1. Geben Sie eine charakterisierende Definition für ein "Verteiltes System" an. Nennen Sie die wichtigsten Design-Ziele bzw. charakteristischen Eigenschaften von Verteilten Systemen. Stellen Sie weiters den Zusammenhang zu den typischen Fallstricken beim Entwurf verteilter Systeme her.

Verteiltes System: Eine Sammlung von unabhängigen Computern, welche dem User als ein zusammenhängendes System erscheint.
Charakteristisch Eigenschaften:

- besteht aus autonomen Komponenten
- erscheint dem Benutzer als zusammenhängendes System
- Benutzer können konsistent und einheitlich auf dem System arbeiten
- Es steht ununterbrochen zur Verfügung
- Erweiterbar und Skalierbar

Design Ziele:
Um heterogene Computer und Netzwerke zu unterstützen und gleichzeitig das Erscheinungsbild eines einzigen Systems anzubieten, werden verteilte Systeme häufig mit Hilfe einer Softwareschicht angeordnet. Ein solches verteiltes System wird als Middleware bezeichnet. Verteilte Systeme unterscheiden sich insofern von herkömmlicher Software, da ihre
Komponenten über ein Netz verteilt sind. Beim Entwurf muss die Verteilung berücksichtigt werden, da ansonsten die Systeme
unnötig zu komplex werden.

- Transparenz
- Erweiterbarkeit
- Fehlertoleranz
- Openess
- Concurency (Gleichzeitges Benutzen)
- Resource Sharing (Verteilen und auslagern von Resourcen)

Selbstverständlich unterscheiden sich verteilte Systeme von herkömmlicher Software, weil die Komponenten über ein Netz verteilt sind. Diese Verteilung während des Entwurfs nicht zu berücksichtigen macht viele Systeme unnötig komplex und führt zu Fehlern die später oft behobenwerden müssen. Folgende typischen Fehlannahmen werden beim Entwurf verteilter Systeme oft getroffen:

- Das Netzwerk ist zuverlässig.
- Das Netzwerk ist sicher.
- Das Netzwerk ist homogen.
- Die Topologie ändert sich nicht.
- Die Latenzzeit beträgt null.
- Die Bandbreite ist unbegrenzt.
- Die Übertragungskosten betragen null.
- Es gibt genau einen Administrator.

2. Was ist Transparenz? Beschreiben Sie die standardisierten Arten von Transparenz und erklären Sie den Zusammenhang zischen den einzelnen Transparenz-Definitionen. Was ist der Nachteil von Transparenz?

Transparenz: Verstecken, dass Prozesse und Ressourcen physisch über mehrere Computer verteilt sind. (Verteilung des Systems soll versteckt werden)

Arten:

- **Zugriffstransparenz**): Verbirgt Unterschiede in der Datendarstellung und die Art und Weise, wie auf eine Ressource zugegriffen wird (Unterschiede in Computerarchitektur; wie Daten von verschiedenen Computern und Betriebssystemen dargestellt werden
- **Ortstransparenz**: die geographische Position soll verborgen werden.
- **Migrationstransparenz:** Verbirgt, dass eine Ressource an einen anderen Ort verschoben werden kann (ohne dass sich an der Art des Zugriffs etwas ändert)
- **Relokationstransparenz**: Verbirgt, dass eine Ressource an einen anderen Ort verschoben werden kann, während sie genutzt wird.
- **Replikationstransparenz:** Verbirgt, dass eine Ressource repliziert ist, also dass mehrere Kopien einer Ressource vorhanden sind. (notwendig dass alle Replikas denselben Namen tragen → System sollte daher auch Ortstransparenz unterstützen)
- **Nebenläufigkeitstransparenz** (Concurency): Verbirgt, dass eine Ressource von mehreren konkurrierenden Benutzern gleichzeitig genutzt werden kann
- **Fehlertransparenz**: Fehler werden verborgen. Verbirgt den Ausfall und die Wiederherstellung einer Ressource.

Nachteile:

Nicht immer Sinnvoll (z.B. Drucker --> man möchte schon den Ort des Druckers kennen) und nicht immer möglich (Hohe Latenzzeiten z.B bei Updates --> kann nicht versteckt werden).

Trade-Off zwischen einem hohen Grad an Transparenz und der Performance des Systems! (z.B. wenn Replikas auf verschiedenen Kontinenten immer konsistent sein müssen, kann eine einzelne Update-Operation Sekunden dauern, was vor dem Benutzer nicht mehr verborgen werden kann;

Man sollte daher die Transparenz so gestalten, um das Verhalten eines Systems für den Benutzer nachvollziehbar zu machen.

3. Was versteht man unter "Openness"?

Unter “Openness” versteht man die Bereitstellung der Services nach Standardsregeln, die die Syntax und Semantik beschreiben. Also prinzipiell um standardisierte Schnittstellen, die vom System zur Verfügung gestellt werden sollten. Dieses Verhalten wird mithilfe von Protokollen und Interfaces erreicht. Um Flexibilität zu erreichen sollte das System aus kleinen, einfach austauschbaren und anpassbaren Komponenten bestehen.

Maßzahlen:

Vollständigkeit: Beschreibt das Ausmaß an Interoperatibilität → alles was für eine Implementierung erforderlich ist wurde spezifiziert

Neutralität: Beschreibt das Ausmaß an Portierbarkeit (ob eine Komponente auch auf einem anderen System mit gleichem Interface läuft) → Spezifikation schreibt nicht vor, wie eine Implementierung aussehen soll

Vollständigkeit ist wichtig für Interoperabilität (beschreibt den Grad bis zu dem zwei Implementierungen von Systemen/Komponenten verschiedener Hersteller zusammenarbeiten

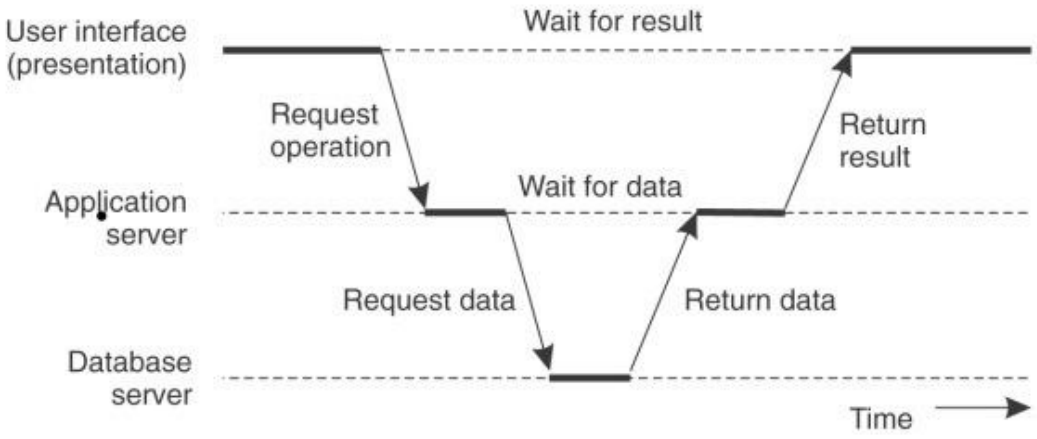
können, indem sie sich lediglich auf die Dienste der anderen verlassen, die nach einem gemeinsamen Standard spezifiziert sind). Neutralität ist wichtig für Portabilität (beschreibt, in welchem Ausmaß eine Anwendung, die für VS A entwickelt wurde, ohne Veränderungen auf VS B ausgeführt werden kann, das dieselben Schnittstellen wie A implementiert).

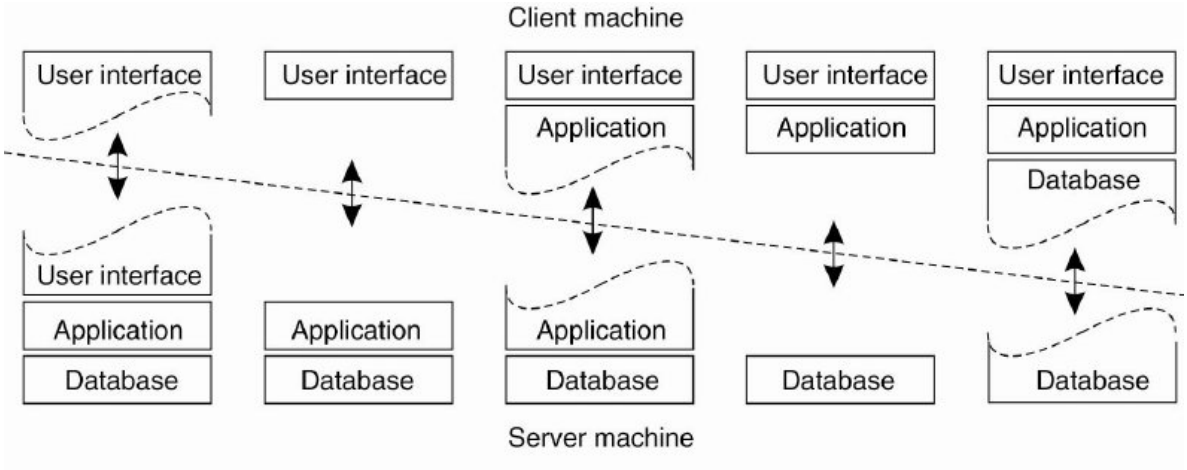
4. Erläutern Sie Probleme und Lösungsansätze für "Scalability".
Scalability: Die Fähigkeit, des Systems bei wachsenden Anforderungen noch einsatzfähig zu sein. Man kann unterscheiden zwischen Skalierbarkeit mit der Anzahl der User und Ressourcen, geographische Ausdehnung und Erstreckung über mehrere Organisationen als ursprünglich geplant.

Probleme
- Größe: Es werden mehr User oder Ressourcen ins System integriert, als ursprünglich geplant --> Man stößt an Grenzen von zentralisierten Diensten, Daten und Algorithmen
- Größere geografischer Ausdehnung --> Latenzzeit wird größer
- Administrativ: Es werden mehr Organisationen/Firmen eingebunden (also über mehrere Domänen) , die sich jedoch in den Schnittstellen (bezüglich Ressourcen/Verwaltung/Sicherheit) wiedersprechen

Lösungen:
- Verbergen der Latenzzeiten durch asynchrone Kommunikation bzw. einen Teil der Verarbeitung auf den Client verschieben (um die Gesamtkommunikation zu verringern).
- Replikation: Erhöht Verfügbarkeit und verringert die Kommunikation (wenn die Replikate auf nahegelegenen Rechnern liegen, oder durch Caching auf lokalen Computer)
- Hierarchien: Verteilung von Funktionen auf verschiedene Domänen

5. Was versteht man unter der vertikalen Verteilung bzw. N-Schichten-Systemen? Diskutieren Sie dabei alle Grundvarianten von Client/Server-Systemen. Ist folglich ein Java Applet eher ein Thick Client oder ein Thin Client?
Vertikale Verteilung/N-Schichten System
Ein System wird in n-Schichten eines Systems (zumindest Presentation- Businesslogic-Persistance) aufgeteilt und auf (physikalisch) verschiedene Systeme verteilt → Vertikale Verteilung





Ein Java Applet ist ein Thick Client, die meiste Funktionalität auf dem Client liegt.

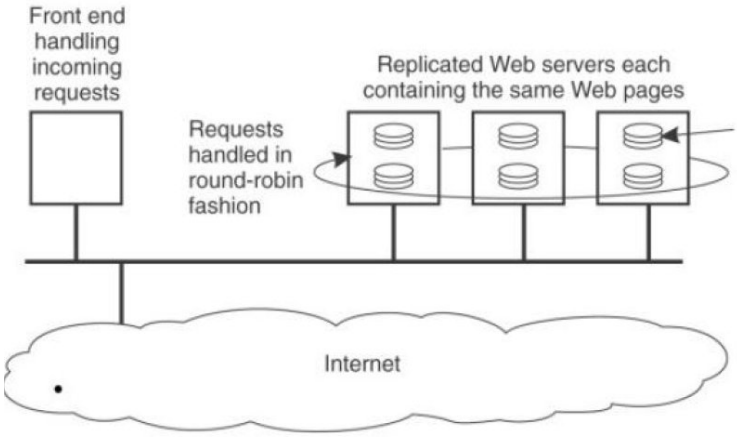
6. Was ist horizontale Verteilung? Mit welchen grundlegenden Design-Fragen müssen Sie sich beim Entwurf der horizontalen Verteilung eines Systems beschäftigen? Gibt es einen Zusammenhang zur vertikalen Verteilung?

Bei der horizontalen Verteilung werden Server oder Clients in logisch äquivalente Teil aufgeteilt, wobei jeder mit seinem eigenen Anteil des Datenbestandes arbeitet. Dies dient zur Lastverteilung.

Designfragen

- Es wird ein geeignetes Konsistenzmodel benötigt, um sicherzustellen, dass alle Teile mit dem gleichen Datenbestand arbeiten.
  Datenzentrierte Konsistenzmodelle (strenge-, sequezielle-, kausale-, FIFO-Konsistenz) versus Clientzentrierte Konsistenzmodelle (Eventuelle Konsistenz, Monotones Lesen, etc)
- Membership Management: Wie sich verschiedene Knoten im System organisieren

Beispiel: Webserver

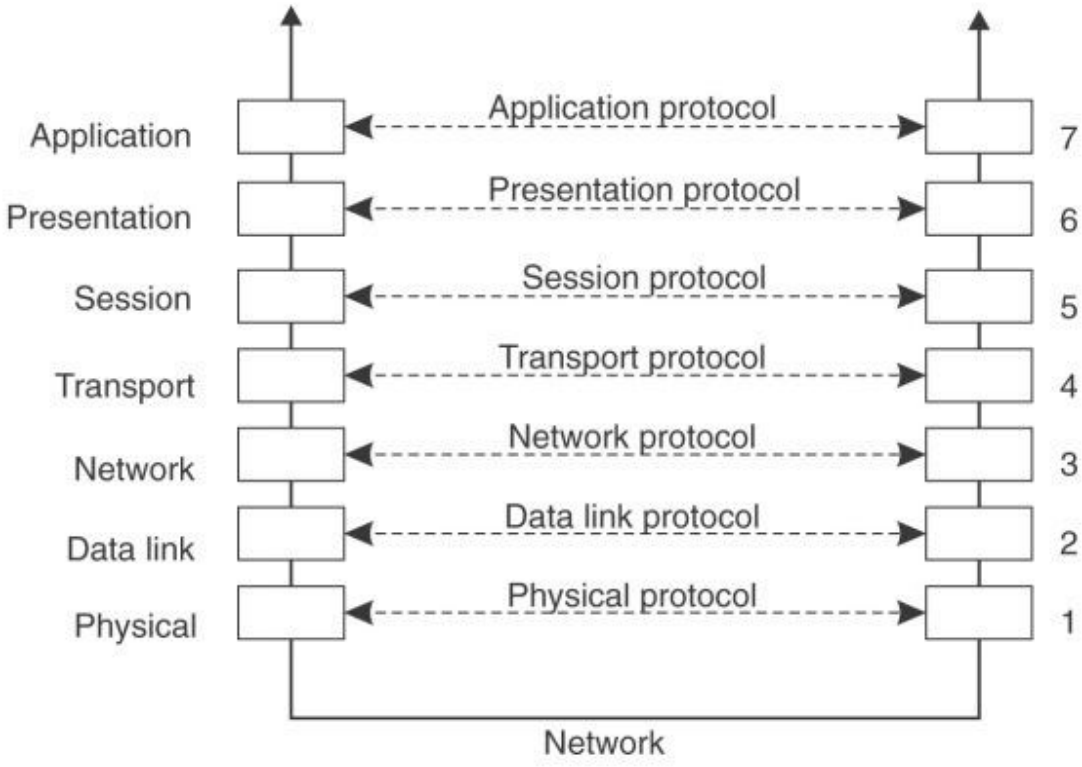


Jeder Server verwaltet dieselbe Menge an Webseiten, und immer wenn eine Webseite aktualisiert wird, wird eine Kopie davon auf jeden Server platziert (je nach Konsistenzmodell verschieden schnell). Trifft eine Anforderung ein, wird sie unter Verwendung einer Round-Robin-Strategie an einen Server weitergegeben. Genügend Bandbreite vorausgesetzt, ist diese Form der horizontalen Verteilung sehr effektiv.

Eine Klasse moderner Systemarchitekturen, die die horizontale Verteilung unterstützen, sind die sogenannten Peer-to-Peer-Systeme. Die Prozesse, die ein P2P System bilden sind alle gleich, die Funktionen, die ausgeführt werden müssen werden also von jedem Prozess des verteilten Systems dargestellt. Folglich ist die Interaktion zwischen den Prozessen symmetrisch: jeder Prozess agiert gleichzeitig als Client und als Server (Servent). Ein Problem hierbei ist, dass ein Prozess i.A. nicht direkt mit einem beliebigen andernen Prozess kommunizieren kann, was aber erforderlich ist, um die Nachricht durch die verfügbaren Kommunikationskanäle zu senden.

7. Beschreiben Sie das ISO-OSI Modell der geschichteten Protokolle (Grundprinzip). Stellen Sie den Bezug zu den Internet-Protokollen (TCP/IP) her. Warum sind Transport-Layer Protokolle für Verteilte Systeme oft nicht ausreichend?

Im OSI Schichtenmodel dient zur Kommunikation in offenen Systemen. Im Detail handelt es sich um 7 Schichten (Layer), wobei jede Schicht nur auf die darunterliegende Schicht Zugriff hat.



1. Physical Layer (Bitübertragung)
Die Bitübertragungsschicht ist für das Übertragen der Einsen und Nullen zuständig. Das Protokoll der Bitübertragungsschicht kümmert sich um die Standardisierung der elektrischen, mechanischen und Signalschnittstellen, sodass eine von einem Rechner ausgesendete 1 auch tatsächlich als 1 und nicht als 0 empfangen wird. Die Bitübertragungsschicht sendet lediglich Bits. Hierbei können allerdings Fehler auftreten, sodass ein Verfahren benötigt wird, um sie zu entdecken und zu korrigieren.
2. Data Link Layer (Sicherung)
Dies ist Aufgabe der Sicherungsschicht. Sie gruppiert die Bits in Frames und berechnet ene Prüfsumme. Der Empfänger berechnet die Prüfsummer erneut. Stimmen beide überein wird der Frame als korrekt anerkannt und angenommen, wenn nicht bittet der Sender den Empfänger um die erneute Übertragung.

3. Network Layer (Vermittlung)
Damit eine Nachricht vom Sender zum Empfänger gelangen kann können (in WANs) mehrere Teilstrecken (Hops) erforderlich sein. Die Auswahl des besten Weges (Routing) ist Aufgabe des Network Layer. Momentan ist das am weitesten verbreitete Netzwerkprotokoll das verbindungslose Internetet Protocol (IP)
4. Transport Layer (Transport)
Die Transportschicht bildet den letzten Teil des grundlegenden Netzwerkprotokollstacks. Die Anwendungsschicht kann somit eine Nachricht an die Transportschicht in der Erwartung übermitteln, dass sie ohne Verluste ihr Ziel erreicht. Wenn die Transportschicht eine Nachricht von der Anwendungsschicht erhält, zerteilt sie sie in kleine Stücke (die klein genug für eine Übertragung sind) weist jedem eine fortlaufende Nummer zu und sendet dann alle. Transportverbindungen können auf verbindungslosen oder verbindungsorientierten Netzwerkdiensten aufsetzten. Bei verbindungslosen Diensten kann nicht garantiert werden, dass alle Pakete in der selben Reihenfolge ankommen, in der sie gesendet wurden. Es ist Aufgabe der Software der Transportschicht, alles wieder in die richtige Reihenfolge zu bringen. Das Transportprotokoll des Internets wird TCP (Transmission Control Protocol) genannet. Die Kombination TCP/IP ist heute der Standard für Netzwerkkommunikation. DieInternetprotokollsuite unterstützt auch ein verbindungsloses Transportprotokoll namens UDP (Universal Datagram Protocol).
Vorhandene Transportprotokolle sind oft nicht ausreichend, so werden regelmäßig neue vorgeschlagen, wie beispielsweise RTP (real-time transport protocol) zur Datenübertragung in Echtzeit.
5. Session Layer (Sitzung)
Die Sitzungsschicht ist im Wesentlichen eine erweiterte Version der Transportschicht. Sie bietet eine Dialogkontrolle, um zu verfolgen welcher Teilnehmer gerade spricht, sowie Funktionen zur Synchronisierung. In der Praxis sind nur wenige Anwendungen an der Sitzungsschicht interessiert. In der Internetprotokollsuite ist sie nicht vorhanden.
6. Presentation Layer (Darstellung)
Die Darstellungsschicht beschäftigt sich mit der Bedeutung der Bits. Hier können spezielle Konvertierungen stattfinden.
7. Application Layer (Anwendung)
Bsp.:
File Transfer Protokoll (FTP): Übertragung von Dateien zwischen Client und Server (nicht mit dem Programm FTP zu verwechseln!)
Hyper Text Transfer Protokoll (HTTP): Übertragung von Webseiten
Im OSI Modell fehlt eine klare Unterscheidung zwischen Anwendungen, anwendungsspezifischen Protokollen und Protokollen für allgemeine Zwecke.

Transport Layer Protokolle nicht ausreichend weil:

- Keine Session Kontrolle
- Bedeutung der Bits ist nicht gegeben (Dateitypen etc.)
- Kein Authentifizierungsprotokoll
- Kein Locking Protokoll (gemeinsamer Zugriff auf Daten)
- Keine Verschlüsselung

8. Was ist Middleware? Welche Anforderungen stellt man an Middleware? Welche Services soll Middleware bieten? Erläutern Sie den Zusammenhang zwischen Middleware und Architectural styles.

Middleware fast häufig verwendete Services von verteilten Systemen zusammen. Dazu zählen:

- Verschlüsselung
- Authentifizierung
- Security
- Replication
- Access Transparenz
- Naming

Die Middleware liegt somit zwischen dem Transport Layer und dem Applikation Layer.
Anforderungen: Sie soll einen Grad an Verteilungskompetenz bieten, der im gewissen Maße die Verteilung der Daten, Verarbeitung und Steuerung übernimmt. Ein Architekturstil hat den Vorteil, dass die Middleware einfacher geformt werden kann, jedoch nicht mehr 100% für den Anwendungszweck geeignet ist.
Important styles of architecture for distributed systems (Folien)

- Layered architectures (OSI)
- Object-based architectures (and components)
- Data-centered architectures (file based, database, resourceful WS, ...)
- Event-based architectures
- .... and combinations thereof

9. Wie kann man die Flexibilität der Middleware erhöhen sowie die Zusammenarbeit von Middleware und Anwendung effizienter gestalten? Erläutern Sie dabei die Grundprinzipien von Interceptoren, Adaptivität und Self-Management.

Middleware soll einfach zu konfigurieren, adaptierbar und anpassbar sein

Eben mithilfe dieser 3 Grundprinzipien
Interceptors: Sind Software-Konstrukte, die es ermöglichen den normalen Kontrollfluss zu unterbrechen und anderen Code auszuführen. Ermöglicht somit die Middleware anzpassen.
Adaptivität: Middleware soll sich an Veränderungen in der Umgebung eines DS anpassen → Konstruktionadaptiver Software

- Separation: Traditionelle Art, Systeme zu modularisieren (d.h. Trennen der Teile eines Systems, die Funktionalität implementieren, von denen, die für Extrafunktionalitäten wie Reliability, Performance, Security etc. zuständig sind.) – ist in VS nicht einfach (Thema für aspect-oriented software development)
- Reflection: Fähigkeit eines Programms, sich selbst zu überprüfen und, wenn notwendig, sein Verhalten anzupassen
- Component-based design: unterstützt Anpassung durch Komposition (dynamisch zur Laufzeit)

Self-Management in Verteilten Systemen

Selbstmanagement ist in großen, verteilten Systemen von entscheidender Bedeutung. Für viele Systeme stellen eine zentrale Verwaltung und Organisation keine optimale Lösung dar. Die meisten selbstverwaltenden Systeme haben gemeinsam, dass Anpassungen mithilfe einer odermehrerer Rückkopplungsschleifen (feedback-control-loops) erfolgen.

- Selbstverwaltend
- Selbstheilend
- Selbstkonfigurationierend
- Selbstoptimierend

10. Erläutern Sie das Grundprinzip des Remote Procedure Call. Gehen Sie auf die Begriffe "client stub" und "server stub" näher ein.

