Netzwerkprogrammierung mit C++

Bernhard Trummer
bernhard.trummer@gmx.at

14. Mai 2005

Übersicht

System-Calls: Wie arbeitet man mit Sockets?

Applikationsdesign: Was ist alles zu berücksichtigen, wenn man einen Server schreiben will?

Beispielprogramme: Vorstellung einiger kleiner Beispielprogramme.

Q&A, Diskussion: Ev. auch nach dem Workshop.

System-Calls
Erzeugen von Sockets

- `int socket(int domain, int type, int protocol);`
- `int close(int fd);`
- `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);`
- `int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);`
- `int listen(int s, int backlog);`
- `int accept(int s, struct sockaddr *addr, socklen_t *addrlen);`

System-Calls
Senden und Empfangen von Daten

- `int send(int s, const void *msg, size_t len, int flags);`
- `int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);`
- `int recv(int s, void *buf, size_t len, int flags);`
- `int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);`

System-Calls
Socket-Optionen

- `int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);`
- `int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);`
- `int ioctl(int d, int request, ...);`
- `int fcntl(int fd, int cmd, ...);`

System-Calls
Socket Multiplexing

- `int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
- `int poll(struct pollfd *ufds, unsigned int nfds, int timeout);`

Was hat das ganze mit C++ zu tun?

- In C++ können Wrapper für die System-Calls der libc geschrieben werden.
- Plattformunabhängigkeit bzw. Kapselung von betriebssystemspezifischen Eigenheiten.
- Einfachere Applikationsentwicklung durch Verwendung von C++-Features (z.B. Exception-Handling).
- Es gibt viele Möglichkeiten, wie man die C++-Wrapper erstellt (Aggregation vs. Derivation vs. Templates).

Was hat das ganze mit C++ zu tun?

- In C++ können Wrapper für die System-Calls der libc geschrieben werden.
- Plattformunabhängigkeit bzw. Kapselung von betriebssystemspezifischen Eigenheiten.
- Einfachere Applikationsentwicklung durch Verwendung von C++-Features (z.B. Exception-Handling).
- Es gibt viele Möglichkeiten, wie man die C++-Wrapper erstellt (Aggregation vs. Derivation vs. Templates).

Was hat das ganze mit C++ zu tun?

- In C++ können Wrapper für die System-Calls der libc geschrieben werden.
- Plattformunabhängigkeit bzw. Kapselung von betriebssystemspezifischen Eigenheiten.
- Einfachere Applikationsentwicklung durch Verwendung von C++-Features (z.B. Exception-Handling).
- Es gibt viele Möglichkeiten, wie man die C++-Wrapper erstellt (Aggregation vs. Derivation vs. Templates).

Was hat das ganze mit C++ zu tun?

- In C++ können Wrapper für die System-Calls der libc geschrieben werden.
- Plattformunabhängigkeit bzw. Kapselung von betriebssystemspezifischen Eigenheiten.
- Einfachere Applikationsentwicklung durch Verwendung von C++-Features (z.B. Exception-Handling).
- Es gibt viele Möglichkeiten, wie man die C++-Wrapper erstellt (Aggregation vs. Derivation vs. Templates).

Verbreitete Libraries mit Netzwerk-Support

- Qt
- SDL_net (eigentlich eine C-Library)
- ClanLib
- CommonC++
- ACE

Applikationsdesign
Allgemeine Richtlinien

- Jeder System-Call muß auf Fehler überprüft werden (Returnwert und errno).
- Netzwerkfehler können immer passieren (z.B. Timeouts, Verbindungsabbrüche).
- Empfangenen Daten darf nie vertraut werden (Buffer Overflows, Denial of Service).

Applikationsdesign
Allgemeine Richtlinien

- Jeder System-Call muß auf Fehler überprüft werden (Returnwert und errno).
- Netzwerkfehler können immer passieren (z.B. Timeouts, Verbindungsabbrüche).
- Empfangenen Daten darf nie vertraut werden (Buffer Overflows, Denial of Service).

Applikationsdesign
Allgemeine Richtlinien

- Jeder System-Call muß auf Fehler überprüft werden (Returnwert und errno).
- Netzwerkfehler können immer passieren (z.B. Timeouts, Verbindungsabbrüche).
- Empfangenen Daten darf nie vertraut werden (Buffer Overflows, Denial of Service).

Bearbeiten von Requests

- Direkte Bearbeitung
- Forken von Child-Prozessen
- Multithreading
- Design-Patterns

Bearbeiten von Requests
Direkte Bearbeitung

- Für kleinere Server durchaus ausreichend (KISS-Prinzip).
- Leicht zu debuggen.
- Wenn der Prozeß abstürzt, ist der ganze Server weg.

Bearbeiten von Requests
Direkte Bearbeitung

- Für kleinere Server durchaus ausreichend (KISS-Prinzip).
- Leicht zu debuggen.
- Wenn der Prozeß abstürzt, ist der ganze Server weg.

