21. Was ist der Unterschied zwischen Prozess und Thread? Was ist beim Einsatz von Multi-Threading zu beachten? Welche Bedeutung haben Threads in verteilten Systemen, insbesonders in Client/Server-Umgebungen?
Ein Prozess ist ein Programm in Ausführung, und wird vom Betriebssystem auf einem eigenen, virtuellen Prozessor ausgeführt. Er hat einen eigenen Adressraum (kann nicht auf RAM der anderen Prozesse zugreifen), CPU Registerwerte, offene Dateien, Speicherabbildungen, gewisse Rechte,...
Das Erstellen eines Prozesses ist relativ teuer, es muss u.a. ein eigener Adressraum generiert werden, und dieser muss "gereinigt" werden -> alle Bits auf 0 setzen.
Ein Prozess kann aus mehreren Threads bestehen, Threads teilen sich den selben Adressraum (hierbei muss der Programmierer dafür sorgen, dass ein Thread keinen Schaden/unerwünschte Zustände bei konkurrierenden Speicherzugriffen erzeugt) Das Switchen zwischen Prozessen ist teurer, als das Switchen/Wechseln zwischen Threads, hierbei muss nur der CPU Kontext gewechselt werden, bei Prozessen noch mehr (Adressraum, offene Dateien,..) Blockiert ein Thread aufgrund eines System Calls, so blockiert der komplette Prozess.

2 Levels:
- user level thread: Ein Thread-Library wird komplet im user modus ausgeführt. Der Thread wird im User-Adressraum ausgeführt, was es billig macht Threads zu erstellen und zwischen ihnen zu wechseln. Nachteil: Ein System Call blockiert den gesamten Prozess
- kernel level thread: Thread läuft im Kernel des Betriebssystems → System Calls können den Thread nicht den ganzen Prozess blockieren, jedoch hat dies dieselben Nachteile wie bei Prozessen.

Threads bringen:
- einfachere Implementierung: (divide&conquer) jeder Thread muss nur mehr einen Teil des Gesamtproblems lösen
- höherer Durchsatz: blockiert ein Thread, so kann billig auf den nächsten gewechselt werden.
- Es kann höhere Transparenz erreicht werden (z.B.: verbergen der Latenzzeiten, während auf das Ergebnis des Servers gewartet wird, kann ein anderer Thread weiterarbeiten)

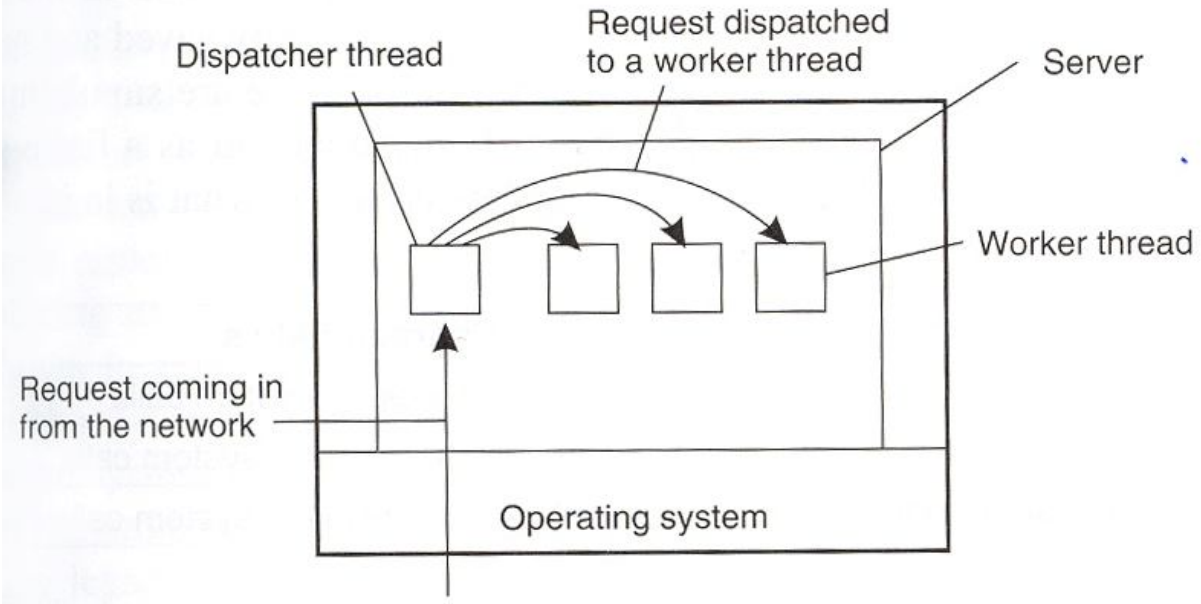
Es muss jedoch darauf geachtet werden, dass ein blockierender Thread nicht den ganzen Prozess lahm legt, dies kann entweder durch die Vermeidung von System calls erreicht werden, oder durch die Verwendung von z.B. LWP's (lightweight processes).
Worauf bei Multithreading geachtet werden muss:
- Safety — Synchronisieren damit sich die Threads nicht gegenseitig beeinflussen (Speicher überschreiben,...)
- Liveness — Deadlocks vermeiden und dafür sorgen dass jeder Thread fair behandelt wird
- Performance – Overhead durch context switching und synchronisieren vermeiden.