Programme erhalten die Möglichkeit Prozeduren auf entfernten Rechnern auszuführen. Wenn ein Prozess auf Rechner A eine Prozedur auf dem Rechner B aufruft, wird der Prozess auf A angehalten und die Ausführung der aufgerufenen Prozedur findet auf Rechner B statt. Die Parameter können Informationen vom Aufrufer zum Aufgerufenen übertragen werden und im Ergebnis der Prozedur zurückkommen.

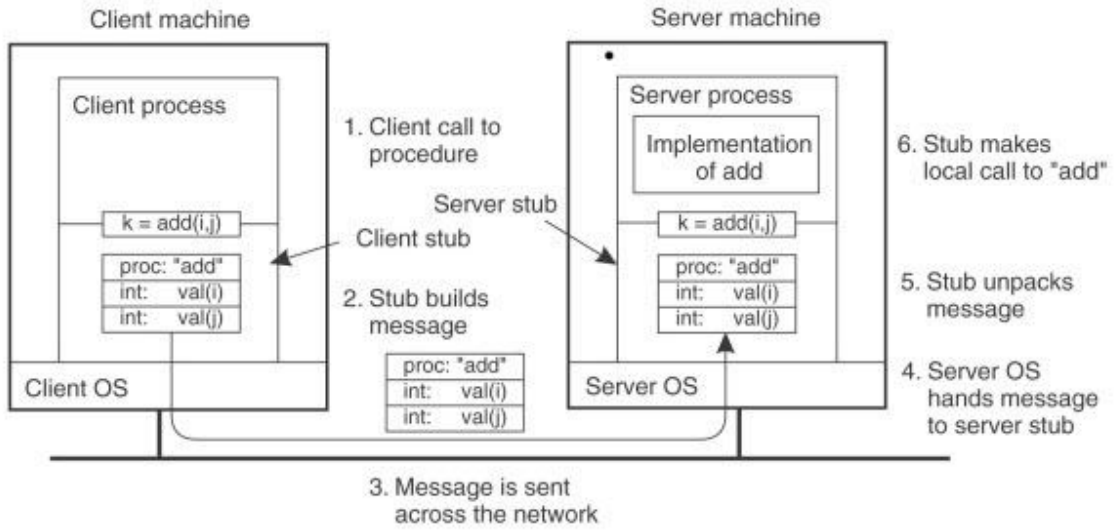
Der Client-Stub (auf dem Rechner des Clients) kapselt den entfernten Aufruf und wird zur Parameterkonvertierung verwendet. Für den Aufrufer sieht der Aufruf also als lokal aus. Erst durch den Client-Stub erfolgt die Weiterleitung des Aufrufs an den Server.

Der Server-Stub: Auf dem Server kapselt die eigentliche Prozedur, die aufgerufen wird. Der Server-Stub nimmt den entfernten Aufruf an. Für die Server-Prozedur sieht der Aufruf so aus als käme er von einem lokalen Programm

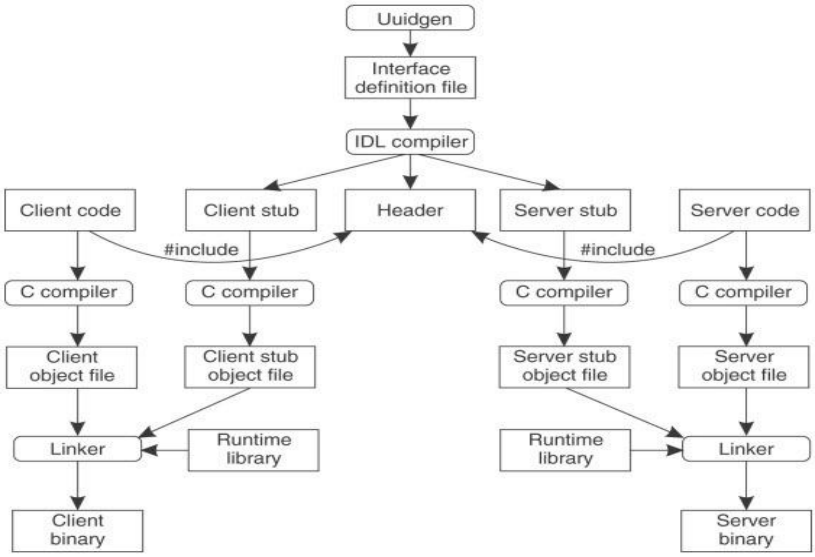
11. Wie können Variablen bei Prozedur-Aufrufen grundsätzlich übergeben werden? Wie werden Sie bei RPC gehandhabt und welche Probleme gibt es dabei? Was versteht man in diesem Zusammenhang unter "parameter marshalling"?

Die Übergabe erfolgt Grundsätzlich entweder als Referenz oder als Wert. Für die aufgerufene Prozedur ist ein Wertparameter lediglich eine initialisierte lokale Variable. Sie kann ihn ändern, aber solche Änderungen wirken sich nicht auf den Originalwert auf der Seite des Aufrufers aus.
Bei Übergabe einer Referenz, wird bei Änderung der Originalwert verändert (keine Kopie).

Die Variablen werden in Nachrichten verpackt und und an den Server-Stub gesendet. Diesen Vorgang nennt man parameter marshalling.
Probleme: In großen verteilten Systemen kommt es häufig vor, dass unterschiedliche Maschinen verwendet werden. Dabei kann es vorkommen dass die unterschiedlichen Maschinen eigene Darstellungen für Zahlen, Zeichen und andere Datensätze verwenden (IBM Mainframe verwendet den EBCDIC-Zeichencode - IBM PC´s verwenden den ASCII Zeichensatz, Intel verwendet Little Indian - SPARC verwendet Big Indian, etc.)



12. Wie schreibt man für RPCs Client und Server und welche Rolle spielt dabei die IDL? Welche Ziele werden mit dem Einsatz einer IDL in einem verteilten System verfolgt? Was versteht man in diesem Zusammenhang unter "binding"?



In einem Client-Server-System ist die Schnittstellendefinition (angegeben in der IDL - InterfaceDefinition Langugae) der Kleber, der alles zusammenhält. In den IDL-Dateien sind Prozedurdeklarationen(in ähnlicher Form wie Funktionsprototypen in C), Typendefinitionen, Konstantendeklarationen und andere Infomationen für die korrekte Verpackung der Parameter und das Auspacken der Ereignisse enthalten. Ein extrem wichtiges Element jeder IDL-Datei ist ein global eindeutiger Bezeichner für die spezifizierte Schnittstelle. Versucht ein Client versehentlich sich zu einem falschen Server zu binden, oder auch zu einer älteren Version des richtigen Servers, wird der Fehler erkannt und das Binden findet nicht statt.

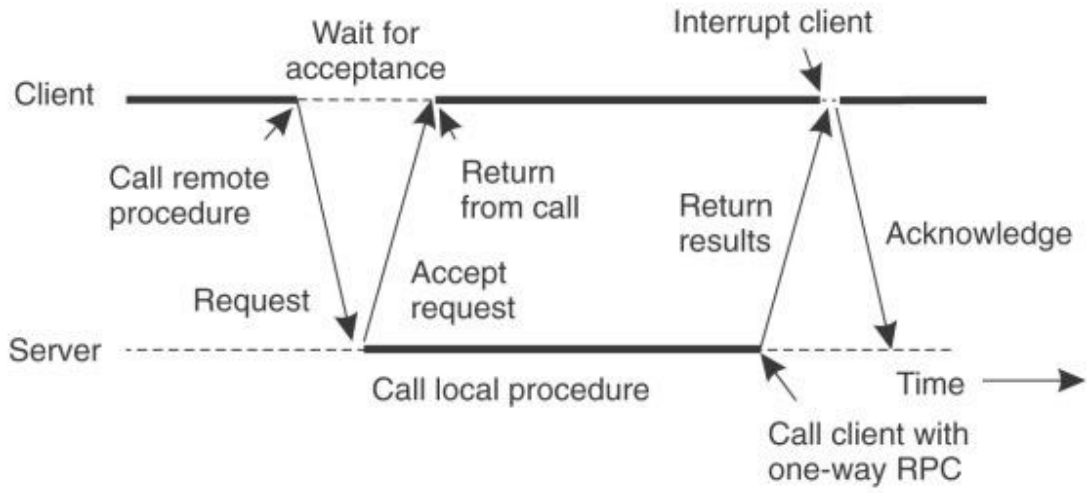
Nachdem die IDL-Datei bearbeitet wurde (Namen der Prozeduren und ihre Parameter eingetragen), wird der IDL Compiler aufgerufen. Die Ausgabe des IDL-Compilers besteht aus drei Teilen:

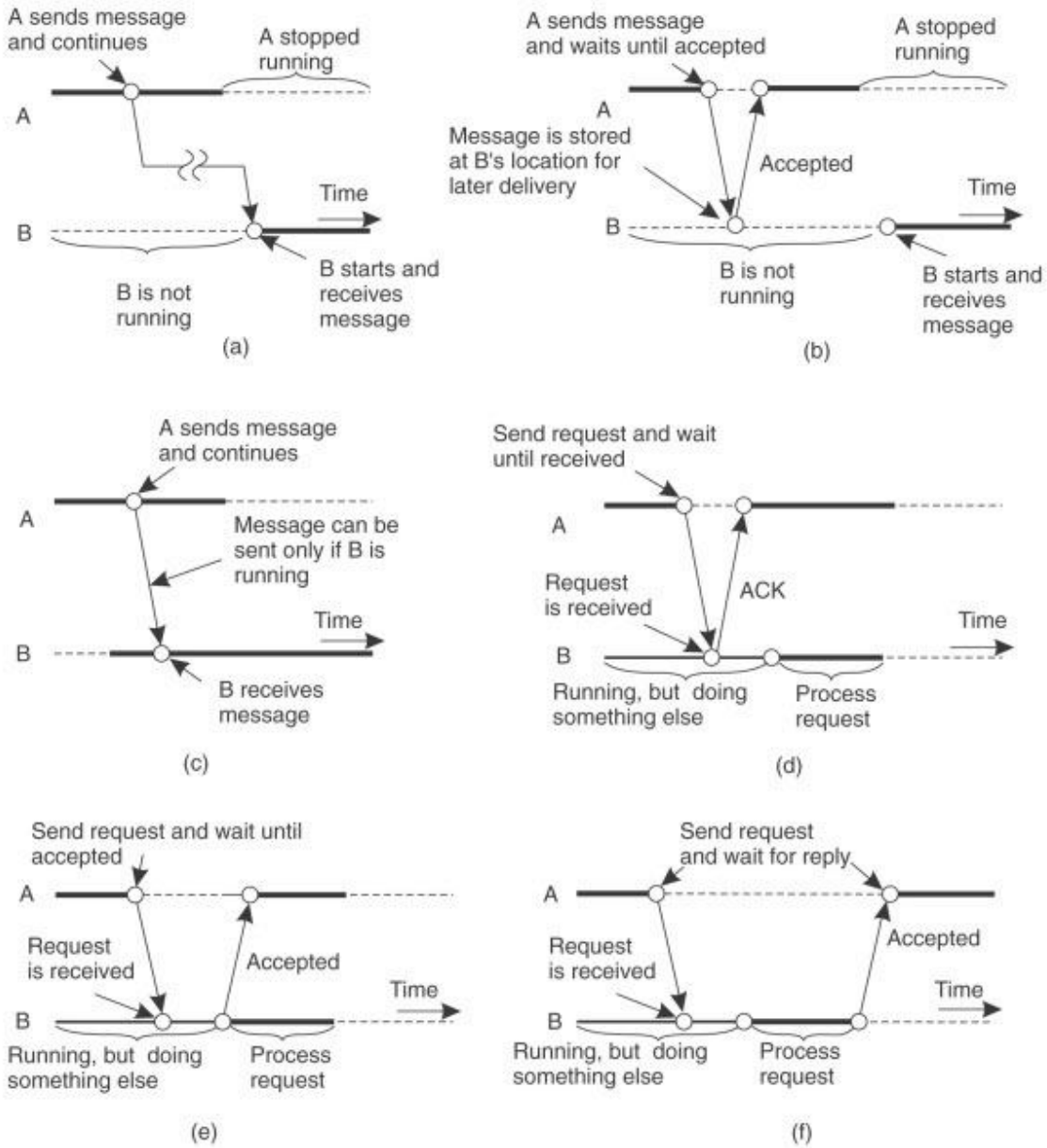
- Header-Datei (wird jeweils in den Sever- und Client-Code inkludiert)
- Client-Stub
- Server-Stub

Anschießend werden der Client sowie der Server Code kompiliert, ebenso wie die beiden Stub-Prozeduren. Die resultierenden Client-Code- und Client-Stub-Objektdateien werden anschließend zu der Laufzeitbibliothek gebunden, um die ausführbare Programmdatei für den Client zu erzeugen. Analog dazu werden der Server-Code und der Server-Stub kompiliert und gebunden, um die Programmdatei des Servers zu erzeugen. Mit dem Einsatz der IDL als Schnittstellendefinitionssprache wird die Applikationsentwicklung wesentlich vereinfacht. Daher bieten alle RPC basierenden Middleware-Systeme eine IDL, bei manchen ist sie sogar zwingend vorgeschrieben.

13. Welche Arten von asynchronen RPCs gibt es? Geben Sie auch einen verallgemeinerten Überblick über verschiedene Typen der Kommunikation (persistent/transient bzw. synchron/asynchron).

Bei asynchronen RPCs kann das Antwortverhalten des Servers unterschieden werden:
Normalerweise antwortet der Server sofort nach Empfangen einer Nachricht mit einem "Accepted". Sobald der Client dieses Acknowledgment erhält, fährt dieser mit seiner Arbeit fort. Dann gibt es noch sogenannte Einweg-RPCs wo der Client unmittelbar nach absetzten einer Anforderung fortgesetzt wird, und nicht auf eine Bestätigung vom Server wartet.
Als weitere Variante gilt der verzögerte synchrone RPC, welcher aus einer Verschachtelung eines "normalen" asynchronen RPCs und einem Einweg-RPC entsteht.
Verzögerter synchroner RPC:





(a) Persistente asynchrone Kommunikation;
(b) Persistente synchrone Kommunikation;
(c)Transiente asynchrone Kommunikation;
(d) Empfangsbasierte transiente synchrone Kommunikation;
(e) Auslieferungsbasierte transiente synchrone Kommunikation;
(f) Antwortbasierte transiente synchrone Kommunikation

Bei einer persistenten Kommunikation wird eine zur Übertragung übergebene Nachricht so lange von der Kommunikations-Middleware gespeichert, wie es dauert, sie an den Empfänger auszuliefern. Im Gegensatz dazu wird bei transienter Kommunikation eine Nachricht nur so lange gespeichert, wie die sendende und empfangende Anwendung ausgeführt werden. Die charakteristische Eigenschaft asynchroner Kommunikation ist, dass der Sender sofort fortfährt, nachdem er seine Nachricht zur Übertragung abgegeben hat. Bei der synchronen Kommunikation ist der Sender gesperrt, bis er weiß, dass seine Anforderung akzeptiert wurde.

14. Erläutern Sie die Grundprinzipien verteilter Objekte sowie der Remote Object (bzw. Method) Invocation. Gehen Sie auf die Begriffe "proxy" und "skeleton" ein. Erklären Sie den Unterschied zwischen "Compile-time" und "Run-time" Objekten. Erklären Sie den Unterschied zwischen persistenten und transienten Objekten.

Ein verteiltes Objekt realisiert als Grundlage die Trennung zwischen Schnittstelle und Objektimplementierung. Diese Trennung ermöglicht es uns, die Schnittstelle auf einen Rechner zu legen und das Objekt selbst auf einen anderen. Dieser Aufbau wird als verteiltes Objekt bezeichnet.
RMI und RPC unterscheiden sich vor allem darin, dass RMI systemweite Objektreferenzen anbietet (Objektserver). Der Client bindet sich an ein entferntes Objekt, dabei wird die Proxy (Client-Stub) geladen.

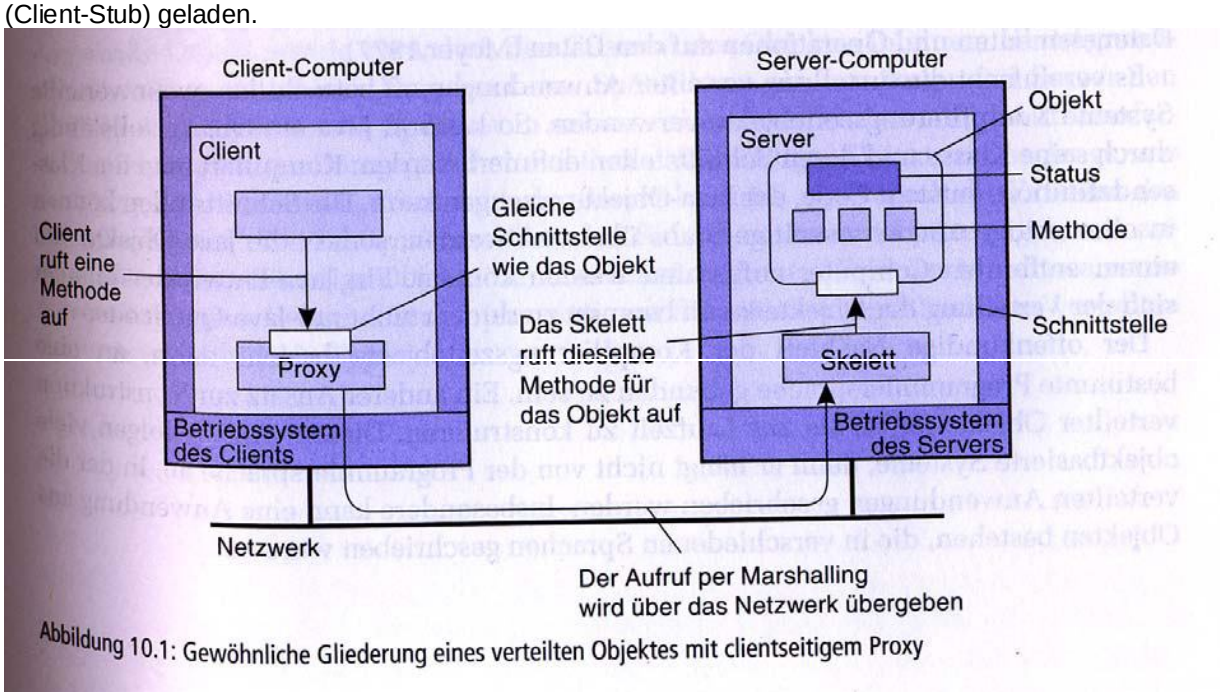


Abbildung 10.1: Gewöhnliche Gliederung eines verteilten Objektes mit clientseitigem Proxy

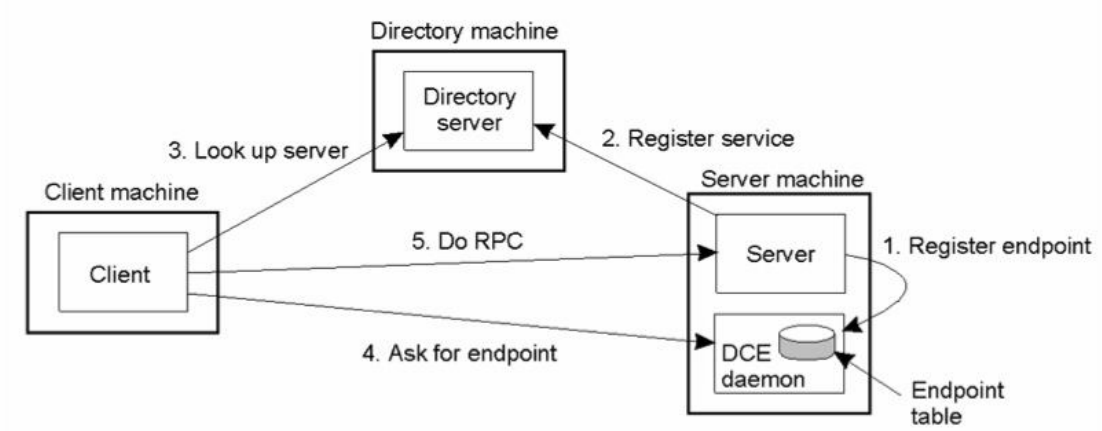
Objekte haben einen bestimmten Zustand sowie Schnittstellen um diesen zu verändern. Ein verteiltes Objekt speichert seinen Zustand auf dem Server während die Schnittstelle auf einen entfernten Client gespeichert werden kann. Wenn sich der Client den das verteilte Objekt bindet, wird die Proxy (Client-Stub) Implementierung geladen. Diese übernimmt das Marshalling der Parameter und das Unmarshalling der Antwort. Der Skeleton (Server-Stub) übernimmt das Parameterunmarshalling, das Ausführen des Methodenaufrufs des Objekts und Marshalling der Antwort am Server.
Compile-Time Objekte basieren auf Klassendefinitionen (beschreibt einen abstrakten Datentyp). Diese Definition beinhaltet alle Methoden, die das Objekt aufweist und seine Struktur kann zu Laufzeit nicht geändert werden. Von einer kompilierten Klasse können dann zur Laufzeit die Objekte instanziiert werden.
Vorteil: Aus einer Klassendefinition können Client & Server Stubs abgeleitet werden (IDL).
Nachteil: weniger flexibel als reine Laufzeitobjekte.

Sprachunabhängig Run-Time-Objekte werden zur Laufzeit konstruiert. D.h. Die Implementierung ist weitgehend offen.
Vorteil: verschiedene Programmiersprachen lassen sich leichter verbinden.
Nachteil: automatische Generierung von Stubs nicht möglich.

Transiente Objekte existieren nur im (Arbeits-) Speicher des Servers, d.h. Wenn der Server beendet wird hört das Objekt auf zu existieren.
Persistente Objekte sind auf einen persistenten Speicher gesichert und können nach einem Neustart wieder von diesem gelesen werden.

15. Wie funktioniert das Binding bei RMI? Welchen Zusammenhang gibt es zu den verschiedenen Arten, eine object reference zu implementieren. Vergleichen Sie (exemplarisch) CORBA und Java in Bezug auf Objektreferenzen.

Um eine fremde Prozedur aufzurufen, muss eine Nachricht vom Client-Prozess zum Server-Prozess versendet werden. In dieser müssen der Name der Prozedur (oder eine ID) und die zugehörigen Parameterwerte enthalten sein. Die Nachricht sollte letztlich bei einem Server-Prozess ankommen, der genau diese Prozedur implementiert.
Binding:



1. Server registriert einen Endpoint
2. Server registriert einen Service
3. Client looks up the Service at a Directory Service
4. Client asks Server for Endpoint
5. Client sends RPC

Binden führt dazu, dass im Adressraum des Prozesses am Client ein Proxy angelegt wird, der eine Schnittstelle mit Methoden enthält, die der Prozess aufrufen kann. CORBA verwendet außerdem die Interoperable Object Reference (IOR), welche alle Informationen
beinhaltet um einen Kommunikationskanal zu einem Objekt aufzubauen. Diese beinhalten:

- Art des Kommunikationsprotokolls
- Angaben zum Ort des Objektes, wie im Fall von TCP/IP eine IP-Adresse und ein Port
- Vorgaben zur Darstellung von Text und Daten, wie Codepages, Endian, Verschlüsselung oder Kompression der Daten
- Angaben zur Identität des Objekts im Zielsystem

Um eine IOR zu erhalten, kann der Client den CORBA Naming Service verwenden.
Es gibt globale und locale object references.
Global References werden bei implizitem Binding verwendet. Explizites Binding verwendet sowohl globale als auch lokale Referenzen. Hier muss das verteilte Objekt manuell an den lokalen Proxy gebunden werden.

Während RMI Java spezifisch ist, muss es sich nicht um sprachenunabhängige Darstellungen von Objekten und somit auch von Objektreferenzen kümmern. CORBA hingegen (siehe 10.4.1 im Buch) hat eine andere Darstellung. Für CORBA hat eine Darstellung einer Referenz immer nur im Kontext des speziellen Prozesses eine Bedeutung. Man kann also nicht so einfach eine Referenz von C++ auf ein Java System übergeben. Deshalb hat CORBA die IORs eingeführt, um eine system- und sprachenunabhängige Darstellung von Objekten und Referenzen zu erreichen.