Bearbeiten von Requests
Direkte Bearbeitung

- Für kleinere Server durchaus ausreichend (KISS-Prinzip).
- Leicht zu debuggen.
- Wenn der Prozeß abstürzt, ist der ganze Server weg.

Bearbeiten von Requests
Forken von Child-Prozessen

- Ein Child-Prozeß pro Request bzw. Anlegen eines Child-Pools.
- Die Childs laufen unabhängig voneinander.
- Wenn ein Child abstürzt, sind der Hauptprozeß und die anderen Childs davon nicht betroffen.
- Overhead durch je einen `fork()` Call pro Request.
- Nur für zustandslose Requests.

Bearbeiten von Requests
Forken von Child-Prozessen

- Ein Child-Prozeß pro Request bzw. Anlegen eines Child-Pools.
- Die Childs laufen unabhängig voneinander.
- Wenn ein Child abstürzt, sind der Hauptprozeß und die anderen Childs davon nicht betroffen.
- Overhead durch je einen `fork()` Call pro Request.
- Nur für zustandslose Requests.

Bearbeiten von Requests
Forken von Child-Prozessen

- Ein Child-Prozeß pro Request bzw. Anlegen eines Child-Pools.
- Die Childs laufen unabhängig voneinander.
- Wenn ein Child abstürzt, sind der Hauptprozeß und die anderen Childs davon nicht betroffen.
- Overhead durch je einen `fork()` Call pro Request.
- Nur für zustandslose Requests.

Bearbeiten von Requests
Forken von Child-Prozessen

- Ein Child-Prozeß pro Request bzw. Anlegen eines Child-Pools.
- Die Childs laufen unabhängig voneinander.
- Wenn ein Child abstürzt, sind der Hauptprozeß und die anderen Childs davon nicht betroffen.
- Overhead durch je einen `fork()` Call pro Request.
- Nur für zustandslose Requests.

Bearbeiten von Requests
Forken von Child-Prozessen

- Ein Child-Prozeß pro Request bzw. Anlegen eines Child-Pools.
- Die Childs laufen unabhängig voneinander.
- Wenn ein Child abstürzt, sind der Hauptprozeß und die anderen Childs davon nicht betroffen.
- Overhead durch je einen `fork()` Call pro Request.
- Nur für zustandslose Requests.

Bearbeiten von Requests
Multithreading

- Ein Thread pro Request bzw. Anlegen eines Thread-Pools.
- Threads können i.A. schneller erzeugt werden als Child-Prozesse.
- Alle Threads teilen sich den selben Heap. D.h. es können Zustandsinformationen im RAM gehalten werden.
- Die Synchronisation dieser Zustandsinformationen kann sich als sehr schwierig herausstellen (Race-Conditions, Löschen der Zustandsinformationen).
- Wenn ein Thread abstürzt, kann der ganze Prozeß als abgestürzt betrachtet werden.

Bearbeiten von Requests
Multithreading

- Ein Thread pro Request bzw. Anlegen eines Thread-Pools.
- Threads können i.A. schneller erzeugt werden als Child-Prozesse.
- Alle Threads teilen sich den selben Heap. D.h. es können Zustandsinformationen im RAM gehalten werden.
- Die Synchronisation dieser Zustandsinformationen kann sich als sehr schwierig herausstellen (Race-Conditions, Löschen der Zustandsinformationen).
- Wenn ein Thread abstürzt, kann der ganze Prozeß als abgestürzt betrachtet werden.

Bearbeiten von Requests
Multithreading

- Ein Thread pro Request bzw. Anlegen eines Thread-Pools.
- Threads können i.A. schneller erzeugt werden als Child-Prozesse.
- Alle Threads teilen sich den selben Heap. D.h. es können Zustandsinformationen im RAM gehalten werden.
- Die Synchronisation dieser Zustandsinformationen kann sich als sehr schwierig herausstellen (Race-Conditions, Löschen der Zustandsinformationen).
- Wenn ein Thread abstürzt, kann der ganze Prozeß als abgestürzt betrachtet werden.

Bearbeiten von Requests
Multithreading

- Ein Thread pro Request bzw. Anlegen eines Thread-Pools.
- Threads können i.A. schneller erzeugt werden als Child-Prozesse.
- Alle Threads teilen sich den selben Heap. D.h. es können Zustandsinformationen im RAM gehalten werden.
- Die Synchronisation dieser Zustandsinformationen kann sich als sehr schwierig herausstellen (Race-Conditions, Löschen der Zustandsinformationen).
- Wenn ein Thread abstürzt, kann der ganze Prozeß als abgestürzt betrachtet werden.

Bearbeiten von Requests
Multithreading

- Ein Thread pro Request bzw. Anlegen eines Thread-Pools.
- Threads können i.A. schneller erzeugt werden als Child-Prozesse.
- Alle Threads teilen sich den selben Heap. D.h. es können Zustandsinformationen im RAM gehalten werden.
- Die Synchronisation dieser Zustandsinformationen kann sich als sehr schwierig herausstellen (Race-Conditions, Löschen der Zustandsinformationen).
- Wenn ein Thread abstürzt, kann der ganze Prozeß als abgestürzt betrachtet werden.