22. Welche Aspekte Verteilter Systeme sind auf Client-Seite zu berücksichtigen? Wie werden User Interfaces in die Architektur Verteilter Systeme eingebunden? Wie können dabei verschiedene Arten der Transparenz unterstützt werden?
Clients müssen in Verteilten Systemen mit dem User und dem Remote Server parallel kommunizieren. Dies kann durch Multithreading am besten und einfachsten geschehen. Weiters können dadurch zB Latenzzeiten bei der Kommunikation überbrückt werden. Wenn es die Server unterstützen, so kann auch über Load-Balancing nachgedacht werden, so dass sich ein Client von mehreren Server-Replikas die Daten parallel besorgt. Auch die lokale Abarbeitung von Daten steigert die Performance, da zB der Netzwerktransfer minimiert werden kann (zB durch Formularkontrolle am Client anstatt am Server).
User Interfaces
- Model
- Control
- View

UIs werden (wie auch Verteilte Systeme) oft schon in einem Schicht-Model realisiert (Vertikale Verteilung). Einzelne Schichten können auf verschiedenen Maschinen laufen (Horizontale Verteilung) (Distributed User Interface). z.B.: kann das Rendering und die Interpretation der Usereingaben komplett am Server erfolgen und der Client dient nur noch als "dumme" Konsole (Remote Application ==>

Rechner-Last auf Server). Damit wird die Application unabhängig von der Client-Plattform (Location, Migration, Relocation Transparency).
Arten der Transparenz: Neben Benutzeroberfläche und anderer für die Applikation benötigter Software besteht die Client-Software aus Komponenten, mit denen Verteilungstransparenz erzielt werden soll. Im Idealfall sollte ein Client nicht erkennen, dass er mit einem entfernten Prozess kommuniziert.
Die Zugriffstranzparenz wird häufig realisiert, indem aus der vom Server gebotenen Schnittstellendefinition ein Client-Stub erzeugt wird. Der Stub bietet dieselbe Schnittstelle, die auch auf dem Server zur Verfügung steht, verbirgt jedoch die möglichen Unterschiede in Hinblick auf die Maschinenarchitektur, sowie die eigentliche Kommunikation. Es gibt unterschiedliche Möglichkeiten, Orts-, Migrations-und Relokationstransparenz zu realisieren. Die Verwendung eines praktischen Namenssystem ist wesentlich. In vielen Fällen ist auch die Zusammenarbeit mit Client-seitiger Software wichtig. Ist beispielsweise ein Client bereits zu einem Server gebunden, kann der Client direkt informiert werden wenn sich die Position des Servers ändert. In diesem Fall kann die Middleware des Clients die aktuelle Position des Servers vor dem Benutzer verbergen und die erneute Bindung zu dem Server ggf. transparent vornehmen.
Auf ähnliche Weise implementieren viele Verteilte Systeme die Replikationstransparenz mithilfe Client-seitiger Lösungen.
Die Maskierung von Kommunikationsfehlern mit einem Server erfolgt normalerweise über Client-Middleware. Beispielsweise kann eine Client-Middleware so konfiguriert werden, dass sie wiederholt versucht, eine Verbindung mit einem Server aufzunehmen, oder nach mehreren Versuchen einen anderen Server auszuprobieren (Fehlertransparenz).

