Beispiel `11_daytime`

Dr. Günter Kolousek

10. Januar 2019

1 Allgemeines

- **Backup** nicht vergessen!
- Hier nochmals zwei Erinnerungen:
  - Regelmäßig Commits erzeugen!
  - Backup deines Repos nicht vergessen (am Besten nach jedem Beispiel)!!!
- In diesem Sinne ist jetzt ein neues Verzeichnis `11_daytime` anzulegen.

2 Aufgabenstellung

Dieses Beispiel behandelt die einfache Stream-orientierte Kommunikation über Sockets an Hand des daytime-Protokolls. Ein Client nimmt Kontakt zu einem Server auf, empfängt die Serverzeit über TCP und gibt diese aus. Die Kommunikation findet zeichenbasiert statt, wobei das Format der Zeit nicht spezifiziert ist.
Los geht's!

3 Anleitung

1. "Installiere" zuerst die asio Bibliothek an einem geeigneten Platz deiner Wahl indem du das Archiv an einen Ort deiner Wahl entpackst, aber nicht im Verzeichnisbaum deines Repositories!!! D.h. irgendwo in deinem Dateissystem, aber so, dass es letztendlich **nicht** abgegeben wird.

   Die Bibliothek wird von mir in Form eines Archives am ifssh zur Verfügung gestellt.

   Das von mir bereitgestellte `template.tar.gz` enhtält sowohl eine Datei `meson.build` als auch eine Datei `meson_options.txt`. Ändere beide Dateien entsprechend ab! Bzgl. `meson_options.txt` bedeutet das, dass du in das `value` - Feld das `include` - Verzeichnis von `asio`.

Andere ”header-only” Bibliotheken werden bei Bedarf analog ”installiert”!

Beachte bitte auch **wie** ich `asio` in `main.cpp` eingebunden habe. Der verwendete Ansatz verhindert Unmengen von Warnungen und damit auch Übersetzungsfehler (siehe `meson.build`)! Alle nicht benötigten `#include` - Präprozessoranweisungen sind zu kommentieren.

Übungszweck

- externe header-only Bibliothek ”installieren” und Buildsystem entsprechend konfigurieren

2. Entwickle einen Daytime-Client auf TCP Basis, der sich zu einem lokalen Daytime-Server (siehe `DaytimeServer.class`) mit einem konfigurierbaren (mittels Kommandozeilenparameter) Port verbindet, die aktuelle Zeit erfragt und danach diese auf der Konsole ausgibt. Implementiert wird also das daytime Protokoll, das einfach nur aus einem leeren Request und einem Response mit Zeichendaten besteht.

   Dieses Programm `daytime_client` ist in einem **eigenem** Verzeichnis `daytime_client` im Verzeichnis `src` zu entwickeln. Als Programmname für den eigentlichen Sourcecode bietet sich `main.cpp` an.

   Der zur Verfügung gestellte Daytime-Server kann folendermaßen gestartet werden:

   ```
   $ java DaytimeServer 1113
   just before accept
   ```

   erfolgen. Die Ausgabe ”just before accept” gibt lediglich an, dass der Server auf eine Verbindungsanfrage wartet und dient dazu die Funktionsbereitschaft des Servers zu zeigen.

   Alternativ kann unter Unix auch der Server einfach als Hintergrundprozess gestartet werden:

   ```
   $ java DaytimeServer 1113&
   just before accept
   ```

   Der Client soll einfach einen Stream zum Server aufbauen, eine Zeile davon lesen, das Gelesene als eigene Zeile ausgeben und den Stream schließen. Fertig. Eine Fehlerbehandlung ist bis jetzt noch nicht notwendig.

   Nachdem der Server läuft, kann der Client gestartet werden und die Ausgabe des Clients sollte jetzt folgendermaßen aussehen:

   Beispiel:

   ```
   $ daytime_client
   Tue Nov 25 12:07:57 CET 2003
   ```

   Für die Anzeige muss lediglich der vom Server übertragene String auf der Ausgabe ausgegeben werden.

   Der Server wird danach wieder ”just before accept” ausgeben.

Übungszweck

- Socketverbindung zu lokalem Host aufbauen. Übertragung von Zeichendaten (Client-Seite) mittels stream-basierter Kommunikation.

3. Beende den Server und starte den Client neu. Wenn du absolut keine Fehlerbehandlung eingebaut hast, wird einfach eine Leerzeile ausgegeben werden. Warum?

   Erweitere den Client, dass jetzt sowohl das korrekte Aufbauen der Verbindung überprüft wird. Kann keine Verbindung aufgebaut werden, dann soll eine Fehlermeldung auf `stderr` ausgegeben werden.

