

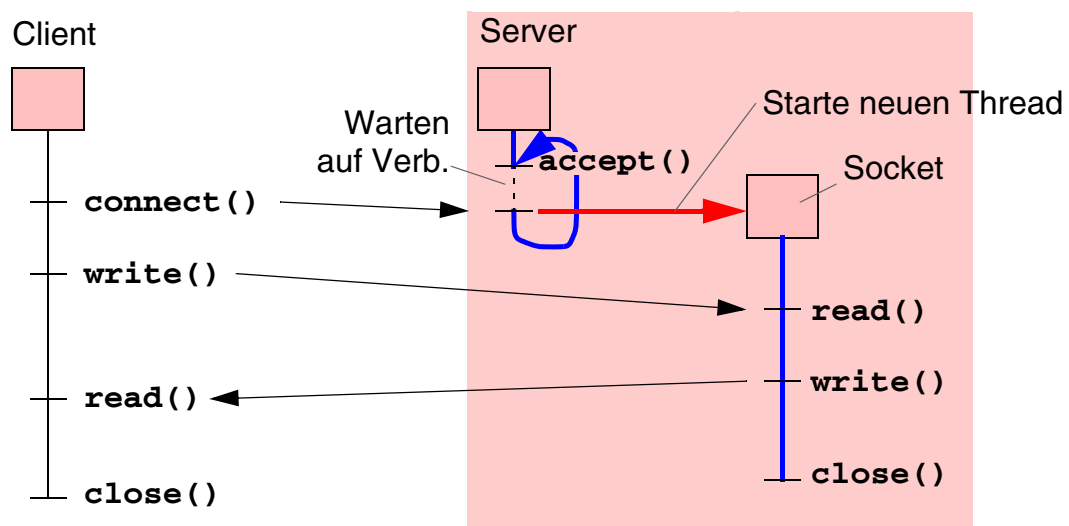
1.5 Threaded Server

- Server als ein Prozess mit mehreren Threads
- Threads
 - ◆ Thread als Aktivitätsträger
 - „virtueller Prozessor“
 - eigener Programmzähler
 - eigener Stackbereich
 - ◆ eingebettet in den Kontext eines Prozesses
 - gemeinsamer Adressraum
 - gemeinsame Betriebsmittel (Sockets, Dateien etc.)
- ★ Vorteil
 - ◆ Thread-Erzeugung kostengünstiger als Prozesserzeugung
 - ◆ Thread-Umschaltung kostengünstiger als Prozessumschaltung



1.5 Threaded Server (2)

- Beispiel mit Sockets



- ◆ mehrere Threads im Serverprozess



1.5 Threaded Server (3)

★ Weitere Vorteile

- ◆ Thread-Umschaltung durch Betriebssystem oder Thread-Bibliothek
- ◆ volle Nebenläufigkeit im Server
 - geringst mögliche mittlere Bearbeitungszeit

▲ Nachteile

- ◆ Nebenläufigkeitskontrolle erforderlich (Koordinierung der Threads)
- ◆ Thread-Erzeugung evtl. teuer
- ◆ Obergrenze für sinnvolle Thread-Anzahl notwendig
 - Ressourcenverbrauch wirkt sich nachteilig aus
 - z.B. zusätzliche Speicherauslagerungen
 - z.B. Timeouts durch faires Scheduling führen zu unnötigen Nachrichten



1.6 Einsatz von Threads

■ Zwei Implementierungsmöglichkeiten

- ◆ Kernel-Level-Threads
 - Threads im Betriebssystem implementiert
 - Betriebssystem schaltet um
 - **Vorteil:** bei blockierenden Systemaufrufen wird nur ein Thread blockiert
- ◆ User-Level-Threads
 - Threads im Prozess implementiert
 - Thread-Software schaltet den **einen** Aktivitätsträger des Prozesses auf **viele** Threads um
 - **Vorteil:** effizientere Umschaltung