23. Geben Sie grundlegende Design-Entscheidungen für Server an und bewerten Sie diese. Gehen Sie auf den Unterschied zwischen stateful und stateless Servern genauer ein und geben Sie Beispiele an. Erläutern Sie anhand einer Skizze Architektur und Funktionsweise eines multi-threaded Servers (zB File- oder Web-Server). Was ist beim Einsatz von Multi-Threading vom Entwickler besonders zu beachten?



Grundlegende Design-Entscheidungen sind zB:
- benötigt man einen iterativen Server (Server handled alles selbst)
- oder einen concurrent server (nebenläufig, gibt alle Aufgaben an eigene Worker-Threads oder andere Prozesse weiter)

wie werden Interrupts behandelt
- Server Interrupt (???): Client beenden und neu starten -> Server wird Verbindung aufgeben, da er denkt der Client ist abgestürzt.

- out of band control: Client und Server so entwickeln dass sie in der Lage sind out of band daten zu senden. Der Server behandelt diese Daten mit höchster Priorität vor allen Anderen.

ist der Server / besitzt der Server
- stateless Server hat keine Information über den Status des Clients und kann eigenen Status ändern ohne den Client darüber zu informieren. (z.b. Webserver). Zustandslose Server enthalten in Wirklichkeit zwar dennoch Informationen über die Clients, aber entscheidend ist dass der Verlust dieser Informationen zu keinen Unterbrechungen führt.
- stateful Server enthält beständige Information über Clients. Diese werden so lange behalten bis sie ausdrücklich gelöscht werden. (z.B. Fileserver: Tabelle mit Clients und Berechtigungen). Sind aus Clientsicht schneller als stateless Server. Nachteil: Stürzt der Server ab, muss der Status vor dem Absturz wieder hergestellt werden. Gelingt das nicht kommt es zu Problemen.
- soft state (der Server kennt den Client nur für eine bestimmte Zeit und verwirft danach alles wieder)
- session state (hier werden Aktionen immer in einer Session ausgeführt, in der sich ein Client authentifiziert hat - zB WebServer mit Session à la Cookies)

wie kann man den Server finden? (Binding, Name server, directory server)

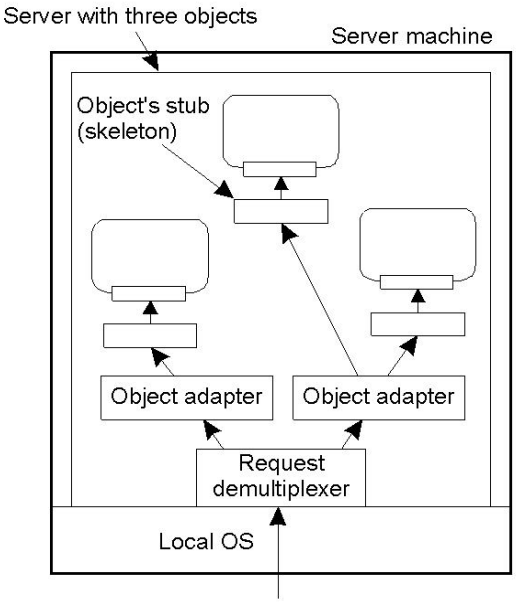
Arten von Servern:
- multithreaded server: bestehend aus einem dispatcher und mehreren worker threads. Der dispatcher thread wartet auf eingehende anfragen, und startet pro Anfrage einen worker thread, an welchen die Anfrage weitergereicht wird. Somit ist es möglich, dass während eine Anfrage bearbeitet wird, der dispachter thread wieder auf neue Anfragen warten kann.
- single-threaded server: Bestehend aus nur einem einzigen Thread, welcher die Anfragen entgegennimmt, bearbeitet und das Ergebnis an den Client sendet. Die Implementierung ist recht einfach, jedoch kann immer nur ein Client zur selben Zeit bearbeitet werden.
- finite-state machine: Ebenfalls nur ein Thread. Die finite state machine hält eine Tabelle mit Ergebnissen bereit, sodass eine Anfrage, wo das Ergebnis bekannt ist, sofort beantwortet werden kann. Falls das Ergebnis nicht in der Tabelle enthalten ist, wird eine Anfrage an die Festplatte geschickt, jedoch sofort weitergearbeitet. Wenn das Ergebnis dieser Anfrage eintrifft, wird es in die Tabelle eingetragen und an den Client geschickt.

