Verteilte objektorientierte BS: Clouds

Überblick

- ❑ verteilte Betriebssysteme
- ❑ Clouds-Architektur
- ❑ Clouds-Objekte
- ❑ Threads
- ❑ Programmierung
- ❑ Minimalkern Ra & Betriebssystemobjekte
- ❑ Distributed Shared Memory
- ❑ Transaktionskonzepte

Literatur

CP89 R.C. Chen and P. Dasgupta, "Linkinging Consistency with Object/Thread Semantics: An Approach to Robust Computation", *Proceedings of the Ninth International Conference on Distributed Computing Systems*, June 1989.

DCM+90 P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J. M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, C. J. Wileknloh, "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems Journal*, Vol 3, USENIX, Winter 1990.

DLA+91 P. Dasgupta, R. LeBlanc Jr., M. Ahamad, U. Ramachandran, "The Clouds Distributed Operating System", To appear in *IEEE computer*.

Handapparat: Ordner Clouds

verteilte Betriebssysteme

❑ Betriebssystem für verteiltes System

- Heutzutage sind viele Betriebssysteme mit Mechanismen ausgestattet, die einen Einsatz zur Nutzung eines verteilten Systems ermöglichen:
  - Netzwerkprotokolle (TCP/IP, ISO-Protokolle)
  - *Remote-Procedur-Call*-Implementierungen (z. B. Sun RPC)
  - netzwerkweite Dateisysteme (NFS, RFS, AFS)
- In aller Regel präsentieren alle diese Betriebssysteme dem Anwender aber nach wie vor die Sicht auf einzelne Rechner, die miteinander kommunizieren:
  - Lediglich im Bereich der Dateisysteme ist es durch entsprechende Administration möglich, die tatsächliche Verteilung der Hintergrundspeicher im Netzwerk für den Anwender weitgehend transparent zu halten.
  - Für die Ausführung von Programmen bestimmt der Anwender normalerweise aber den Rechner explizit selbst — transparente Prozeßverlagerungen, z. B. zur Lastverteilung, sind sehr selten möglich.

❑ verteiltes Betriebssystem

ein verteiltes Betriebssystem verwaltet das verteilte System als eine Einheit — die vorhandenen Ressourcen werden gemeinsam verwaltet und nach Bedarf (und entsprechenden Strategien) auf die Anwender verteilt.

- compute server
- data server
- user workstations

bilden logische Knoten in dem verteilten System. Ein realer Rechner kann dabei gleichzeitig mehrere Aufgaben wahrnehmen — ein Knoten, der sowohl über eine leistungsfähige CPU als auch über Hintergrundspeicher verfügt, kann dabei beispielsweise zur Implementierung eines *Compute Servers* und eines *Data Servers* genutzt werden.

➔ Benutzer sieht ein virtuelles System

– Probleme im Umgang mit verteiltem System werden vermieden

Der Benutzer sieht ein zentrales System, das ihm eine Menge von Ressourcen anbietet — er muß nicht selbst explizit entscheiden, welche Knoten des verteilten Systems für seine Anwendung genutzt werden.

– Vorteile von Verteilung werden erhalten

Es gibt Anwendungen, die von einem Wissen über das verteilte System, in dem sie bearbeitet werden und seinen besonderen Eigenschaften, profitieren können (z. B. können im Gegensatz zu einem zentralen Mehrrechnersystem einzelne Knoten unabhängig voneinander ausfallen — für Fehlertoleranzmaßnahmen im Rahmen einer Anwendung könnten Informationen über die Systemkonfiguration dabei genutzt werden).

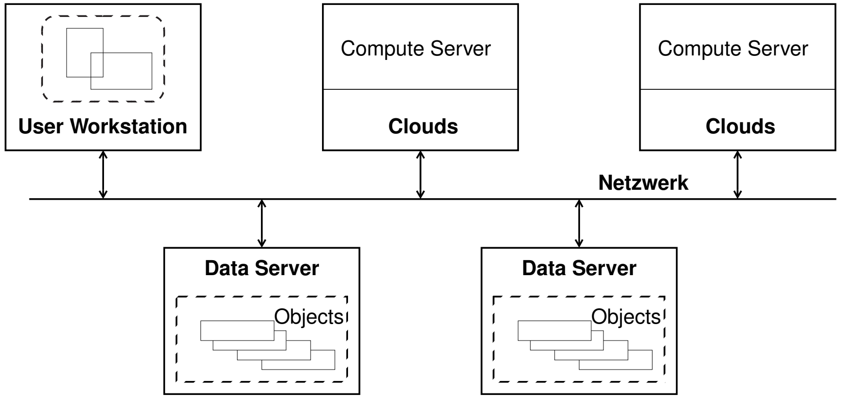
- Strukturierung solcher Systeme
  - Message-basiert

