2 Definition und Eigenschaften Verteilter Systeme

2.1 Definition des Begriffs Verteiltes System

Definition 2.1:

Ein Verteiltes System ist eine Menge vernetzter Computer, auf denen gemeinsam eine Anwendung läuft.

Naive Definition!

Definition 2.2 (Sloman, Kramer, 1989):

Ein verteiltes Dateisystem ist eines, in dem mehrere autonome Prozessoren und Datenspeicher […] so kooperierend zusammenarbeiten, dass ein gemeinsames Ziel erreicht wird. Die Prozesse koordinieren ihre Aktivitäten und tauschen Informationen über ein Kommunikationsnetzwerk aus.

Definition 2.3 (Tanenbaum, van Steen):

Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes köhärentes System erscheint

Definition 2.4 (Coulouris, Dollimore & Kindberg)

A distributed system is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages.

Die Folgen:

- Nebenläufigkeit
- Keine globale Uhr
- Unabhängiges Versagen

Ziele verteiler Systeme:

a) historisch

- Reduzierte Kosten
- Modularität und einfachere Software
- Flexibilität und Erweiterbarkeit
- Verfügbarkeit
- Leistung
- Lokale Kontrolle

b) neu

- Neue Anwendungsmöglichkeiten, die nur als VS realisierbar sind
- File Sharing
- Chats
- Elektronische Meetings
- Mobile Anwendungen
- „Social Software“

Besondere Anforderungen:

- Heterogenität
    - Netzwerke
    - Betriebssysteme
    - Programmiersprachen
    - Hardware
    - Software-Implementierungen von verschiedenen Entwicklern
- Offenheit
    - Dynamische Erweiterung des Systems („Plug-In“) sollte möglich sein
    - Essentiell: Technische Schnittstellen müssen veröffentlicht sein
    - Prozess der Weiterentwicklung z.B. über RFC‘s
- Sicherheit
    - Vertraulichkeit
    - Datenintegrität
    - Verfügbarkeit
- Skalierung
    - System muss auch bei erheblicher Erhöhung der beteiligten Ressourcen funktional bleiben
    - Dimensionen: Anzahl der Rechner, User, Verbindungen, Dateien, …
    - Kostenkontrolle physikalischer Ressourcen
    - Kontrolle des Performance-Verlusts
    - Sicherung der Verfügbarkeit von Software-Ressourcen
    - Verhinderung von Leistungs-Bottlenecks
- Fehlerbehandlung
    - Erkennung von Fehlern
    - Maskierung von Fehlern (z.B. Doppelte Speicherung von Daten, Ignorieren und Neuanfordern bei fehlerhaften Nachrichten)
    - Fehlertoleranz
    - Erholung von Fehlern (Recovery)
- Nebenläufigkeit
    - Synchronisation verschiedener unabhängiger Prozesse, Aufgaben, Nutzer
    - Kontrolle des Zugriffs auf gemeinsam genutzte Ressourcen
- Transparenz (Unsichtbarkeit):
    - Zugangstransparenz: gleicher Zugriff auf lokale & entfernte Ressourcen
    - Ortstransparenz: Adresse von Ressourcen (physisch, Netzwerk)
    - Nebenläufigkeitstransparenz: keine sichtbare Störung durch andere Prozesse
    - Fehlertransparenz: Fehler werden verborgen, Benutzer und Anwendungen können Ziel trotzdem erreichen
    - Replikationstransparenz: Existenz mehrerer Datenkopien ist dem Benutzer/Anwendungsprogramm nicht bekannt

- Mobilitätstransparenz: Ressourcen (Daten, Programme, Rechner) können ohne Einfluss auf das System migriert werden
- Leistungstransparenz: Dynamische Rekonfiguration zur Leistungssteigerung möglich und nicht vom Benutzer sichtbar
- Skalierungstransparenz: Dynamische Größenänderung möglich und nicht vom Benutzer sichtbar

Zusammenfassend bedeutet dies für Verteilte (Rechner-)Systeme, dass alles für ein offenes System konzipiert wird, also oberste Priorität Flexibilität gepaart mit Zuverlässigkeit.

In Verteilten, eingebetteten Systemen liegt diese Priorität im Allgemeinen auf ganz anderem Gebiet: Echtzeitfähigkeit und Zuverlässigkeit des Systems (und damit des Netzwerks).