Der Overhead an Komplexität, den CORBA aufgrund seiner Sprachunabhängigkeit mit sich bringt, entfällt dann.

16. Was versteht man unter "static" und "dynamic" Invocation von verteilten Methoden? Geben Sie Beispiele an.

Static Invocation setzt voraus, dass die Schnittstellen eines Objektes bereits bekannt sind. Eine Änderung erfordert eine Neukompilierung am Client.

Bsp. MyObject.doSomething(in1, in2, out1, out2)

Dynamic Invocation erlaubt es den Namen der Methode des verteilten Objekts, die es aufrufen will, zur Laufzeit zu bestimmen.

Bsp. invoke(MyObject, id(doSomething), in1, in2, out1, out2).
Id() bezeichnet hier den Look-up nach dem Methodennamen (z.B. aus einem Directory Service).

Anwendungen von dynamic Invocation:
object browser, run-time inspection, batch processing, frameworks

17. Wie funktioniert die Parameterübergabe bei RMI? Gehen Sie dabei auf jene Eigenschaften der Objektorientierung ein, welche den Vorteil von RMI gegenüber RPC bewirken.

In Java war es wichtig, ein hohes Maß an Verteilungstransparenz zu erreichen und Remote Calls soweit wie möglich wie lokale aussehen zu lassen.
Das Prinzip der Stubs bleibt unverändert. Der Client-Stub wird hier als Proxy, der Server-Stub als Skeleton bezeichnet. Der Client übergibt bei RMI lokale Objekte als Methodenparameter dem Proxy mittels Copy/Restore, entfernte Objekte hingegen können als Referenz übergeben werden → Entspricht den Eigenschaften der OO. Eine Referenz auf ein Verteiltes Objekt beinhaltet folgende Informationen:

- Netzwerkadresse
- Endpunkt des Servers
- lokaler Bezeichner des echten Objekts (nur von Server verwendet)

In einem RPC werden alle Methodenparameter immer mittels Copy/Restore Variante übergeben.

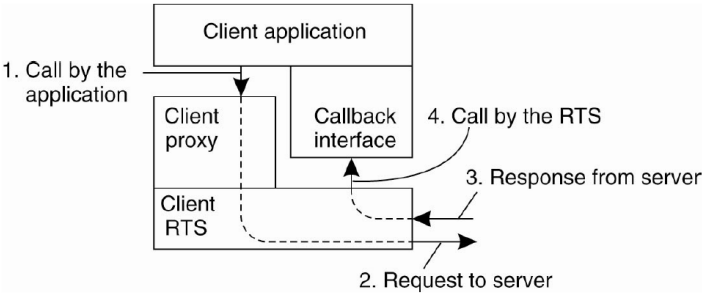
In beiden Fällen werden die Parameter schlussendlich vom Proxy gemarshalled und an den Server-Stub/Skeleton gesendet.

mehr infos => siehe p.460 Kapitel 10.3.3 (lokale Objekte können kopiert werden, remote Objects per Reference übergeben)
Unterschied zu RPC ist, wenn Referenzen zu einem wiederum entfernten Objekt übergeben werden, wird diese nicht durch Copy/Restore übergeben sondern wieder als Referenz, um die sich dann der Server weiter kümmert.

18. Erläutern Sie die Grundprinzipien (Kategorien) von Message-orientierter Kommunikation und gehen Sie auf CORBA Messaging exemplarisch ein. Beschreiben Sie zwei unterschiedliche Methoden, wie asynchrone Methodenaufrufe in CORBA Messaging erfolgen können.

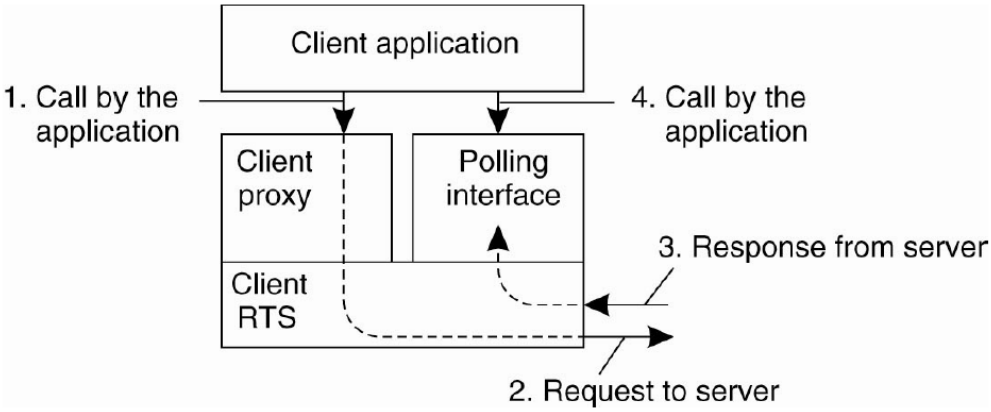
Message Orientierte Kommunikation basiert auf Nachrichten, die zwischen den einzelnen Systemen hin und hergeschickt werden und bieten Unterstützung für persistente/transiente, synchrone/asynchrone Kommunikation (MoM (Message oriented Middleware): persistent asynchron durch Queues; Message Passing Interface: transient synchron/asynchron).
Wichtig zu wissen bei den Queues ist folgendes: es gibt im Prinzip trotzdem keine Garantie, dass eine Message jemals gelesen wird, auch wenn sie in die Queue aufgenommen ist. Es kann also sein, dass es nie zu dem kommt. Dies ermöglicht aber genauso eine sehr lose Kopplung von Software!
CORBA Messaging verbindet Methodenaufrufe und objektbasierte Nachrichtenübermittlung miteinander. CORBA stellt dafür prinzipiell einen Nachrichtendienst zur Verfügung, über den persistente asynchrone Kommunikation trotz konsequenter objektbasierter Handhabung möglich wird.
Es gibt 2 Formen eines asynchronen Aufrufs:

- Callback-Model: Zu jeder ursprünglichen Schnittstelle werden 2 neue Schnittstellen erzeugt, wobei eine dem Aufruf dient (Parameter enthalten keine Return Werte), und eine als Callback fungiert (Parameter sind Rückgabewerte der urspr. Schnittstelle). Sobald eine Antwort eintrifft, wird der Callback mit den Return Werten des Aufrufs ausgelöst. (Resultat der Invocation wird automatisch an Anwendungsebene weitergegeben, sobald sie verfügbar ist). Der Client muss eine Implementierung des Callbacks zur Verfügung stellen.



CORBA's callback model for asynchronous method invocation.

- Polling-Model: Zu jeder ursprünglichen Schnittstelle werden 2 neue Schnittstellen erzeugt, wobei eine dem Aufruf dient (Parameter enthalten keine Return Werte) und eine Schnittstelle (Parameter sind Rückgabewerte der urspr. Schnittstelle), welche es der Anwendungsebene erlaubt, eingegangene Nachrichten (Resultate der Invocation) abzufragen. Dies muss von der Anwendungsebene explizit angestoßen werden.



CORBA's polling model for asynchronous method invocation.

Beide Methoden können aus der ursprünglichen Schnittstelle automatisch generiert werden. In beiden Fällen bleibt es dem Client überlassen, sich für synchrone oder asynchrone Kommunikation zu entscheiden und berührt den Server nicht.

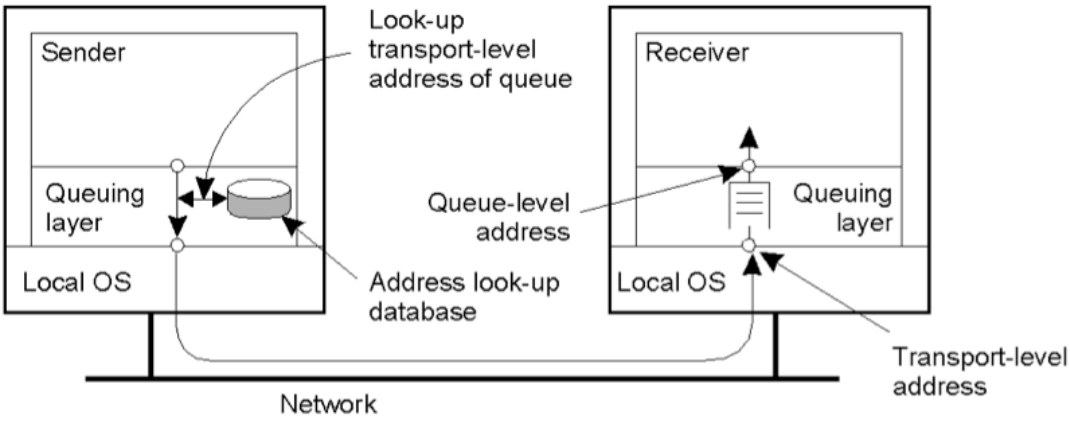
19. Was versteht man unter "Message-oriented Middleware MoM"? Erläutern Sie Modell und Architektur solcher "Message-Queueing"-Systeme. Erklären Sie die Primitivoperationen Put, Get, Poll und Notify eines Message-Queuing Systems. Diskutieren Sie Einsatzzwecke sowie Vor- und Nachteile - gehen Sie insbesonders auf den Begriff des Message Brokers und dessen Bedeutung für EAI ein.

MoM bieten Unterstützung für persistente asynchrone Kommunikation. Anwendungen kommunizieren, indem sie ihre Nachricht in eine Message Queue einstellen. Jede Anwendung hat ihre eigene Queue von der sie liest.
Ein Sender einer Nachricht erhält nur eine Garantie, dass die Nachricht irgendwann in die Queue eingefügt wird.
Vorteil: Empfänger kann passiv sein während Sender sendet und umgekehrt (loosly-coupling).
Für das loosely-coupling gibt es 4 Kombinationen:

- Sender running - Receiver running
- Sender running - Receiver passive
- Sender passive - Receiver running
- Sender passive - Receiver passive

Nachteil: Übertragung kann sich im Minutenbereich abspielen.

Architektur:



- Get: blockt solange die Queue leer ist; gibt ansonsten die erste Nachricht zurück
- Put: hängt eine Nachricht an eine Queue an
- Poll: Queue nach Nachrichten durchsuchen, gibt die erste Nachricht zurück. Blockt niemals.
- Notify: installiert einen Handler, der aufgerufen wird, wenn eine Nachricht in die Queue gestellt wird.

MoM erlaubt es loosly-coupled Systems zusammenzuarbeiten. Unterschiedliche Systeme kommunizieren möglicherweise über unterschiedliche Nachrichtenformate, die von anderen Systemen nicht verstanden werden.
Ein Message Broker (MB) wird verwendet um Nachrichten von einem Format in ein anderes zu konvertieren. Des Weiteren kann ein MB auch als application-level gateway eingesetzt werden, der die Konversationen zwischen Anwendungen managet → enterprise application integration - zB könnte hier auch folgendes geschehen: der MB könnte Messages an interessierte Clients weiterleiten, wenn diese daran interessiert sind (wie zB RSS Feeds) - dh, diese Clients werden benachrichtigt, wenn es was Neues gibt.

20. Erklären Sie "stream-oriented communication". Was ist "QoS" und inwiefern ist es für stream-oriented communication von Bedeutung?

Ein Datenstream ist nichts weiter als eine Folge von Dateneinheiten. Sie können sowohl auf kontinuierliche (Audio, Video), als auch auf diskrete Medien angewendet werden. Die zeitliche Abfolge ist für das Streaming von größter Bedeutung, damit die Daten rechtzeitig beim Client eintreffen.

- asynchron: keine zeitliche Beschränkung (Dateidownload)
- synchron: maximale End-to-End Verzögerung (Sensorenbeispiel: ein Sensor liefert in einer bestimmten Rate Signale. Diese müssen an den Empfänger übertragen werden. Wenn die Übertragung schneller ist als die Erfassungsrate, ist es ok, wenn sie langsamer ist als die Erfassungsrate, dann gibts Probleme)
- isochron: minimale und maximale End-to-End Verzögerung (Audio / Video Stream): hier muss die Übertragung zeitgerecht erfolgen. Siehe Audio + Video Stream - beide sollten die gleiche Verzögerung haben, sonst stimmt das Endergebnis nicht zusammen.

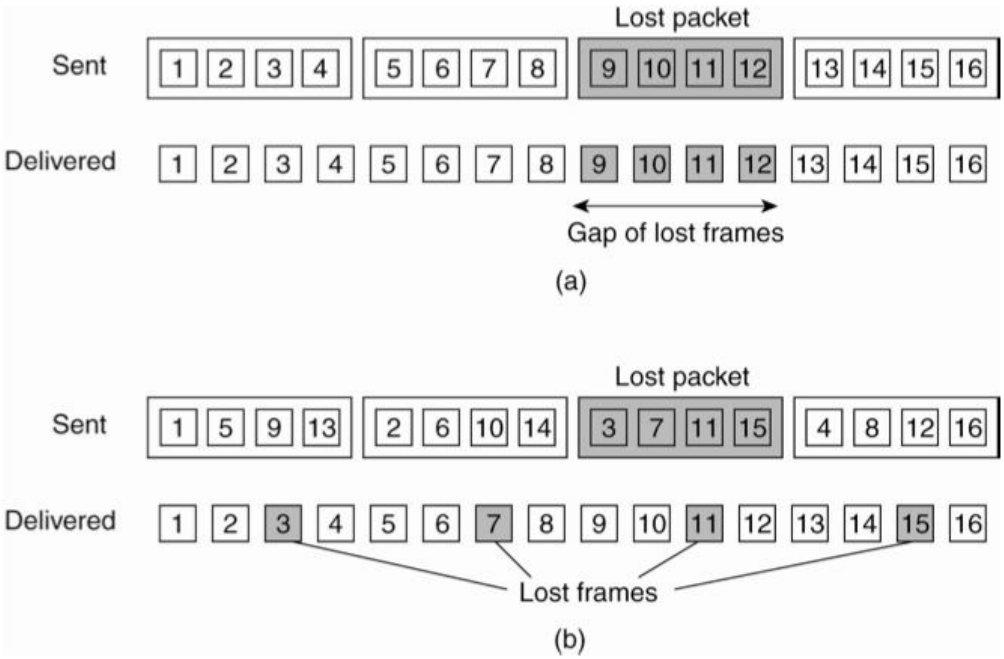
Ein einfacher Stream besteht aus einer Datenfolge (z.b. Mono Audio).
Ein komplexer Stream beinhaltet mehrere Substreams die in zeitlicher Abhängigkeit zueinander stehen (Stereo-Audio, Video mit Tonspuren).

QoS (Quality of Service)= Anforderung an zeitliche Steuerung. Solche Anforderungen beschreiben zB was von einem darunterliegenden VS und Netzwerk erwartet wird um sicherzustellen, dass bestimmte Dienste noch korrekt ausgeführt werden können.

- Erforderliche Bit-Rate
- max. Verzögerung bei Session beginn
- max. End-to-End Verzögerung
- max. Jitter (Verzögerungsvarianz)
- max. Umlaufverzögerung

Um Jitter zu vermeiden, wird ein Buffer verwendet. Somit können kleine Unregelmäßigkeiten im Datenstrom ausgeglichen werden.

Wenn ein Packet mehrere (Audio und/oder Video) Frames enthält und verloren geht, dann kann eine große Lücke beim Abspielen entstehen. Durch verschachtelte Rahmen (interleaved transmission) kann dieser Effekt reduziert werden. Dafür muss aber eine größere Start Verzögerung in Kauf genommen werden.



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)

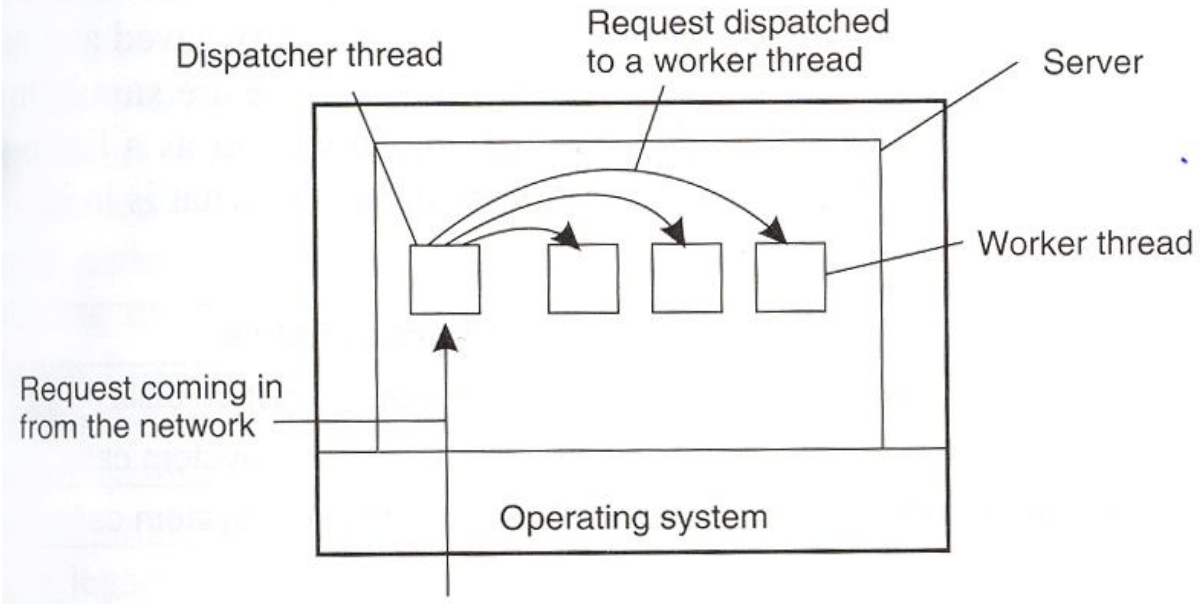
Es muss jedoch darauf geachtet werden, dass ein blockierender Thread nicht den ganzen Prozess lahm legt, dies kann entweder durch die Vermeidung von System calls erreicht werden, oder durch die Verwendung von z.B. LWP's (lightweight processes).
Worauf bei Multithreading geachtet werden muss:
- Safety — 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)

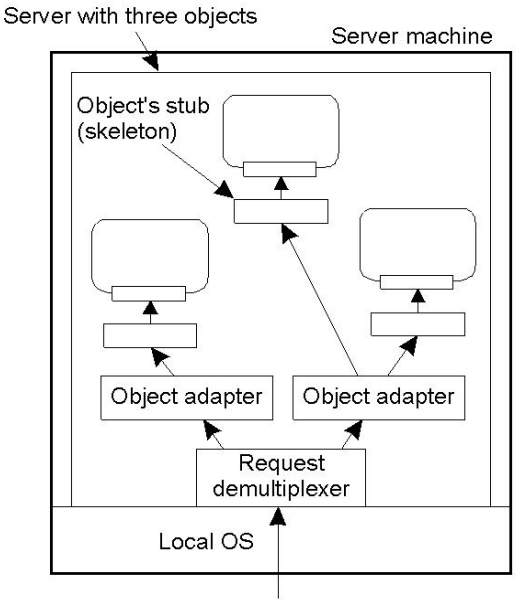
Arten von Servern:
- multithreaded server: bestehend aus einem dispatcher und mehreren worker threads. Der dispatcher thread wartet auf eingehende anfragen, und startet pro Anfrage einen worker thread, an welchen die Anfrage weitergereicht wird. Somit ist es möglich, dass während eine Anfrage bearbeitet 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.

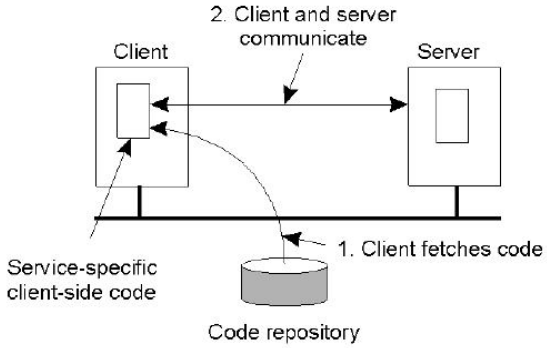


25. Erläutern Sie die wichtigsten Aspekte der Code Migration. Erklären Sie "strong mobility" und "weak mobility" und geben Sie für "weak mobility" ein Beispiel an.
Bei der Code Migration werden nicht nur Daten, sondern ganze Programme verschickt, dies ist sogar im laufenden Zustand möglich. – wird aus 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,..)

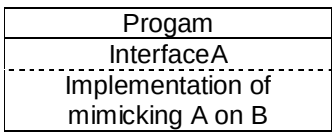
Weak Mobility: Hier wird nur das Code Segment transportiert, möglicherweise mit Initialisierungsdaten. Es ist hier nur möglich das Programm an einer, von möglicherweise mehreren, vordefinierten Positionen zu starten, wie z.B. bei Java Applets, die immer beim Begin gestartet werden. Der Vorteil dieser 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.

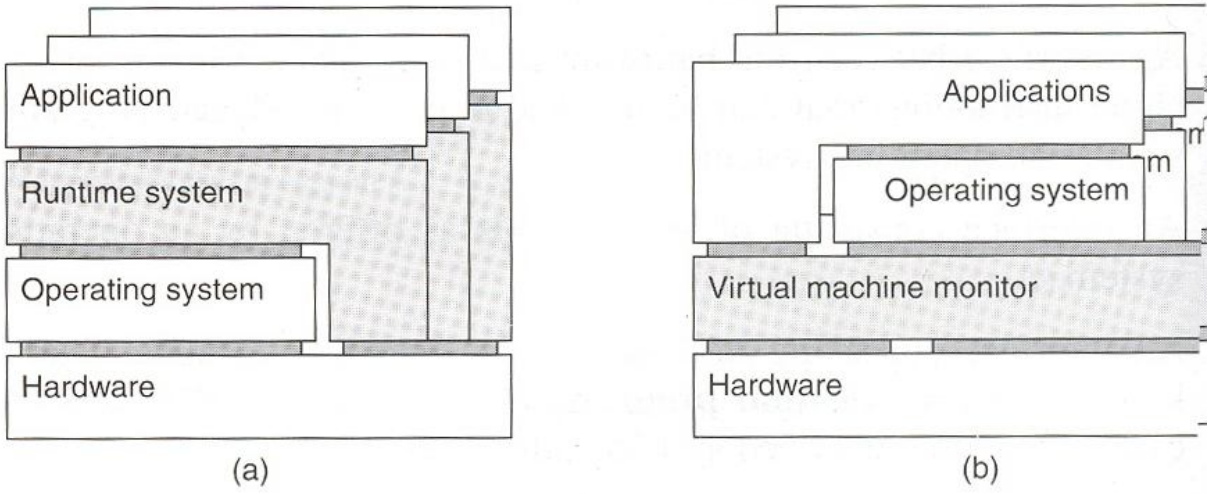
Vorteile:

- 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).

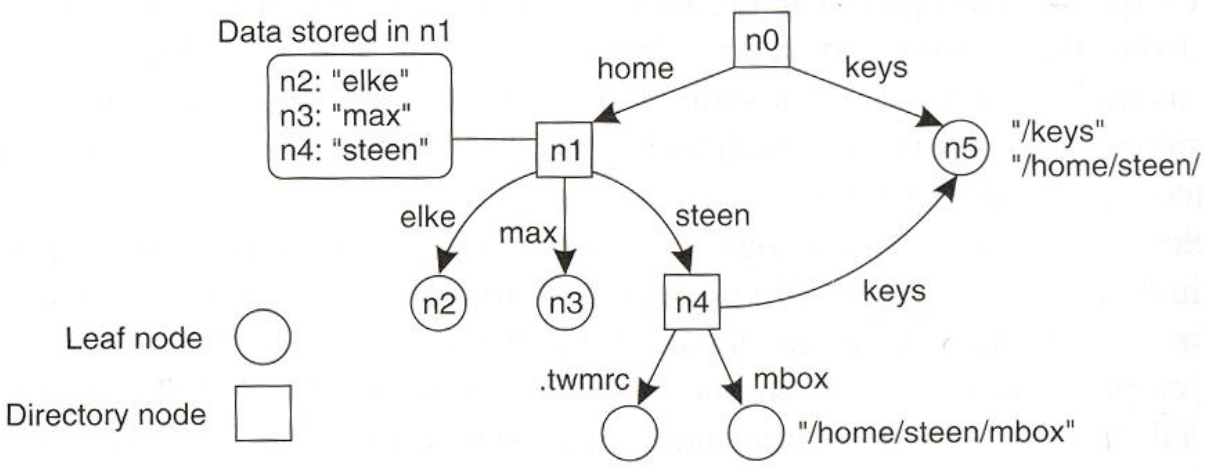


Figure 5-9. A general naming graph with a single root node.