    Prozesse auf den Knoten des verteilten Systems kommunizieren über Mitteilungen durch das Netzwerk.
  - Objekt-basiert

Clouds-Architektur

❑ Integration einer Menge homogener Rechner zu einer Umgebung, die sich als ein großer Rechner präsentiert

nach Tanenbaum: "virtual uniprocessor"



- Die dargestellten Einheiten des Systems sind logische Einheiten — müssen also nicht 1:1 mit den physisch vorhandenen Knoten identisch sein!
- *Compute Server* werden zur Bearbeitung von Aktivitätsträgern eingesetzt. Sie verfügen lediglich über Hauptspeicher, nicht jedoch über Hintergrundspeicher.
- Hintergrundspeicher wird von *Data Servern* zur Verfügung gestellt.
- *User Workstations* unterstützen hochqualitative Grafik- und andere Schnittstellen.

Objekte

❑ Unterscheidung System-Objekte ⇔ Sprach-Objekte

- Normalerweise werden Objekte in der Implementierung als Speicherbereiche in einem virtuellen Adreßraum dargestellt.
- Die Objekte sind lediglich auf der Ebene der Programmiersprache zu sehen - für das Betriebssystem sind einzelne Objekte nicht erkennbar.

❑ System-Objekte (Clouds-Objekte)

- Clouds stellt Programmierumgebungen bereit, in denen Objekte formulierbar sind, die auch zur Laufzeit für das Betriebssystem Verwaltungseinheiten darstellen.

– grob-granular

- Ein Betriebssystem kann wirtschaftlich nur mit größeren Datenblöcken agieren (Hauptspeicherbereiche/Kacheln, Disk-E/A).
- Objekte, die dem Betriebssystem zur Verwaltung übergeben werden, müssen daher zwangsläufig grob granular sein - d. h. ein Objekt liegt mindestens in der Größenordnung von Kilobytes, kann aber auch mehrere Megabytes groß werden.
- Jedes Clouds-Objekt bildet einen eigenen virtuellen Adreßraum.

– passiv

- Clouds behandelt das Konzept des Aktivitätsträgers getrennt, Objekte sind daher grundsätzlich von sich aus passiv.

– persistent

- Clouds-Objekte existieren ab ihrer Erzeugung, bis sie explizit gelöscht werden (vergleichbar mit Dateien). Sie werden also auch nicht automatisch am Ende der Anwendungsausführung, in deren Zusammenhang sie erzeugt wurden, wieder vernichtet.

– mobil

- Clouds-Objekte können als virtuelle Adreßräume auf verschiedene Knoten des verteilten Systems abgebildet werden — prinzipiell sogar gleichzeitig (siehe *Distributed Shared Memory*).

❑ Sprach-Objekte (z. B. C++ oder Eiffel-Objekte)

– fein-granular

- Da Sprach-Objekte nur im Programmiermodell existieren und nur vom Compiler zu behandeln sind, können sie — wie in C++ oder Eiffel üblich — beliebig klein (auch nur ein Integer) sein.

– Bestandteil von Clouds-Objekten

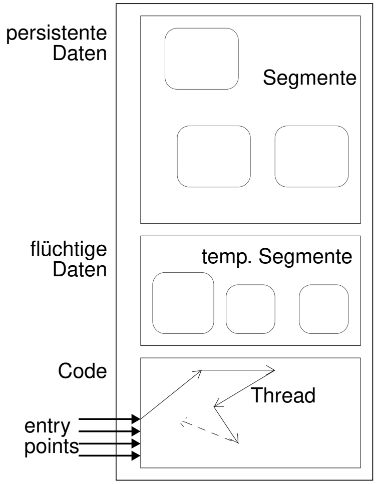
– fest an Clouds-Objekt gebunden

- Der Compiler bindet die Sprachobjekte, die von einem Clouds-Objekt benutzt werden, statisch zusammen — das Ergebnis bildet Code- und Datensegmente des Clouds-Objekts.

