Informatik B

Vorlesung 17
Netzwerkprogrammierung

Rückblick

- **`URL`**-Objekt
- **`Socket`**
- Verbindung zu einem Server aufbauen
- Webserver aus Clientsicht

Serverimplementation

- Server bauen keine eigene Verbindung auf, sondern horchen an ihrem zugewiesenen Port auf Eingaben und Anfragen
- Ein Server wird durch die Klasse **`ServerSocket`** repräsentiert
- Der Konstruktor bekommt einfach die Port-Nummer, zu der sich Clients verbinden können, als Argument übergeben

Portwahl

- Bei der Wahl des Ports muss folgendes bedacht werden:
  - Wird ein bereits belegter Port gewählt folgt daraus eine **`IOException`**
  - Auf Unix basierten Systemen können von einem normalen Benutzer nur Ports oberhalb von 1024 genutzt werden, ansonsten sind root-Rechte erforderlich

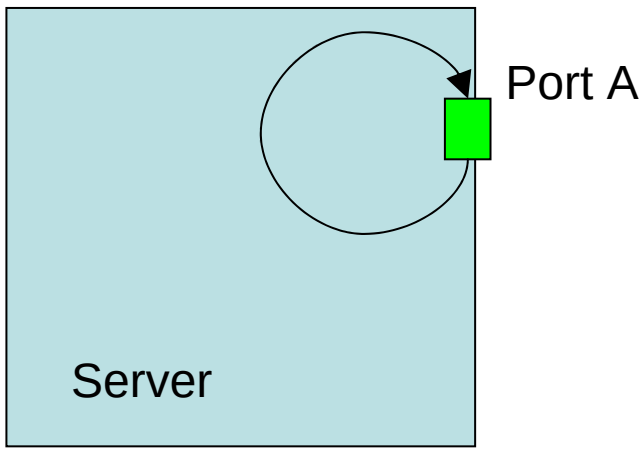
Auf Verbindungen horchen

- Wurde der **`ServerSocket`** instanziiert horcht der Server auf eingehende Verbindungen
- Dazu wird die Methode **`accept()`** verwendet
- Diese Methode blockiert solange, bis eine Verbindung durch einen Client hergestellt wurde
- Wird eine Verbindung hergestellt, liefert die Methode einen **`Socket`** zurück, über den kommuniziert werden kann

Auf Verbindungen horchen

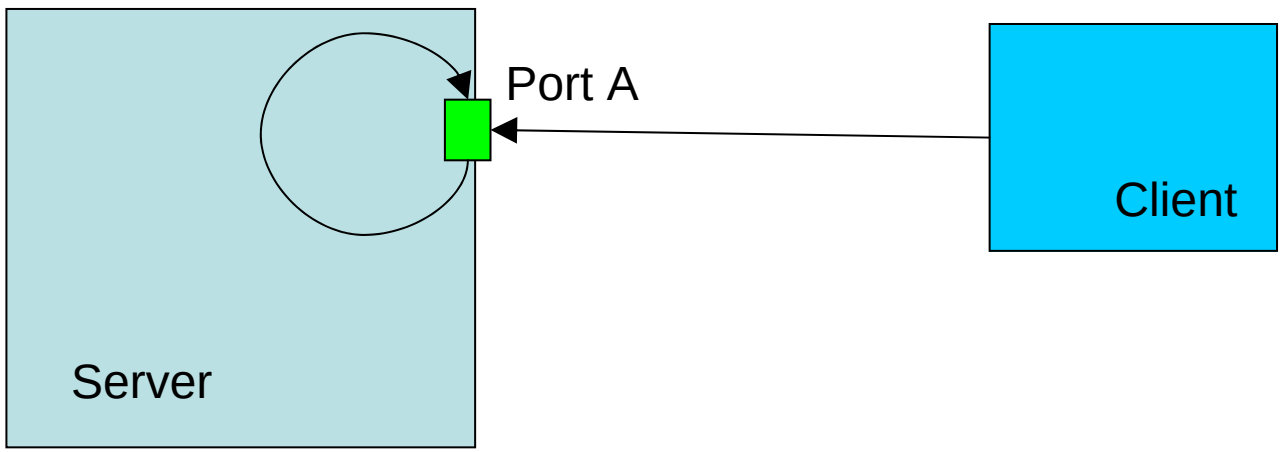
- Damit der Port nicht blockiert wird, wird der von **`accept()`** erzeugte **`Socket`** auf einen freien Port gelegt
- Der **`ServerSocket`** ist somit wieder frei für eine weitere Anfrage
- Beispielhaft:
  - Clienten wollen in einem Bürogebäude (Server) etwas bearbeitet haben
  - Sie wenden sich an die für sie vorgesehene Auskunftsstelle (Port)
  - Sie werden drangenommen und an einen freien Mitarbeiter verwiesen (anderer Port)

