget og aflæst.



```
{"jsonrpc": "2.0", "method": "SETTEMP", "params": [40.1], "id": "jba02"}
```

The screenshot shows the Eclipse IDE interface. The top bar includes tabs for Problems, Tasks, Console, Properties, Call Graph, Remote Shell, and Program. The Console tab is active, showing the output of the JSON-RPC call. The text in the console is: `E4ISD2_PB_TEMP_RPC Debug [C/C++ Remote Application] Remote Shell` followed by the value `40.1`.

6.2 Fejlkoder

Her forsøges det at sende noget skrald for at fremprovokere fejlkode 1:

```
adsklgalsdnglan
{"error":{"code":"1","data":"adsklgalsdnglan\r\n","message":"Parse error, or m
alformed use of protocol."},"id":"","jsonrpc":"2.0"}

E4ISD2_PB_TEMP_RPC Debug [C/C++ Remote Application] Remote Shell
adsklgalsdnglan

message: [json.exception.parse_error.101] parse error at line 1, column 1: syntax error while parsing value - invalid literal; last read: 'a'
exception id: 101
byte position of error: 1
```

Serveren returnerer en fejlbesked som forventes, med indlejret fejlobjekt. Udfaldet af try-catch-blokken vises i Eclipse, og således ses det, at beskeden ikke valideres som JSON. Som ønsket.

Dernæst afsendes et korrekt formatteret JSON-RPC-kald, dog med en ukendt metode. Nedenfor vises, at serveren svarer med fejlkode 3, og beretter, at metoden ikke er genkendt. Som ønsket.

```
{"jsonrpc": "2.0", "method": "POWEROFF", "id": "jba"}
{"error":{"code":"3","data":"Called method: POWEROFF","message":"Method not re
cognized."},"id":"jba","jsonrpc":"2.0"}
```

Endelig forsøges det at fremprovokere fejlkode 2A og 2B. Fejlkode 2A fremprovokeres ved manglende parameterliste, som beskrevet i teksten:

```
{"jsonrpc": "2.0", "method": "SETTEMP", "id": "jba"}
{"error":{"code":"2A","data":{"id":"jba","jsonrpc":"2.0","method":"SETTEMP"},"message":
"Please include params object, like \"params\": [40.4]"},"id":"jba","jsonrpc":"2.0"}
```

Fejlkode 2B fremkommer ved en ugyldig temperatur, her forsøges det at sætte temperaturen til 8000 grader:

```
{"jsonrpc": "2.0", "method": "SETTEMP", "params": [8000], "id": "jba"}
{"error":{"code":"2B","data":{"id":"jba","jsonrpc":"2.0","method":"SETTEMP","params":[8
000]},"message":"Invalid temperature range, must be between 20 and 60.\n"},"id":"jba","
jsonrpc":"2.0"}
```

Dette er den korrekte fejlbesked, og det oprindelige RPC-kald er igen indlejret som data i fejlobjektet, der er medsendt i fejlbeskeden.

6.3 Resultater

Protokollen virker efter hensigten:

- RPC-kald kan foretages til kendte metoder (GETTEMP, SETTEMP), konformt til JSON-RPC-specifikationen.
- For alle andre RPC-kald returneres en fejlbesked, der er konform til JSON-RPC-specifikationen.
 - Fejlbeskeder beskriver fejlen, returnerer en fejlkode, og udløses af:

- 1: Forkert protokolversion, manglende felter eller forkert formatering (besked kan ikke parses).
- 2A: Manglede parameterliste ved SETTEMP.
- 2B: SETTEMP set-point uden for validt temperaturinterval.
- 3: Kald foretaget til ukendt metode.

6.4 Diskussion, forbedringsmuligheder

Benyttelse af JSON-RPC-protokollen giver mere struktureret kommunikation. Især når der skal sendes metode-kald med en tilhørende parameterliste, samt når der returneres fejlbeskeder inklusive fejlobjekter (fejlbeskrivelser, oprindeligt RPC-kald, osv.).

Det bliver altså lettere at opbygge en omfattende to-vejs ”grammatik” for RPC. Prisen for dette er væsentligt mere omfattende kode, og en del jongleren med JSON.

Hvis man blot har behov for at sende en enkelt kommando som ”GETTEMP”, så overstiger ulemper / overhead / dødvægt i at benytte en JSON-struktureret protokol *tydeligt* fordelene.

Hvis man ved, at RPC API’et skal udvides betydeligt i fremtiden, eller man har eksterne brugere, så giver det mening at fastlægge en klar og gennemanalyseret struktur fra begyndelsen³.

Det er en generel observation, at jo mere fleksibilitet og jo flere muligheder og parametre, der skal indbygges i kommunikationsprotokollen, jo flere ting kan ”gå galt” – og derfor bliver validering, fejlhåndtering og fejlrapportering en omfattende (bekostelig) affære. Både i kodelinjer og udviklertid.