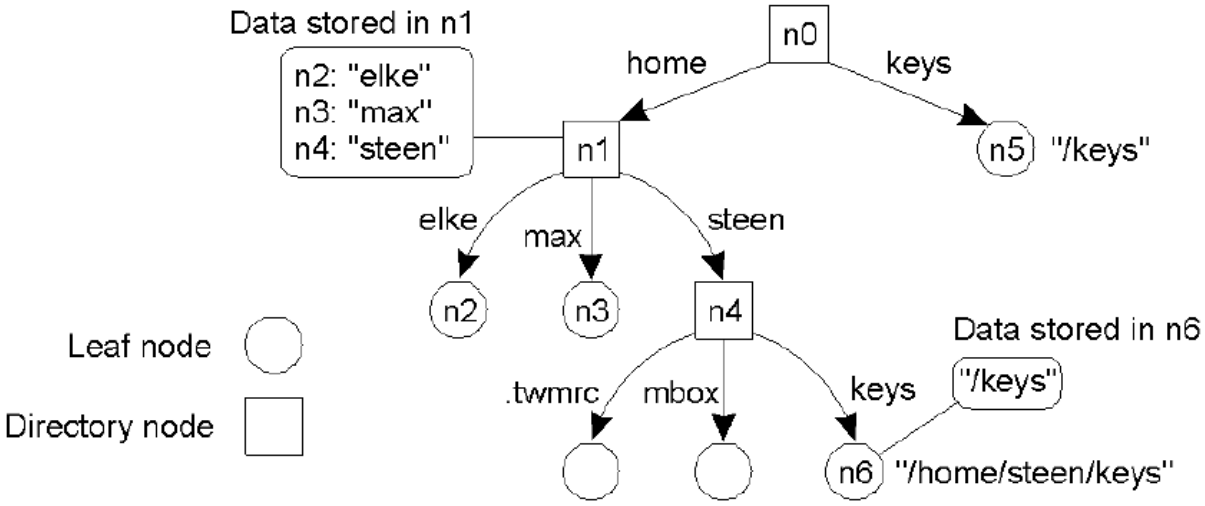
Name Space: Namen werden in einem Verteilten System mithilfe eines Namespace organisiert. Ein Name identifiziert ein Objekt. Zur eindeutigen Zuordnung ist jedoch der entsprechende Kontext – eben der Namensraum zu beachten. Hierarchische Namen können als beschriftete gerichtete Graphen dargestellt werden:

- Blattknoten = benannte Entität
- Verzeichnisknoten (hat ausgehende Kanten): speichert eine Verzeichnistabelle → Kantenbeschriftung, Knotenbezeichner Paare

Durch **Aliases** können mehrere Namen auf die gleiche Entität zeigen.

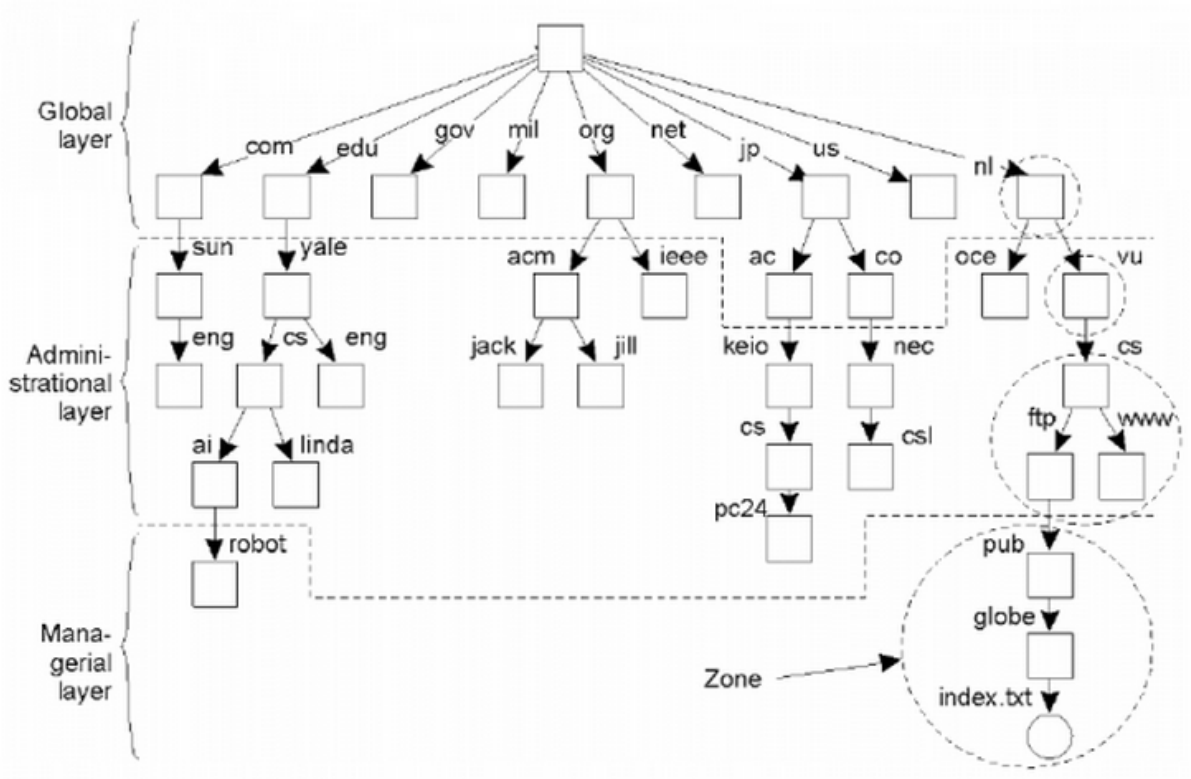
- Hardlinks: mehrere absolute Pfade verweisen auf den selben Knoten im Graphen
- Symbolic link: speichern Pfad zu einer Entität

Durch **Mounting** können Namensräume miteinander kombiniert werden. (benötigt: access protocol, server, mounting point).



Closure Machanismus: zu wissen, wie und wo die Namensauflösung beginnen soll → Auswahl des ersten Knotens.
Bsp. Unix: um absoluten Verzeichnispfad (/users/home) aufzulösen, muss dem Dateisystem der Root-Knoten „/“ bekannt sein. Der tatsächliche Offset des Root-Knotens ist im Superblock des logischen Laufwerks kodiert.
Bsp. DNS: Der Closure Mechanismus bei DNS sind die IP-Adessen der 13 DNS-Root-Server

29. Erklären Sie die Schichten der Verteilung von Name Spaces. Erläutern Sie die Einsatzmöglichkeiten von Replication und Caching in den verschiedenen Schichten. Erklären Sie verschiedene (hierarchische) Möglichkeiten der iterativen/rekursiven "name resolution".



Schichten:

Globale Schicht

- wenige Knoten der höchsten Ebene (Root-Knoten und seine Kind-Knoten)
- sehr stabil, da Änderungen sehr selten sind
- können Organisation oder Gruppen von Organisationen abbilden (z.B. com, org, Länder: at, de)
- geographische Ausdehnung: weltweit
- Reaktionszeit im Sekundenbereich
- Viele Replikate (da wenig Änderungen leicht zu replizieren)
- Effektives Caching durch Clients (da wenig Änderungen zu erwarten sind)

Administrations Schicht

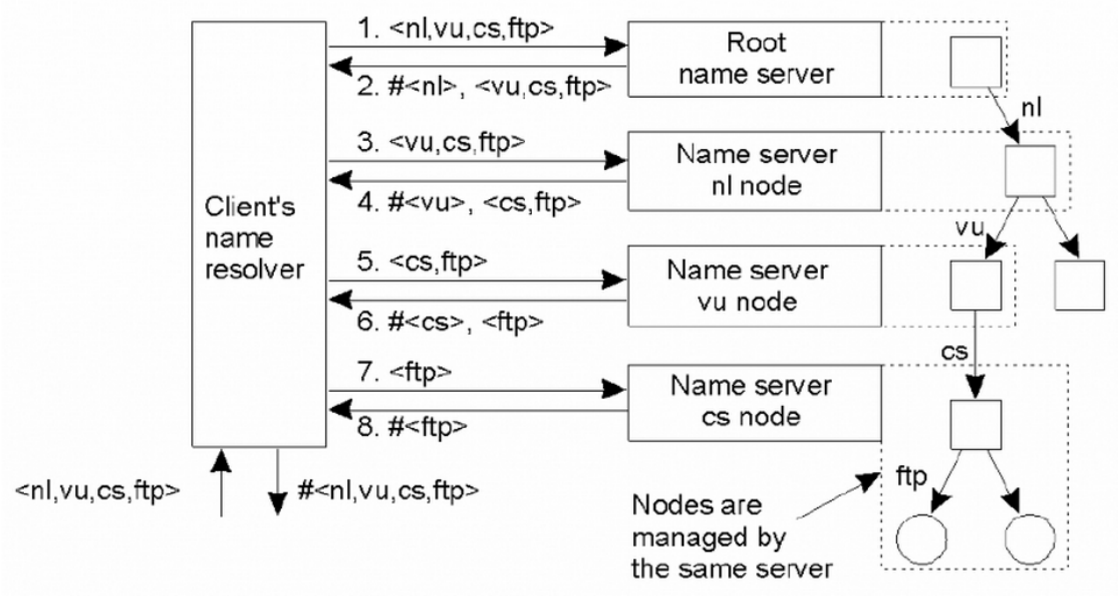
- Knoten die innerhalb einer Organisation verwaltet werden
- Besteht aus vielen Knoten (im Vergleich zur globalen Schicht)
- Reaktionszeit im Millisekundenbereich
- keine bis wenige Replikate
- Caching durch Clients möglich

Management Schicht

- Knoten die innerhalb einer Abteilung verwaltet werden
- besteht aus zahllosen Knoten
- Reaktionszeit: sofort
- keine Replikate
- Caching durch Clients manchmal möglich

iterative Namensauflösung:

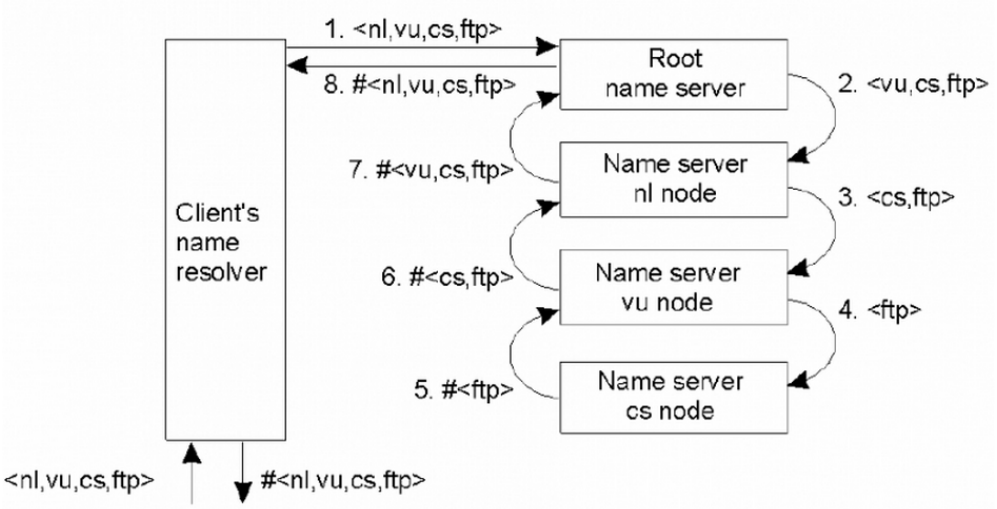
1. der Resolver übergibt den vollständigen Namen an Root-Server.
2. Root-Server löst Name soweit wie möglich auf und gibt als Zwischenergebnis die Adresse des nächsten NS und dem verbleibenden Pfadnamen an den Client/Resolver zurück.
3. Client fragt den retournierten NS nach dem verbleibenden Pfadnamen.
4. Schritt 3 wiederholt sich solange bis der gesamte Namen aufgelöst wurde.



Rekursive Namensauflösung:

1. der Resolver übergibt den vollständigen Namen an den Root-Server.
2. Wenn der NS den Namen nicht vollständig auflösen kann (oder aus dem Cache lesen kann) fragt er automatisch den nächsten NS ohne ein Zwischenergebnis an den Client zu senden.
3. Sobald ein NS den Namen vollständig aufgelöst hat, wird das Endergebnis an den übergeordneten NS weitergegeben.
4. Der Root-Server übergibt das vollständige Resultat an den Client.

- Vorteil: effektiveres Caching, geringere Kommunikationskosten
- Nachteil: benötigt mehr Leistung am Server



Daher: Globale Schicht → iterative, Administrationsschicht → rekursiv

Caching steigert die Geschwindigkeit der Namensauflösung bei einem Cachehit und reduziert die Netzwerklast.

Replikation verbessert die Availability, steigert die Geschwindigkeit durch „nähere“ Replikate und reduziert Hot-Spots.

30. Erläutern Sie das Domain Name System DNS, sowie den Ablauf bei der Namens-Auflösung anhand der DNS Database (Ressource Records). Was ist reverse lookup? Was ist ein zone-transfer?

DNS ist das Namesystem für das Internet. Es wird verwendet um Domainnamen auf IP-Adressen zu mappen. Eine Domain kann man sich als „Unterbaum“ eines hierarchischen Graphen vorstellen. Pfadnamen nennt man „Domainname“.

Knoten speichern ihre Daten in Ressource Records. Ressource Record ist die kleinste Informationseinheit im DNS. Ausgewählte Ressource Record Typen:

- SOA → Start of Authority: Informationen zur dargestellten Zone
- A → IP eines Hosts
- MX → Mailserver
- NS → Name eines NS der die dargestellte Domain implementiert
- CNAME → kanonischer Name des Hosts (für reverse lookup)
- PTR → symbolischer Link auf Primären Namen des dargestellten Knoten

Caching und Replikation werden zur Steigerung der Effektivität verwendet.

Zone-Transfer = Zonenübertragung (Übertragung von Ressource Records) auf einen anderen NS also z.B. von Primary NS auf Secondary NS. Dies dient der Ausfallsicherheit und Performance.

(Ein Secondary NS kann ein Primary NS einer anderen Zone sein. Beide sind authorative).

Reverse-lookup: wird benötigt um von einer IP auf einen Domainname zu schließen. Dazu wurde eine eigene Domain (mit 3 (Ebenen von) Subdomains) begründet: in-addr.arpa-Domäne.
Somit lassen sich z.B. alle IP Adressen die mit 193 beginnen in der 193.in-addr.arpa Domäne finden.

31. Was ist Directory Service bzw. "Attribute-based naming"? Beschreiben Sie den prinzipiellen Aufbau des X.500 Name Space sowie dessen LDAP Implementierung.

Directory Service bzw. Attribute-based naming
- Attributbasierte Namensgebung
- Bietet einen Suchdienst nach Entitäten über deren Attribute
- Suche einer Entität über (Attribut,Wert)-Paare.
- z.B. Wildcards Suche

X.500 (basierend auf OSI) ist ein hierarchisch organisiertes Directory Service. Die Benennung einer Entität basiert auf deren Attribut-Wert Paaren (=attributbasierte Benennung).
Es gibt einen einzigen Directory Information Tree (DIT) welcher Einträge hierarchisch speichert (Verteilung über mehrere Server möglich). Jeder Eintrag kann mehrere Attribute-Wert-Paare speichern.

Jeder Eintrag hat einen eindeutigen Distinguished Name, der eine Kombination aus seiner eigenen RDN (Relative Distinguished Name) und der RDN all seiner übergeordneten Einträgen ist.

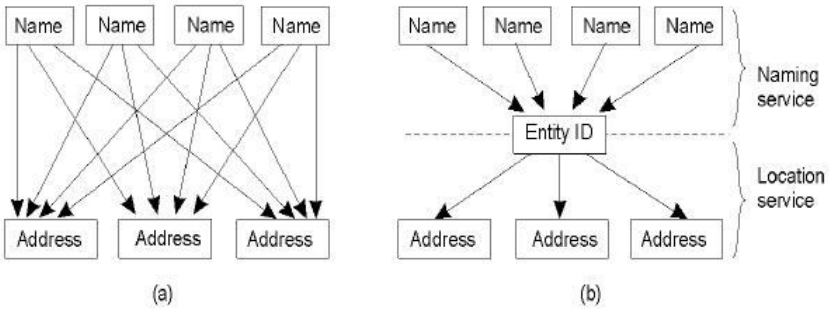
Vorteil: effektives Suchen wird möglich (Besucher gibt Beschreibung dessen ein, was er sucht).

LDAP (Lightweight Directory Access Protocol) ist eine effizientere, einfachere Implementierung auf TCP Basis. Verwendung: z.B. zum Beschreiben von Zugriffsberechtigungen für Benutzerauthentifizierung (MS Active Directory), oder als Verzeichnis für Personen und E-Mailadressen.

32. Wie funktioniert in flachen Namensräumen das Location Service? Geben Sie das Grundprinzip möglicher Lösungen an und gehen Sie dabei auch auf die Begriffe Mobility und Discovery ein. Erläutern Sie Vor- und Nachteile bei der Verwendung von "Forewarding Pointers". Wie funktionieren "Home-based approaches" für mobile Geräte?

Ein Location Service bezeichnet ein Two-Level mapping zum Übersetzen von Namen in Adressen über sogenannte Identities. Es dient dazu, Services zu registrieren und aufzufinden. Beispielsweise bei P2P Netzen, müssen Rechner Services und Ressourcen ankündigen bzw. Ressourcen anderer Rechner auffinden können. Abgrenzung zu Discovery Services: Bei Location Services ändern sich die Anforderungen oft und schnell. Zudem wird ein hoher Grad an Mobility erreicht, da ein Host nicht mehr über eine fixe Adresse (IP) angesprochen wird.

Lösungsansätze



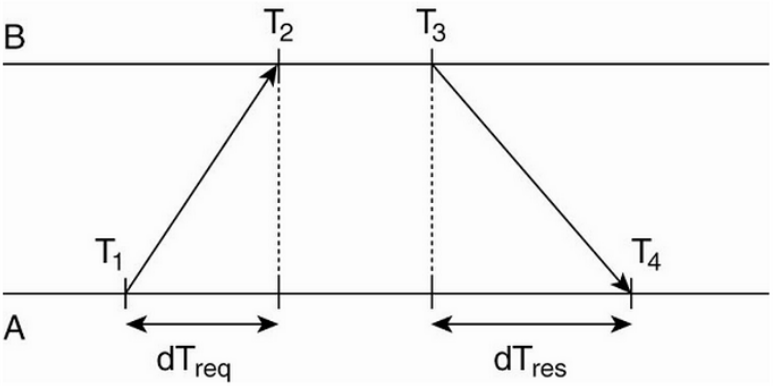
a) Direct, single level mapping between names and addresses.
b) Two-level mapping (e.g., using identities).
Location service maps identifier to address.

Ein Ansatz um Mobility zu erreichen ist die Verwendung von Forwarding Pointers. Wenn eine Entität seinen Ort ändert, lässt sie eine Referenz zurück, welche auf ihren neuen Ort zeigt.
Vorteil: sehr einfach
Nachteil: Lange Ketten, möglicherweise viele Zwischenschritte notwendig, eine falsche Referenz unterbricht gesamte Kommunikation (broken link), daher müssen die Ketten so kurz wie möglich gehalten werden.

Ein Ansatz um Mobility in großen Netzwerken zu erreichen ist die Verwendung von Homebased Approaches. Der Home-Agent speichert dabei immer den aktuellen Standort einer Entität. Ändert das mobile Gerät (Entität) ihren Standort, registriert sie eine Care-Of-Address beim Home-Agent. Ein Client, der den aktuellen Standort nicht kennt, kann ihn also immer beim Home Agent erfragen und anschließend direkt kommunizieren.

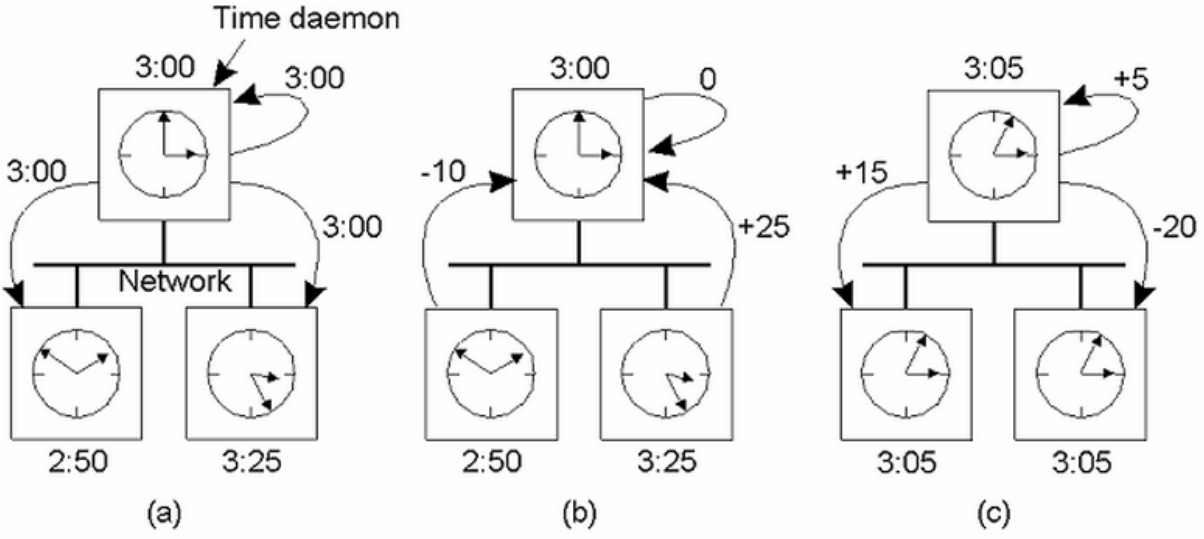
33. Wozu braucht man Uhrensynchronisation? Erläutern Sie das NTP und den Berkeley Algorithmus. Was ist die Problematik bei der Synchronisation von Physical Clocks?

NTP (Network Timing Protocol):
Es handelt sich dabei um eine Synchronisation mit einem **externen Zeitgeber**. Die Genauigkeit von Zeitgebern wird durch das sogenannte Stratum definiert. Je niedriger es ist, desto genauer ist eine Zeitquelle. Beim NTP wird ein höheres Stratum mit einem niedrigeren synchronisiert. Ein Rechner sendet eine Nachricht an den Zeitserver und merkt sich den Sendezeitpunkt t1. Der Server zeichnet sich den Emfangszeitpunkt t2 sowie den Sendezeitpunkt der Antwort (kurz davor) t3 und schickt die Antwort. Der Rechner zeichnet den Empfangszeitpunkt t4 auf. (t1 und t4: lokale Zeit; t2 und t3: Serverzeit) Berechnung der einfachen Übertragung: ((t4 - t1) - (t3 - t2)) / 2. Somit kann durch t4 - t1 einfaches Delay darauf geschlossen werden, um wie viel die Zeit des Senders nach- oder vorgeht. Korrektur: beschleunigen oder bremsen der lokalen Uhr. Dieser Vorgang wird 8-mal wiederholt und das Ergebnis mit dem kürzesten Delay verwendet. Delay über einem Grenzwert --> Zeitquelle nicht vertrauenswürdig.



BERKELEY ALGORITHMUS

Dabei handelt es sich um eine interne Synchronisation. Es gib einen Koordinator (Time daemon), der seine lokale Zeit an alle sendet (auch an sich selbst). Jeder antwortet mit seiner Abweichung von dieser Referenzzeit. Danach berechnet der Koordinator die durchschnittliche Abweichung und sendet sie jedem zurück. Diese beschleunigen oder verlangsamen dann ihre Zeit entsprechend der Korrektur



Was ist die Problematik bei der Synchronisation von Physical Clocks?

- Die Uhr wird dann langsam vorgestellt, jedoch NIEMALS zurückgestellt
  - ➔ zurückstellen ist problematisch wegen Zeitstempel. Zeitstempel müssen immerlinear vorwärts gehen.
- Antwortzeiten über Netzwerk unterschiedlich

34. Was sind die Gründe für die Verwendung von Logical Clocks? Erklären Sie die Unterschiede zu den Physical Clocks. Was ist die "happened-before" Beziehung und wie funktionieren die "Lamport-Timestamps"?