Auf Verbindungen horchen



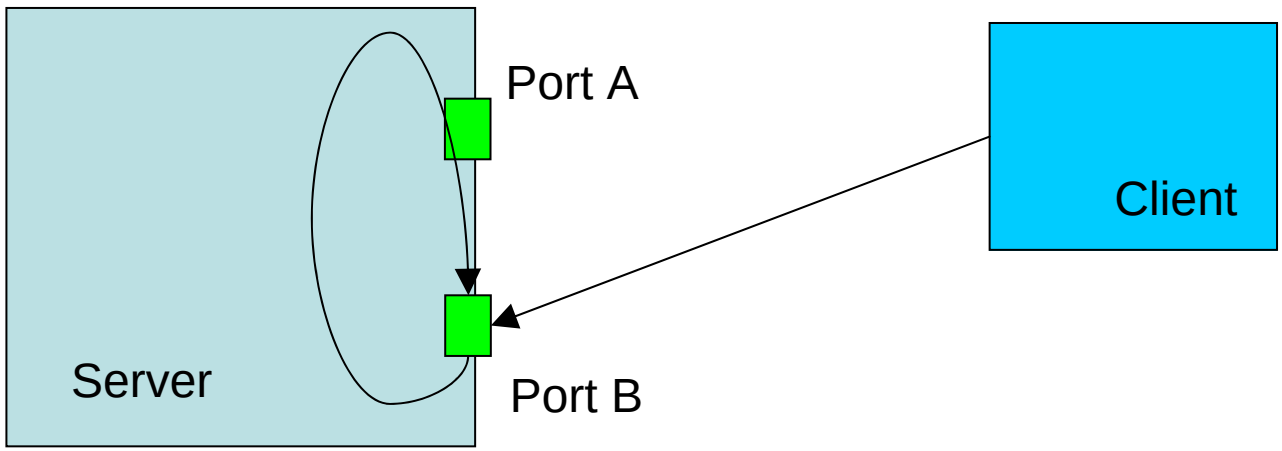
Der Server wartet auf eine Verbindung
und ist in der Methode **`accept()`** blockiert

Auf Verbindungen horchen



Der Client verbindet sich mit dem
Server an dem entsprechenden Port

Auf Verbindungen horchen



Der Server akzeptiert die Verbindung
und erzeugt ein neues Socket Objekt
an einem anderen Port