2.2 Architekturmodelle Verteilter Systeme

Definition:

Architektur eines Systems = Struktur, bestehend aus Komponenten (Funktionen) und deren Zusammenhang (Interaktionen, Abhängigkeiten)

Spezielle Fragen der Architektur eines Verteilten Systems:

- Wo sind Komponenten im Netzwerk?
- Welche Rollen und Kommunikationsmuster gibt es?
- Umgang mit Herausforderungen (aus Anwendungskontext), z.B.
  - *Ausfall eines Rechners in einem File Sharing System (Fehlerbehandlung)*
  - *Spitzen im Zugriff auf Webserver (Skalierung)*
  - *Unterstützung zukünftiger technischer Standards (Heterogenität & Offenheit)*

2.2.1 Grundlegendes Architekturmodell

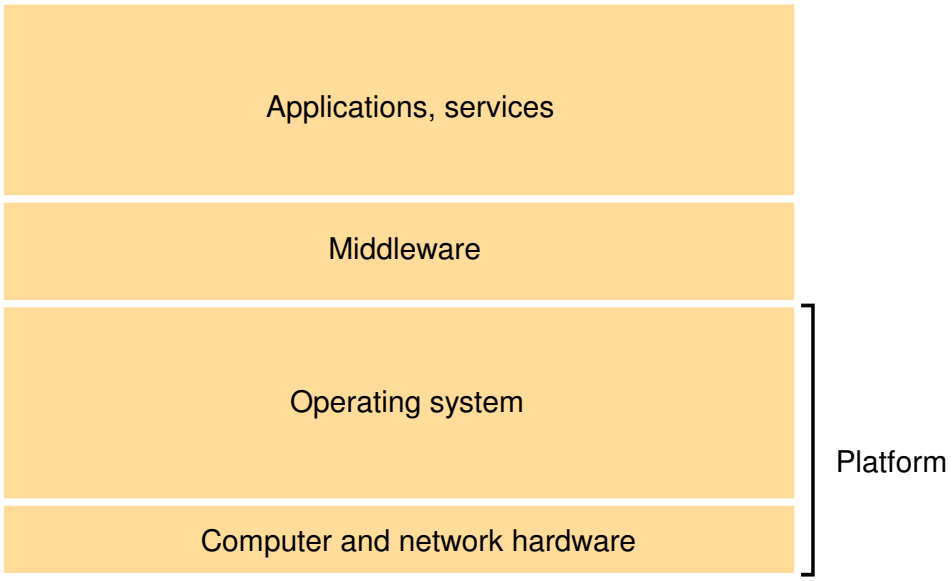

Bild 2.1 Architekturmodell Verteilter Systeme

- Middleware kann von Anwendungen verwendet werden, um Funktionen auf OS/Hardware-Ebene und Netzwerkebene zu realisieren (auch Kommunikation!)
- Middleware leistet Beitrag zur Transparenz des Systems
- Beispiele für Middleware:

- CORBA (Common Object Request Broker Architecture)
- RMI (Remote Method Invocation)
- RPC (Remote Procedure Call)
- Web Services
- DCOM (Distributed Component Object Model)

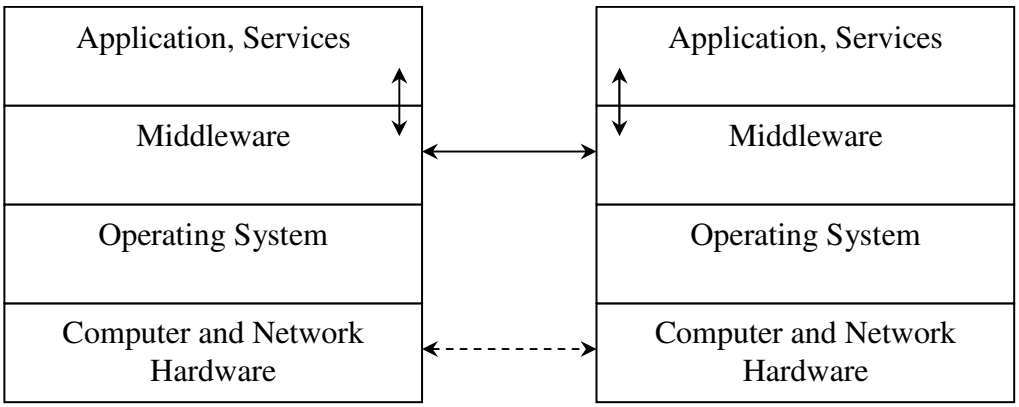

Bild 2.2 Kommunikationen in Verteilten Systemen

Typisch: Delegation der Kommunikation an die Middleware. Dies ist oft eine sinnvolle Strategie bei der Entwicklung von Verteilten Systemen

Aber: Falls Anwendungen spezifische Anforderungen haben (die nicht sinnvoll in allgemeiner Middleware realisierbar sind), findet Kommunikation auch auf Anwendungsebene statt.