   Das sollte dann in etwa so aussehen:

   ```
   Could not connect to server!
   ```

4. Ok, das ist gut, aber unter Umständen interessieren uns auch noch weitere Informationen, die in weiterer Folge *geloggt* werden sollen. In komplexeren Programmen und speziell in Netzwerkprogrammen muss immer wieder geloggt werden. Dazu ist es sinnvoll sich eine entsprechende Unterstützung zu organisieren. Wir werden dazu die Header-only Bibliothek `spdlog` verwenden.

   Auch diese Bibliothek wird von mir in Form eines Archives am ifssh zur Verfügung gestellt. Die Verwendung ist analog zu `asio` (im speziellen ist auch diese Bibliothek **nicht** im Abgabeordner abzulegen!)

   `spdlog.cpp` aus dem Template enthält dafür auch Code, der zeigt wie man damit auf primitiver Ebene umgehen kann.

   Für die gesamte Dokumentation siehe spdlog!

5. Weiter mit einem eigenen Daytime-Server, der für einen Daytime-Client die lokale Zeit zur Verfügung stellt.

   Füge deinem Projekt ein **weiteres** Unterverzeichnis `daytime_server` im Verzeichnis `src` hinzu und passe `meson.build` entsprechend an. Dann werden zwei ausführbare Programme gebaut, vorausgesetzt in `daytime_server` befindet sich auch eine Sourcecode-Datei wie z.B. `main.cpp`.

   Der Server soll vorerst nur einmal eine Verbindung akzeptieren (Port 1113) und danach die aktuelle Zeit ermitteln und als Zeile an den Client zurücksenden. Fertig.

   ```
   $ daytime_client
   2016-01-13 23:27:07
   ```

   Für die Übertragung der aktuellen Zeit kann wiederum die von mir bereitgestellte Headerdatei `timeutils.h` verwendet werden oder – noch einfacher – die mittels `system_clock::now()` ermittelte Zeit in den Ausgabestrom (vom Typ `tcp::iostream`) geschoben (also mit `<<`) werden.

   Der Server wird sich danach beendet haben.

Übungszweck Server-Socket anlegen und Client-Requests annehmen (blocking, single-threaded). Übertragung von Zeichendaten (Server-Seite) mittels stream-basierter Kommunikation.

6. Erweitere den Server, sodass sich dieser nicht mehr beendet, sondern nach erfolgter Bearbeitung der Anfrage wieder bereit für eine erneute Verbindung ist.

   Übungszweck single-threaded Server für beliebig viele Requests.

7. Erweiterung um Verarbeitung von Kommandozeilenparametern. Dazu ist entweder die header-only Bibliothek `clipp` oder die header-only Bibliothek `CLI11` zu verwenden! Wie "gewohnt" stelle ich auch hierfür eine Version zur Verfügung, die an einem entsprechenden Ort zu entpacken ist.

   - Der Port an dem der Server lauscht *muss* als Kommandozeilenparameter übergeben werden. Eine Hilfe soll *optional* angefordert werden können.

     Die Ausgabe soll folgendermaßen aussehen:

     ```
     $ daytime_server
     SYNOPSIS
             daytime_server -p <port> [-h]

     OPTIONS
             <port>      server port
             -h, --help  help
     ```

   - Weiters soll in diesem Zusammenhang auch der Client angepasst werden, dass dieser als Kommandozeilenparameter *optional* den Port akzeptiert. Wird kein Port angegeben, dann soll 1113 zum Einsatz kommen. Eine Hilfe soll optional angefordert werden können.

     ```
     $ daytime_client -h
     SYNOPSIS
             daytime_client [-p <port>] [-h]

     OPTIONS
             <port>      port to connect to
             -h, --help  help
     ```

   Übungszweck Verarbeitung von Kommandozeilenparameter für Server- und Client-Programme.

8. Erweitere jetzt sowohl den Client als auch den Server jetzt dass dieser jetzt auch mit Fehlern umgehen kann. Setze dazu die Möglichkeiten von `spdlog` ein!

Fehler, die behandelt werden sollen:

- Client kann sich nicht verbinden
- Client kann keine Daten einlesen
- Server kann Verbindung nicht fehlerfrei annehmen
- Server kann nicht fehlerfrei senden

Prinzipiell gibt es die Möglichkeit mittels `try/catch` oder mit zusätzlichen Parameter vom Typ `error_code`. Beide Möglichkeiten sind gleichwertig, aber die Lösung mittels `try/catch` für diese Situation besser geeignet, nicht wahr?

Übungszweck Einfachste Fehlerbehandlung kennenlernen.