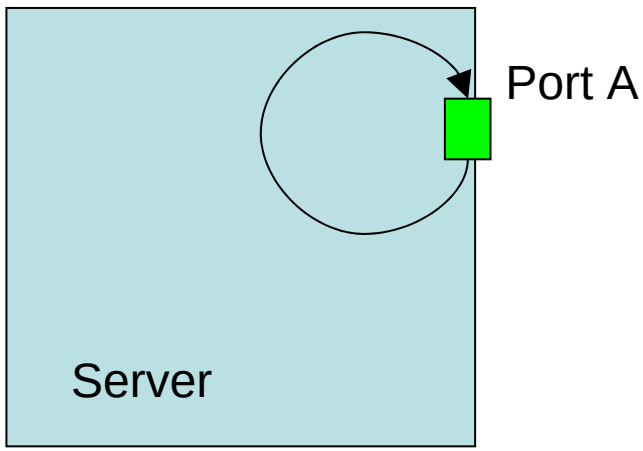
Time-Server

- Will man einen simplen Time-Server implementieren wären folgende Schritte notwendig:
  - Erzeugen eines **`ServerSocket`**-Objektes
  - In einer Endlosschleife immer wieder auf Verbindungseingänge warten (**`accept()`**)
  - Erfolgt eine Verbindung erhält man einen **`Socket`**
  - Schreiben der Informationen
  - **`Socket`** schließen
- Beispiel: **`server1`**

Verarbeitung vieler Anfragen

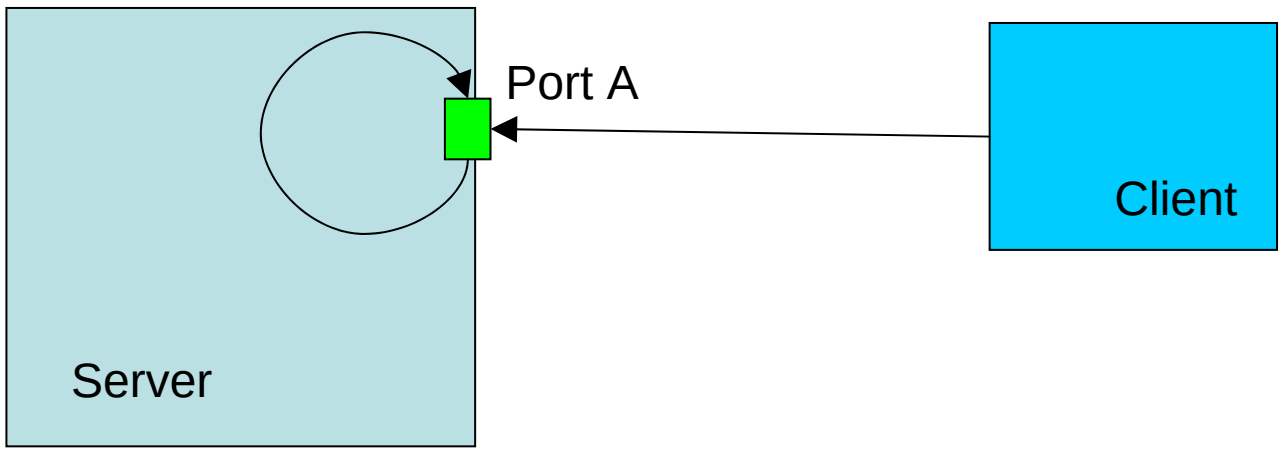
- Im Beispiel **`server1`** können die Anfragen nur nacheinander abgearbeitet werden
- Dies liegt daran, dass zwar der Port des **`ServerSocket`**-Objektes freigegeben wird, aber erst die Kommunikation mit dem Client erfolgt, bevor wieder auf eine Verbindung gehorcht wird
- Abhilfe bieten hier Threads
- Jede eingehende Verbindung wird durch **`accept()`** auf einen freien Port gelegt und sollte in einem separaten Thread abgearbeitet werden
- Beispiel: **`server2`**