Ein stateless Server speichert/hat keinerlei Informationen über den Zustand der Clients (z.B.: HTTP Server), der Client ist eigenverantwortliche bei Änderungen die Initiative zu ergreifen, wobei der stateful server genau über den Status des Clients bescheid weiß, und diesen persistent speichert (z.B: Fileserver, bei Änderungen (Dateien, Rechte,..) informiert er die Clients davon)
Beim Einsatz von Multi-Threaded Servern ist vom Entwickler besonders Wert auf die Nebenläufigkeit zu legen. (vergleiche auch synchronzied in Java). Problem: Threads greifen alle auf denselben Speicherbereich des Prozesses zu und können sich gegenseitig Werte überschreiben. Dies kann zu Inkonsistenzen führen.

24. Was sind die Besonderheiten von Objekt-Servern? Welche Arten gibt es dabei für die "Invocation", also den Aufruf (evtl. auch die Aktivierung/Activation) eines Objektes auf Server-Seite (Policies hinsichtlich thread, code sharing, und object creation)? Was ist in diesem Zusammenhang ein Objekt-Adapter?
Ein Objektserver dient zum Bereitstellen von Objekten in einem Verteilten System. Im Gegensatz zu herkömmlichen Servern, stellt der Objektserver keine spezifischen Dienste zur Verfügung, dafür sind die Objekte des Servers zuständig. Der Objektserver stellt lediglich die Mittel, um lokale Objekte remote zugreifbar zu machen. Bevor ein Objekt aufgerufen kann, muss es in den Adressraum des Servers gebracht werden. Es gibt verschiedene Arten der Aktivierung von Objekten: Sie können beim Start des Servers (alle zugleich) erstellt werden, oder beim ersten Aufruf (können nach dem ersten Aufruf bestehen bleiben bis zur Beendigung des Servers, oder gleich nach Beendigung der Anfrage zerstört werden)

Des weiteren kann die activation policy eines Objektes festlegen, ob es in einem gemeinsamen Speicherbereich mit anderen Objekten, oder in einem eigenen Speicherbereich untergebracht wird (aus Sicherheitsgründen).
Durch Codesharing können sich Objekte den Code teilen, sodass dieser nur einmal am Server geladen werden muss (z.B.: Ein Objekt zum Zugriff auf eine Datenbank, welches von allen Anfragen benutzt werden kann). Es kann vom Objektserver für jedes Objekt ein Thread erzeugt werden, oder ein Thread für alle Objekte (Warteschlange falls Anfragen während der Bearbeitung eintreffen). Alle diese Entscheidungen werden in der activation policy festgelegt.

Object Adapter: Dient als Mechanismus um Objekte nach Policy zu gruppieren. Er beinhaltet dabei ein oder mehrere Objekte, die er bei eine Request aktiviert. Da er generisch daherkommt, kann man ihn für unterschiedliche Objekte verwenden, wobei man nur die spezifische Policy konfigurieren muss.