Clouds-Objekt

❑ Abstraktion für Speicher

- Clouds-Objekte stellen die Basis-Einheiten der Clouds-Speicherverwaltung dar (sowohl für die Externspeicherverwaltung, als auch für die Hauptspeicherverwaltung).
- Clouds-Objekte werden auf *Data-Servern* instantiiert und dort gespeichert. Zur Laufzeit werden einzelne Seiten nach Bedarf in den Hauptspeicher des *Compute-Servers* übertragen und dort temporär zwischengespeichert



❑ persistenter (nicht-flüchtiger) virtueller Adreßraum

- Während virtuelle Adreßräume normalerweise flüchtig sind — Rechnerausfälle also nicht überleben —, sind Clouds-Objekte persistent.

❑ kapselt Menge von Sprach-Objekten

❑ für Betriebssystem sichtbar

❑ global eindeutig identifizierbar

- Jedes Clouds-Objekt hat einen global eindeutigen Namen (*sysname*).
- Benutzer können eigene, symbolische Namen zusätzlich vergeben, die über Nameserver-Dienste in den *sysname* umgewandelt werden.

❑ im verteilten System verlagerbar

- Da Clouds-Objekte auf *Compute-Servern* ohnehin nur temporär zwischengespeichert werden, Daten und Code aber eigentlich auf *Data-Servern* liegen, ist es grundsätzlich möglich, ein Clouds-Objekt auf einem anderen Knoten zwischenzuspeichern und die Berechnung dann dort durchzuführen.

❑ kann auf verschiedenen Knoten "gleichzeitig" vorhanden sein

- Das *Distributed-Shared-Memory*-Konzept von Clouds erlaubt auch eine gleichzeitige Zwischenspeicherung von Objekten auf mehreren *Compute-Servern*. Software-Cache-Kohärenz-Protokolle sorgen dafür, daß ein einheitlicher Objektzustand erhalten bleibt.

❑ Interaktion teuer

- Da jedes Clouds-Objekt einen eigenen virtuellen Adreßraum bildet, muß bei jeder Interaktion zwischen Clouds-Objekten ein Adreßraum-Wechsel vorgenommen werden.
- Die hohen Kosten für eine Interaktion zwischen Clouds-Objekten sind ein weiterer Grund, daß aus Effizienzgründen nur grob-granulare Clouds-Objekte vertretbar sind, in denen ein Großteil der anwendungsinternen Kommunikation lokal (objektintern, zwischen den Sprachobjekten) abläuft.

Threads

❑ Aktivitätsträger zur Ausführung von Objekt-Methoden

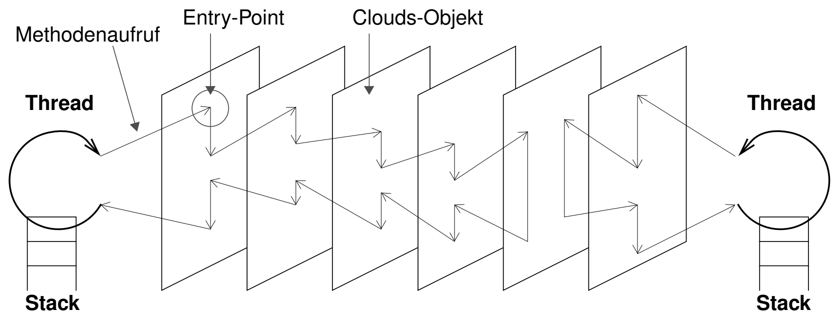
- Threads werden entweder interaktiv durch den Benutzer erzeugt (Kommand-Eingabe) — dies entspricht dem Start einer Anwendungsausführung — oder durch andere Objekte bei deren Ausführung gestartet.

❑ wandert bei Methodenaufrufen zwischen Clouds-Objekten

- Ein Thread ist nicht an einen virtuellen Adreßraum gebunden (im Gegensatz zu Prozessen in vielen traditionellen oder auch moderneren Betriebssystemen wie UNIX oder MACH).
- Der Thread verläßt bei einem Methodenaufruf an ein anderes Clouds-Objekt den aktuellen virt. Adreßraum und kehrt am Ende der Methodenbearbeitung zurück.
- Ein Thread terminiert, wenn die Methode, zu deren Bearbeitung er erzeugt wurde, beendet ist.

