Simulation einer Datenübertragung mit LEB128

Michael Domanek

Jänner 2021

Contents

1 Aufgabenstellung

Simulation einer Datenübertragung von ganzen Zahlen basierend auf der Kodierung Signed LEB128 wobei Zahlen (zufällig zwischen -100000 und 100000; über die Kommandozeile konfigurierbar) zwischen zwei Threads übertragen werden sollen. Es sollen permanent Zahlen im Sekundentakt übertragen werden, wobei die Übertragung als String stattfinden soll. Es sind promise und future Paare zur Kommunikation zu verwenden.

2 Einleitung

In meinem Projekt geht es darum Dezimalzahlen zwischen 2 Threads zu übertragen. Dafür wird die Dezimalzahl in unsigned/signed LEB128 kodiert dann übertragen und dann wieder zurück konvertiert.
Um die Zahlen zu unsigned LEB128 zu kodieren wurde folgendes Schema[1] verwendet:

1. Zahl binär darstellen
2. 0en bis auf Vielfaches von 7 links auffüllen
3. in 7er Gruppen teilen
4. auf 8 Bits bringen: MSB setzen in jeder Gruppe außer der höchstwertigsten
5. Daten beginnend mit dem niederwertigsten Byte übertragen

Um die Zahlen zu signed LEB128 zu kodieren wurde folgendes Schema[2] verwendet:

1. Zahl binär darstellen (negativ → positiv, 0 Bit hinzu, 2er-Komplement)
2. VZ bis auf Vielfaches von 7 links auffüllen
3. in 7er Gruppen teilen
4. auf 8 Bits bringen: MSB setzen in jeder Gruppe außer der höchstwertigsten
5. Daten beginnend mit dem niederwertigsten Byte übertragen

Für die Übertragung zwischen den Threads wurden promise und future Paare verwendet. Die LEB128 kodierten zahlen werden als Strings übertragen.

[1] Schema aus ihrer pdf 21_encoding S. 6 verwendet
[2] Schema aus ihrer pdf 21_encoding S. 8 verwendet

Im folgenden Code sieht man verkürzt wie das Programm und die Übertragung funktioniert:

```
LEB128 leb128{logger};

while (true) {
    promise<string> promise;
    future<string> future{promise.get_future()};

    thread t1{[&]{
        value = gen(rd);
        binary = leb128.toSignedLeb128(value);
        promise.set_value(binary);
        this_thread::sleep_for(chrono::milliseconds(delay));
    }};

    thread t2{[&]{
        string binary = future.get();
        value = leb128.signedLeb128toDecimal(binary);
        cout << value << endl;
    }};
}

t1.join();
t2.join();
```

3 Parameter des Komandozeileinterface

In diesem Kapitel geht es darum welche Option das Komandozeileinterface hat. Im Bild 1 sieht man eine Übersicht der Optionen.

Standartmäßig werden zufällige Zahlen alle 1000ms zwischen -100000 und 100000 übertragen und die signed LEB128 Kodierung verwendet.

--help
Eine Ausgabe alle Komandozeilenoptionen wie in der Übersicht.

--unsigned
Es wird die unsigned LEB128 Kodierung verwendet.

--show-encoded
Es werden die Kodierten Zahlen in die Kommandozeile ausgegeben.

--start & --end
Mit diesen Optionen wird der Bereich der Zufallszahlen festgelegt. Dieser muss zwischen -100000 und 100000 liegen. Beides müssen ganzahlige Werte sein und die Option **--values** darf nicht verwendet werden.

Option	Typ	Bedingungen	Beschreibung
-h,--help	FLAG		Print this help message and exit
-u,--unsigned	FLAG		Encode with unsigned LEB128
--show-encoded	FLAG		Show encoded values
-s,--start	INT:INT in [-100000 - 100000]	Excludes: --values	Start of the range of random numbers
-e,--end	INT:INT in [-100000 - 100000]	Excludes: --values	End of the range of random numbers
-d,--delay	UINT		Delay between data transfer in ms
-v,--values	INT:INT in [-100000 - 100000]	Excludes: --start --end	Values to transfer in a loop
--json-output-name	TEXT		Name of json output file
--toml-path	TEXT:FILE		Name of toml configuration file (overrides)

Figure 1: Eine Übersicht der Optionen der CLI

--values
Damit können Werte eingeben werden, die dann wiederholt übertragen werden. Für diese Werte gelten die selben Bedingungen wir für start und end.

