ulm university universität
uulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

28. April 2017
Blatt 2

Parallele Programmierung mit C++ (SS 2017)

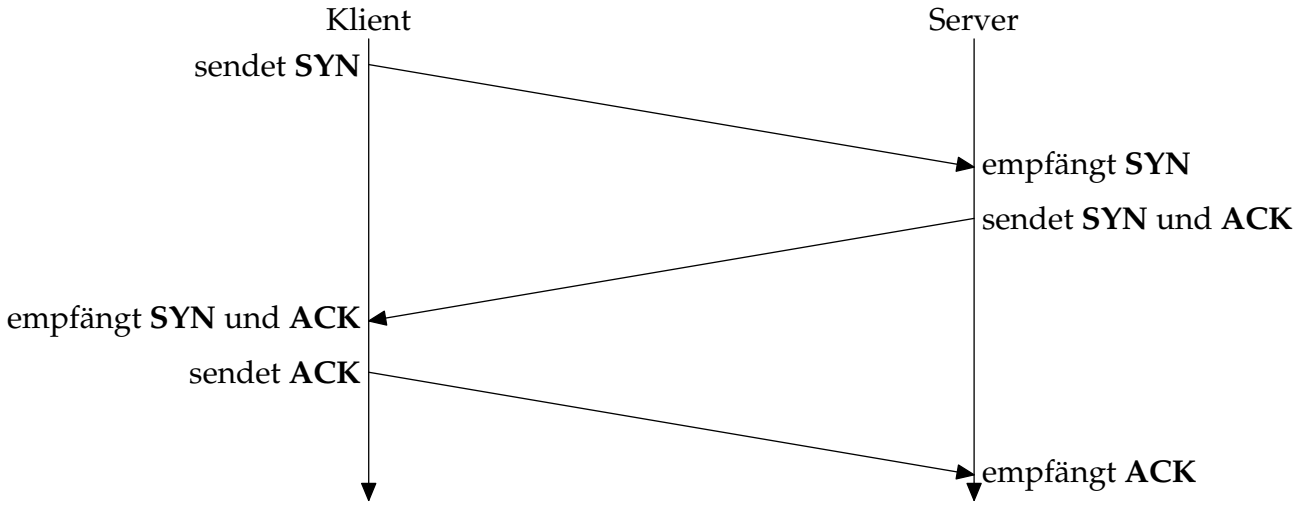
Abgabe bis zum 5. Mai 2017, 14:00 Uhr

Lernziele:

- Modellierung paralleler Prozesse mit CSP
- Dynamischer Polymorphismus in C++

Aufgabe 3: Three-Way Handshake

Modellieren Sie den Beginn einer TCP-Verbindung mit Hilfe von CSP:



Der Klient beginnt mit dem Systemaufruf *connect*, sendet dann ein SYN-Paket, wartet dann auf das SYN-und-ACK-Paket, sendet dann ein ACK, worauf es die Verbindung als etabliert betrachtet. Der Server beginnt mit dem Systemaufruf *listen*, wartet dann auf den Eingang des SYN-Pakets, schickt dann das SYN-und-ACK-Paket, wartet dann auf das ACK und betrachtet dann die Verbindung als etabliert. Modellieren Sie dabei auch das dazwischenliegende Netzwerk, das jeweils ein Paket transportiert. Achten Sie dabei darauf, dass die Schnittmenge der Alphabete des Klienten und des Servers leer ist. Sobald der Klient bzw.

der Server die Verbindung als etabliert betrachten, können sie jeweils erfolgreich terminieren (in CSP mit Hilfe von *SKIP*).

Eine Ergänzungsmöglichkeit wäre der Fall, dass der Server noch nicht *listen* aufgerufen hat, aber schon ein SYN-Paket erhält. In diesem Falle würde er ein RST-Paket schicken (*connection refused*).

Ihre Lösung können Sie einreichen mit

```
submit pp 3 handshake.csp
```

Aufgabe 4: Unzuverlässiges Netzwerk

Modellieren Sie die Beziehung eines Klienten mit einem Dienst und einem unzuverlässigem Netzwerk mit einer Bandbreite von drei Paketen, die parallel behandelt werden können. Der Klient schickt bis zu drei Mal ein Anfragepaket über das Netzwerk (*send_request*). Wenn keine Antwort innerhalb einer Zeitschranke eintrifft (das wird durch die Ereignisklasse *timeout* modelliert), dann wird ein Anfragepaket erneut verschickt. Wenn der Dienst über das Netzwerk ein Anfragepaket erhält (*receive_response*), dann bearbeitet er die Anfrage und sendet eine Antwort zurück (*send_response*), die, falls das Netzwerk nicht ausfällt, vom Klienten empfangen wird (*receive_response*).

Dieses Modell findet Anwendung bei DNS (*domain name service*), das über UDP-Pakete (UDP = *User Datagram Protocol*) abgewickelt ist, bei denen ein Verlust oder auch eine mehrfache Ankunft denkbar sind. Jeder DNS-Klient muss entsprechend Anfragen wiederholen, wenn innerhalb einer Frist keine Antwort kommt.