❑ nebenläufig

- Mehrere Aktivitätsträger können gleichzeitig Code in einem Clouds-Objekt ausführen.
- Nebenläufigkeit muß durch den Programmierer koordiniert werden. Das Clouds-System bietet dazu Koordinierungsmechanismen wie Locks oder Semaphore an.



BS-Besonderheiten

Neben einer strengen Trennung der Konzepte für Aktivitätsträger und virtuelle Adreßräume unterscheidet sich das Clouds-Betriebssystem auch in Bezug auf weitere, allgemein übliche Betriebssystemmechanismen sehr von herkömmlichen Betriebssystemen:

❏ Interprozeßkommuniktion
- einheitliches Konzept: Methodenaufrufe
  - Methodenaufrufe können verwendet werden bei:
    - Clouds-Objekt-lokaler Objektinteraktion (zwischen Sprachobjekten)
    - Interaktion zwischen verschiedenen Clouds-Objekten einer Anwendung (eine Anwendung kann damit auch auf mehrere virtuelle Adreßräume aufgeteilt sein — Schutz zwischen Komponenten einer Anwendung möglich!)
    - Interaktion zwischen verschiedenen Anwendungen (= verschiedene Clouds-Objekte)

❏ Dateisystem
- entfällt
- persistente Objekte bilden mächtigeren Mechanismus
- Datei durch Clouds-Objekt mit Methoden read und write realisierbar
  - Da Objektzugriffe ortstransparent in dem verteilten System möglich sind, entfällt gleichzeitig auch ein Bedarf für *Network-File-System*-Konzepte oder ähnliches.

❏ Message-Passing
- entfällt
- Messages durch Puffer-Objekte implementierbar
  - Auch Message-Kommunikation zwischen Anwendungen auf verschiedenen Knoten im verteilten System ist durch Pufferobjekte, auf die ortstransparent im verteilten System zugegriffen werden kann, realisierbar.
  - Da Objekte im verteilten System auch migrieren können — und sich damit zu ihrem Kommunikationspartner hin bewegen können, kann intensive Kommunikation über Knotengrenzen leichter vermieden werden.

Programmierung & Ausführung

❏ Klassen
  - Schablone für die Erzeugung von Instanzen
  - vergleichbar mit einem übersetzten Programm (oder einer .o-Datei) unter UNIX.

❏ Methoden
  - *Entry-points* der Clouds-Objekte
  - Funktionen, die von anderen Clouds-Objekten aus aufrufbar sind
  - Methoden von Sprach-Objekten sind aus anderen Clouds-Objekten nicht ansprechbar

❏ Instanzen
- Instantiierung über Kommando
- Instantiierung durch Objekte
- Benutzerspezifizierbare Namen für Instanzen

❏ Nameserver
  - bilden Objektnamen des Anwenders in den systemweit eindeutigen *sysname* ab
  - Vergleich mit Spring:
    - Nameserver = Context
    - *sysname* = Objektreferenz
    - benutzerspezifische Objektname = Namenseintrag in Context

❏ Statische Typprüfung
  - Clouds-Objekte und deren Methoden (*entry-points*) werden bei der Programmierung mit einem Typ gekennzeichnet.
  - Typprüfung erfolgt nur statisch zur Übersetzungszeit.
  - Zur Laufzeit können Objekt-Referenzen an bereits existierende, persistente Clouds-Objekte (über deren Namen) gebunden werden — hierbei erfolgt keine dynamische Typprüfung. Dadurch sind Typ-Zugriffsfehler möglich.

Programmbeispiel

```
clouds_class rectangle;
int x, y;              // persistente Daten von rectangle
entry rectangle;       // Konstruktor
entry size (int x, y);// Größe des Rechtecks setzen
entry int area();      // Fläche des Rechtecks zurückgeben
end_class
```

```
rectangle_ref rect;    // rectangle_ref ist eine Klasse, die auf
                       // ein Objekt des Typs rectangle zeigt
String s;
s.strinit("R1");       // String-Objekt für Objekt-Namen
rect.Create(s, ...);   // Instantiierung (... = Konstruktor-Param.)
```

```
rectangle_ref rect;    // Referenz auf rectangle anlegen
rect.bind("R1");       // Nameserver-Aufruf: Objekt R1
                       // wird an rect gebunden

rect.size(5, 10);      // Größe auf 5 x 10 setzen
printf("%d\n", rect.area() );// Fläche (50) ausgeben
```