Verarbeitung vieler Anfragen



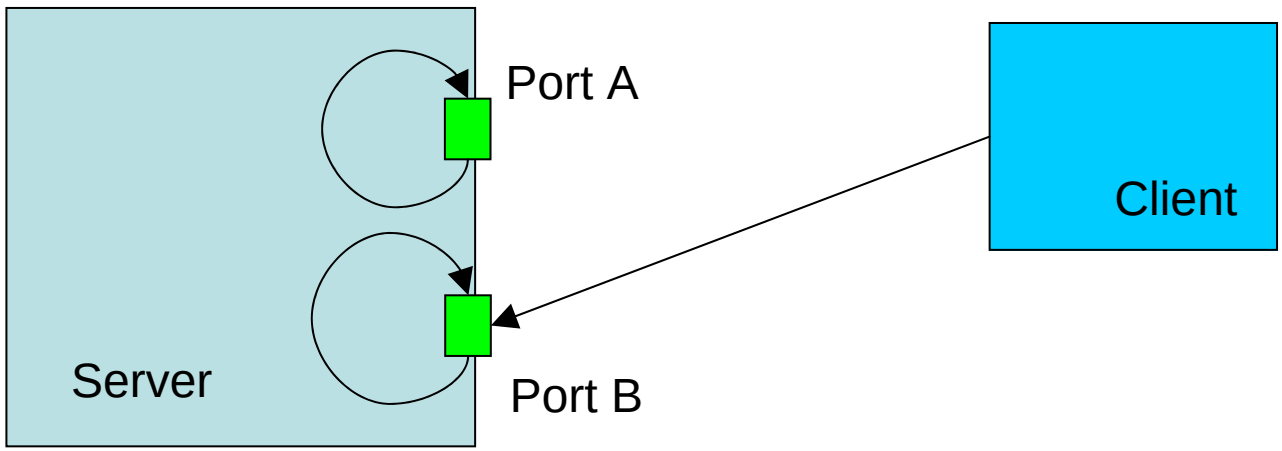
Der Server wartet auf eine Verbindung
und ist in der Methode accept() blockiert

Verarbeitung vieler Anfragen



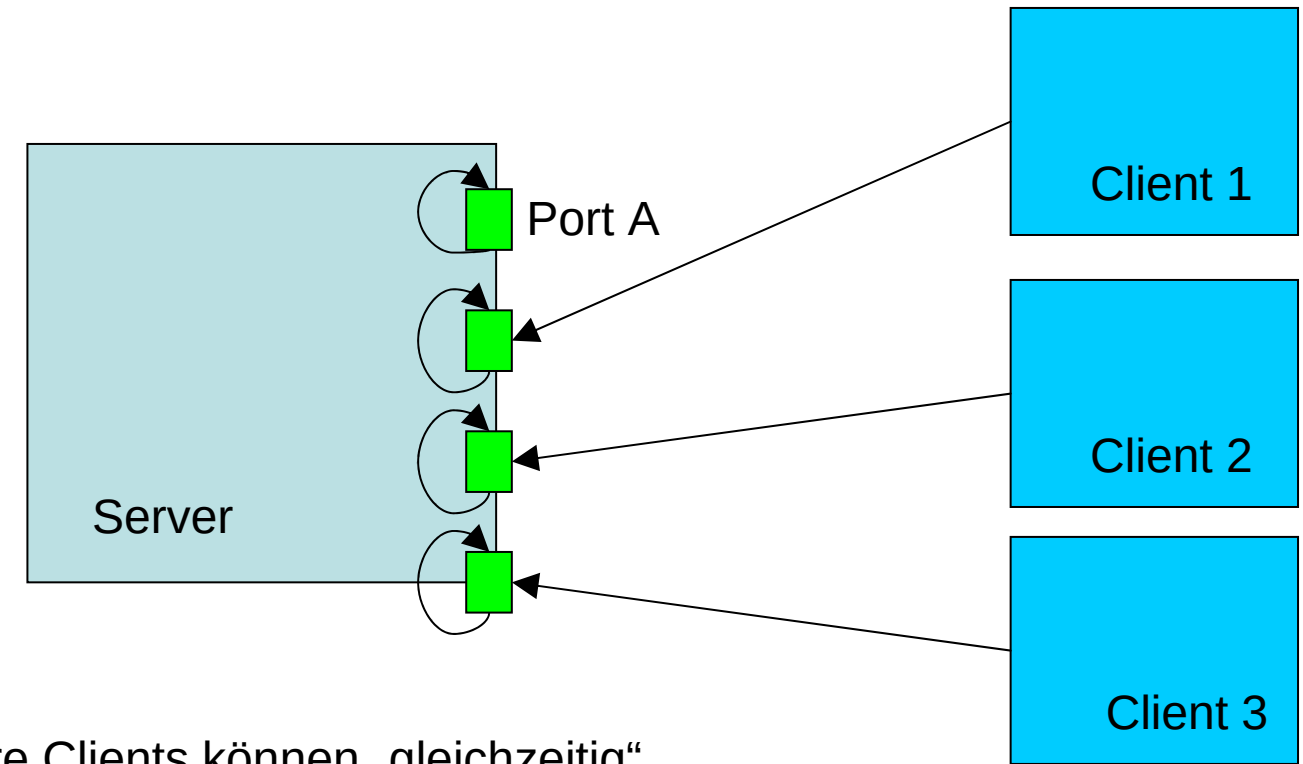
Der Client verbindet sich mit dem
Server an dem entsprechenden Port