Verwendung von Logical Clocks: Es ist meist ausreichend, wenn alle dieselbe Zeit haben, auch wenn diese nicht mit der realen Zeit zusammenpasst. Es ist oft nur die Reihenfolge der Operationen wichtig.

Unterschiede: Die physikale Uhr ist die "reale" Uhr (Kristall-Oszillator, mittlere Sonnensekunde, Atomuhr (TAI, International Atomic Time), UTC (Universal Coordinated Time), wobei eine Logische Uhr eine Reihung der Ereignisse darstellt.

"happend-before"-Beziehung: a -> b wird gelesen als "a passiert vor b". Dies bedeutet, dass alle Prozesse wissen, dass a vor b eingetreten ist. Zwei Situationen:

- wenn a und b im selben Prozess sind, und a tritt vor b ein dann ist "a->b" = true
- wenn a das Ereignis darstellt, dass eine Nachricht von einem Prozess gesendet wird, und b ist das Ereignis, dass die Nachrricht von einem anderen Prozess empfangen wird. Eine Nachricht kann nicht empfangen werden, bevor sie gesendet wird (und auch nicht gleichzeitig).

Dies ist eine transitive Relation: a -> b und b -> c, dann a -> c

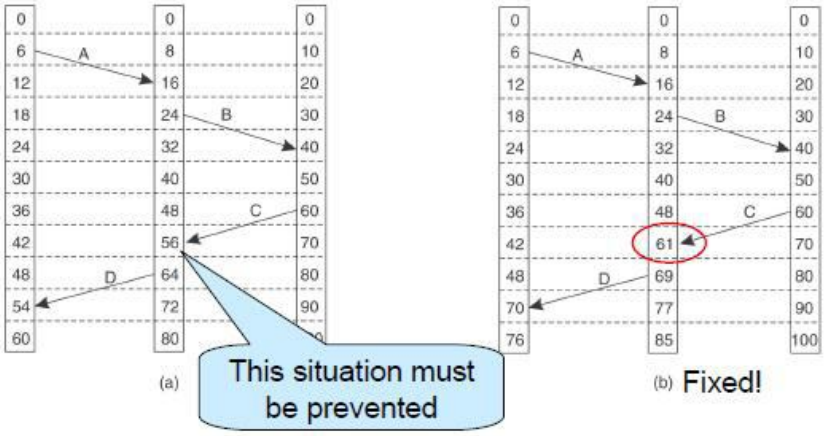
Lamport-Timestamps: Ereignisse werden nur durchnummeriert. Potentiell kausal abhängige Ereignisse (im Sinne der "happend-before"-Beziehung) bekommen dabei Nummern so zugewiesen, dass das abhängige Ereignis eine größere Nummer hat als die Ursache, und dass diese Nummern bei allen Prozessen gleich sind. Jeder Prozess hat seinen eigenen Counter (logische Uhr). Bei jedem Ereignis (Berechnung, Senden, Empfangen) wird er um 1 erhöht und hängt diese Zahl der Nachricht an. Empfänger setzt seinen Counter auf das Maximum aus seinem aktuellen Counter und der mitgeschickten Zahl, anschließend erhöht er seinen Counter um 1. Es muss immer gelten:

- T(b) > T(a), falls a und b in dieser Reihenfolge im gleichen Prozess stattfinden

Eigenschaften:

- 2 aufeinanderfolgende Ereignisse a und b im gleichen Prozess: T(a) < T(b)
- Der Empfang einer Nachricht hat immer höhere Zahl als das Senden
- Die Relation ist transitiv abgeschlossen

Es kann jedoch nicht auf kausale Abhängigkeiten geschlossen werden, d.h. T(a) < T(b) bedeutet nicht unbedingt dass a->b gilt (nur in die andere Richtung).



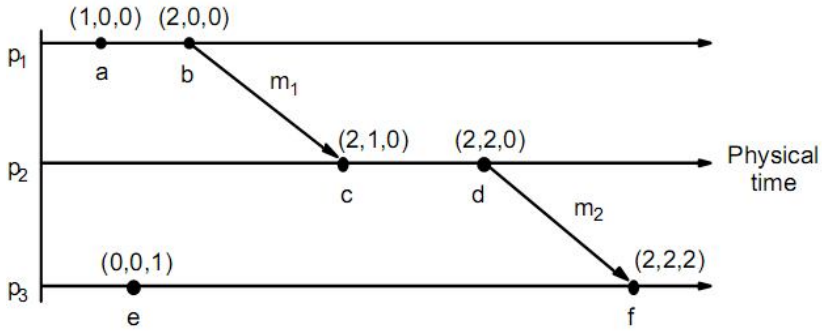
35. Welchen Nachteil haben die Lamport-Timestamps und wie kann dieser durch Vector-Timestamps überwunden werden?

Nachteil: keine kausalen Abhängigkeiten. Nur weil T(a) < T(b) gilt sagt das noch nicht aus, dass a auch vor b stattgefunden hat. Es kann sich ja um Zeiten von 2 oder mehreren verschiedenen unabhängigen Prozessen handeln.

Vector-Timestamp:
Ein Vector Timestamp (VT oder Vector Clock VC) hat die Eigenschaft, dass wenn VC(a) dem Event a zugeordnet ist, dann hat dieser Vektor die Eigenschaft dass wenn VC(a) < VC(b) für ein Event b ist, a kausal vor b ausgeführt worden ist.
Jeder von n Prozessen hat einen Vektor von der Länge n (initialisiert mit dem Null-Vektor). Bei jedem Ereignis in Prozess i zählt dieser Element i um 1 hoch. Der Vektor wird an die gesendete Nachricht angehängt. Beim Empfang durch Prozess j wird das Weitergeben der Nachricht an die Applikation solange verzögert bis gilt (m ist die Message; m[i] beschreibt den Counter des Prozesses i im Vector der Message):

- m[i] = Vj[i] + 1
- m[k] <= Vj[k] für alle k außer i

Potentiell kausal abhängige Nachrichten können durch elementweisen Vergleich der Vektoren gefunden werden. Es handelt sich dabei um eine partielle Ordnung der Ereignisse.



36. Wie funktioniert Distributed Mutual Exclusion. Wie verhalten sich verschiedene Algorithmen (centralized, distributed, token-ring) hinsichtlich Skalierbarkeit und Fehlertoleranz?

Das Ziel von Distributed Mutual Exclusion ist, in einem Verteilten System den exklusiven Zugriff auf Ressourcen zu vergeben, es soll verhindert werden, dass 2 Prozesse gleichzeitig auf dieselbe Ressource zugreifen. Dazu werden mehrere Ansätze verfolgt:

- **token ring**: Ein Token wird zwischen den Prozessen weitergereicht, nur der Prozess, der das Token hält, ist zum Zugriff auf die Ressource berechtigt. Benötigt ein Prozess keinen Zugriff, so reicht er einfach das Token zum nächsten Prozess weiter. (Vorteile): Einfache Implementierung; keine Starvation (jeder Prozess kommt an die Reihe); keine Deadlocks (Nachteile): stürzt der Prozess ab, der das Token hält, geht es verloren, es ist schwer zu erkennen wann ein Token verloren gegangen ist; auch wenn kein Prozess auf die Ressource zugreifen will, muss trotzdem ständig das Token weitergereicht werden.
- **centralized**: ein Prozess agiert als Koordinator, er erhält die Anfragen der anderen Prozesse und gewährt oder verwehrt den Zugriff auf die Ressource, sodass immer nur ein Prozess zur selben Zeit mit der Ressource arbeitet. Ist dieser Prozess mit seiner Arbeit fertig, teilt er dies dem Koordinator mit. Falls ein Prozess eine Anfrage stellt und die Ressource in Verwendung ist, kann der Koordinator die Anfrage in einer Warteschlange zwischenspeichern. (Vorteile):hohe Effizienz; Einfache Implementierung; keine Starvation (jeder Prozess kommt an die Reihe); keine Deadlocks (Nachteile): Der Koordinator ist ein Single-point of failure und ein bottleneck
- **distributed**: Wenn ein Prozess exklusiven Zugriff auf eine Ressource haben will, so schickt er eine Nachricht an alle anderen Prozesse (einschließlich sich selbst). Dieser Nachricht hängt er seine logische Zeit (Lamport clock) an. Der Empfänger dieser Nachricht hat 3 Antwortmöglichkeiten:
    - Greift er selbst auf die Ressource zu, so antwortet er mit denied
    - Greift er nicht darauf zu, und hat es auch nicht vor, so ist die Antwort OK
    - Greift er noch nicht darauf zu, will es aber, so werden die logischen Zeiten der Nachrichten (seine eigene + die Anfrage des anderen Prozesses) verglichen, die niedrigere gewinnt, je nachdem antwortet der Prozess mit denied oder OK

  Wird von jedem der n Prozesse auf ein OK gewartet, so gibt es nicht mehr einen single-point of failure, sondern n (sobald einer der n Prozesse gecrasht ist funktioniert dieses Prinzip nicht mehr); Bei dieser Variante kommt es zu einem sehr hohen Kommunikationsaufwand; Dieser Algorithmus beseitigt keine Bottlenecks, da jeder Prozess befragt wird und jeder dieselbe Arbeit erledigen muss -> keine Aufteilung der Arbeit.

37. Sie sollen einen Dateiserver implementieren der mehrere Clients gleichzeitig bedienen kann. Nennen und beschreiben Sie einen (konkreten) Mechanismus der garantiert dass immer nur einer der Clients gleichzeitig eine Datei schreiben darf. Die anderen Clients sollen solange blockiert werden (egal ob schreibend oder lesend) bis der schreibende Client seine Arbeit beendet hat. Warum ist das sinnvoll?

CENTRALIZED Algorithmus

- Ein zentraler Koordinator nimmt die READ und WRITE Anforderungen der Clients entgegen.
- READ dürfen parallel von mehreren Clients durchgeführt werden.
- WRITE darf nur ein einzelner CLIENT durchführen.
  -> sobald WRITE freigegeben wird, darf kein READ mehr durchgeführt werden. Alle READ-Requests kommen in eine Warteschlange. Sobald WRITE-Request fertig gestellt ist, werden weitere WRITE-Anfragen in der Warteschlange bearbeitet, ansonsten werden die anderen READ-Requests bearbeitet.
- WRITE –Requests kommen ebenso in eine Warteschlange.

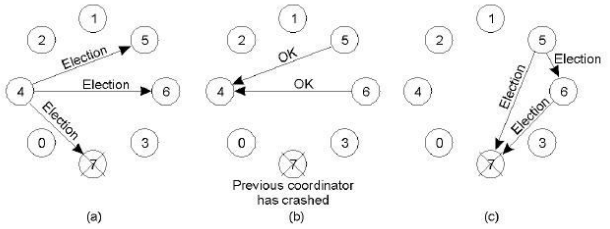
Sinnvoll:

- Wenn ein Schreibvorgang noch nicht beendet ist und ein READ ausgeführt wird, würde der Client eine ungültige Datei erhalten
- Ein Schreibvorgang darf nicht in einen anderen Schreibvorgang hineinschreiben → würden sich gegenseitig die Daten zerstören

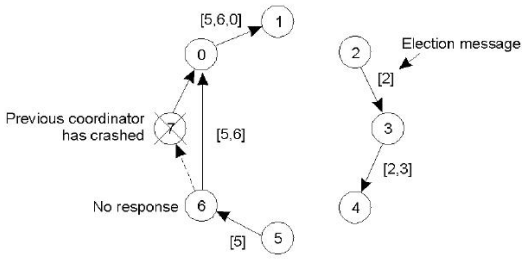
38. Erläutern Sie den "Bully" und den "Ring"-Algorithmus für Election und vergleichen sie die beiden hinsichtlich Fehlertoleranz. Warum sind diese Algorithmen für ad-hoc oder large-scale Systeme weniger geeignet und welche grundsätzlichen Lösungsansätze verfolgt man daher dort?

Sowohl bully, als auch der ring election Algorithmus basieren auf der Idee, dass jeder Prozess eine eindeutige ID besitzt (z.B.: ProzessID), und der Prozess mit der höchsten ID als Koordinator fungiert.

- **bully algorithmus**: Sobald ein Prozess bemerkt, dass der Koordinator ausgefallen ist, sendet er eine ELECTION Nachricht an alle Prozesse mit einer ID höher als seiner eigenen ID. Wenn er keine Antwort erhält, so ist er der neue Koordinator (schickt COORDINATOR Nachricht an die anderen Prozesse), wenn jedoch einer der Prozesse antwortet, so übernimmt dieser die Kontrolle, wer neuer Koordinator wird (in diesem Fall schickt der Prozess wieder eine ELECTION Nachricht, die geschieht so lange, bis der Prozess mit der höchsten ID Koordinator geworden ist und eine COORDINATOR Nachricht an die anderen Prozesse sendet).



- **ring algorithmus**: Die Prozesse sind logisch in Form eines Ringes angeordnet. Der Prozess, der den Ausfall des Koordinators bemerkt, sendet eine ELECTION Nachricht an seinen Nachfolger (wenn dieser nicht antwortet an dessen Nachfolger, so lange bis einer antwortet). Jeder Prozess hängt seine ID an die Nachricht an. Wenn die Nachricht den Ring einmal durchlaufen hat, und beim Initiator angelangt ist, sendet dieser eine COORDINATOR Nachricht aus, da ihm nun der Prozess mit der höchsten ID bekannt ist. Der neue Koordinator kann seine Arbeit aufnehmen, nachdem die COORDINATOR Nachricht wieder beim Initiator angelangt ist, wird sie entfernt. Falls mehrere Prozesse gleichzeitig den Ausfall des Koordinators bemerken und eine ELECTION Nachricht aussenden, ändert es nichts am Ergebnis, es wird lediglich ein wenig zusätzlicher Traffic erzeugt.



Beide Algorithmen sind sehr Fehlertolerant, da immer wieder ein neuer Koordinator gewählt werden kann. Es besteht KEIN Single-Point-Failure Problem.

Bei **ad-hoc-Netzwerken** kann sich die Topologie schnell ändern und man kann nicht davon ausgehen dass die Kommunikation zuverlässig (reliable) ist. Der Bully- und Ring-Algorithmus sind daher für derartige Netzwerke nicht geeignet. Außerdem ist es durch die heterogene Struktur wichtig dass nicht irgendein Koordinator gewählt wird, sondern der beste (anhand eines Kriteriums, z.B. Bandbreite).
Lösung: Bei der Auswahl sendet ein Knoten die ELECTION Nachricht an all seine Nachbarn. Empfängt ein Nachbar diese Nachricht setzt er den Sender als Parent und leitet die ELECTION Nachricht an alle Nachbarn außer den Parent weiter. Dadurch entsteht eine Baumstruktur. Jeder Parent wartet nun nach dem Weiterleiten auf die Response aller seiner Kinder. Die Response enthält die ID und das Score (Bewertung) des besten Prozesses im Subtree. Hat ein Parent von allen Kindern eine Response erhalten, kann er wiederum eine Response an seinen (mit dem besten Prozess) Parent senden. Der Root-Prozess weiß daher welcher Prozess der geeignetste Leader ist.

Bei **large-scale-Netzwerken** werden Superpeers ausgewählt, da ein Koordinator zu wenig sein wird. Auswahl mehrerer ausgezeichneter Knoten, die sich jeweils um eine Anzahl anderer Knoten kümmern. Dabei müssen Superpeers folgende Kriterien erfüllen:

1. Superpeers sollten geringe Latenzzeiten gegenüber normalen Peers haben
2. Superpeers sollten im Netzwerk gleichmäßig sein.

3. Die Anzahl der Superpeers sollte eine definierte Anzahl relativ zur Anzahl normaler Knoten nicht unterschreiten
4. Jeder Superpeer sollte nur eine gewisse Anzahl an Knoten bedienen

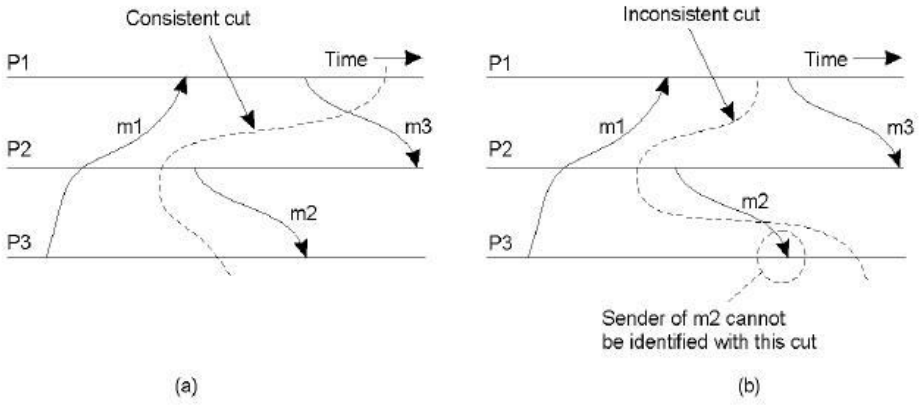
Lösung: Es werden für n Superpeers n Token zufällig im Netz verteilt. Kein Knoten kann mehr als einen Token gleichzeitig haben. Außerdem wirken auf die Token so genannte Kräfte die sicherstellen sollen, dass sich Token voneinander abstoßen (vgl. Teilchen mit gleicher Ladung). Erfährt ein Knoten von ein oder mehreren Token in der Umgebung kann aufgrund der Richtung, in der die anderen Token liegen jene Richtung bestimmt werden in die sein Token weitergegeben wird. Hält ein Knoten einen Token für eine gewisse Zeit (Token haben sich stabilisiert) wird dieser zu einem Superpeer. Im englischen Buch S. 270 wird darauf hingewiesen, dass diese mysteriöse Kraft mit einem Gossip-Protokoll realisiert werden kann.

39. Welche Probleme gibt es bei der Ermittlung des "Global State" und wie können diese überwunden werden? Geben Sie zumindest einen Algorithmus an.

Global State

Der Globale State eines verteilten Systems besteht aus
- dem lokalen Zustand eines jeden Prozesses
- zusammen mit den Nachrichten die gerade unterwegs sind (gesendet aber noch nicht empfangen)

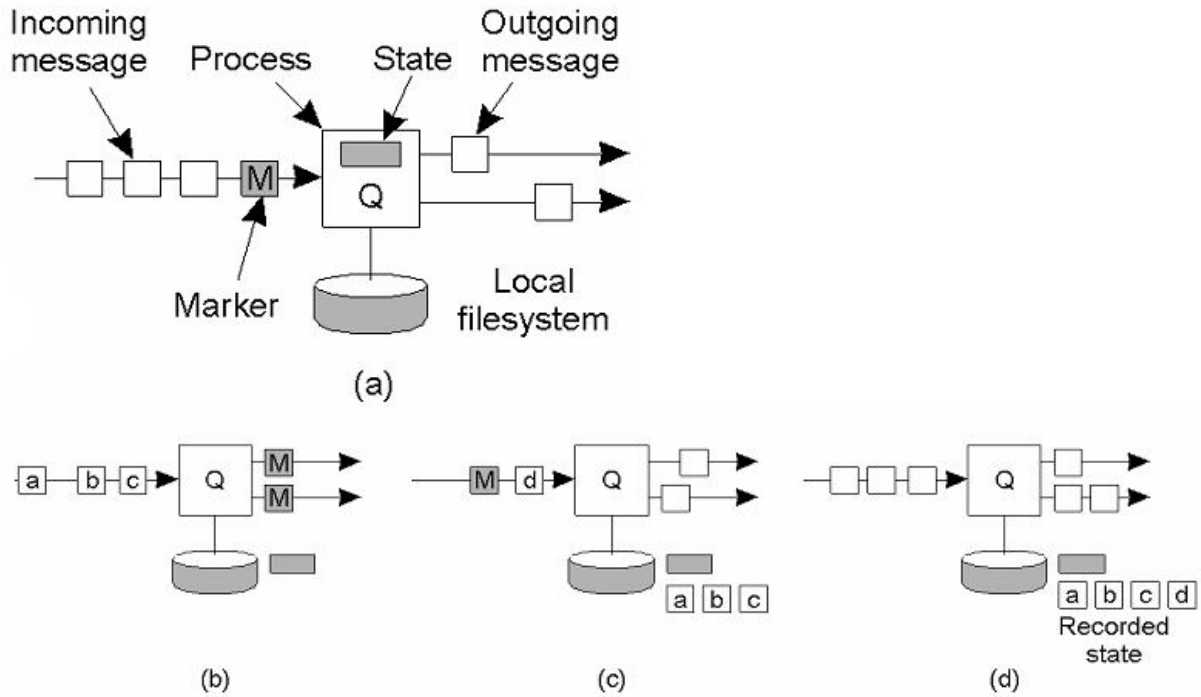
Probleme:
- Es könnte ein inkonsistener "Schnitt" durch das System gemacht werden. (Ergebnisse ohne Ursache. z.B.: Empfangen einer Nachricht ohne Senden dieser Nachricht)



a) A consistent cut
b) An inconsistent cut (effect without cause)

- Keiner hat eine globale Sicht auf das System.
- Es gibt keine gemeinsame Zeit für die Aufzeichnung ("Stichtag")

Lösung der Probleme durch Ermittlung des Globale State durch Chandy und Lamport-Algorithmus (Snapshot vom Global State).



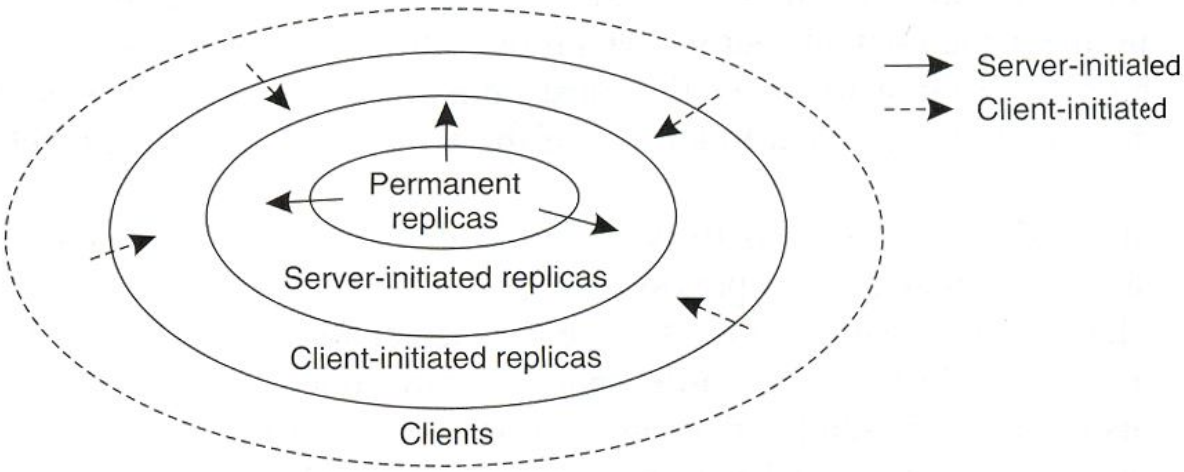
Ein Initiator beginnt seinen eigenen Status aufzuzeichnen und schickt einen Marker aus. Beim Empfang des 1. Markers zeichnet jeder Rechner seinen lokalen Status auf und schickt den Marker weiter (b). Bis zum Empfang des Markers zum zweiten Mal zeichnet jeder Prozess alle eingehenden Nachrichten auf (c). Beim Empfang des Markers zum zweiten Mal ist die Aufzeichnung abgeschlossen (d): der lokale Status und die aufgezeichneten Nachrichten können an den ursprünglichen Initiator gesendet werden, wo dann die Auswertung passiert. Der aufgezeichnete Status ist garantiert konsistent, kann aber Kombinationen von lokalen Zuständen enthalten, die so nie aufgetreten sind.

40. Was sind die Hauptgründe für den Einsatz von Replikation in verteilten Systemen? In welcher Beziehung stehen Replikation und Skalierbarkeit zueinander? Erläutern Sie in diesem Zusammenhang verschiedene Varianten der Content Replication und des Content Placement.
Bei Replikation führt man mehrere Kopien einer Entität auf verschiedenen Knoten.