--delay
Damit wird der Dauer zwischen den Übertragungen in Millisekuden angegeben.

--json-output-name
Mit dieser Option legt man fest das ein JSON Datei erstellt wird mit dem angegeben Namen. In diesem JSON stehen die Konigurationsparameter und die übergeben Werte von thread 1 und die erhaltenen Werte von thread 2.

--toml-path
Damit wird ein Pfad des Konfigurationsfiles festgelegt, das in TOML Syntax muss sein. Mit dieser Datei kann man alle oberhalb genannten Option festlegen bzw. man überschreibt sie wenn sie per Kommandozeile eingestellt wurden.

4 Implementierung

Die Implementierung des Hauptprogramms wurde großteils schon beschrieben. Zuerst werden die Kommandozeilenoptionen eingelesen und verarbeitet. Wenn irgendeine Bedingung nicht erfüllt wird, wird ein ValidationError geworfen und geloggt. In dem Programm werden entweder Zufallszahlen oder die Werte von **--values** verwendet und dauerthaft übertragen.

Für die Übertragung wird die Klasse LEB128 verwendet. Diese Klasse enthält die 4 public Methoden:

```
string toSignedLeb128(const int &number);
string toUnsignedLeb128(const int &number);
int signedLeb128toDecimal(string value);
int unsignedLeb128toDecimal(string value);
```

Mit diesen Methoden kann man entweder Dezimalzahlen in signed oder unsigned LEB128 Kodierung kovertieren oder umgekehrt.

```
1  string toSignedLeb128(const int &number) {
2      if (!number) {
3          return "00000000";
4      }
5
6      string binary{bitset<17>(abs(number)).to_string()};
7      binary.erase(0, binary.find_first_not_of('0'));
8      binary = "0" + binary;
9
10     if (number < 0) {
11         binary = getTwoscomplement(binary);
12     }
13
14     fillWithSign(binary, number >= 0 ? '0' : '1');
15
16     return translatePosition(binary);
17 }
18
19 string translatePosition(string binary) {
20     string leb128binary{"0" + binary.substr(0, 7)};
21     for (size_t i = 7; i < binary.length(); i += 7) {
22         leb128binary = "1" + binary.substr(i, 7) + leb128binary;
23     }
24     return leb128binary;
25 }
```

Diese Methode ***toSignedLeb128*** funktioniert folgendermaßen:

Zuerst wird überprüft ob die Zahl 0 ist, da sich 0 nicht verändert kann man ”00000000” zurückgeben. Danach wird die absolute Zahl in binär umgewandelt (Zeile 6). Da die Binärzahl 17 Stellen hat werden in Zeile 7 und 8 alle 0 vor der eigentlich Zahl bis auf eine entfernt. Danach wird bei negativen Zahlen das Zweierkomplement gebildet. Dann wird an die Binärzahl auf ein Vielfaches von 7 Bit von links aufgefüllt (0 bei positiver Zahl, 1 bei negativer Zahl). In *translatePosition* werden die Zahlen von 7 in 8 Bit umgewandelt und beginnend mit dem niederwertigsten Byte übertragen.

Die Methoden **toUnsignedLeb128** ist genau gleich aufgebaut, da es aber keine negativen Zahlen gibt wird kein Zweierkomplement gebildet und es wird bei *fillWithSign* immer 0 als Vorzeichen übergeben.

```
1  int signedLeb128toDecimal(string value) {
2      string binary{""};
3      bool isNegative{false};
4
5      while (true) {
6          bool isLastByte{value.front() == '0'};
7
8          binary = value.substr(1, 7) + binary;
9          value.erase(0, 8);
10
11         if (isLastByte) {
12             break;
13         }
14     }
15
16     if (binary.front() == '1') {
17         binary = getTwoscomplement(binary);
18         isNegative = true;
19     }
20
21     binary.erase(0, binary.find_first_not_of('0'));
22     int decimal = (int)bitset<17>(binary).to_ulong();
23
24     return isNegative ? decimal : -decimal;
25 }
```

Diese Methode ***signedLeb128toDecimal*** funktioniert folgendermaßen:

In Zeile 5 – 14 werden die Bytes so lange übertragen bis das erste Bit 0 ist. 0 als MSB bedeuted, dass das letzte Byte übertragen wurde. In der Schleife wird der Wert gleich in die richtige Reihenfolge gebracht und die 8 Bits werden zu 7 Bits umgewandelt.