Verarbeitung vieler Anfragen



Der Server akzeptiert die Verbindung und erzeugt ein neues Socket Objekt an einem anderen Port, die Abarbeitung erfolgt in einem separaten Thread

Verarbeitung vieler Anfragen



Mehrere Clients können „gleichzeitig“ versorgt werden

Chat-Server

- Der Time-Server war eine sehr einfache Anwendung, die eine Information an den Client schreibt
- Der Chat-Server sollte folgende Eigenschaften haben:
  - Es können sich „beliebig“ viele Chatter anmelden
  - Ein Chatter muss zunächst einen Namen angeben
  - Schreibt ein Chatter etwas, sollen es alle anderen sehen
  - Es soll vor einer Nachricht der Name des Chatters ausgegeben werden
  - Ein Chatter kann den Chat abbrechen, indem er einen Punkt auf einer leeren Zeile eingibt

Chat-Server

- Der Chat-Server muss sich merken, welche Chatter sich angemeldet haben
- Jeder Chatter muss in einem eigenen Thread abgearbeitet werden
- Wird eine Nachricht an alle gesendet, darf dies nicht durch einen anderen Thread unterbrochen werden, da sonst die Nachrichten in unterschiedlicher Reihenfolge ankommen => Synchronisation!!!
- Beispiel: **`chatserver1`**

Chat-Server

- Eine Verbindung kann „hart“ abgebrochen werden, wenn der Client beendet wird
- Wenn dies geschieht werden im Beispiel **`chatserver1`** trotzdem noch alle anderen Chatter darüber informiert
- Wo ist das im Code geregelt?
  - Wenn die Verbindung eines Chatters unterbrochen wird, wird automatisch der Socket geschlossen
  - Das führt dazu, dass sein **`InputStream`** geschlossen wird
  - Daher bricht die Schleife ab und die Nachricht wird an die anderen Chatter übermittelt
- Beispiel: **`chatserver1`**

`flush()`

- In einer Client/Server-Anwendung mit Sockets werden oft gepufferte Ströme eingesetzt, um nicht laufend kleine Datenpakete zu senden
- Insbesondere wenn Ströme wie **`BufferedInputStream`** oder **`BufferedOutputStream`** eingesetzt werden, werden die Informationen im Puffer zwischengespeichert und nicht direkt zum anderen Rechner übertragen
- In einem Frage-Antwort-Szenario muss der Server oder Client die Anfrage direkt übertragen und die Nachricht darf nicht im Puffer verweilen
- Daher müssen die **`flush()`**-Methoden der Puffer-Klassen die aufgenommenen Daten vermitteln, damit die Kommunikation weitergeht

Serverprotokoll

- Der Server kann ein Protokoll definieren
- Dass heißt er hat ein bestimmtes Set von Nachrichten, auf die er reagieren kann
- Erhält er eine Nachricht, wird überprüft, ob diese bekannt ist, falls ja wird die entsprechende Aktion ausgeführt
- Falls nein wird eine Fehlermeldung zurückgegeben
- Es sollte dem Benutzer eine Übersicht über alle Meldungen gegeben werden können