7 Samlet konklusion

I denne opgave er systemet fra øvelse 4 ”opgraderet” med en RPC-protokol baseret på JSON-RPC [4]. JSON-håndteringen er implementeret vha. et header-only bibliotek til C++ [3]. Opgaven har således demonstreret, hvordan der relativt nemt implementeres en RPC-protokol op imod en given standard.

JSON er ofte anvendt til at strukturere data og beskeder i kommunikation. Det er desuden en lidt nemmere struktur at arbejde med end det også meget anvendte XML. Det var alt i alt en nyttig øvelse.

Der er analyseret fordele og ulemper ved brugen af en RPC-protokol til et system af denne størrelse versus til større systemer / anderledes behov.

Med en JSON-RPC-protokol implementeret i systemet, er det nu relativt nemt at udvide systemet yderligere til at håndtere en mere omfattende ”grammatik” i API’et.

³ Men, som bekendt: *“Premature optimization is the root of all evil.”* - Sir Tony Hoare

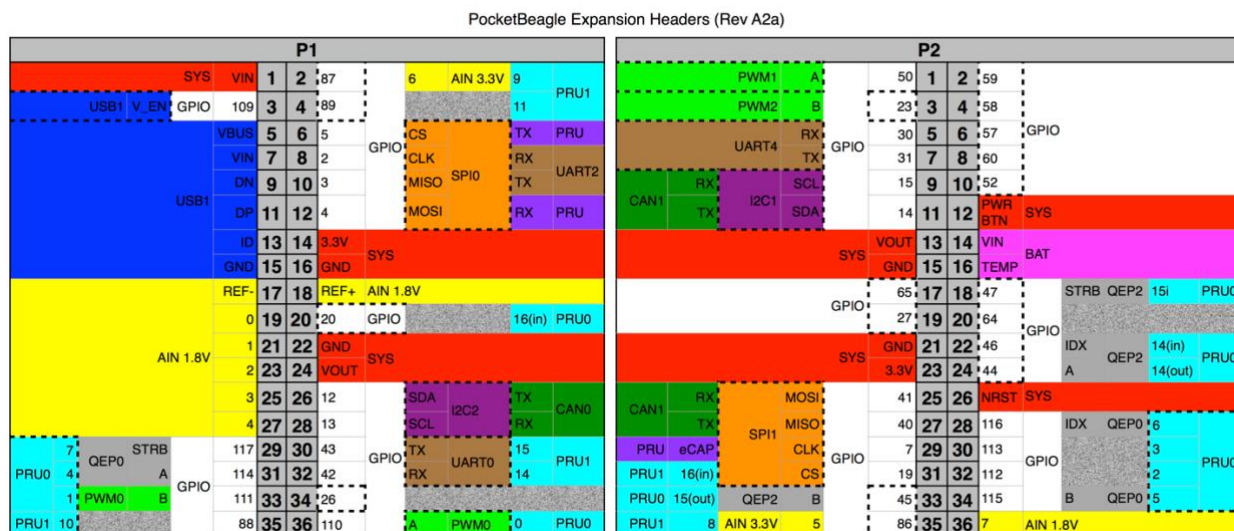
8 Referencer

- [1] MOLLOY, DEREK. *Exploring BeagleBone*. 2. udgave. Wiley, 2019.
- [2] Texas Instruments. LM35 Precision Centigrade Temperature Sensors. Tilgængelig online (sidst set 29/03/2020). URL: <http://www.ti.com/lit/ds/symlink/lm35.pdf>
- [3] LOHMANN, NIELS. JSON for Modern C++. Tilgængelig online (sidst set 11/04/2020). URL: <https://github.com/nlohmann/json>
- [4] JSON-RPC Working Group. JSON-RPC 2.0 Specification, 2013-01-04. Tilgængelig online (sidst set 11/04/2020). URL: <https://www.jsonrpc.org/specification>
- [5] ANTHONY, R. J. Systems Programming. Designing and Developing Distributed Applications. Elsevier, 2016.
- [6] Tencent. RapidJSON. *A fast JSON parser/generator for C++ with both SAX/DOM style API*. Tilgængelig online (sidst set 11/04/2020). URL: <https://github.com/Tencent/rapidjson>
- [7] Tencent. RapidJSON Documentation. Tilgængelig online (sidst set 11/04/2020). URL: <http://rapidjson.org/>

9 Bilag

9.1 Pin-outs på Pocketbeagle

Nedenfor ses pin-outs.



Issues I Rev. A2:

https://github.com/beagleboard/pocketbeagle/wiki/System-Reference-Manual#223_Rev_A2

Power pins:

https://github.com/beagleboard/pocketbeagle/wiki/System-Reference-Manual#54_Power