Beispiele:

- Transfer sehr großer Daten über unsichere Netze
- Logisches FIFO in Verteilten Systemen

2.2.2 Verfeinerung des Architekturmodells

Zentrale Frage bei der Architektur ist: Welche Komponenten liegen wo?

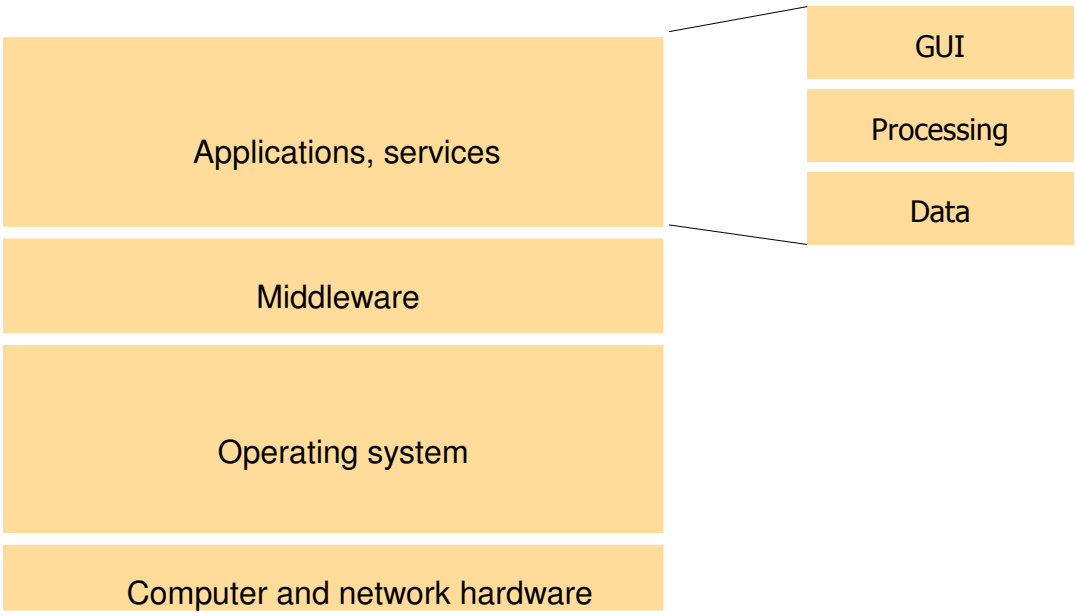

Bild 2.3 Verfeinerung Application Tier

Architekturmodell 1: Client/Server

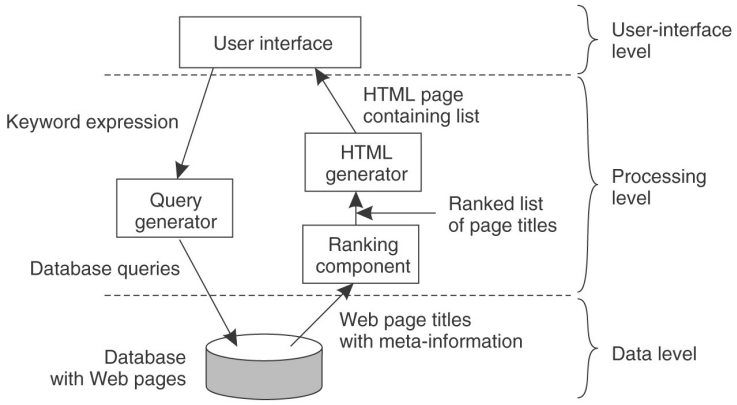

Bild 2.4 Architekturmodell Internet-Suchmaschinen

- „Klassisches“ Modell Verteilter Systeme: Client-Anwendungen stellen Anfragen an Server und erhalten Antwort
- Server fragen dazu ggf. bei anderen Servern nach.