- Zuerst wird die Clouds-Klasse **`rectangle`** definiert.
- Nachdem eine Klasse übersetzt wurde, kann sie instantiiert werden.
- Der Compiler erzeugt automatisch zu jeder Clouds-Klasse eine zusätzliche **`_ref`**-Klasse. Ein Objekt der Klasse **`rectangle_ref`** (= eine Referenz auf ein **`rectangle`**-Objekt) besitzt alle Methoden der Clouds-Klasse **`rectangle`** und zusätzlich Methoden zur Instantiierung neuer **`rectangle`**-Objekte, zum Binden der Referenz an bereits exitierende, persistente **`rectangle`**-Objekte, zur Erzeugung neuer Threads, die eine Methode eines **`rectangle`**-Objekts ausführen sollen, etc.
- Das zweite Code-Beispiel zeigt eine Instantiierung aus einem CC++-Programm heraus. Eine Instanz der Klasse **`rectangle`** mit dem Namen **`Rect01`** wird angelegt. Da keine Zeiger als Parameter übergeben werden können, wird die Klasse **`String`** verwendet, um den Instanz-Namen zu übergeben.
- Das dritte Code-Beispiel zeigt, wie man aus einem CC++-Programm heraus Zugriff auf die bereits existierende Instanz **`Rect01`** bekommt. Auch hierfür wird wieder ein Referenz-Objekt der Klasse **`rectangle_ref`** verwendet.

Minimalkern Ra

❑ unterste Betriebssystemebene

❑ unterstützt Basismechanismen für
  – virtuelle Speicherverwaltung
  – einfaches Scheduling

Abstraktionen

❑ Segmente
- Bytesequenz auf Disk oder im Hauptspeicher
- Systemweit eindeutiger Name für jedes Segment
- Existieren nach ihrer Erzeugung, bis sie explizit zerstört werden

❑ virtuelle Adreßräume
- Bereiche von virtuellen Adressen
- jeder Adreßbereich ist auf ein Segment (oder einen Teil eines Segments) abgebildet

❑ IsiBas (leichtgewichtige Prozesse)
- leichtgewichtiger, adreßraum-gebundener Prozeß
- verbunden mit leichtgewichtigem Kontext (Registersatz, Stack)
- Clouds-Prozeß = IsiBa + Anwendungs-Stack + virtueller Adreßraum
- Clouds-Thread wird durch einen oder mehrere Clouds-Prozesse implementiert (wenn ein Thread in ein anderes Clouds-Objekt wandert, wird die Kontrolle an einen anderen, dort "ansässigen" Clouds-Prozeß weitergegeben, der aufrufende Clouds-Prozeß wird bis zum return blockiert).
- IsiBas werden innerhalb des Systems außerdem für weitere Dienste wie Interrupt-Bearbeitung eingesetzt.

❑ Partitionen
- Realisiert nicht-flüchtigen Speicher für ein Segment.
- Einem *Compute-Server* können ein oder mehrere Partitionen zugeteilt sein, jedes Segment liegt aber nur in genau einer Partition.
- Eine Partition kommuniziert mit dem *Data-Server*, auf dem das Segment abgelegt ist, um die Segment-Daten ein- oder auszulagern (seitenweise).
- Ra definiert lediglich die Schnittstelle zu Partitionen, Partitionen selbst sind als Systemobjekte realisiert.

Betriebssystem-Objekte

❑ Module zur Realisierung der Clouds-Betriebssystemfunktionen
- Die eigentlichen Clouds-Betriebssystemfunktionen sind nicht als Betsandteil des Ra-Kerns implementiert, da Ra lediglich minimale Funktionalität enthalten soll.

❑ bauen auf den Abstraktionen von Ra auf
- Ra implementiert Basis-Abstraktionen, auf denen — objektorientiert konstruierte — Moduln (sog. System-Objekte) aufbauen, die die Abstraktionen des Clouds-Systems implementieren.