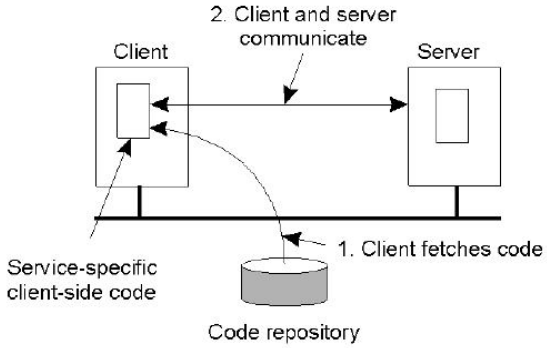


25. Erläutern Sie die wichtigsten Aspekte der Code Migration. Erklären Sie "strong mobility" und "weak mobility" und geben Sie für "weak mobility" ein Beispiel an.
Bei der Code Migration werden nicht nur Daten, sondern ganze Programme verschickt, dies ist sogar im laufenden Zustand möglich. – wird aus Performance Gründen gemacht, z.B.: Auf dem Server läuft eine Datenbank, die Client-Application stellt viele Anfragen. Hier könnte es gut sein, einen Teil des Client-Codes auf den Server zu bringen, damit nur noch die Antworten gesendet werden müssen und das Netzwerk entlastet wird.
Eine weitere Möglichkeit ist ein **Mobile Agent**. Ein Mobile Agent wandert von Seite indem er sich selbst bzw. seinen Status zu einer anderen Seite weiterkopiert (z.B. durch RPC). Durch die Parallelität erreichen wir eine lineare Steigerung der Geschwindigkeit im Vergleich zu einer einzelnen Programminstanz.

Code Migration reduziert somit den Kommunikationsoverhead, indem der Code dort hingebracht wird, wo die Daten liegen:

- query processing to database machine
- moving code to client → improve scalability

Ein weiterer Vorteil ist die **Flexibilität**. Wird z.B. auf der Clientseite ein spezielles Protokoll benötigt, um auf gewisse Funktionen des Servers zugreifen zu können (z.B. ein Video Codec), müsste dieses zum Entwicklungszeitpunkt des Clients bereits verfügbar sein. Es macht jedoch mehr Sinn, dieses Protokoll dynamisch zum Zeitpunkt des Bindings mit dem Server heruntergeladen wird, und dann erst ausgeführt wird.



The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server (e.g. Java applet).

Sender-initiatet migration: Wird vom System gestartet, wo sich der Code befindet bzw. gerade ausgeführt wird, etwa bei File Uploads oder beim Mobile Agents.
Receiver-initiatet migration: Wird vom System gestartet, das den Code benötigt, etwa bei Java-Applets.
Ein Prozess besteht aus 3 Segmenten:

- Code (tatsächlicher Programmcode)
- Resourcen (Präferenzen zu externen Resourcen , die für den Prozess benötigt werden, wie Files, Drucker, Geräte, andere Prozesse, ...)
- Execution (Speichert den Exekutionsstatus des Prozesses: private Daten, Stack, Programm Counter,..)

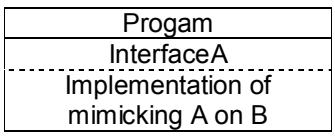
Weak Mobility: Hier wird nur das Code Segment transportiert, möglicherweise mit Initialisierungsdaten. Es ist hier nur möglich das Programm an einer, von möglicherweise mehreren, vordefinierten Positionen zu starten, wie z.B. bei Java Applets, die immer beim Begin gestartet werden. Der Vorteil dieser Methode ist die Einfachheit, die einzige Anforderung ist, dass der Empfänger den Code ausführen kann.
Beispiel → Java Applet

Strong Mobility: Hier wird auch das Execution Segment übertragen. Charakteristisch ist, dass der laufende Prozess angehalten, auf eine andere Maschine übertragen, und dort weiterlaufen kann. Es ist schwieriger zu implementieren als Weak Mobility.