Zugriff auf gemeinsame Daten
(Zustandsinformationen)

- Multithreading.
- Ein Shared Memory Segment, in dem alle gemeinsamen Daten liegen (siehe SIP Express Router).
- Ein dezidierter (Child-)Prozeß, der über ein lokales Protokoll angesprochen wird.
- Eine Datenbank.

Zugriff auf gemeinsame Daten
(Zustandsinformationen)

- Multithreading.
- Ein Shared Memory Segment, in dem alle gemeinsamen Daten liegen (siehe SIP Express Router).
- Ein dezidierter (Child-)Prozeß, der über ein lokales Protokoll angesprochen wird.
- Eine Datenbank.

Zugriff auf gemeinsame Daten
(Zustandsinformationen)

- Multithreading.
- Ein Shared Memory Segment, in dem alle gemeinsamen Daten liegen (siehe SIP Express Router).
- Ein dezidierter (Child-)Prozeß, der über ein lokales Protokoll angesprochen wird.
- Eine Datenbank.

Zugriff auf gemeinsame Daten
(Zustandsinformationen)

- Multithreading.
- Ein Shared Memory Segment, in dem alle gemeinsamen Daten liegen (siehe SIP Express Router).
- Ein dezidierter (Child-)Prozeß, der über ein lokales Protokoll angesprochen wird.
- Eine Datenbank.

Design-Patterns

- Leader/Follower
- Active Object
- Reactor
- Proactor (fully asynchronous)
- Half-sync/Half-async
- `http://www.cs.wustl.edu/~schmidt/patterns-ace.html`

Entscheidungskriterien
für die Wahl des „richtigen“ Designs

- Können Requests zustandslos bearbeitet werden?
- Ist das Handling eines Requests eher CPU- oder I/O-lastig?
- Wie kritisch ist ein Absturz des Servers?
- Performance, Latenz, Skalierbarkeit.
- Testbarkeit, Debugging.

Entscheidungskriterien
für die Wahl des „richtigen“ Designs

- Können Requests zustandslos bearbeitet werden?
- Ist das Handling eines Requests eher CPU- oder I/O-lastig?
- Wie kritisch ist ein Absturz des Servers?
- Performance, Latenz, Skalierbarkeit.
- Testbarkeit, Debugging.

Entscheidungskriterien
für die Wahl des „richtigen“ Designs

- Können Requests zustandslos bearbeitet werden?
- Ist das Handling eines Requests eher CPU- oder I/O-lastig?
- Wie kritisch ist ein Absturz des Servers?
- Performance, Latenz, Skalierbarkeit.
- Testbarkeit, Debugging.

Entscheidungskriterien
für die Wahl des „richtigen“ Designs

- Können Requests zustandslos bearbeitet werden?
- Ist das Handling eines Requests eher CPU- oder I/O-lastig?
- Wie kritisch ist ein Absturz des Servers?
- Performance, Latenz, Skalierbarkeit.
- Testbarkeit, Debugging.

Entscheidungskriterien
für die Wahl des „richtigen“ Designs

- Können Requests zustandslos bearbeitet werden?
- Ist das Handling eines Requests eher CPU- oder I/O-lastig?
- Wie kritisch ist ein Absturz des Servers?
- Performance, Latenz, Skalierbarkeit.
- Testbarkeit, Debugging.

01-echo.cpp
Ein simpler UDP-Echoserver

- Endlosschleife von `recvfrom()`- und `sendto()`-Aufrufen.
- Primitives Signalhandling, um den Server „sauber“ zu beenden.

02-echo_unix_server.cpp

Ein simpler TCP-Echoserver

- Wie 01-echo.cpp.
- Verwendung von TCP statt UDP.
- Verwendung eines Unix-Domain-Sockets.
- Als Client muß 02-echo_unix_client.cpp verwendet werden.

03-echo_fork.cpp
Ein UDP-Echoserver mit Child-Prozessen

- Forken von vier Child-Prozessen.
- Parallele Endlosschleife von `recvfrom()`- und `sendto()`-Aufrufen in allen Childs.
- Entspricht in etwa dem Leader/Follower-Pattern.
- Das Betriebssystem sorgt dafür, daß eingehende Requests auf die Childs verteilt werden.
- Der SIP Express Router (SER) arbeitet nach einem ähnlichem Prinzip.

04-chat.cpp
Ein primitiver TCP-Chat-Server

- Verwendung von `poll()` für das Multiplexing.
- Unterstützung von primitiven Kommandos.

Weiterführende Literatur

- Die man-Pages der jeweiligen System-Calls. :-)
- Advanced Programming in the Unix Environment (Addison-Wesley)
- Unix Network Programming (Addison-Wesley)
- C++ Network Programming (Addison-Wesley)
- PThreads Programming (O'Reilly)
- Secure Coding: Principles and Practices (O'Reilly)
- Der Quellcode bestehender Open-Source-Applikationen (Apache, SER, etc.).

Ende

- Questions & Answers.
- `exit()` ;-)