❑ konzeptionell wie Clouds-Objekte aufgebaut
- Die System-Objekte sind ähnlich wie Clouds-Objekte realisiert, die jeweils in eigenen Adreßräumen plaziert werden könnten, Methoden anbieten und Zugriff auf die Funktionen des Ra-Kerns haben.

❑ aus Effizienzgründen zusammen mit Ra zu einem Kern zusammengebunden
- Um innerhalb des Betriebssystems teuere Adreßraumwechsel zu vermeiden, werden die Clouds-Systemobjekte jedoch — im Gegensatz zu Clouds-Anwendungsobjekten — zu Übersetzungszeit statisch zusammengebunden und auch mit dem Ra-Kern statisch gebunden. Ra-Kern und System-Objekte werden dann beim Booten in einen priviligierten Adreßraum geladen.
- Die Methoden von System-Objekten sind dadurch auch nicht direkt durch Anwendungsobjekte aufrufbar, sondern werden durch eine spezielle Systemcall-Schnittstelle vermittelt (herkömmliche Methode über trap-Befehl).

❑ System-Objekte
- Thread Manager
  - implementiert Clouds-Threads mit Hilfe von Ra-Isibas
- User Object Manager
  - realisiert die Erzeugung und das Löschen von Clouds-Objekten und unterstützt Methodenaufrufe zwischen Clouds-Objekten
- DSM Clients & Server
  - implementieren zusammen die Funktionalität zur Koordinierung von *distributed-shared-memory* (siehe Abschnitt DSM)
- User I/O Manager
  - realisiert die Ein-/Ausgabe auf *user-workstations*
- Networking & RaTP
  - Clouds-Netzwerk-Protokoll, aufbauend auf den Funktionen des Ethernet-Treibers.

verteilte Anwendungen

❑ Anwendungsausführung: Compute-Server

❑ Objektspeicherung: Data-Server

➔ Transfer von Objekten zwischen Data-Server und Compute-Server
- Der Transfer von Code und Daten zwischen Compute-Servern und Data-Servern erfolgt durch Paging. Da Data-Server und Compute-Server in Clouds generell voneinander ortsunabhängig sind, erfolgt Paging damit grundsätzlich netzwerk-transparent:
  - ➔ es ist egal, auf welchem Compute-Server ein Clouds-Objekt ausgeführt werden soll!

❑ Clouds-Objekte einer Anwendung auf verschiedenen Compute-Servern
- Für die Kommunikation und Synchronisation/Koordinierung zwischen Anwendungen bzw. zwischen verschiedenen Komponenten einer Anwendung — hier Clouds-Objekte — gibt es zwei grundlegende (prinzipiell gleichwertige, weil ineinander überführbare) Paradigmen:
  - message-passing
  - shared memory
- In der Praxis wird heutzutage shared memory in erster Linie bei Mehrprozessor-Systemen mit gemeinsamem Hauptspeicher (eng gekoppelte Systeme) eingesetzt, während bei lose gekoppelten Systemen mit verteiltem Speicher meist message passing verwendet wird. Bei dieser Vorgehensweise werden die Eigenschaften der Hardwarearchitektur auf das Programmiermodell übertragen.
- Nachteil dieser Vorgehensweise ist, daß Anwendungen zwischen eng und lose gekoppelten Systemen nicht übertragbar sind.

➔ Methodenaufruf:
- Zielobjekt wird zum Aufrufer migriert
  - Da das Paging ohnehin knoten-unabhängig ist, ist es grundsätzlich kein Problem, die Seiten des aufgerufenen Objekts auf einem Knoten freizugeben und auf dem Knoten des Aufrufers einzupagen. Der aufrufende Thread muß dadurch nicht zu einem anderen Knoten wandern.
- Aufruf wird als RPC implementiert
  - Herkömmliche Implementierung eines Remote-Procedure-Calls — die Aufrufparameter werden verpackt, zum Knoten, auf dem das aufgerufene Objekt liegt, gesendet und das Ergebnis wird am Ende der Methodenbearbeitung zurückgeschickt.

❑ Clouds-Objekte auf verschiedenen Knoten rufen gleichzeitig Methoden eines Objekts auf
- Zielobjekt wird auf beiden Compute-Servern gleichzeitig gelagert
  - ➔ Distributed Shared Memory
  - Die Seiten des Zielobjekts werden zusätzlich zu seinem aktuellen Aufenthaltsort (Compute-Server) auf dem Knoten des Aufrufers eingepaged — es existieren damit zwei Kopien der Objektdaten, so daß für deren Konsistenz zu sorgen ist!