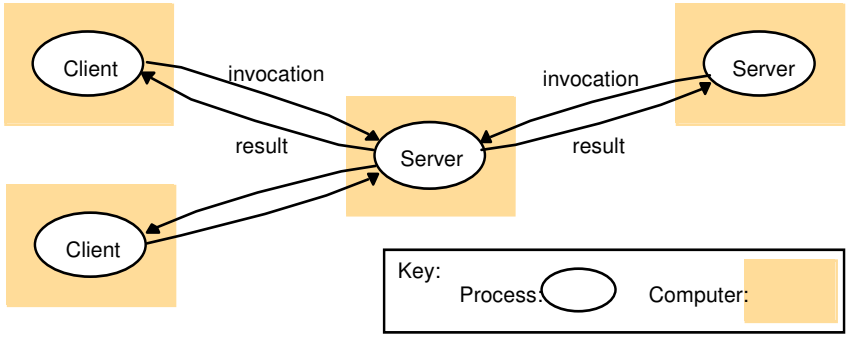

Bild 2.5 Client-Server- Modell

Beispiel: Virtual Network Computing

- 2-Tier Architektur (direkte Client-Server-Kommunikation)
- Client erhält Bildschirminhalt des Servers
- Aktionen des Clients (Mausklicks, Tastatureingaben etc) werden an Server weitergeleitet
- Ausführung der Aktionen auf Server: Änderung des Zustands der Anwendung
- „Thin Client“-Ansatz