26. Erläutern Sie das Konzept der Virtualisierung und in weiterer Folge deren Bedeutung für die Code Migration in heterogenen Umgebungen. Beschreiben Sie die zwei verschiedenen Arten von Architekturen von "virtual machines".

Threads und Prozesse können als eine Möglichkeit gesehen werden mehrere Dinge zur selben Zeit zu erledigen. Es können (teile von) Programme(n) gleichzeitig ausgeführt werden. Auf einem Computer mit einem einzigen Prozessor ist diese Gleichzeitigkeit natürlich eine Illusion, die durch schnelles Umschalten zwischen beiden Prozessen erreicht wird. (wird auch als **resource virtualization** bezeichnet)

Es gibt viele unterschiedliche Arten von Interfaces, die von den einfachen Befehlssätzen einer CPU bis über eine enorme Sammlung von Application Programming Interfaces die mit vielen Middlewaresystemen geliefert werden. Hier dient Virtualisierung dazu, dass ein Interface erweitert oder ersetzt wird, damit es die Eigenschaften eines anderen Systems imitieren kann.



Interace B
Hardware/software system B

Virtualisierung hilft:

- Altlasten-Interfaces (Legacy-Interfaces) auf neue Plattformen zu transportieren.
- Durch Virtualisierung kann man die Unterschiedlichkeit von Plattformen und Maschinen reduzieren, indem man jede Applikation auf einer eigenen Virtual Maschine laufen lässt und sowohl die Libraries als auch das Betriebssystem übernimmt.
- In Content Delivery Netzwerken die die Replikation von dynamischen Content realisieren sollen, ermöglicht Virtualisierung einfacheres Management indem die gesamte Seite inklusive Environment kopiert wird.

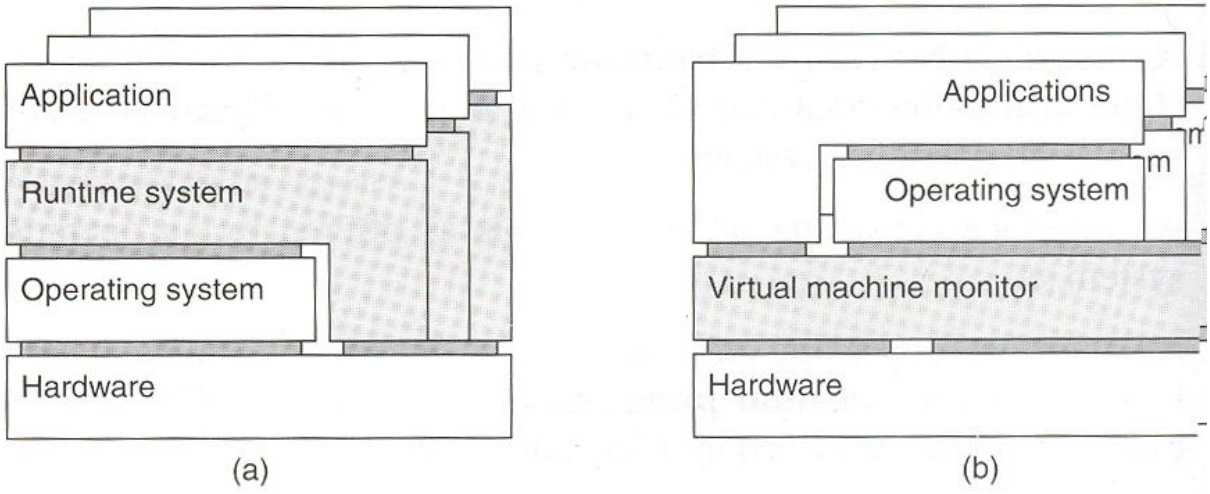
Vorteile:

- Erreichen von Flexibilität
- Erreichen von Portabilität
- Austauschbarkeit (in Bezug auf das hostende OS. Sollte dies nicht übereinstimmen, so kann durchaus eine neue VM installiert werden, um die Applikation wieder zum Laufen zu kriegen)