Distributed Shared Memory

- ❏ Präsentation von verteilten, lokalen Speichern als gemeinsamer, globaler Speicher
- ❏ Probleme:
  - Konsistenz
    - Eine bestimmte Invariante ist immer erfüllt
  - Kohärenz
    - *A memory is coherent if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address*
- ❏ Multi-Prozessor Cache-Kohärenz
  - In *Shared-memory*-Multiprozessoren sind Cache-Kohärenz-Protokolle im allgemeinen vorzufinden. Es wäre naheliegend, die dort realisierten Protokolle (Write-Through, Copy-Back, ...) auch für Distributed-shared-memory zu verwenden.
  - Problem: Multiprozessorsysteme besitzen normalerweise Hardware-/Firmware-Unterstützung für die Cache-Kohärenz:
    - – Maschinenbefehle, die Atomarität gewährleisten
    - – spezielle Signale auf dem Systembus zur Koordinierung der Caches
  - Ein Nachbau entsprechender Mechanismen in Software für ein verteiltes System wäre teuer und könnte damit nur schwer eine akzeptable Performance erreichen.
- ❏ Nutzung von Anwendungsspezifischen Informationen
  - Ideal wäre es, wenn man von der Anwendung Informationen erhalten könnte, die eine Optimierung der Cache-Kohärenz-Sicherung erlauben. Z. B.:
    - – welche Methoden modifizieren den Objektzustand
    - – welchen Größenumfang haben die Modifikationen

Distributed Shared Memory (2)

- ❏ Kohärenz-Semantiken
  - One-Copy-Semantik
  - schwächere Formen
- ❏ Protokolle zur Kohärenzerhaltung
  - write-broadcast
    - Geänderte Daten werden an alle anderen Caches übertragen.
  - write-invalidate
    - Bei einer Änderung werden die anderen Caches informiert. Die entsprechenden Daten werden dort daraufhin für ungültig erklärt und müssen beim nächsten Zugriff neu geladen werden.
  - lock-based
- ❏ unterschiedliche Zugriffsmodi
  - – read/write
    - *exclusive lock*
  - – read-only
    - *non-exclusive lock*
  - – weak-read
  - – none
    - Man erhält exklusiven Zugriff auf ein Segment, jedoch ohne *lock*. Das Segment kann einem jederzeit wieder entzogen werden.

Nebenläufigkeit

Die höhere Gesamtleistung eines verteilten Systems oder Mehrprozessorsystems läßt sich problemlos nutzen, wenn verschiedene, voneinander unabhängige Anwendungen auf die vorhandenen Prozessoren verteilt werden können. Eine Anwendung alleine kann jedoch nur profitieren, wenn sie für eine parallele Abarbeitung programmiert wurde.

Zur Erzeugung paralleler Abläufe gibt es im wesentlichen zwei Paradigmen:

- explizite Erzeugung neuer Aktivitätsträger (vergleichbar mit einem fork() in UNIX)
- asynchrone Ausführung von Funktionen bzw. — im objektorientierten Programmiermodell — von Methoden.

Asynchrone Methodenaufrufe

❑ Synchroner Methodenaufruf

– Thread wandert

- Bei einem synchronen Methodenaufruf wandert der Thread aus dem aufrufenden in das aufgerufene Objekt, führt dort die aufgerufene Methode aus (wobei er ggf. in weitere Objekte wandert und wieder zurückkehrt) und kehrt am Ende der Methodenbearbeitung in das aufrufende Objekt zurück. In der Zwischenzeit ist das aufrufende Objekt passiv (da ja kein Aktivitätsträger in ihm vorhanden ist).
- In der Praxis werden normalerweise nur synchrone Methodenaufrufe innerhalb eines virtuellen Adreßraums wirklich durch "Wandern" des Aktivitätsträgers (PC umsetzen, neuen Stack-Frame aufbauen, am Ende zur Rücksprungadresse zurückkehren) implementiert. Methodenaufrufe an andere virtuelle Adreßräume oder sogar an andere Knoten des verteilten Systems werden dagegen durch Beauftragung eines Prozesses in der Zielumgebung realisiert, wobei der aufrufende Prozeß (in Clouds wäre das ein Ra-Isiba) in der Zwischenzeit blockiert wird.