1.6 Einsatz von Threads (2)

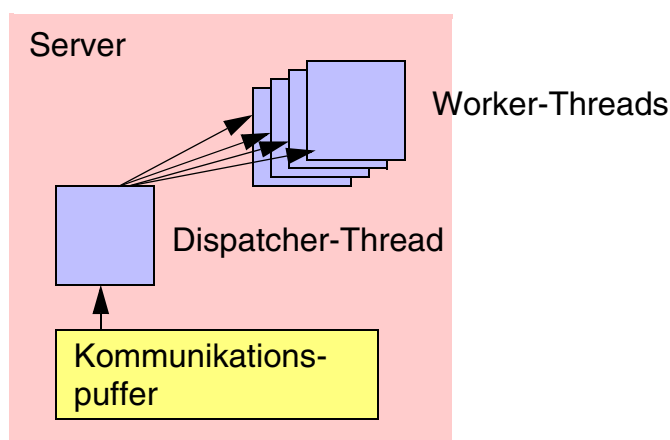
- Thread-Software
 - ◆ Integration in die Sprache
 - Thread-Konzept und Koordinierungsmittel nahtlos in Sprache eingebettet
 - z.B. Java, Ada
 - ◆ Thread-Bibliotheken
 - zusätzliche Software macht Sprache threading-fähig
 - z.B. C++, C

- Effizientere Thread-Erzeugung
 - ◆ Pool von Arbeitsthreads
 - ◆ Threads im Pool warten auf Aufträge
 - ◆ Neuerzeugung nur bei leerem Pool erforderlich
 - ◆ Poolgröße ist Tuningparameter



1.7 Implementierungsstrukturen

- Dispatcher/Worker-Modell, Master-Slave-Modell
 - ◆ Haupt-Thread empfängt Aufträge und
 - ◆ gibt diese an Bearbeiter-Thread (Worker) weiter (evtl. dynamisch erzeugt)
 - siehe Beispiel auf Folie G.10



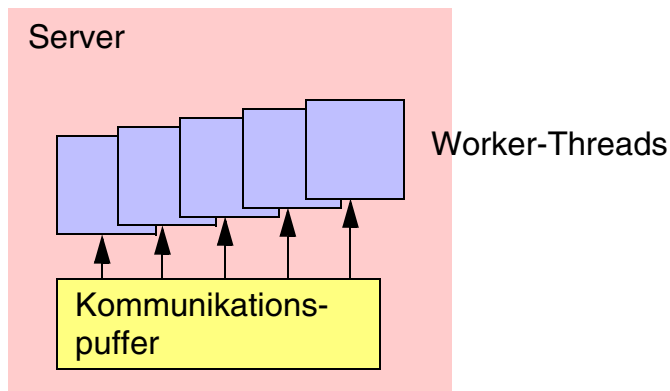
nach Weber



1.7 Implementierungsstrukturen (2)

■ Team-Modell

- ◆ alle Threads arbeiten gleichberechtigt
- ◆ zentrale Komponente stellt Aufträge bereit
 - z.B. alle rufen `accept()` am Socket auf
 - z.B. Abruf von Aufträgen an einem Auftrags- oder Nachrichtenpuffer
 - Koordinierung beim Zugriff erforderlich!



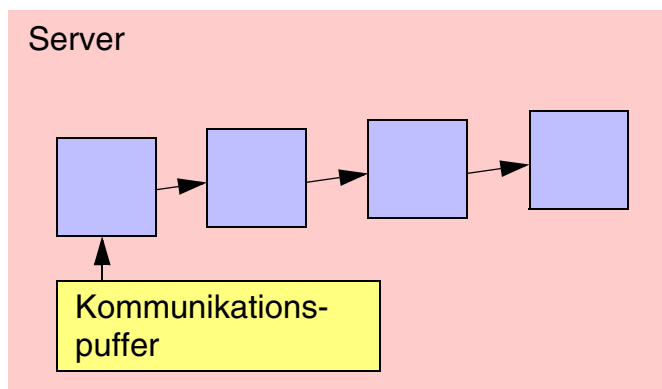
nach Weber



1.7 Implementierungsstrukturen (3)

■ Pipeline-Modell

- ◆ jeder Thread bearbeitet nur Teilauftrag
- ◆ Anordnung der Threads in einer Bearbeitungskette (Pipeline)
- ◆ Weitergabe der Ergebnisse an Folge-Thread
 - Zwischenpuffer zur Entkopplung benachbarter Threads sinnvoll