- **Zuverlässigkeit**
  - Umschalten im Fehlerfall → **Fehlertoleranz**
  - Schutz gegen beschädigten Daten (corrupted data)
- **Leistung durch Skalierbarkeit**
  - **Größe**: Die Skalierung nach der Größe tritt beispielsweise auf, wenn immer mehr Prozesse auf Daten zugreifen müssen, die von einem einzigen Server verwaltet werden. In diesem Fall kann die Leistung verbessert werden, indem man diesen Server repliziert und damit die Arbeit aufteilt.
  - **geographischem/topologischem Kontext**: Die Skalierung in Hinblick auf die Größe eines geografischen Bereichs kann ebenfalls eine Replikation erforderlich machen. Dabei wird eine Kopie der Daten in die Nähe des Prozesses platziert, der sie nutzt, und somit die Datenzugriffszeit reduziert.

Skalierbarkeit und Replikation stehen insofern in der Beziehung, dass die Replikation einen deutlichen Mehraufwand bedeutet, da die Daten konsistent gehalten werden müssen. Man muss also abschätzen ob das Einsetzen von Replikaten den zusätzlichen Aufwand (Traffic, Ressourcen) rechtfertigt. (Medizin schlimmer als Krankheit → Kompromiss)

Content Replication und Content Placement



Content Replication: Sind Replikate permanent vorhanden oder nur zur Laufzeit
Content Placement: Wo werden die Replikate eingerichtet? (Client oder Server)

Permanente Repliken

- Cluster Based: Server befinden sich am selben Ort, es kann z.B. Round Robin durchgeführt werden
- Mirroing: gespiegelte Server die sich an einem anderen Ort befinden

Server-initiierte Repliken:

- Replikat wird dynamisch initialisiert, wenn es u vielen Requests kommt. Dazu ist es notwendig die Anzahl der Zugriffe auf ein File zu zählen und zu wissen woher der Zugriff kommt
- Schwellenwert für Replikat erstellen (ab wann wird ein Replikat erstellt)
- Replikat wird gelöscht, wenn es nicht mehr benötigt wird (wobei es immer eine Replikat geben muss)
- Replikat wird dort möglichst dort erstellt, wo die meisten Requests herkommen
- Ideal um flash crowds abzufangen

Client-initiierte Repliken: Auch bekannt als Cache, werden auf dem lokalen Speicher oder im lokalen Netzwerk platziert. Dienen zum Verbessern der Zugriffszeiten auf Daten, werden jedoch nur eine begrenzte Zeit gespeichert um zu verhindern, dass extrem veraltete Daten verwendet werden, Effizienz kann gesteigert werden wenn ein gemeinsamer Cache verwendet wird.

41. Geben Sie verschiedene Möglichkeiten der Update Propagation (Content Distribution) an und bewerten Sie diese hinsichtlich Vor- und Nachteilen sowie Einsatzmöglichkeiten.

Bei der Update Propagation von Replikas ist es nötig verschiedenen Aspekte zu betrachten und gegen einander abzuwiegen.

Status versus Operation:
Ein wichtiger Entwurfsaspekt kümmert sich darum, was überhaupt weitergegeben werden soll. Dafür gibt es grundsätzlich 3 Möglichkeiten:

- Weitergabe der Aktualisierung nur durch eine Benachrichtigung:
  In einem Invalidierungsprotokoll werden andere Kopien darüber informiert, dass eine Aktualisierung stattgefunden hat und dass ihre Daten nicht mehr gültig sind. Erst bei einem Lesezugriff wird diese Daten aktualisiert

  Vorteil: wenig Netzwerkbandbreite verbraucht, da nur eine Benachrichtigung versendet wird.
  Einsatzbereich: wenn es viele Aktualisierungsoperationen im Vergleich zu Leseoperationen gibt.

- Übertragung der Aktualisierungs-Daten:

  Nachteil: Viel Bandbreitenverbrauch
  Einsatzbereich: dort wo das Lese/Schreib Verhältnis hoch ist.

- Weitergabe der Aktualisierungs-Operation:
  Hier werden keine Änderungen an den Daten übertragen, sondern jeder Replik wird mitgeteilt, welche Aktualisierungsoperation sie ausführen soll. Dieser Ansatz wird auch aktive Replikation genannt.

  Vorteil: Aktualisierungen können mit minimalen Kosten für die Bandbreite weitergegeben werden.
  Nachteil: Jede Replik benötigt möglicherweise mehr Rechenleistung, insbesondere wenn die Operationen relativ komplex sind.

Pull- oder Push-Protokolle:
Dieser Entwurfsaspekt unterscheidet sich darin, ob Aktualisierungen abgeholt (pull) oder automatisch verschickt werden (push).

- In einem push-basierten Ansatz, auch als Server-basierte Protokolle bezeichnet, werden Aktualisierungen an andere Repliken weitergegeben, ohne dass diese welche angefordert haben. Server-basierte Protokolle werden hauptsächlich dort eingesetzt, wo die Repliken im Allgemeinen einen relativ hohen Konsistenzgrad aufweisen müssen. Dieser Bedarf eines hohen Konsistenzgrades hat mit der Tatsache zu tun, dass permanente und Server-initiierte Repliken sowie große Caches häufig von vielen Clients gemeinsam genutzt werden, die wiederum hauptsächlich Leseoperationen durchführen. Daher ist das Lese/Aktualisierung Verhältnis relativ hoch (eine Aktualisierung wird oft gelesen).

  Ein wichtiger Aspekt bei push-basierten Ansätzen ist, dass der Server alle Clients verwalten muss, welche Aktualisierungen von ihm erhalten. Neben der Tatsache, dass statusbehaftete Server weniger fehlertolerant sind, kann die Verwaltung aller Client-Repliken und -Caches einen wesentlichen Overhead auf dem Server verursachen

- In einem pull-basierten Ansatz, auch als Client-basierte Protokolle bezeichnet, fordert ein Server oder Client einen anderen Server auf, ihm Aktualisierungen zu senden, die ihm zu diesem Zeitpunkt vorliegen.
  Dieser Ansatz wird häufig von Web-Caches verwendet, wo der Browser zuerst prüft ob die im Cache befindlichen Datenelemente noch aktuell sind, und dann bei Bedarf die Aktualisierung vom Webserver holt.
  Client-basierte Protokolle sind effizient, wenn das Lese/Aktualisierung Verhältnis relativ niedrig ist. Der größte Nachteil ist, dass die Antwortzeit im Falle eines veralteten Cache-Eintrag steigt.

Aspekt	Push-basiert	Pull-basiert
Status am Server	Liste mit Client-Repliken und Caches	Keine
Gesendete Nachrichten	Aktualisierung (und möglicherweise später die Aktualiserung laden)	Pull und Aktualisieren
Antwortzetien auf dem client	Unmittelbar	Laden/Aktualiserungszeit

Häufig wird eine Mischform aus beiden Ansätzen verwendet, die auf Leases basiert. Ein Lease ist eine Zusage des Servers, dass er dem Client eine bestimmte Zeit lang Aktualisierungen bereitstellt. Wenn ein Lease abläuft, ist der Client gezwungen, den Server für mögliche Aktualisierungen zu kontaktieren, um diese bei Bedarf per Pull zu erhalten.
Wie die Expiration time bei Leases gewählt wird:

- Age-based: Annahme das länger unveränderte Daten auch weiterhin wenig Änderungen haben werden. Daten mit länger zurückliegender Änderungszeit bekommen höhere Expiration time.
- Renewal-frequency-based: Clients die eine hohe Update Rate haben bekommen einen höhere Expiration time
- State-space overhead: Schutzmechanismus um eine Überlastung des Servers zu vermeiden. Wenn der Server eine Überlastung feststellt verringert er die Expiration time der Leases und reguliert somit seine Last.

Unicasting versus Multicasting:
Bei einer Unicast-Kommunikation sendet ein Server, der Teil des Datenspeichers ist, seine Aktualisierung an N andere Server, indem er N separate Nachrichten sendet, je eine an jeden Server. Beim Multicasting übernimmt das zugrunde liegende Netzwerk die Aufgabe, eine Nachricht effizient an mehrere Empfänger zu senden.
Das Multicasting kann häufig effizient mit einem push-basierten Ansatz kombiniert werden, um Aktualisierungen weiterzugeben. In diesem Fall verwendet ein Server eine Multicast-Gruppe, um mehrere andere Server (welche in der Multicast-Gruppe sind) Aktualisierungen bereitzustellen.
Bei einem pull-basiertem Ansatz ist es häufig nur ein einziger Client der eine Aktualisierung anfordert. In diesem Fall wird Unicasting die effizientere Lösung sein.

42. Erläutern Sie die Funktionsweise der "primary-based" Protokolle. Bewerten und vergleichen Sie die verschiedenen Arten.

In Primary Based Protokollen ist jedem Datenelement x im Speicher ein primärer Server zugeteilt, der für die Koordination von Schreiboperationen für x verantwortlich ist. Ein Schreibrequest auf einem beliebigen anderen Replikat wird an den Primary-Server weitergereicht (remote-write). Geschrieben wird immer nur auf einem Replikat. Der Primary kann dabei für jedes Datenitem ein anderer Host sein. Alternativ dazu gibt es noch die Möglichkeit, das Datenitem zuerst auf den Server zu migirieren, auf dem der Schreibzugriff abgesetzt wird (Änderung des Primarys) und die Operation dann dort auszuführen (local-write).

Remote-write
Bei dem einfachsten Primary-based Protokoll werden alle Operationen zu einem bestimmten (fixierten) single-server weitergeleitet. Wenn z.B. eine Schreiboperation stattfinden soll, wird die Operation zuerst weitergeleitet an den Primary Server für das item. Der Primary Server führt daraufhin ein Update auf der lokalen Kopie des Items durch und leitet das Update an alle Backup-Server weiter. Jeder Backup-Server führt das Update genauso durch und sendet ein Acknowledge zurück an den Primary Server. Wenn alle aktualisiert wurden, schickt der Primary Server ein Acknowledge zurück zu dem initialisierenden Prozess. Das Performance Problem bei diesem Schema ist, dass es relativ lange braucht, bevor ein Prozess der ein Update initiiert hat fortfahren darf. Eine Alternative wäre einen nonblocking Ansatz zu wählen. Das Problem bei einem nonblocking System ist die Fehlertoleranz. Der Vorteil ist die Geschwindigkeit.

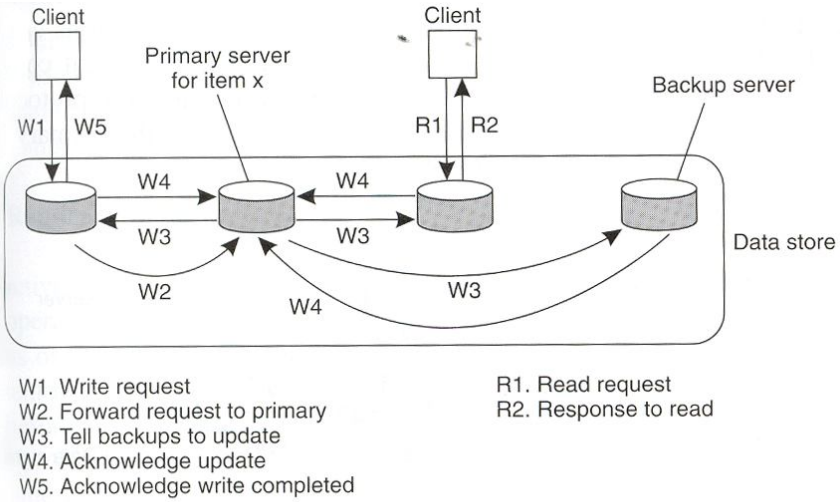


Figure 7-20. The principle of a primary-backup protocol.

Local-write:
Wenn ein Item geupdated werden soll, wird zuerst die primäre Kopie lokalisiert (wie zuvor). Und dann wird diese Kopie auf die eigene location kopiert. dazu gibt es noch die Möglichkeit, das Datenitem zuerst auf den Server zu migrieren, auf dem der Schreibzugriff abgesetzt wird (Änderung des Primarys) und die Operation dann dort auszuführen (local-write). Der Hauptvorteil dieser Vorgehensweise besteht darin, dass mehrere Schreiboperationen nacheinander lokal erfolgen können, während lesende Prozesse weiterhin auf ihre lokalen Kopien zugreifen können. eine Verbesserung dieser Art kann jedoch nur dann erfolgen, wenn ein nicht sperrendes Protokoll beachtet wird, das Aktualisierungen an die Replikate weiterleiten, nachdem der primäre Server sie lokal erfolgreich ausgeführt hat.

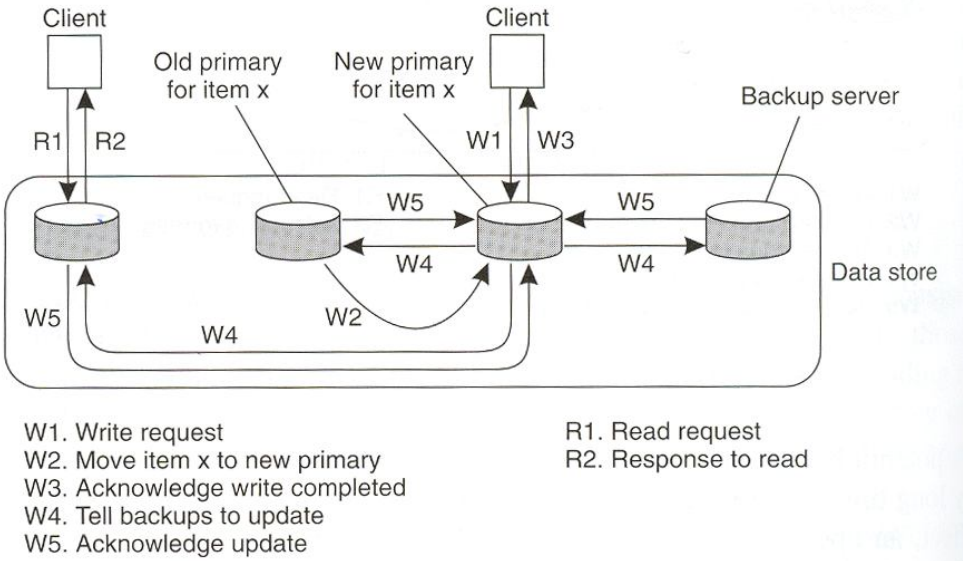


Figure 7-21. Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

43. Erläutern Sie die Funktionsweise der "replicated-write" Protokolle. Bewerten und vergleichen Sie die verschiedenen Arten. Welche Probleme können bei "Active Replication" auftreten?

Erklären Sie "quorum-based" Replikationsprotokolle und geben Sie die Bedingungen an, um Read-Write und Write-Write Konflikte zu verhindern.

Im Gegensatz zu Primary-Based Protokollen erlauben Replicated-Write-Protokolle den Zugriff auf beliebige Replikate. Die Änderungen werden dann an alle anderen Replikate propagiert, entweder über

- **active replication:** Die Operation wird an jedes Replica gesendet. Das Problem ist, dass Operationen überall in der selben Reihenfolge geliefert werden müssen. Was es daher braucht ist einen "totaly-ordered multicast- Mechanismus". Hier gibt es aber in groß skalierten Systemen Probleme. Als Alternative kann ein zentraler Koordinator (Sequencer) verwendet werden. Meistens ist eine Mischung aus beiden nötig.
- **quorum-based protocolls**: Hier wird ein Voting verwendet. Bsp.: Um ein File zu aktualisieren, muss der Client zuerst mindestens die Hälfte der Server plus 1 dazu bringen dem Update zuzustimmen. Um das File zu lesen muss der Client wiederum mindestens die Hälfte der Server plus eins dazu bringen ihre Versionsnummern des Files zu schicken. Wenn alle Versionsnummer gleich sind, muss es sich um die aktuelle Version handeln.

  Eine Verallgemeinerung hierfür ist das **Gifford's scheme**, das folgenden Regeln folgt: Der Client muss eine Mehrheit der Server mit den Update versorgen. Beim Lesezugriff muss auch wieder von zumindest der Anzahl der nicht upgedateten Server +1 gelesen werden. Dabei gewinnt die Dateneinheit mit der höchsten Versionsnummer.

  Nr + Nw > N (verhindert read-write Konflikte)
  Nw > N/2 (verhindert write-write Konflikte)

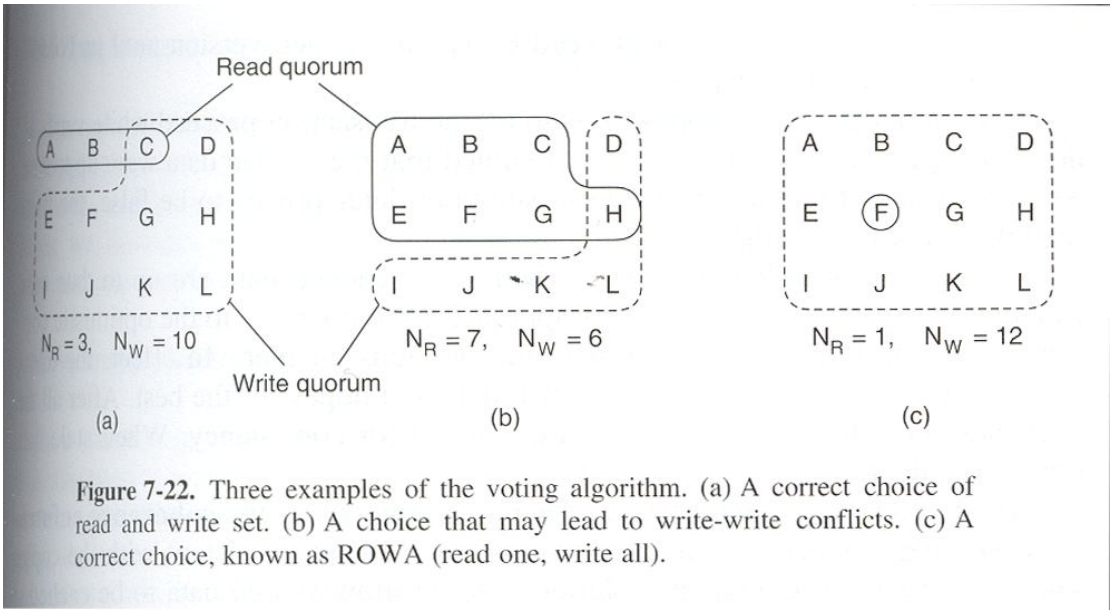


Figure 7-22. Three examples of the voting algorithm. (a) A correct choice of read and write set. (b) A choice that may lead to write-write conflicts. (c) A correct choice, known as ROWA (read one, write all).

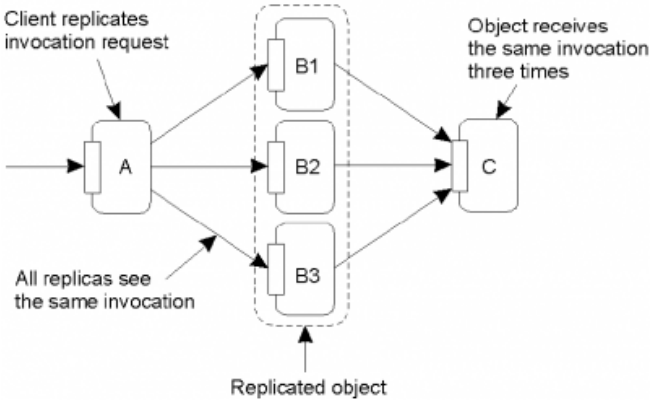
Eine Spezialform von Replicated-Write-Protokollen ist **Coordinator-Cohort-Replication**. Dabei wird die Anfrage an ein (beliebiges) Replikat gesendet, das die Updates synchron an alle anderen propagiert. Dazu ist ein distributed-Locking-Mechanismus erforderlich.

44. Welche Besonderheiten sind bei der Replikation von Objekten zu beachten? Erläutern Sie (inkl. genauer Skizze) wie man "Replication transparency" in Objektsystemen umsetzen könnte ("replicated invocation").

Problem: Wenn Objekte in verteilten Systeme verwendet werden und ebenfalls verteilt sind, dann müssen auch die Zugriffe auf Objekte konsistent gehalten werden, da es sonst zu inkonsitenten Zuständen innerhalb des Systems führen kann (entry consistency). Wir müssen also eine gleichzeitige Ausführung von Methoden am Objekt verhindern (durch Locking relativ einfach zu lösen) und wir müssen Änderungen (Methodenaufrufe) auf ALLE Replikate des Objektes verteilen um sicherzustellen, dass nicht 2 unterschiedliche Aufrufe auf ein verteiltes System zur gleichen Zeit geschehen.
Dies kann folgendermaßen geschehen:

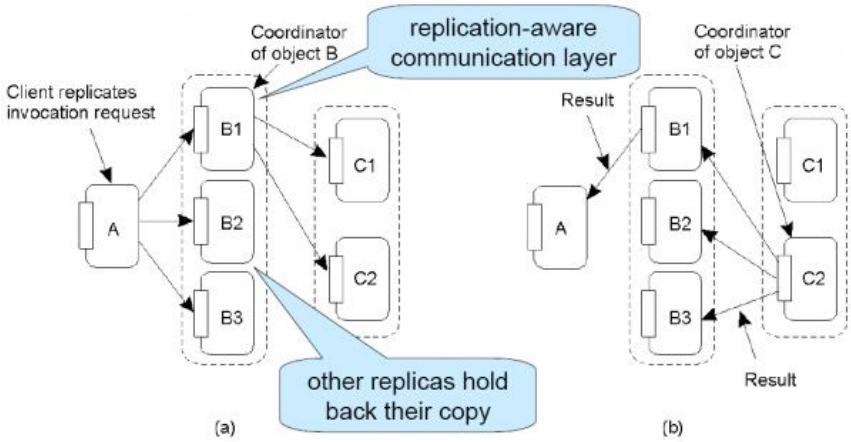
- **primary-based approach**: ein "Koordinator" übernimmt die Kontrolle und delegiert die Aufrufe. Problem: Overhead, Skalierbarkeit geht verloren, "Koordinator" muss andere verteilte Objekte kennen.
- **totally-ordered-multicasts**: Methodenaufrufe werden zB durch Lamport clocks durchnummeriert und dann in der Reihenfolge auf jedes Objekt ausgeführt. Problem dabei: sehr aufwändig, viel Overhead
-

Replicated Invocation Problem: Methodenaufrufe auf ein vielleicht verteiltes Objekt werden mehrmals ausgeführt.



nicht

Lösung: Realisierung von **Replication Transparency**:
Falls von A aus ein Aufruf in B erfolgt, so ruft A alle Replikate in B (B1, B2, B3) auf. Wenn B nun seinerseits wieder eine Methode von C aufruft, so wird das nur vom Koordinator von B (z.B. B1) gemacht (und zwar bei C1 und C2). Andernfalls (wenn alle 3 Replikate von B den Aufruf durchführen würden) wäre die Methode von C zu oft aufgerufen worden. Die Antworten werden nur vom Koordinator zurückgegeben, also von z.B. C2 zu allen 3 Replikaten von B, und von B1 zu A.



45. Was sind Epidemic Protocols. Welche Vor- und Nachteile haben diese? Erklären Sie "gossiping" ("rumor spreading") im Zusammenhang mit Replica update propagation. Erläutern Sie Vor- und Nachteile. Erklären Sie das Anti-Entropy Modell im Zusammenhang mit Replica update propagation. Erläutern Sie Vor- und Nachteile.

Epidemic Protocols sind für Datenspeicher gedacht, welche nur eine eventuelle Konsistenz aufweisen müssen. Das heißt: Wenn es eine Aktualisierung gibt, muss nur sichergestellt sein, dass alle Replikas irgendwann identisch sind.
Das Hauptziel dieser Protokolle ist es, schnell Information an viele Knoten zu verbreiten während man nur lokale Informationen verwendet.

- Vorteile: Gute Skalierbarkeit, Aktualisierung an alle Repliken erfolgt in so wenigen Nachrichten wie möglich -> geringe Netzwerkbelastung.
- Nachteile: Weitergabe des Löschens eines Datenelements ist schwierig, löst keine Aktualisierungskonflikte, nur eventuelle Konsistenz (sehr schwache Konsistenz).

Das **Anti-Entropy Modell** ist ein Weitergabemodell für Epidemische Protokolle welches per Zufall einen anderen Server auswählt, und mit diesem dann Aktualisierungen austauscht. Es kann pull- oder pushed basiert arbeiten (Ein Server schickt seine Updates zu einem anderen oder holt sie von einem anderen). Wenn viele Knoten infiziert sind, ist die Chance relativ gering über den push-Ansatz weitere Knoten für ein Update zu finden, im Gegensatz zum Push Ansatz.