In Zeile 16 – 19 wird überprüft ob die Binärzahl negative ist und wenn wird das Zweierkomplement gebildet. Danach werden die links aufgefüllten 0 entfernt und die Binärzahl in eine Dezimalzahl umgewandelt. Danach wird die positive oder negative Dezimalzahl zurückgegeben.

Die Durchschnittszeit von der Umwandlung dauert ca. 1 – 2 Millisekunden. Messungsstart vor der Methode *toSignedLeb128* und Ende nach der Methode *signedLeb128toDecimal*

Um das Programm übersichtlicher zu machen und die Erklärung einfachen werden in den Codebeispielen loggen mit spdlog entfernt und das einlesen und ausgeben von JSON & TOML ausgelasen bzw. entfernt.

5 Ausgabe-Beispiele

5.1 Konsolenausgabe

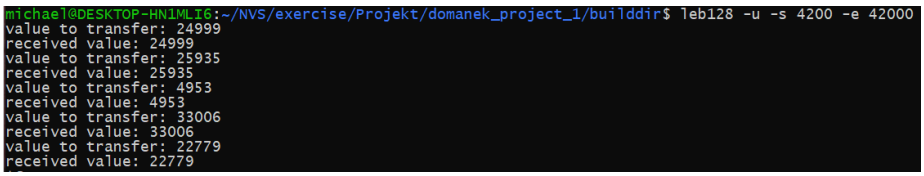

Figure 2: Beispiel eines Aufrufs auf der Konsole

Figure 3: Beispiel eines Aufrufs auf der Konsole

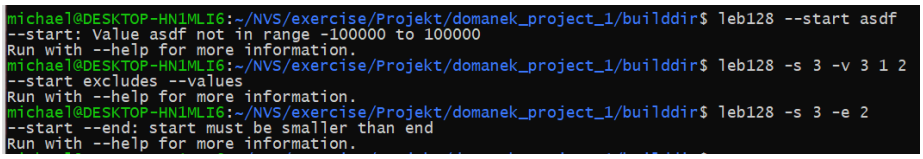

Figure 4: Beispiel von Aufrufen mit Fehler auf der Konsole

5.2 Beispiel mit TOML Konfigurationsfile und JSON Outputfile

Befehl: **leb128 --toml-path ../examples/example-configuration2.toml**

Wenn man diesen Befehl eingibt und nach 2 Übertragung stoppt erhält man folgendes JSON.

```
json-output-name = "toml.json"
values = [
  1,
  42,
  7007
]
```

Listing 1: TOML-Konfigurationsfile

```
{
    "data": [
        {
            "received": 1,
            "transfered": 1
        },
        {
            "received": 42,
            "transfered": 42
        }
    ],
    "delay": 1000,
    "show-encoded": false,
    "unsigned": false,
    "values": [
        1,
        42,
        7007
    ]
}
```

Listing 2: JSON Outputfile

5.3 Auschnitt aus dem Logfile

Die Zeit wurde nach dem nach dem erste Befehlt ausgeschnitten, damit die Zeile nicht zu lang wird

```
[error] [thread 680] --start --end: start must be smaller than end
[2021 01 10 23:07:17,047] [info] [thread 633] ==========
[info] [thread 633] stared new simulation of data transfer with LEB128
[debug] [thread 634] Convert to signed LEB128
[debug] [thread 634] Number: 126
[debug] [thread 634] Binary number: 01111110
[debug] [thread 634] Binary with sign: 00000001111110
[debug] [thread 634] Binary with translated positions: 1111111000000000
[debug] [thread 635] Convert signed LEB128 to decimal
[debug] [thread 635] LEB128 encoded binary: 1111111000000000
[debug] [thread 635] Binary with translated positions: 00000001111110
[debug] [thread 635] Binary number: 1111110
[debug] [thread 635] Number: 126
```

Listing 3: Auschnit aus dem Logfile