nach Weber



2 Lastverteilung

2.1 Verteilter Server

- Mehrere Serverprozesse auf verschiedenen Rechnerknoten
 - ◆ Beispiel: Dispatcher-Modell mit fest konfigurierten, verteilten Bearbeiterprozessen
- ★ Last lässt sich auf mehrere Knoten verteilen
- Arten von Last
 - ◆ Verbrauch von Betriebsmitteln
 - Prozessor: Rechenlast
 - Speicher: Speicherlast
 - Netzwerkbandbreite: Kommunikationslast
 - u.a.



2.1 Verteilter Server (2)

- Anwendung der Implementierungsstrukturen für Threads
 - ◆ Dispatcher-Modell
 - zentraler Server gibt Aufträge an fest konfigurierte Prozesse weiter
 - zentraler Server startet für Aufträge neue Prozesse auf verschiedenen Rechnern
 - z.B. Ray-Tracing (Push-Modell)
 - ◆ Team-Modell
 - Bearbeiterprozesse holen sich Aufträge von zentraler Stelle
 - z.B. Ray-Tracing (Pull-Modell)
 - ◆ Pipeline-Modell
 - Zwischenergebnisse werden von Prozess zu Prozess weitergereicht
 - z.B. Trainingsläufe für Spracherkenner



2.2 Einsatz bei Arbeitsplatzrechnern

- Viele Arbeitsplatzrechner nicht ausgelastet
 - ◆ Benutzer häufig mit anderen Dingen beschäftigt
 - ◆ nachts inaktiv

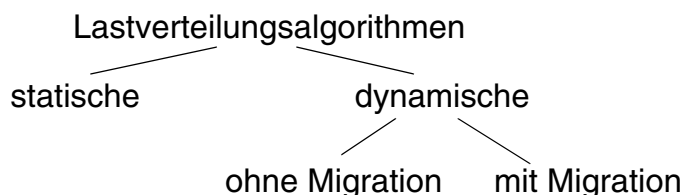
- Starten von Bearbeiterprozessen auf ungenutzten Arbeitsplatzrechnern
 - ◆ Problem: Sicherheit
 - vertrauenswürdige Arbeitsplatzrechner erforderlich
 - ◆ Problem: Benutzer kommt zurück, will rechenintensive Software nutzen
 - Bearbeiterprozess stoppen
 - Bearbeiterprozess abrechnen und anderswo neu starten
 - laufenden Bearbeiterprozess verlagern (Migration)



2.3 Lastverteilungsalgorithmen

- Problem
 - ◆ gegeben mehrere Prozesse mit gegebenem Kommunikationsverhalten
 - ◆ welche Verteilung für optimale Lastbalancierung
 - Vermeidung von Ressourcenengpässen
 - effiziente Abarbeitung

- Einteilung der Algorithmen



2.3 Lastverteilungsalgorithmen (2)

- Statische Algorithmen
 - ◆ berechnen eine optimale initiale Prozessverteilung
- Dynamische Algorithmen
 - ◆ berechnen für jede Prozesserzeugung idealen Ort
 - ◆ berücksichtigen aktuelle Last an den Orten (Messung)
 - ◆ mit Migration
 - zur Lastveränderung werden Prozesse verlagert



3 Migration

- Verlagerung einer Softwarekomponente an einen neuen Ort
 - ◆ Prozesse
 - ◆ Objekte
 - ◆ ...
- Relokationstransparenz
 - ◆ Verlagerung bleibt den Kommunikationspartnern verborgen
- Passive Komponenten
 - ◆ einfache Verlagerung durch Übertragung von Daten und Code
 - ◆ notwendiges Warten bis alle laufenden Operationen beendet
 - ◆ **schwache Migration**