Architekturen:

Process Virtual Machine: Wir erzeugen eine Laufzeitumgebung, die einen abstraktes Instruction-Set zur Ausführung von Applikationen liefert. Die Befehle können entweder direkt Interpretiert werden (wie beim Java Runtime Environment), oder um eine andere Laufzeitumgebung nachzuahmen, etwa um Windowsapplikationen unter Unix laufen zu lassen. Dabei sollte beachtet werden, dass auch das Verhalten von System-Calls imitiert werden sollte, und normalerweise nur in einem einzigen Prozess laufen kann.

Virtual Maschine Monitor: Hierbei wird die ein Layer implementiert, der die original Hardware abschirmt, aber das Originale Instruction Set liefert. Somit können z.B. mehrere verschiedene Betriebssysteme auf derselben Platform laufen. Diesen Layer nennt man Virtual-Maschine-Monitor. VMware und Xen sind gute Beispiele dafür.



VMMs werden immer wichtiger im Kontext von Verlässlichkeit und Sicherheit für Verteilte Systeme. Da sie eine komplette Isolation der Anwendung und der Umgebung davon durchführen kann ein Fehler oder eine Sicherheitsattacke nicht mehr die ganze Maschine lahm legen, sondern nur den Teil der VMM. Die restlichen laufen also normal weiter. Weiters wird bei VMMs das Decoupling noch weiter getrieben, was eine noch bessere Portabilität bewirkt - nichts von den Schichten über dem VMM ist mehr abhängig von einer speziellen Hardware darunter.

Code Migration in heterogeneous systems

Anfangs war es recht schwer, Code Migration zwischen unterschiedlichen Plattformen durchzuführen, z.B. Pascal Code zu migrieren. Durch die Einführung von Virtualisierung wurde dies enorm erleichtert. Entweder man setzt überhaupt auf ein virtuelles Betriebssystem um die Migration so einfach als möglich zu gestalten, oder man geht den Weg wie Java ihn gegangen ist.
Code wird in Java nicht mehr direkt auf Maschinenbefehle runterkompiliert sondern auf eine Art "intermediate language", der dann Plattform unabhängig ist und von einer Plattform abhängigen JVM (java virtual machine) interpretiert wird. Dadurch erreicht man ein hohes Maß an Portabilität und Flexibilität was das Betriebssystem und dergleichen betrifft.

27. Erläutern Sie die Begriffe "Name", "Identifier", "Address" sowie den Bezug zwischen diesen Begriffen in der Praxis.

Name: benennt eine Entität (meistens benutzerfreundlich)

Vorteile:
- unabhängig von Adresse (ortsunabhängig) durch Namensauflösung
- für Benutzer sind (hierarchische) Namen leichter zu merken als z.B. reine (IP) Adressen

Eigenschaften:
- Location independent
- Unique

Address: Den Namen eines Zugriffspunktes einer Entität nennt man Adresse. Direkte Verwendung einer Adresse ist problematisch, da dadurch die Location-, Replication- und Migration-Transparenz nicht unterstützt wird und Adressen auch oft für Menschen schwer lesbar sind. Gute Namen sollten daher die Adresse weder direkt, noch versteckt codieren.

Identifier: für Eindeutige Kennzeichnung einer Entität, die folgenden Kriterien entspricht:
- Jeder Identifier verweist höchstens auf eine Entität
- Auf jede Entität verweist höchstens ein Identifier
- Ein Identifier verweist immer auf die gleiche Entität

Identifier können durch Counter oder Zufallszahlgeneratoren erzeugt werden. Wobei bei letzterem sichergestellt werden muss dass sich Zahlen mit hoher Wahrscheinlichkeit NICHT wiederholen.

28. Was ist ein "Name Space"? Erläutern Sie das Grundprinzip des "Closure Mechanismus" anhand eines Beispieles (zB Unix File System).