Gossiping oder auch „**rumor spreading**“ genannt ist eine spezielle Form des Anti-Entropy Modells und ein effizientes Weitergabemodell für Epidemische Protokolle. Die Funktionsweise ist einfach: Wenn ein Datenelement aktualisiert wurde, wendet sich der Server an einen beliebigen anderen Server um ihm die "Neuigkeit" mitzuteilen. Dieser wiederum macht dasselbe und kontaktiert den nächsten Server um die Aktualisierung vorzunehmen usw. Wenn ein Server erreicht wird, der schon "infiziert" wurde, ist die "Neuigkeit" nicht mehr so interessant und macht nur mehr mit einer gewissen Wahrscheinlichkeit weiter mit "gossiping".

- Vorteil: Aktualisierung wird schnell weitergereicht
- Nachteil: Es kann nicht garantiert werden, dass alle Server "infiziert" werden, sprich mit Aktualisierungen versorgt werden.

46. Erläutern Sie die grundlegenden Begriffe der Dependability: Nennen Sie die fünf wesentlichen Attribute (bzw. Requirements) eines "dependable system". Was ist der Unterschied zwischen Availability und Reliability? Erläutern Sie die "dependability threats" Failure, Error und Fault sowie den Zusammenhang zwischen den drei. Erläutern Sie "permanent", "transient" und "intermittent" faults anhand von Beispielen.

Dependability meint die Eigenschaft eines Systems, Failures zu verhinden, die häufiger oder schwerwiegender sind, als akzeptiert werden kann. Die ist eng verbunden mit Fault Tolerance, was bedeutet, dass ein System auch dann noch sein Service fehlerfrei anbieten kann, wenn Faults im System vorhanden sind (diese müssen maskiert werden).

Attribute:

- Availability/Verfügbarkeit wird als Wahrscheinlichkeit angegeben, mit welcher ein Service/System ein korrekt arbeitet zu einer beliebigen Zeit, sagt jedoch nichts darüber aus, wie lange ein System am Stück Verfügbar ist.
- Reliability/Ausfallsicherheit: Definiert wie lange (Zeitspanne) ein System ohne Ausfall korrekt arbeiten kann. Bsp: Ein System welches 2 Wochen in Jahr gewartet wird, sonst jedoch nie ausfällt, hat eine hohe Ausfallsicherheit, jedoch nur eine Verfügbarkeit von 96%

    ➔ Im Gegensatz zur Verfügbarkeit ist hier von einem Zeitraum und nicht von einem Zeitpunkt die Rede. Wenn ein System pro Stunde eine Millisekunde lang nicht funktioniert ist, hat es zwar eine hohe Verfügbarkeit, aber keine hohe Ausfallsicherheit. Andererseits hat ein System welches ganze 11 Monate am Stück fehlerfrei läuft aber

dann den ganzen Dezember über abgeschalten wird, hat eine hohe Ausfallsicherheit aber ist nicht hoch verfügbar.

- Safety/Sicherheit: Definiert, wie sicher ein System im Falle eines Fehlers agiert, sodass keine größeren Schäden passieren. Bsp: Bei einem Software Fehler in einem Atomkraftwerk sollen das System keinen Katastrophen ähnlichen Zustand annehmen.
- Integrity/Integrität: Es dürfen keine ungültigen Systemzustände (Errors) angenommen werden
- Maintainability/Wartbarkeit: Gibt an, wie leicht ein System im Falle eines Fehlers repariert werden kann. Eine hohe Maintainability kann zu einer hohen Verfügbarkeit führen, da im Falle eines Fehlers dieser möglicherweise automatisch erkannt und repariert werden kann.

dependability threats:

- Fault: ist die Ursache eines Errors/Fehlers
- Error: Ein Teil des System Status, welcher durch einen Fault ausgelöst wurde und schlussendlich zu einem failure führt
- Failure: Ein Failure tritt ein, wenn ein System durch einen Error seine Dienste nicht mehr vollständig oder überhaupt nicht mehr anbieten kann.

Faults können wie folgt klassifiziert werden:

- transient fault: Treten zufällig, jedoch nicht wiederholt auf. Bsp: Ein Vogelschwarm stört die Funkverbindung.
- intermittent fault: treten zufällig auf, verschwinden wieder, und kommen dann wieder; diese Klasse der Faults ist besonders schwer zu identifizieren. Bsp: ein loser Netzwerkstecker, welcher zu einem Wackelkontakt führt.
- permanent: Treten solange dauerhaft auf, bis die fehlererzeugende Komponente ersetzt wurde. Bsp: Hardwaredefekte, Software Bugs

47. Wozu benötigt man Fehlermodelle ganz allgemein? Geben Sie verschiedene Fehlermodelle für "fail-controlled systems" an und diskutieren Sie diese v.a. hinsichtlich des benötigten Aufwandes für die Maskierung. Inwiefern ist es u.U. heikel zu spezifizieren, dass ein System "k-fault-tolerant" sein soll?

Fehler werden in Klassen (Fehlermodelle) unterteilt, um die Auswirkungen besser einschätzen zu können .

- **crash failure (Absturtzausfall)**: Ein Server stürzt ab und gibt keine Rückmeldungen mehr, bis er neugestartet wird. Bis der Crash eingetroffen ist, arbeitet der Server korrekt. Mithilfe von mehreren redundanten Servern lässt sich dieser Fehler relativ gut maskieren (da leicht Feststellbar), jedoch steigt dadurch der Traffic und es ergeben sich neue Probleme wie z.B. Konsistenz
- **omission failure (Dienstausfall)**: Ein Server antwortet nicht auf Anfragen. Das kann verschiedene Gründe haben, beispielsweise, dass er nie die Anfrage erhalten hat (receive omission). Oder der Server hat die Anfrage erhalten, beantwortet, ist aber nicht in der Lage die Antwort zu versenden (send omission). Auch bei einem Dienstausfall lässt sich der Fehler mit Redundanten Servern leicht maskieren.
- **timing failure (Zeitbedingter Ausfall)**: Wenn eine Antwort länger als eine definierte Response-Time benötigt, spricht man von einem Timing failure. Maskierung abhängig von der definierten Responsetime → zu kurz gewählt → Fehler tritt oft auf, zu groß gewählt → Fehler kann nicht wird zu spät erkannt.
- **response failure (Ausfall korrekter Antwort)**: wenn die Antwort eines Servers inkorrekt ist. Es gibt 2 Arten von response Fehlern:
    - value failure der Server liefert eine falsche Antwort (Suchmaschine liefert Webseiten, nach welchen nicht gesucht wurde)

    - o state transition failure: passieren, wenn der Server unerwartet auf eine Anfrage reagiert, also vom Programmablauf abweicht (z.B.: ein Server erhält eine Anfrage, welche er nicht erkennen kann und daraus Aktionen startet, welche niemals passieren dürften)

      Da er sich nur schwer erkennen lässt (meist nur durch den User) lässt er sich meist nicht maskieren.
- **arbitrary failure/bizantine failure**: ist eine sehr schwerwiegende Klasse von Fehlern, wenn z.B. wenn ein Server eine Antwort liefert, welche niemals erstellt werden dürfte (die aber nicht als fehlerhaft erkannt werden kann) → Da sich die Fehler nicht erkennen lassen, lassen sie sich auch nicht maskieren.

Ein k-fault-tolerant System ist ein System, bei welchen bis zu k Komponenten ausfallen können, ohne dass das System davon beeinträchtigt wird und immer noch korrekte Ergebnisse liefern. Da die Annahme ist, dass die fehlerhaften k Prozesse keine Antworten mehr liefern. Das Problem bei der k-fault-tolerance ist, dass Prozesse auch weiterhin arbeiten können, nachdem sie in einen fehlerhaften Zustand geraten sind. Wenn ein byzantinischer Fehler auftritt, und die fehlerhaften k Prozesse sowohl korrekte, als auch falsche Ergebnisse liefern, ist es nötig, dass mindestens 2k+1 Prozesse nötig sind, um k-fault-tolerant zu sein, damit immer mindestens k+1 Prozesse funktionstüchtig sind und in jedem Fall die fehlerhaften k Prozesse überstimmen.

48. Wieso benötigt man Redundanz zur Maskierung von Fehlern? Welche Arten von Redundanz gibt es?

Um auch in der Zwischenzeit, während ein Fault vorhanden ist, fehlerfreies Verhalten garantieren zu können, sind Redundanzen notwendig. Diese können entweder **zeitlich** (z.B. mehrmalige Übertragung wie bei TCP), **datentechnisch** (Prüfsummen, Hamming-Distanzen, etc.) oder **physikalisch** (doppelte Ausführung eines Servers wie z.B. im DNS) realisiert sein. Im Falle von physikalischen Redundanzen kann man unterscheiden ob die redundanten Systeme ständig im Gesamtsystem mitlaufen (aktiv) oder nur im Falle eines Faults angekoppelt werden (passiv). Bei passiven Systemen kann weiter unterschieden werden ob die (nicht angeschlossenen) redundanten Teile nebenbei mitlaufen (hot standby) oder komplett abgeschaltet sind (cold standby). Ein Beispiel für eine redundante Systemarchitektur ist Triple Modular Redundancy (TMR).Ein Schaltkreis, bei dem jedes Element einer sequentiellen Aneinanderreihung von Komponenten 3 Mal ausgeführt wird. Als Input bekommt jedes Element das (von einem Voter ermittelte) Mehrheitsergebnis der vorherigen Komponente. Die Voter müssen, da diese selbst fehlerhaft sein können, ebenfalls repliziert sein. So kann bei einem fehlerhaften Komponent, ausgeglichen werden, ohne dass das System den Fehler mitbekommt.

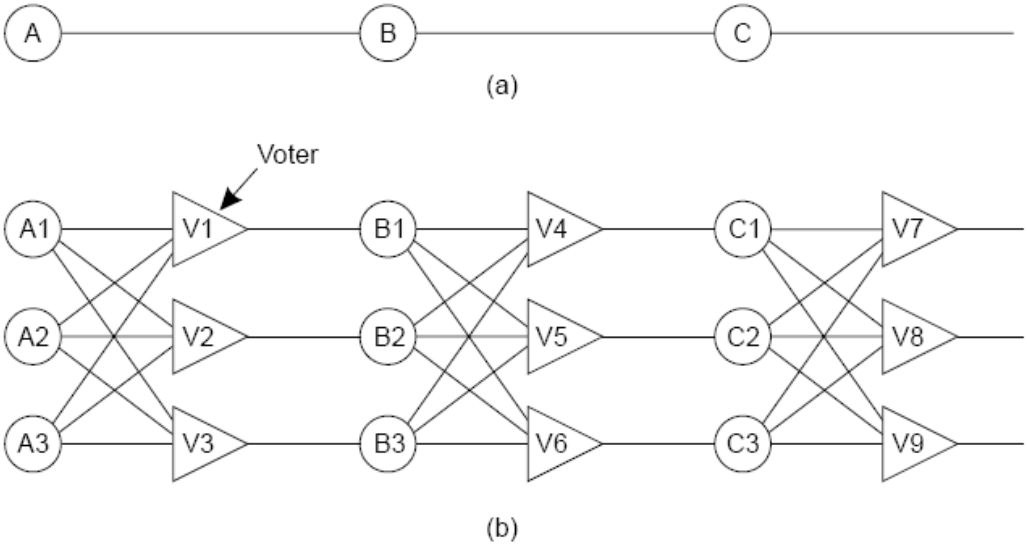
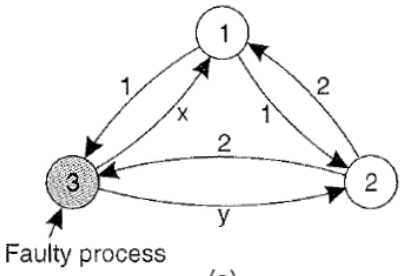


Figure 7-2. Triple modular redundancy.

49. Erläutern Sie die Aussage des "two-army" Problems

Im two-army Problem geht es darum, dass mit unzuverlässiger Kommunikation 2 Prozesse nicht zu einem gemeinsamen Konsens kommen können.
Die Annahme dieses Problems ist, dass es eine rote und eine blaue Armee gibt. Die blaue Armee, bestehend aus 2 Generälen, an verschiedenen Orten, möchte sich einen Angriffzeitpunkt ausmachen. So wird eine Nachricht mittels eines Boten von General A zu General B geschickt, wann dieser Angriff erfolgen soll. General B bekommt die Nachricht und schickt sie an A zurück. A bemerkt, dass B nicht weiß, ob die Nachricht bei A angekommen ist und so möglicherweise nicht angreift. Deswegen schreibt A zurück an B, welcher sich bei Erhalt der Nachricht denkt, dass A nicht sicher weiß, ob die Nachricht angekommen ist. So werden lauter Nachrichten verschickt, es gibt bei unsicheren Verbindungen (der Bote in diesem Fall) keine Sicherheit über den Erhalt der Nachricht und somit keine letzte Nachricht, und damit auch keine Übereinkunft.

50. Erläutern Sie die Aussage der "Byzantinischen Generäle".
Das Problem der Byzantinischen Gerneräle sieht folgende Situation vor: Eine Stadt wird von einer roten Armee gehalten, während auf jedem der umliegenden Berge ein General mit seiner Armee auf den Angriff wartet. Nun sollen die Truppen zwischen den Generälen so ausgetauscht werden, dass die Armee eines jeden Generals die gleiche Truppenstärke aufweist. Die Generäle sind alle loyal, bis auf einen, welcher versucht die anderen so zu stören, dass es niemals zu einem Angriff kommt. Im Gegensatz zu dem two-army Problem geht man hier davon aus, dass eine verlässliche Verbindung (Telefon) verwendet wird, immer 2 Generäle kommunizieren direkt und verlässlich miteinander und teilen sich ihre Truppenstärke mit. Jeder General berichtet seine wahre Truppenstärke, bis auf den fehlerhaften, welcher jedesmal eine andere Truppenstärke liefert. Danach hat jeder General die Informationen aller anderen. Nun teilt jeder General seine gesammelten Informationen den anderen mit, wieder lügt der falsche General und teilt jedem anderem, fehlerhaftem Werte mit. Es werden nun die Ergebnisse verglichen, die Mehrheit gewinnt, hat kein Wert die Mehrheit, so wird der Wert als unbekannt maskiert. Nun sollte jeder General die Truppenstärke der anderen wissen, nur die des falschen Generals ist unbekannt, er hat es nicht geschafft seine Manipulation erfolgreich durchzusetzen
Damit dieses Schema (Lamport 1982) funktioniert, benötigt man bei k fehlerhaften Prozessen mindestens 2k+1 Prozesse, welche ordnungsgemäß arbeiten, somit werden 3k+1 Prozesse benötigt, um k fehlerhafte zu tolerieren.
Um das hier an einem Beispiel zu zeigen. Bei 3 Teilnehmern und einem fehlerhaften davon kann es zu keinem Konsens kommen, da keiner eine Mehrheit aus 2 Werten bilden kann.



(a)

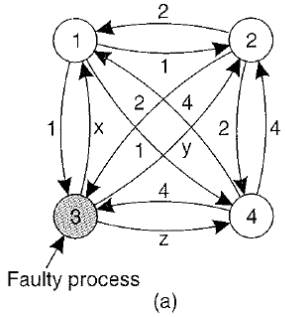
1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

Hingegen kann bei 4 Teilnehmern mit einem fehlerhaften sofort eine Mehrheit gebildet werden:



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Der Algorithmus, welcher in dem Bild dargestellt wird arbeitet in vier Schritten:

1) Punkt (a): Jeder nicht-fehlerhafte Knoten sendet vi zu jedem anderen Prozess mittels reliable unicasting. Da Multicasting verwendet wird können fehlerhafte Prozesse verschiedene Werte an die unterschiedlichen Prozesse schicken (im Bild: x, y, und z).
2) Punkt (b): In diesem Schritt werden die Ergebnisse aus dem ersten Schritt in Form eines Vektors gesammelt.
3) Punkt (c): Jeder Prozess schickt seinen Vektor aus Schritt 2 zu den anderen Prozessen. Jeder Prozess erhält also drei Prozesse, jeweils einen von den allen anderen Prozessen.
4) Jeder Prozess kontrolliert das i-te Element jedes neu empfangenen Vektors und fügt dieses in den result vector ein, wenn eine Mehrheit gebildet werden kann (im Bild Werte von v1, v2, v4). Kann keine Mehrheit gebildet werden, wird das Element im result vector mit UNKNOWN markiert

51. Erläutern Sie die Fehlerklassen in RPC-Client/Server-Umgebungen. Gehen Sie besonders auf das "lost reply" Problem ein.

Fehlerklassen von RPC Systemen:

- Der Client kann den Server nicht finden.
- Die Frage geht zwischen Client und Server verloren
- Der Server stürzt nach Erhalt der Nachricht ab
- Die Antwort des Servers geht auf dem Weg zum Client verloren
- Der Client stürzt nach versenden der Anfrage ab

Falls die Antwort vom Server verloren geht (lost reply) ist es für den Client schwer zu sagen, ob die Antwort oder die Anfrage verloren gegangen ist, oder der Server in der Zwischenzeit abgestürzt ist. Eine Möglichkeit wäre es, die Anfrage erneut zu stellen, dies ist aber nur sinnvoll bei idempotenten Operationen (Operationen welche wiederholt ausgeführt werden können, z.B.: Daten lesen, während eine Überweisung von einem Konto zu einem anderen nicht idempotent ist). Man könnte nun versuchen alle Request idempotent aufzubauen, was aber nicht immer möglich ist. Eine weitere Möglichkeit wäre es, jeder Anfrage eine ID zu vergeben, sodass der Server erkennen kann, ob die Anfrage schon bearbeitet wurde, oder eine neue ist. In diesem Fall stellt sich die Frage, wie lange der Server sich die Anfragen merken soll. Zusätzlich könnte der Client bei jeder Nachfrage ein Bit anhängen, ob es sich um eine neue, oder eine wiederholte Anfrage handelt.

52. Was versteht man unter reliable bzw. ordered multicast (group communication) in statischen Gruppen von Prozessen? Was muss man bedenken, wenn sich die Gruppen dynamisch verändern können? Erläutern Sie das Prinzip des "atomic multicast" ("virtual synchrony").

Ein reliable bzw. ordered multicast Service garantieren, dass Nachrichten an alle Mitglieder einen Prozessgruppe gesendet werden.
Meistens wird eine gesicherte point-to-point Verbindung von der Transportschicht bereit gestellt. Schwieriger ist es allerdings ein gesichertes Multicasting anzubieten. Eine Nachricht die zu einer Prozessgruppe gesendet wird, soll jedes Mitglied dieser Gruppe erreichen. Es müssen aber auch Situationen abgedeckt sein, wie das crashen eines Mitglieds, das Hinzukommen eines neuen Mitglieds und dass alle Mitglieder die Nachrichten in der gleichen Reihenfolge erhalten. Um die Ordnung einzuhalten kann eine Sequenznummer an die Nachrichten angehängt werden.
Ein Hauptproblem bei Multicasting ist die Unterstützung von vielen Empfängern. Gibt es N Empfänger, kommen in der Regel auch N Acknowledgements zurück, dadurch kann der Sender "überschwemmt" werden. Eine Möglichkeit ist, dass Empfänger den Sender nur informieren, wenn sie eine Nachricht nicht erhalten haben. Aber in diesem Fall wird der Sender die Nachrichten nicht ewig speichern, sondern sie nach einer gewissen Zeit löschen.

Ordered Multicast
Es gibt 4 verschiedene Arten Multicasts zu ordnen:

- Unordered Multicasts: Es ist nichts definiert
- FIFO ordered Multicasts: Nur Nachrichten vom selben Prozess werden in einer definierten Reihenfolge zugestellt (nämlich in der, in der sie gesendet wurden).
- Causally ordered Multicasts: Kausal abhängige Nachrichten (2 Nachrichten sind abhängig, wenn sie im Sinne von FIFO abhängig sind oder wenn eine gesendet wurde nachdem eine andere empfangen wurde – siehe Vector Timestamps) werden in der Reihenfolge der Abhängigkeit zugestellt.
- Totally ordered Multicasts: Alle Nachrichten werden bei allen Prozessen in der gleichen Reihenfolge zugestellt

Atomic Multicast
Meistens ist es nötig zu garantieren, dass Nachrichten entweder alle oder kein Empfänger erhalten, und dass die Nachrichten bei allen in der gleichen Reihenfolge ankommen. Das wird "Atomic Multicast Problem" genannt.
Bsp.: An eine Gruppe von Prozessen werden Mutlicast Update-Nachrichten verschickt. Ein Replica crasht. Das Update wurde aber auf den anderen Replicas trotzdem durchgeführt. Wenn das Replica wieder aktiv wird, wird es einige Updates vermissen. Bei Atomic Multicast wird das Update erst durchgeführt, wenn alle anderen Mitglieder sich einig sind, dass das abgestürzte Replica nicht mehr Teil der Gruppe ist. Wenn es wieder aktiv wird, bekommt es keine Updates, bis es sich wieder als Mitglied registriert hat und somit alle Daten von einem anderen System geholt hat.

Virtual Synchrony ist eine stärkere Art von Atomic Multicast:
Jede Message wird allen nicht fehlerhaften Group Membern zugestellt oder keinem, in derselben Reihenfolge Es soll sichergestellt werden, wenn ein Prozess die Gruppe verlässt (z.B. durch einen Crash), sollen die Nachrichten zu allen Verbleibenden geschickt werden oder zu keinem.

Synchrony Multicasting kann wie folgt realisiert werden. Jeder Prozess in der Gruppe buffert sämtliche Mutlicast-Nachrichten. Sobald ein Prozess die Gruppe verlassen will, beitreten will oder den Ausfall eines Prozesses entdeckt hat schickt er eine View-Change-Nachricht an alle in der Gruppe. Das veranlasst die Prozesse sämtliche von ihnen gebufferten Nachrichten in die Gruppe auszuschicken und mit einer Flush-Nachricht abzuschließen (was bedeutet, dass der Prozess keine gebufferten Nachrichten mehr hat). Damit wird gewährleistet, dass vor dem View Change alle Nachrichten an alle Prozesse in der Gruppe gehen, falls diese mindestens einen Prozess erreicht haben (eventuell haben sie, als sie ursprünglich ausgeschickt wurden, nicht alle Prozesse erreicht, weil der Sender abgestürzt ist bevor er den Multicast abschließen konnte). Sobald ein Prozess die Flush-Nachricht von allen anderen erhalten hat kann er sicher sein dass er keine an diese Gruppe adressierten Nachrichten verpasst hat und kann somit den View Change durchführen.

53. Erläutern Sie die Definition von Security mit den Attributen Availability, Confidentiality und Integrity anhand von Beispielen. Beschreiben Sie jeweils vier Security Threats und Security Mechanisms.

Security Atribute:

- **Availability**: Die Wahrscheinlichkeit, dass ein Service zu einem beliebigen Zeitpunkt verfügbar ist. Security Maßnahmen sollten natürlich die ganze Zeit, und überall verfügbar sein. Bezieht sich eine Zugriffskontrolle nur auf einen Teil des Systems, kann man den ungeschützten Teil für Angriffe missbrauchen.
- **Confidentially**: Es dürften nur berechtigte Benutzer oder Prozesse Zugriff auf ein System und die darin gespeicherten Daten haben. Ist z.B. mit einem Authentifizierunsdienst wie Cerberus oder Active Directory zu bewerkstelligen.
- **Integrity**: Änderungen an Systembestandteilen (Hardware/Software/Daten) sollen nur autorisiert erfolgen. Das heißt auch, dass missbräuchliche Änderungen erkannt und wiederhergestellt warden können.