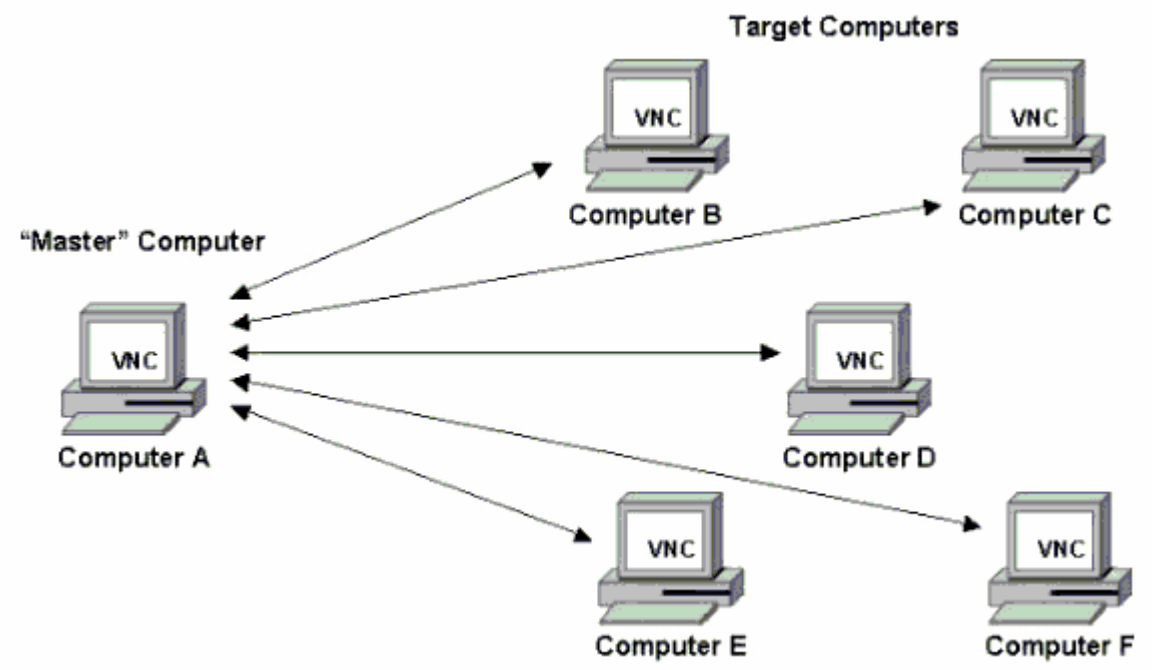

Bild 2.6 Virtual Network Computing

Beispiel 3-Tier-WEB-Architektur

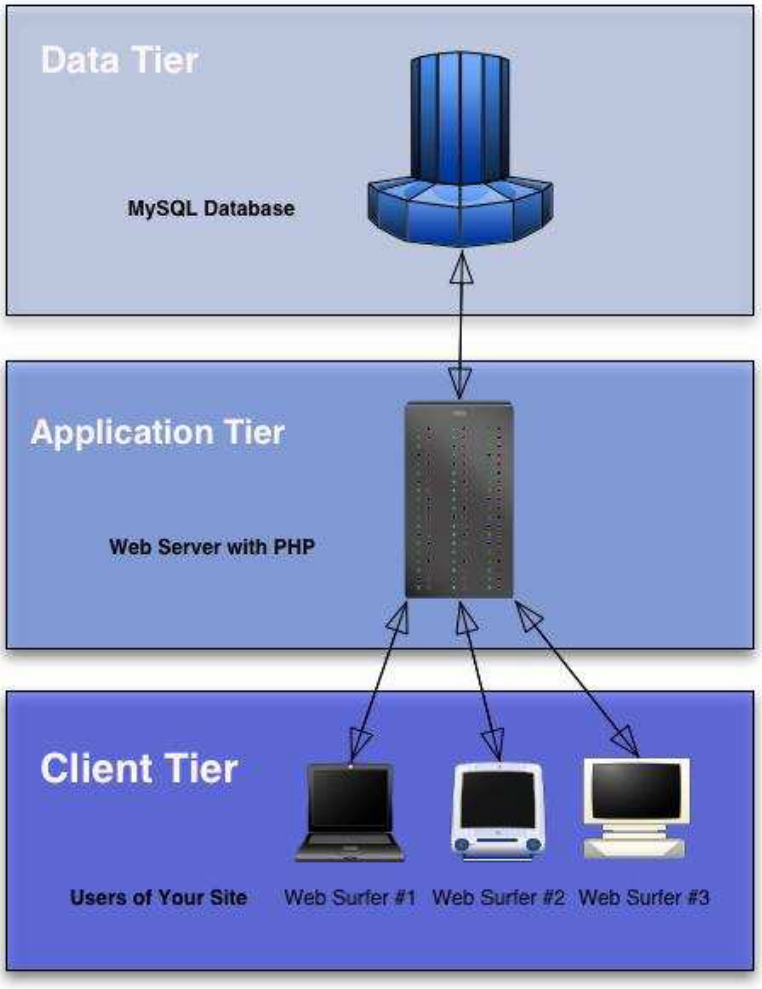

Bild 2.7 3-Tier WEB-Architektur

Beispielszenario: Eine WWW-Suchmaschine reagiert auf Anfragen von Clients und gibt auf Suchworte eine Liste von URLs zurück. Gleichzeitig hat diese Suchmaschine mehrere Crawler, die das Web nach URL‘s durchsuchen und die Datenbank aktualisieren.

Welche Anforderungen bestehen hinsichtlich der Synchronisation der Prozesse?

- Immer die auf die Anfrage passende Antwort geben, auch bei vielen nebenläufigen Anfragen
- Ergebnisse eines Crawlers sollten die eines anderen nicht überschreiben
- Während der Datenbankeintrag zu einem Schlüsselwort geändert wird, müssen Antworten zu diesem Anfragewort warten (Aktualität der Suchergebnisse: Neue Seiten in Datenbank, Löschen von Einträgen aus Datenbank)