So könnte ein beispielhafter Verlauf aussehen, bei dem das erste Anfragepaket beim Senden verloren geht und beim zweiten Versuch das Antwortpaket des Dienstes vom Netzwerk verschluckt wird:

```
thales$ trace -ap dnsreq.csp
Acceptable: {send_request}
send_request
Acceptable: {timeout}
timeout
Acceptable: {send_request}
send_request
Acceptable: {receive_request, timeout}
receive_request
Acceptable: {process_request, timeout}
process_request
Acceptable: {send_response, timeout}
send_response
Acceptable: {timeout}
timeout
Acceptable: {send_request}
send_request
Acceptable: {receive_request, timeout}
receive_request
```

```
Acceptable: {process_request, timeout}
process_request
Acceptable: {send_response, timeout}
send_response
Acceptable: {receive_response, timeout}
receive_response
Acceptable: {process_response}
process_response
OK
thales$
```

Sollten alle drei Versuche schiefgegen, sollte der Klient mit dem *failure*-Ereignis aufhören:

```
thales$ $S/trace -ap dnsreq.csp
Acceptable: {send_request}
send_request
Acceptable: {timeout}
timeout
Acceptable: {send_request}
send_request
Acceptable: {timeout}
timeout
Acceptable: {send_request}
send_request
Acceptable: {timeout}
timeout
Acceptable: {failure}
failure
OK
thales$
```

Hinweise: Es empfiehlt sich, zuerst mit einem funktionierenden Netzwerk zu beginnen. Dann sollten Sie die Bandbreite des Netzwerks erhöhen, so dass mehrere Pakete gleichzeitig über das Netzwerk transportiert werden können. Sie können das z.B. mit mehreren konkurrierenden Servern und Klienten testen.

Im nächsten Schritt kehren Sie wieder zur ursprünglichen Variante mit einem Klienten und einem Dienst zurück, sehen aber beim Klienten das *timeout*-Ereignis und die bis zu drei Versuche. Den Klienten müssen Sie dann entsprechend erweitern. Das geht inkrementell mit *Client1*, der einen Versuch unternimmt und nach dem *timeout* mit *failure* endet. Darauf aufbauend kann dann *Client2* geschrieben werden, der es einmal selbst versucht und bei einem *timeout* das weitere dem *Client1* überlässt usw. Beachten Sie hierbei, dass auch Antworten auf frühere Anfragen nach einem Timeout kommen können. Wenn Sie alles am Ende mit *SKIP* schön beenden wollen, müssen Sie darauf achten, dass da mindestens das Alphabet aus der Schnittmenge der Alphabeten zwischen dem Netzwerk und dem Klienten explizit angegeben wird.

Dann sollten Sie einen verlässlichen Netzwerktransport mit einem Pakete verlierenden Netzwerktransport nicht-deterministisch mit dem $\sqcap$-Operator verknüpfen.

Ihre Lösung können Sie einreichen mit

```
submit pp 4 udp.csp
```

Aufgabe 5: Abstrakte Spielereien

In der im ersten Übungsblatt vorgestellten und in der Aufgabe 2 umgesetzten Implementierung eines Spiels hängt der Spielablauf von der Art der Spieler ab. Bei menschlichen Spielern muss ein Zug interaktiv eingelesen und überprüft werden, bei einem rechnerseits implementierten Spieler muss anhand irgendwelcher Kriterien ein Zug gefunden werden. Wenn ein Computer gegen einen menschlichen Spieler zu spielen hat, führt dies zu Konstruktionen wie die folgende:

```
while (!game.finished()) {
    // print current state
    std::cout << "Heaps:";
    for (unsigned int i = 0; i < number_of_heaps; ++i) {
        std::cout << "␣" << game.get_heap_size(i);
    }
    std::cout << std::endl;
    NimMove move;
    if (game.get_next_player() == NimGame::PLAYER1) {
        // human player
        move = fetch_move_from_human_player(game);
    } else {
        // computer
        move = fetch_move_from_computer(game);
        std::cout << "Taking␣" << move.get_count() << "␣from␣heap␣"
            << move.get_heap() << std::endl;
    }
    game.execute_move(move);
}
```

Dies ist natürlich unbefriedigend, da die beiden Spieler-Implementierungen so ein fester Bestandteil des Hauptprogramms bilden. Es erscheint daher erstrebenswert,

- eine abstrakte Klasse *NimPlayer* zu entwickeln,
- mehrere Erweiterungen dieser Klasse zu implementieren wie beispielsweise *HumanNimPlayer*, *RandomNimPlayer* oder *OptimalNimPlayer*,
- und diese beim Start des Programms auswählbar zu machen, z.B. indem die gewünschten Implementierungen dynamisch nachgeladen werden.

Im Rahmen dieser Aufgabe ist dies umzusetzen, wobei Sie mindestens zwei Implementierungen entwickeln sollten, wozu auch *HumanNimPlayer* gehören muss.

Hinweise: Denken Sie darüber nach, wie das Ende der Eingabe bei *HumanPlayer* zu behandeln ist. Hier ist es keine Lösung, die Ausführung zu beenden. Stattdessen ist es geschickter,

einen Zug einzuführen, bei dem ein Spieler „aufgibt“. Bei einer Aufgabe hat ein Spieler sofort verloren.

Wenn Sie Ihre Lösung einreichen, sollten Sie diese bitte zunächst mit Hilfe von *tar* verpacken und komprimieren:

```
thales$ tar cvfz Nim.tar.gz *.?pp [mM]akefile
```

Sie können dann Ihre Lösung wieder mit *submit* einreichen:

```
thales$ submit cpp 5 Nim.tar.gz
```

Viel Erfolg!