Verteilte Systeme

...für C++ Programmierer

Serverprogrammierung

by

Dr. Günter Kolousek

Serverprogrammierung

- Server
  - stellt Dienste (services) zur Verfügung
  - läuft im Hintergrund (siehe Folien ”Prozesse”)
  - läuft ”ewig”
    - soll nicht ”sterben”
  - meist: wartet auf Request und antwortet mit Response
    - im allgemeinen Sinne
    - startet u.U. weitere Prozesse, Threads
- Programmierung von Server-SW
  - Funktionalität
  - zuverlässig, robust !
  - sicher !
  - performant, skalierbar, wartbar,…

Entwicklungsentscheidungen

- Technologieauswahl
- Entwicklungsprozess
  - Anforderungsmanagement, Qualitätssicherung, Peer reviews, Codegenerierung,...
- SW-Architektur und Entwurf
  - $\rightarrow$ siehe "`software_architecture`"
- Implementierung
- Testen

Technologieauswahl

- Prozessor, Netzwerk,…
- Betriebssystem
  - Server
    - Standardanwendungen → Windows…
    - Serverdienste → Windows vs. Unix
  - Internet → Unix (Linux, BSD)
  - Eingebettetes System → QNX, VxWorks, embedded Linux (z.B. OpenWrt, RTAI), Windows embedded,…
- Middlewaretechnologie
- Programmiersprache
  - Funktionalität, Produktivität, sicheres Programmieren, Performance, Ressourcen-Bedarf
  - Assembler, C, C++, Java, C#, Python, Erlang, Go,…
- Tools: Entwurf, Debugging (→ Memory leaks,…), Test

Implementierung

- Fehlerbehandlung, Fehlerüberprüfungen
  - Exceptions vs. Error-Codes
- Speicherverwaltung
  - manuell vs. automatisch
  - heap vs. stack
- Verwendung von Constraints
  - precondition, postcondition, invariant
- Dokumentation
- Codegenerierung
  - z.B.: FSM (finite state machine), Parsergenerierung, MDA (model-driven archtitecture),…

Implementierung – 2

- Kommunikation
  - Schließen der Verbindung

    z.B. bei Request/Response:

    1. Client: sendet Request
    2. Client: shutdown auf output stream
    3. Server: empfängt Request (bis keine weiteren Daten)
    4. Server: shutdown auf input stream (kein weiteres Lesen)
    5. Server: sendet Response
    6. Server: shutdown auf output stream (kein weiteres Senden)
    7. Client: empfängt Response (bis keine weiteren Daten)
    8. Client: schließt Socket

Implementierung – 3

- Kommunikation
  - Verbindung bricht ab (z.B. Client-Prozess stirbt) $\rightarrow$ Server hängt (Wartezeit) $\rightarrow$ keine Locks halten bei Aufruf blockierender Aufrufe!
  - $\rightarrow$ *Be strict in what you send and tolerant in what you receive*
    - aber: Validierung aller empfangenen Daten $\rightarrow$ security!
- Serverarten
  - Iterativer vs. nebenläufiger Server
  - Daemon-Server vs. Super-Server
  - statusloser vs. statusbehafteter Server
  - Objektserver: Unterstützung verteilter Objekte
    - $\rightarrow$ Folien `communication`
- Daemonizing

Iterativer vs. nebenläufiger Server

- Iterativer (oder sequentieller) Server
  - verarbeitet Anforderung selbst
  - blocking server: blockierende Funktionsaufrufe
    - nur 1 Verbindung zu einem Client
  - nonblocking server: nicht-blockierende Funktionsaufrufe
    - polling oder event-driven
    - z.B. select - API, `asio`, Java, .Net,...
- Nebenläufige (parallele) Server
  - verarbeitet Anforderung nicht selbst
    - $\rightarrow$ eigener Prozess oder eigener Thread

select-basierte Server

```
import select, socket, struct, time
PORT = 8037
TIME1970 = 2208988800L # secs since 1.1.1970
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind(("", PORT)); serversock.listen(1)
print("listen on port", PORT)

isreadable,iswriteable,iserr = [serversock],[],[serversock]
while 1:
    # time-out of 1s (default: blocking)
    r,w,e = select.select(isreadable, iswriteable, iserr, 1)
    if r:
        client, info = serversock.accept()
        print("connection from", info)
        t = int(time.time()) + TIME1970
        t = struct.pack("!I", t)  # network-byte-order, uint
        client.send(t)  # 4 bytes certainly will not block
        client.close()
    else:
        print("further waiting")
```

Nebenläufiger Server

- Je Request, Verbindung, Client
  - ein Thread, ein Prozess
  - u.U. Thread-Pool, Prozess-Pool
- multi-process Server
- multi-threaded Server

Multi-threaded Server

- `main`-Thread wartet auf Verbindung
  - startet Client-Thread je Verbindung zu Client
- Client-Threads warten blockierend auf Anfragen des Clients
- Vorteil: einfach
- Nachteile:
  - Erzeugen, Löschen und Verwalten (inkl. Context-Switch) sind kostspielige Operationen
  - Skalierbarkeit kann leiden
  - Sychnronisation
    - Overhead durch Synchronisationsmechanismen
    - Wahrscheinlichkeit von Programmierfehlern in Synchronisation höher

Daemon-Server vs. Super-Server

2 spezielle Aspekte:

- Wie erfährt Client zu welchem Port verbunden werden muss?
  - fixe Zuordnung: `/etc/services` oder systemspezfisch

```
http            80/tcp
http            80/udp
www             80/tcp
www             80/udp
www-http        80/tcp
www-http        80/udp
http            80/sctp
```

  - dynamische Zuordnung $\rightarrow$ Daemon-Server
- Lebenszeit des Serverprozesses
  - Wird bei Systemstart gestartet

Daemon-Server vs. Super-Server

2 spezielle Aspekte:

- Wie erfährt Client zu welchem Port verbunden werden muss?
  - fixe Zuordnung: `/etc/services` oder systemspezfisch

```
http            80/tcp
http            80/udp
www             80/tcp
www             80/udp
www-http        80/tcp
www-http        80/udp
http            80/sctp
```

  - dynamische Zuordnung $\rightarrow$ Daemon-Server
- Lebenszeit des Serverprozesses
  - Wird bei Systemstart gestartet
    - Was ist wenn dieser nie gebraucht wird?

Daemon-Server vs. Super-Server

2 spezielle Aspekte:

- Wie erfährt Client zu welchem Port verbunden werden muss?
  - fixe Zuordnung: `/etc/services` oder systemspezfisch

```
http            80/tcp
http            80/udp
www             80/tcp
www             80/udp
www-http        80/tcp
www-http        80/udp
http            80/sctp
```

  - dynamische Zuordnung $\rightarrow$ Daemon-Server
- Lebenszeit des Serverprozesses
  - Wird bei Systemstart gestartet
    - Was ist wenn dieser nie gebraucht wird?
  - Wird bei Bedarf gestartet $\rightarrow$ Super-Server

Daemon-Server

- Ablauf/Funktion
  - Server startet sich und registriert sich bei Daemon $\rightarrow$ freier Port wird zugewiesen
    - z.B. http, smtp, imap,... ($\rightarrow$ `/etc/services`) oder applikationsspezifisch...
  - Daemon lauscht an definierten Port und beantwortet Anfragen des Clients bzgl. Diensten mit der entsprechenden Portnummer
  - Client kann danach direkt mit dem Server kommunizieren
- Vorteil: Client muss keinen speziellen Serverport kennen
- Nachteil: zusätzlicher Dienst, zusätzliche Abfrage
- Beispiel: portmapper Mechanismus (Unix)

Super-Server

- Ablauf/Funktion
  - Super-Server läuft permanent und lauscht an *allen* Ports, die den angebotenen Diensten zugeordnet sind
  - Client verbindet sich mit spezifizierten Port
  - Super-Server startet bei Bedarf den entsprechenden Serverprozess
  - Client kommuniziert danach direkt mit Server
- Vorteil: Minimierung der gestarteten Server-Prozesse am Server
- Nachteil: erstmalige Anfrage dauert länger
- Beispiel: inetd Modell (Unix)

Statusloser vs. statusbehafteter Srv

- statusloser Server
  - speichert keine Information über Clients
  - Vorteil: robust gegenüber Abstürzen
  - Nachteil: Status muss vom Client verwaltet werden und jedes Mal übertragen werden
- statusbehafteter Server
  - verwaltet Status der Clients
  - Vorteil: komplexere Operationen möglich
  - Nachteil: Recovery nach Absturz kann problematisch sein, da
    - Nachrichten von vorhergehenden Nachrichten abhängig sein können
    - nicht jede gesendete Nachricht einfach nochmals gesendet werden kann (”überweise 100€”)

Daemonizing

- Daemon
  - Hintergrundprozess
  - (fast) ohne Interaktion mit Benutzer
  - z.B. httpd (apache, nginx), ntpd, sshd,…
- Tätigkeiten
  - Forking und Elternprozess beenden
    - neuer Child $\rightarrow$ orphaned $\rightarrow$ init übernimmt!
  - Neue eindeutige Sessions ID anlegen
    - Signale werden vom Terminal an Prozess gesendet
    - Kindprozess erbt Terminal von Elternprozess
    - Kindprozess erbt Session von Elternprozess
    - $\rightarrow$ neue Session (ohne Terminal)

Daemonizing – 2

- Tätigkeiten – 2
  - (Geerbte) Dateideskriptoren schließen
  - Ändern der umask (Maske benutzt für Rechte bei Dateierzeugung)
  - in das richtige Arbeitsverzeichnis wechseln
  - sicherstellen, dass nur ein Prozess je Daemon läuft (mittels Lock)
  - Signale abfangen und behandeln
  - Logs anlegen/öffnen
  - Privilegien abgeben (setuid,…)
    - $\rightarrow$ Ports, Rechte zum Anlegen von Dateien,…

Testen

- Testen allgemein: siehe POS
- Funktion
  - Black-Box und White-Box Tests
  - formale Spezifikation und Verifikation
- Last, Lastschwankungen, Langzeittests
  - Speicher, CPU, IO (Netzwerk, Massenspeicher, Geräte)
- Fehler: Verhalten bei definierten Fehlersituationen
  - $\rightarrow$ Zuverlässigkeit
- Stresstests: Verhalten in Ausnahmesituationen
  - Crashtests: Versuche System $\rightarrow$ Absturz
- Wiederinbetriebnahme
- Sicherheit: potentielle Sicherheitslücken