Variation der Client-Server-Architektur

Dienste mit mehreren Servern

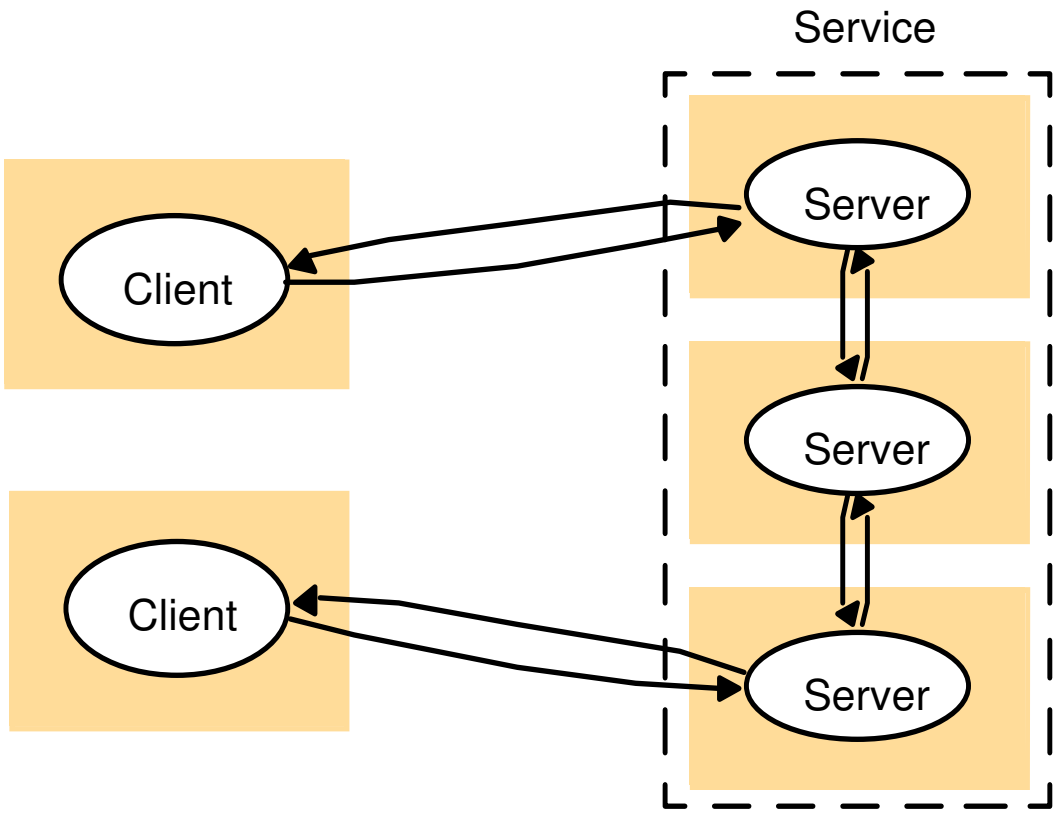

Bild 2.8 Dienste mit mehreren Servern

Proxy-Server (von *proximus*, lat., der Nächste):

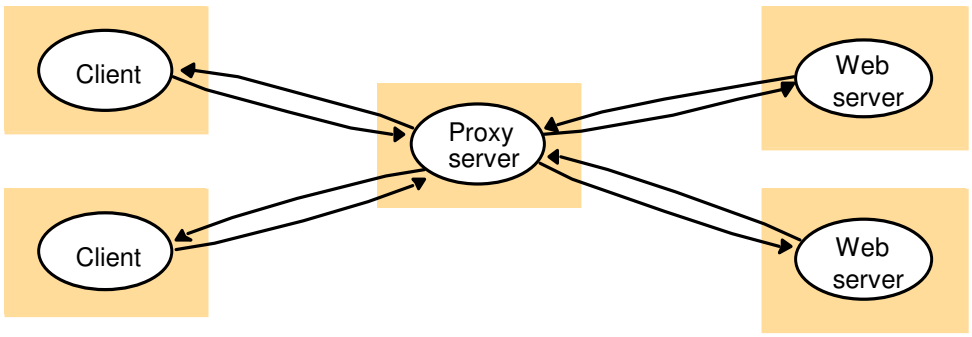

Bild 2.9 Proxy Server

- Caching von Daten (Zwischenspeicherung), z.B. zur Verbesserung der Antwortzeit
- Firewall zur Kontrolle des Zugriffs auf Ressourcen