3 Migration (2)

- **Aktive Komponenten**
 - ◆ Verlagerung eines laufenden Aktivitätsträgers
 - ◆ zusätzlich Übertragung von
 - Programmzähler
 - Registersatz
 - Stackspeicher
 - ◆ Hilfe des Betriebssystems notwendig
 - ◆ **starke Migration**

- ▲ **Problem**
 - ◆ Umgebung der Komponente muss erhalten bleiben
 - z.B. offene Dateien eines verlagerten Prozesses



3 Migration (3)

- ▲ **Problem**
 - ◆ verschiedene Rechnerarchitekturen
 - unterschiedliche Datenspeicherung
 - z.B. Little-Endian vs. Big-Endian
 - unterschiedliche Adressräume
 - z.B. 32 Bit vs. 64 Bit
 - unterschiedliche Maschinenbefehle
 - z.B. IA32 vs. Sparc

- ★ **Teilweise lösbar**
 - ◆ Umwandlung in architekturunabhängiges Übertragungsformat und zurück
 - ◆ Laden äquivalenter Code-Module anstatt Migration von Code



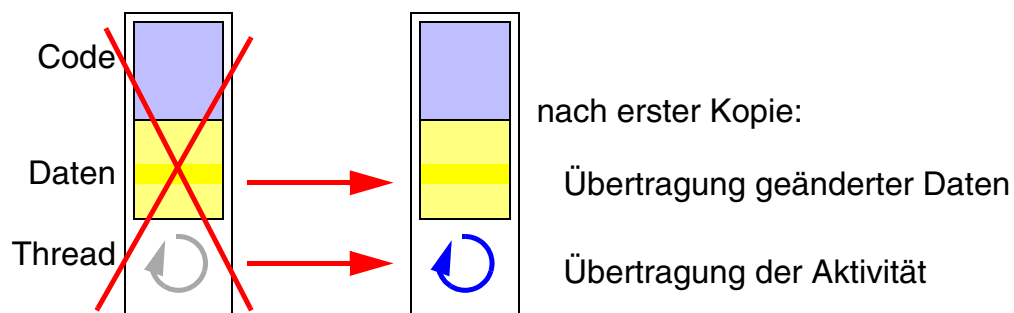
3.1 Mobile Agenten

- Mobile Objekte/Prozesse mit proaktivem Verhalten
 - ◆ selbständig und zielgerichtet tätig
 - z.B. Suche nach einem geeigneten Online-Shop
- Einsatz von Migrationstechniken
 - ◆ schwache Migration
 - Agent sichert seinen Bearbeitungszustand in Datenstrukturen
 - nach Migration wird Datenstruktur analysiert und geeignet fortgefahren
 - ◆ starke Migration
 - Agent setzt einfach Tätigkeit fort



3.2 Migration großer Komponenten

- Optimierung der Übergangszeit
 - ◆ Komponente soll bald möglichst am neuen Ort einsatzbereit sein
- Vorabübertragung
 - ◆ Kopieren der Daten und des Codes der laufenden Komponente an neuen Ort
 - ◆ Erkennung und Nachlieferung veränderter Daten
 - ◆ kurze Übergangszeit mit letzter Übertragung



3.2 Migration großer Komponenten (2)

- Nachgelagerte Übertragung der Daten
 - ◆ Vorabübertragung des Codes
 - ◆ Stoppen der Komponente an altem Ort, Starten an neuem
 - ◆ sukzessives Nachliefern der Daten
 - ◆ bei Zugriffsfehlern: sofortiges Nachliefern der Daten
 - z.B. seitenbasiert, gesteuert über Seitenfehler



4 Zusammenfassung

- Lokale Serverstrukturen
 - ◆ iterative Server
 - ◆ nebenläufige Server
 - Multiplex-Server, Mehrprozess-Server, Threaded-Server
- Verteilte Serverstrukturen
 - ◆ Lastverteilung von Serverkomponenten
 - z.B. Bearbeiterprozesse
- Lastverteilungsalgorithmen
- Migration von Software-Komponenten
 - ◆ (transparente) Verlagerung