❑ Asynchroner Methodenaufruf

– zur Methodenausführung wird neuer Thread erzeugt

– aufrufender Thread kann weiterarbeiten

- Die Erzeugung eines neuen Aktivitätsträgers ist bei dieser Vorgehensweise mit dem Konstrukt des Methodenaufrufs kombiniert. Der Programmierer muß dadurch nicht explizit einen neuen Aktivitätsträger generieren, anschließend festlegen, welche Methode dieser ausführen soll und schließlich irgendwann einmal explizit auf das Ende des Aktivitätsträgers warten, sondern er kann bei einem Methodenaufruf durch einfache syntaktische Änderungen gegenüber einem synchronen Aufruf angeben, daß die Methode parallel zum weiteren Ablauf des Aufrufers abzuarbeiten ist.
- Während die explizite Erzeugung neuer Aktivitätsträger durch Systemaufrufe realisierbar und damit orthogonal zum Programmiermodell ist, müssen asynchrone Methodenaufrufe in der Programmiersprache verankert werden. In Clouds-CC++ wird dies beispielsweise in der Form `Zielobjekt!methode(…)`, statt `Zielobjekt.methode(…)` für synchrone Aufrufe ausgedrückt.
- In den meisten Programmierumgebungen, in denen asynchrone Methodenaufrufe unterstützt werden, bekommt man als Ergebnis eines asynchronen Aufrufs ein *Aufruf-Handle* (auch ***Future*** genannt) zurück, über das man Status und Ergebnisse des Aufrufs abfragen kann.

Nebenläufigkeit (2)

Datenkonsistenz

Es gibt im wesentlichen zwei Ursachen, die die Konsistenz der Zustände von Clouds-Objekten gefährden:

- Nebenläufigkeit — Parallelbearbeitung von Objektmethoden
- Systemfehler, durch die die Bearbeitung von Methoden abgebrochen wird

Das bekannteste Paradigma zu Gewährleistung von Datenkonsistenz sind atomare Transaktionen (ACID-Eigenschaften — Atomarität, Konsistenz, Isolation und Dauerhaftigkeit)

❑ Transaktionen

- Transaktionssysteme stellen sehr restriktive Auflagen an die Abarbeitung. Insbesondere die Kriterien Atomarität und Isolation sind mit dem Ziel einer parallelen Anwendungsabarbeitung nur schwer verträglich.
- Die notwendige Serialisierung von Abläufen führt in der Praxis meist dazu, daß andere Aktivitätsträger blockiert werden müssen, während ein Aktivitätsträger Änderungen an einem Objektzustand durchführt.
- Viele Systeme, die Transaktionen implementieren, bieten dem Programmierer deshalb gleichzeitig Mechanismen an, die Transaktionsrestriktionen aufzuweichen.

– Atomarität, Serialisierung

– kaum verträglich mit Nebenläufigkeit

❑ Optimierung von Nebenläufigkeit und Konsistenzanforderungen

- Während für das Betriebssystem bzw. für allgemeine Transaktionssysteme nicht feststellbar ist, unter welchen Umständen von den strengen Transaktionsregeln abgewichen werden kann, ohne das Gesamtergebnis der Anwendung zu gefährden, liegen im Rahmen der Anwendungsprogrammierung Informationen vor, mit deren Hilfe eine Konsistenzerhaltung gewährleistet werden kann, obwohl man auf die Nutzung von strengen ACID-Transaktionen verzichtet.
- Hierzu müssen in der Programmierumgebung Mechanismen zur Steuerung der Konsistenzerhaltungsmechanismen angeboten werden.

• unterschiedliche Konsistenzanforderungen

– globale Konsistenz

- Eine Gruppe von Clouds-Objekten muß zusammen in konsistentem Zustand erhalten werden (Koordinierung von Nebenläufigkeit und Recovery).

– lokale Konsistenz

- Der Zustand eines Clouds-Objekts muß konsistent erhalten werden (unabhängig von anderen Clouds-Objekten)

– keine Konsistenzgarantie

❑ Konsistenzattribute für Threads und Methoden

- Die Konsistenzattribute werden an Threads vergeben, bei deren Bearbeitung für die Einhaltung der jeweiligen Konsistenzanforderungen gesorgt wird.