Architekturmodell 2: Peer-to-Peer-Architektur

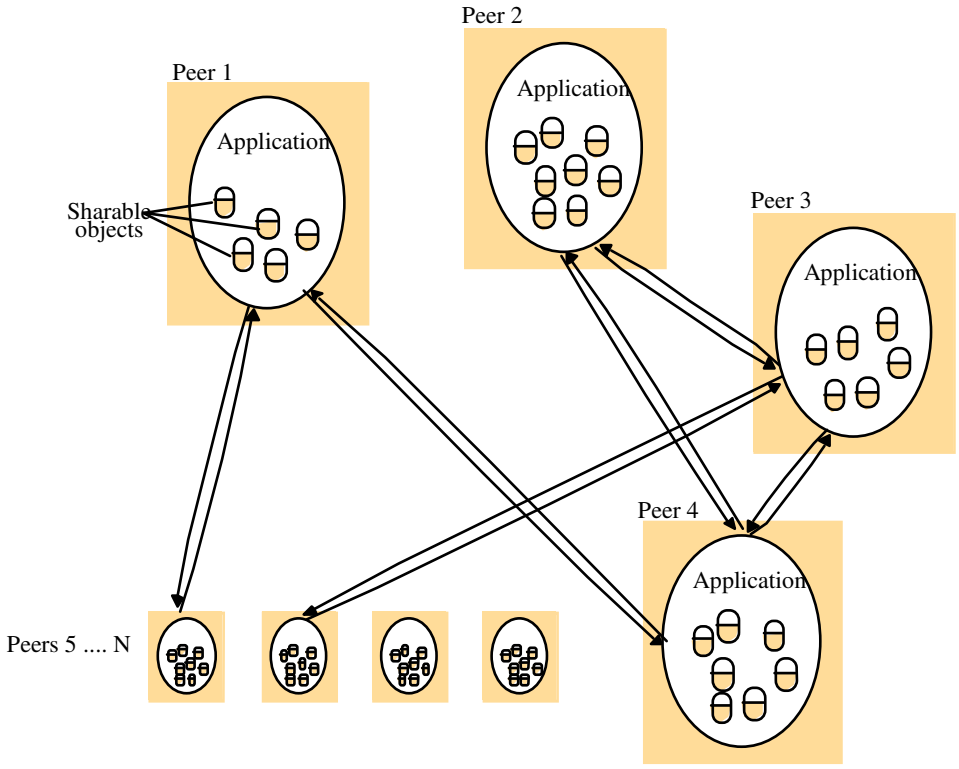

Bild 2.10 Peer-to-Peer-Architektur

Das Wort *Peer* bedeutet Gleichgestellter oder Ebenbürtiger. Jeder Peer ist gleichzeitig Server und Client!

Jeder Peer hat (evtl. verschiedene) Daten und (den gleichen) Programmcode, insbesondere zur Koordination.

Allgemeine Vor- und Nachteile von P2P-Architekturen:

+ Keine Bottlenecks
+ Robustheit
+ Flexible Verteilung der Belastung
– Schwierigkeit, globale Information zu erhalten
– Kontrolle & Administration

Vergleich zwischen C/S- und P2P-Architektur

Ein Programm zum Filetransfer soll entworfen werden. Als Architektur kommt entweder ein Client-Server-Modell oder P2P in Frage. Was sind die Vor- und Nachteile dieser beiden Wahlmöglichkeiten?

C/S Vorteile:

- Zentraler Server: es ist sehr einfach, Daten zu indexieren.
- Absolute Kontrolle über die Daten: Verfügbarkeit und Zugang durch denjenigen der den Server kontrolliert.

C/S Nachteile:

- Zentraler Server kann schnell ein Bottleneck bezüglich Bandbreite werden

- Dateien müssen zu jedem Client vollständig übertragen werden, das bedeutet hohe Kosten wegen des erzeugten Netzwerkverkehrs
- Zentraler Server kann ein „Single Point of Failure“ sein: wenn nur dieser ausfällt ist ein Datenaustausch unmöglich

P2P Vorteile:

- Kein zentraler Server: das ganze Netz ist schwerer angreif- und abschaltbar (z.B. um staatlicher Zensur zu entgehen). Viele Knoten können ausfallen und doch wird das Netzwerk noch funktionstüchtig bleiben
- „Verteilte“ Übertragung der Dateien: man ist nicht auf die maximale Übertragungsgeschwindigkeit eines Knotens angewiesen, sondern kann – bei entsprechender Verbreitung der Datei im Netzwerk – von beliebig vielen Knoten gleichzeitig Teile der Datei herunterladen, bis die eigene maximale Bandbreite ausgereizt ist.
- Wenn eine Datei einmal im Netzwerk ist, übernehmen die Nutzer die weitere Verbreitung. Der eigentliche Einsteller erreicht somit ein sehr weites Publikum unter Aufwendung sehr weniger Ressourcen. Beispiel: eine Firma die ein Update über Bittorrent Technologie verbreitet benötigt dazu wesentlich weniger Bandbreite als wenn sie jedem Client einzeln das komplette Update überträgt.

P2P Nachteile:

- Kein zentraler Server – Clients müssen sich irgendwie finden, es muss eine geeignete Lösung gefunden werden die anderen Clients und Dateien zu lokalisieren (Name-Service Problem)
- Man hat quasi keine Kontrolle über die Daten im Netz, Dateien wieder aus dem Netz zu entfernen ist schwierig. Auch können über meinen Rechner/Knoten Informationen, Dateien oder Teile davon, von anderen Nutzern übertragen werden welches ethische, moralische oder auch rechtliche Probleme darstellen kann.