Telefonbuchserver

- Es soll ein einfacher Telefonbuchserver implementiert werden
- Die Namen/Nummer-Paare sind fest in einer Map verdrahtet
- Der Server soll die Nachrichten
  - **`HELP`**: Übersicht aller Kommandos
  - **`GET name`**: Anfrage nach exaktem Namen
  - **`LIKE pattern`**: Testet ob ein Name vorhanden ist, der das **`pattern`** enthält
- Beispiel: **`telbookserver1`**

Telefonbuchserver

- Das Erfragen der einzelnen Messages führt zu einer komplexen **`run()`**-Methode
- Eine Verbesserung ist die starke Modularisierung der Methode (auslagern in einzelne Methoden)
- Eine andere Möglichkeit stellt das Reflection-Modell zur Verfügung
- Beispiel: **`telbookserver2`**

Telefonbuchserver

- Im Beispiel **`telbookserver2`** wurde eine einfache Möglichkeit gezeigt die Nachrichten zu verarbeiten
- **Achtung**: Wird mit Reflection gearbeitet, muss sichergestellt werden, dass nur Methoden aufgerufen werden, die auch wirklich erlaubt sind!!!!! Von daher muss man wirklich wissen was man tut!!!!!

TCP vs. *UDP*

- *TCP* ist ein verbindungsorientiertes Protokoll, dass auf der Basis von IP eine gesicherte Punkt zu Punkt Verbindung zur Verfügung stellt (Analogie: telefonieren)
- Im Gegensatz dazu gibt es das *UDP* (*User Datagram Protocol*)
- UDP ist verbindungslos, d.h.:
  - Die Anwendung muss selbst dafür sorgen, dass alle Pakete ankommen
  - Die richtige Reihenfolge der Pakete muss sichergestellt werden
  - Durch geringere Überprüfung ist das Protokoll schneller

TCP vs. UDP

- UDP erzeugt weniger Netzbelastung als TCP:
  - Keine Verbindung, sondern Pakete
  - Ungesichert
- UDP wird bei Kommunikationen eingesetzt, die vom Client und Server selbst überwacht werden kann
  - Wenn die Netzlast gering gehalten werden muss
  - Wenn keine Gewähr für das Ankommen der Pakete übernommen werden muss
  - Bei Broadcast (einer sendet an viele)

UDP

- In Java wird das UDP durch die Klassen **`DatagramPacket`** und **`DatagramSocket`** abgebildet
- **`DatagramSocket`** ist eine Schnittstelle zum senden und empfangen von Paketen
- **`DatagramSocket()`** // am ersten freien Port
  `DatagramSocket(int port)`
  `DatagramSocket(int port, InetAddress iaddr)`
- **`send(datagrampacket)`**
  `receive(datagrampacket)`
  `close()`

UDP

- **`DatagramPacket`** stellt ein zu versendendes Paket dar
- Serverseitig
  `DatagramPacket(byte[] buf, int buflen)`
- Clientseitig mit Angabe des Empfängers
  `DatagramPacket(byte[] buf, int buflen, InetAddress iaddr, int port)`
- Ein Datenpaket kann also mit einer Adresse und Inhalt versehen und direkt verschickt werden
- Der Server erhält ein Datenpaket, kann den Inhalt austauschen und das Paket wieder direkt zurücksenden

TimeServer mittels UDP

- Der Server wartet an einem Port auf ein Paket
- Der Client verschickt mittels UDP ein Datenpaket an den Server
- Der Server erhält das Paket mit dem enthaltenen Absender
- Der Server erzeugt ein Paket, kopiert den Absender in das neue Paket als Adresse
- Packt einen String mit der aktuellen Zeit in das Paket
- Der Server schickt das Datenpaket mittels UDP zurück
- Beispiel: **`udp1`**

Zusammenfassung

- ServerSocket
  - Verbindung wird angenommen
  - Auf einen freien Port gelegt
  - Kommunikation mit dem Client in einem eigenen Thread
- UDP
  - Verbindungslos
  - Nicht gesichert

Ausblick

- GUI (Graphical User Interface)
- MVC (Model View Controller)