Es gibt folgende **security threats**:

- **Lauschangriff** "Interception" (Unauthorisierter Zugriff)
- **Unterbrechung** "Interruption" (File ist beschädigt oder verloren - Services oder Daten sind nicht mehr zugänglich bzw. unbrauchbar)
- **Modifizierung** "Modification" (Unauthorisierte Änderung von Daten)
- **Erzeugung** "Fabrication" (Zusätzliche Daten werden generiert, die normalerweise nicht existieren würden. Ein Eindringling legt laufend fehlerhafte Daten an um das System zu blockieren.) ==> dagegen Authentifizierung/Autorisierung

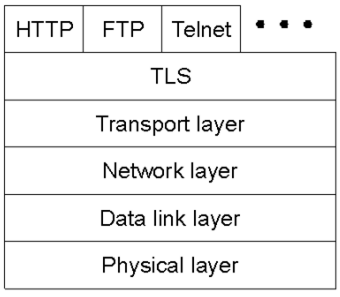
Security Mechanismen (dienen zur "Umsetzung" von Security Policies):

- **Encryption**: Verschlüsselt Daten, damit sie ein Attackierender nicht verstehen kann.
- **Authentication**: Wird verwendet um die Identität eines Users, Clients, Servers oder Hosts zu verifizieren.
- **Authorization**: Wird verwendet um festzustellen, ob der Client eine bestimmte Aktion durchführen darf.
- **Auditing**: Wird verwendet um festzuhalten, welcher Client worauf zugegriffen hat und wie.

(**Security Policies** hingegen beschreiben nur, welche Aktionen eine Entität in einem System durchführen darf)

54. Was ist der Secure Socket Layer (SSL, auch TLS genannt)? Positionieren Sie SSL/TLS im Internet Protokoll Stack.

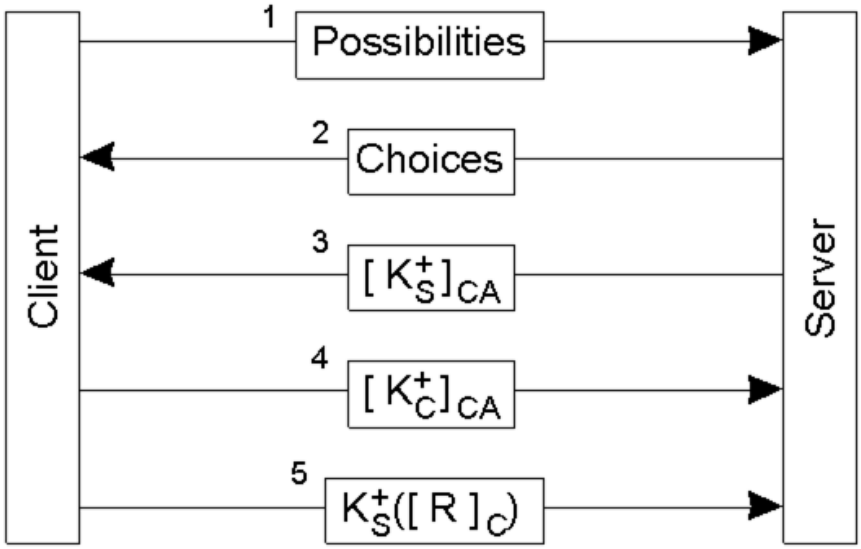
SSL (auch TLS) ist ein Verschlüsselungsprotokol zur sicheren Datenübertragung im Internet. SSL steht für Secure Socket Layer und verschlüsselt eine TCP-Verbindung. SSL befindet sich direkt über dem Transportlayer.



Der Ablauf ist wie folgt:

- Der Client teilt dem Server seine Verschlüsselungs- und Kompressionsmöglichkeiten mit (1)
- Der Server wählt welche davon aus und teilt dies dem Client mit (2)
- Der Server authentifiziert sich mittels Zertifikat am Client, welches den Public Key beinhaltet und von einer Certification Authority (CA) signiert wurde. (3) (Eventuell authentifiziert sich auch der Client mittels Zertifikat (Schritt 4))
- Der Client erzeugt eine Zufallszahl, verschlüsselt sie mit dem Public Key des Servers und sendet sie diesem (5). Wenn Client Authentification verlangt wird, dann signiert der Client zuvor die Zahl noch mit seinem Private Key. (das [R]c im Schritt 5)

- Beide leiten aus dieser Zufallszahl einen Session Key ab. Das ist ein einmalig benutzbarer symmetrischer Schlüssel, der während der Verbindung zum Ver- und Entschlüsseln der Daten genutzt wird. Die Nachrichten, die die Kommunikationspartner sich nun gegenseitig zusenden, werden nur noch verschlüsselt übertragen.



55. Geben Sie eine Definition für "Cryptography" an und erläutern Sie die Funktionsweise von symmetrischen und asymmetrischen Verschlüsselungsverfahren. Gehen Sie dabei auch auf die spezifischen Vor- und Nachteile ein und geben Sie konkrete Beispiele für Algorithmen an.

Fest verbunden mit dem Begriff Sicherheit in verteilten Systeme ist der Begriff der Kryptographie. Sie befasst sich im Prinzip mit dem Ver- und Entschlüsseln von Daten. Sie wird auch als Technik des "Versteckens" von Information bezeichnet. Es geht also darum, dass ein Sender seine Nachricht verschlüsselt (encrypt) und den verschlüsselten Text (ciphertext) an den Empfänger schickt, welcher diesen dann wieder entschlüsseln (decrypt) kann um auf den Plain-Text zu kommen.

Bei **symetrischer Verschlüsselung** verwenden beide Kommunikationspartner den gleichen Schlüssel zur Ver- und Entschlüsselung. Es müssen Information und Schlüssel gleichzeitig verschickt werden. Es ist also notwendig den Schlüssel über einen sicheren Kanal zu senden.

- Vorteil: schneller als Asymmetrische Verschlüsselung
- Nachteile:
  - der Schlüssel muss geheim gehalten werden
  - es werden N(N-1)/2 Schlüssel benötigt und jeder Host muss N-1 Schlüssel verwalten ==> KDC (Key Distribution Center) wächst exponentiell
- Bsp: DES (Data Encryption Standard, wegen der geringen Schlüssellänge von 56 Bit wird aus Sicherheitsgründen eher das Triple DES gewählt) Anm.: Ich denke weder DES noch Triple DES kann auch nur irgendwie als sicher beschrieben werden.

Bei **asymetrischer Verschlüsselung** hat jeder Partner einen Schlüssel zum Ver- und einen zum Entschlüsseln. Der zum Verschlüsseln ist öffentlich, der zum Entschlüsseln privat. Die beiden Schlüssel sind unterschiedlich bilden aber zusammen ein eindeutiges Paar - dh, wenn eine Nachricht mit dem public Key von A verschlüsselt wurde, so kann diese NUR A mit seinem private Key entschlüsseln.

- **Vorteil**:
  - nur eine Person kann die Nachricht wieder entschlüsseln (Sicherheit)
  - Schlüsselverteilung einfach, da Public Key eigentlich nutzlos für Angreifer
  - Schlüsselanzahl wächst linear

- **Nachteil**:
    - viel langsamer
    - jeder hält trotzdem noch immer seine eigenen 2 (public + private) + die public keys der N-1 anderen Teilnehmer ==> KDC (kann jetzt nicht als Nachteil gesehen werden)

Bsp: RSA (verwendet für Verschlüsselung und digitale Signatur)

Um die Vorteile beider Verfahren zu nutzen gibt es hybride Verfahren, bei denen mittels asymmetrischer Verschlüsselung ein symetrischer Schlüssel, ein sogenannter Session Key, ausgetauscht wird.

56. Welche Eigenschaften erwartet man sich von einem "Secure Channel"? Geben Sie jeweils auch Beispiele für unerwünschte Effekte an. Erklären Sie wie sich zwei Kommunikationsteilnehmer basierend auf Public Key Cryptography gegenseitig authentifizieren können.

Secure Channel: Wie der Name schon sagt, ist ein Secure Channel ein sicherer Kanal zwischen zwei oder mehreren Teilnehmer, der vor Angriffen (z.B.: Eavesdropping, Spoofing, Message Tampering, etc.) schützen soll. Die meisten Angriffe kann man durch eine Verschlüsselung des Kanals abwehren (Public- und Private-Key Encryption). Er schützt aber nicht gegen Interruption.

Unerwünschte Effekte eines nicht verschlüsselten Kanals

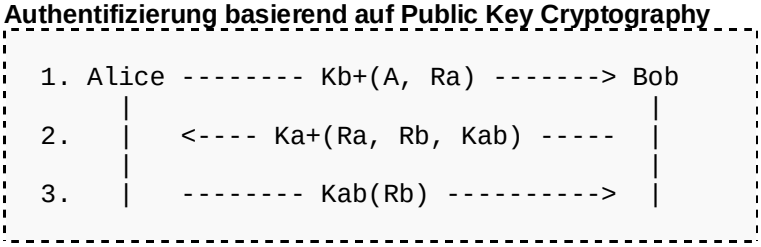
- Sender der Nachricht kann nicht bestimmt werden (Spoofing):

```
'A' -> B : "I want to buy 10 widgets."
B -> A : "Here are your 10 widgets."
A -> B : "I did not order widgets."
```

- Nachrichtenänderung (Message Tampering):

```
A -> B : "I want to buy 10 widgets."
B -> A : "Here are your 100 widgets."
A -> B : "I ordered 10!"
B -> A : "I received an order for 100!"
```

Authentifizierung basierend auf Public Key Cryptography

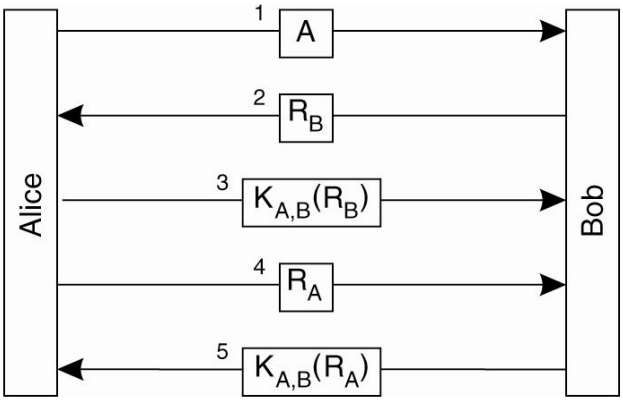


Die Kommunikationspartner besitzen den öffentlichen Schlüssel (K+) des jeweils anderen.

1. Alice schickt Bob eine Nachricht mit dem Inhalt 'A' ("Ich bin Alice") und einer Challenge Ra (z.B.: Zufallszahl). Diese Nachricht wird mit Bobs öffentlichem Schlüssel Kb+ verschlüsselt.
2. Da nur Bob die Nachricht entschlüsseln kann, bekommt er die Challenge Ra. Er generiert einen Session Key Kab, der für die weitere Sitzung des Secure Channels verwendet wird. Bob antwortet mit Ra, einer neuen Challenge Rb und dem Session Key verschlüsselt mit dem öffentlichen Schlüssel von Alice Ka+.
3. Alice ist nun in Besitz des Session Keys und bestätigt Bobs Challenge Rb mit dem Session Key verschlüsselt.

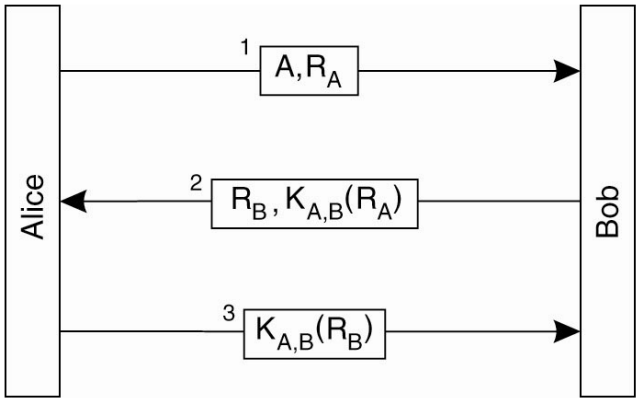
57. Erklären Sie, wie sich zwei Kommunikationsteilnehmer basierend auf symmetrischen Schlüsseln gegenseitig authentifizieren können. Zeigen Sie auch auf, welche Probleme entstehen können, wenn ein Protokoll falsch "optimiert" wird.

1. Alice sendet ihre Identität an Bob um einen Kommunikationskanal zu eröffnen.
2. Bob Sendet eine Aufforderung Rb an Alice (Kann z.B. eine Zufallszahl sein)
3. Alice verschlüsselt es mit dem secret Key Kab (den sie mit Bob teilt) und schickt die Nachricht an Bob. Bob kann die Nachricht entschlüsseln und festzustellen, ob sie Rb enthält, wenn das der Fall ist, weiß er, dass Alice an der anderen Seite sitzt. Alice weiß aber noch nicht, dass es sich sicher um Bob handelt auf der anderen Seite.
4. Alice sendet eine Aufforderung Ra an Bob.
5. Bob sendet Kab(Ra) zurück.

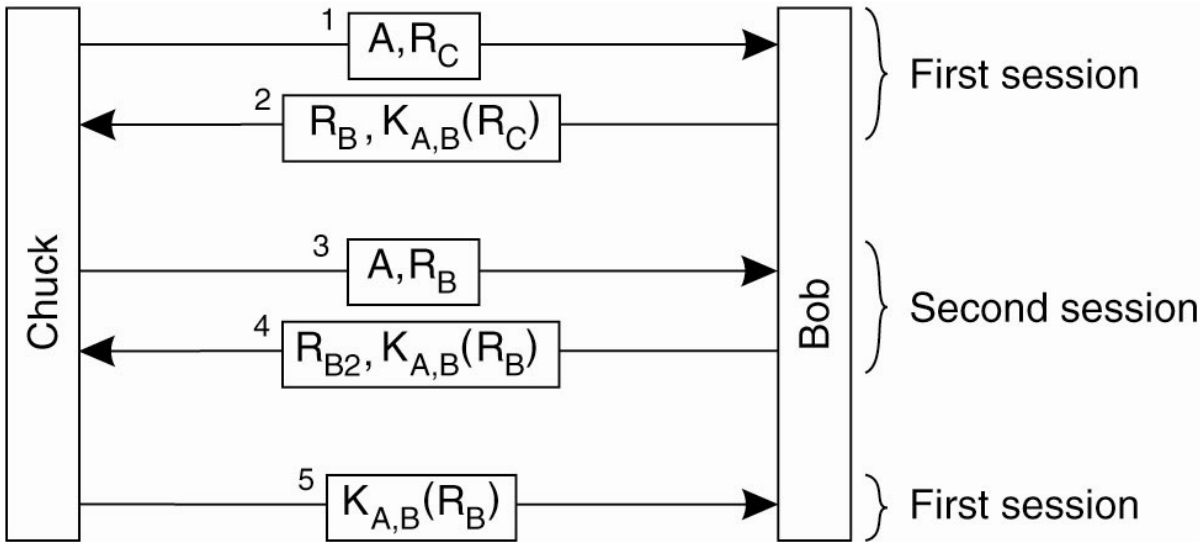


Optimierung durch Reduzierung:

1. Alice sendet ihre Identität an Bob und gleichzeitig eine Zufallszahl
2. Bob schickt ebenfalls eine Zufallszahl, und verschlüsselt mit dem secret Key die die von Alice erhaltene Zufallszahl
3. Wenn die verschlüsselte Zufallszahl gleich der von ihr gesendeten ist, weiß sie, dass Bob auf der anderen Seite sitzt. Bob weiß aber noch nicht, ob es wirklich Alice ist. Deshalb verschlüsselt Alice Bobs Zufallszahl und schickt sie ihm



Problem: Reflaction Attack: Chuck gibt sich als Alice aus. Er lässt Bob die Nachricht selbst in einem zweiten Kanal verschlüsseln.



58. Was ist eine digitale Signatur, welche Garantien bietet sie und wie funktioniert sie? Was ist eine Hash Funktion und welche Eigenschaften muss sie erfüllen, damit sie im Rahmen einer digitalen Signatur eingesetzt werden kann.

Digitale Signatur Eine digitale Signatur ist ein Anhang an eine Nachricht, die die Authentizität des Absenders A (Garantie, dass die Nachricht tatsächlich von A gesendet wurde) dem Empfänger B der Nachricht bestätigt und aber auch gleichzeitig eine Veränderung der Nachricht verhindert, da die Signatur eindeutig an den Inhalt der Nachricht gebunden ist.

Funktionsweise der digitalen Signatur: Von einer zu signierenden Nachricht wird der Hashwert gebildet. Dieser Hashwert wird mit dem privaten Schlüssel des Absenders A verschlüsselt (das ist die Signatur) und an die Nachricht angehängt. Der Empfänger der Nachricht bildet wiederum den Hashwert und entschlüsselt die Signatur mit dem öffentlichen Schlüssel des Absenders. Wenn die beiden Hashwerte überein stimmen, so kann man davon ausgehen, dass die Nachricht tatsächlich vom Absender A stammt und unverändert ist.

Hashfunktion Eine Hashfunktion H() bestimmt von einer Eingabe m einen Wert h konstanter Länge. Dieser Wert wird Hashwert genannt. Folgende Eigenschaften muss/sollte eine gute Hashfunktion erfüllen:

- keine Umkehrungfunktion H'(), die es erlaubt den Eingabewert m anhand des Hashwertes h zu bestimmen. (Hashfunktion sind Einwegfunktionen)
- Strong collision resistance. It is hard to find any x and y such that H(x) = H(y).
- Die Länge des Hashwertes soll immer gleich sein, und unabhängig von der Länge des Plain Textes.

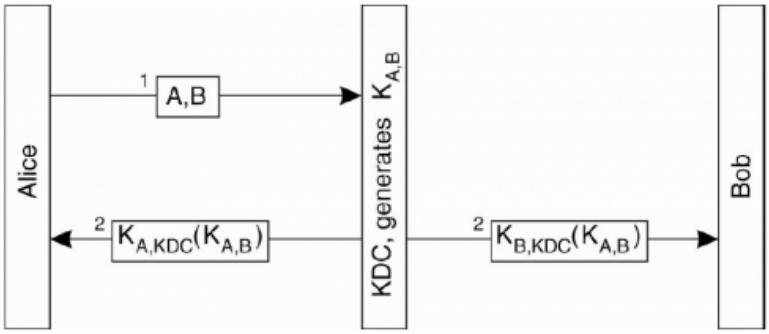
59. Was ist ein Key Distribution Center (KDC), wozu wird es eingesetzt und welchen Vorteil bietet es? Geben Sie ein konkretes Verfahren zur gegenseitigen Authentifizierung von Kommunikationsteilnehmern mit Hilfe eines KDC an und beschreiben Sie für jeden Schritt, wer sich wem gegenüber bereits authentifiziert hat.

Eines der größten Probleme bei Shared Secret Key Authentications ist die Skalierbarkeit der Systeme. Es müssen einfach zu viele Schlüssel gewartet werden. Bei N Hosts im System muss jeder der Hosts mit N-1 anderen einen Secret Key teilen. Im System gibt es dann N(N-1)/2 Schlüssel und jeder Host muss N-1 warten.
Eine Alternative dazu (zur "Schlüsselwartung") stellt das KDC dar (Key Distribution Center). Es teilt einen Secret Key mit jedem Host aber die Hosts untereinander müssen keine Secret Keys mehr teilen. Es müssen also nur noch N Keys gewartet werden.

Authentifizierung mit KDC:

Die Grundidee ist folgende: das KDC händigt jedem der Teilnehmer einen Key aus, mit dem diese dann untereinander kommunizieren können.
Ablauf:

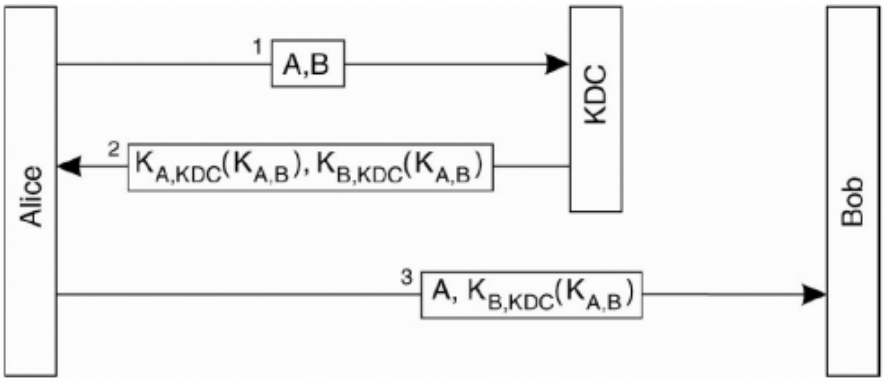
1. Alice sendet eine Message an das KDC wo sie bekannt gibt, mit wem sie sprechen will
2. das KDC retouniert einen Shared Secret Key Ka,b an Alice und Bob, welcher mit dem Secret Key des jeweiligen Empfängers verschlüsselt wurde



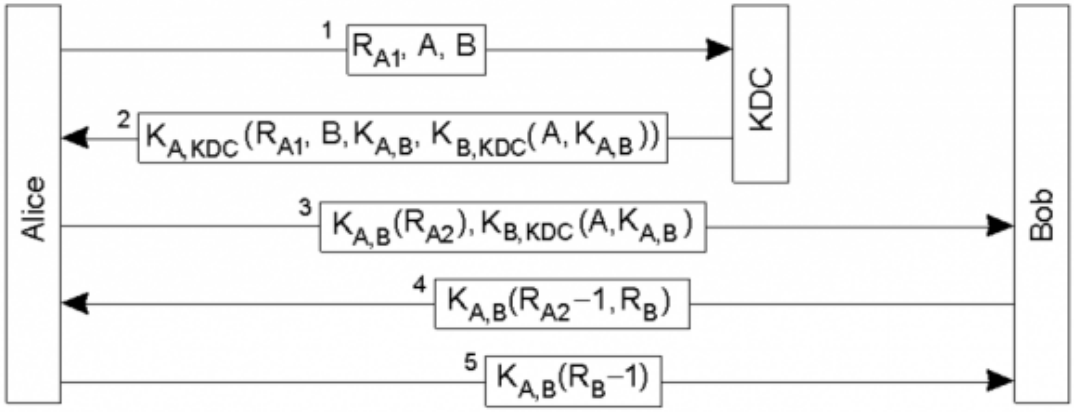
Hierbei gibt es nur ein Problem: Alice könnte den Schlüssel schon bekommen haben und eine Kanal zu Bob aufmachen, obwohl dieser den Schlüssel noch nicht hat (oder noch nicht dazu bereit ist).

Abhilfe schafft folgendes angepasste Protokoll.

Hierbei erhält Bob nicht vom KDC sondern von Alice seinen Schlüssel übermittelt. Dieser Ansatz wird als Ticketing bezeichnet (die Message Kb,kdc(Ka,b) ist das Ticket). Auch hier kann nur Bob etwas mit dem Ticket anfangen, da nur er den Private Key zum Entschlüsseln hat.



Ein konkretes Protokoll für die Authentifizierung, welches nun diesen Ansatz verwendet, ist das **Needham-Schroeder-authentication protocol**.



Ablauf:

1. Alice sendet eine Challange Ra1 und die Teilnehmerdaten an das KDC. Ra1 ist hierbei eine sog. **nonce** - eine zufällige Nummer die nur einmal verwendet wird, um 2 Nachrichten miteinander in Relation zu setzen. Wenn Ra1 nicht in Message 1 gewesen wäre, so könnte Alice die Antwort 2 nicht deuten bzw nicht korrekt deuten, da diese Nachricht von einer alten Anfrage stammen könnte.
2. das KDC liefert das Ticket retour und den Secret Key Kab für die Kommunikation mit Bob. In der Message ist auch B enthalten was gemacht wird, um einen eventuellen Eindringling nicht die Möglichkeit zu geben, die Kommunikation zu verhindern. Annahme: Eindringling C modifiziert Message 1 und gibt statt B seine Identität C mit. Ohne B in der Return-Message wüsste A dann nicht, ob nun wirklich mit B kommuniziert wird.
3. nun kann der Channel aufgebaut werden, indem das Ticket an Bob geschickt wird. Hierfür wird wieder eine neue **nonce** verwendet: Ra2
4. Bob liefert nun Ra2-1 retour. Damit weiß Alice, dass Bob einerseits die Nachricht genau entschlüsseln konnte und andererseits auch, dass die Return Message zur Message 3 gehört. Mit dem -1 weiß Alice aber auch, dass Bob die Challenge lesen konnte, die ja mit dem Shared Key Kab verschlüsselt wurde (wäre aber nicht notwendig)
5. als letztes muss nun nur noch Alice Bob zeigen, dass sie auch in der Lage ist seine Challenge zu entschlüsseln indem sie ihm die Challenge - 1 verschlüsselt retour sendet

60. Ordnen Sie in der folgenden Taxonomie den vier Typen von Koordinationsmodellen jeweils zwei konkrete Techniken, Beispiele oder Systeme zu.

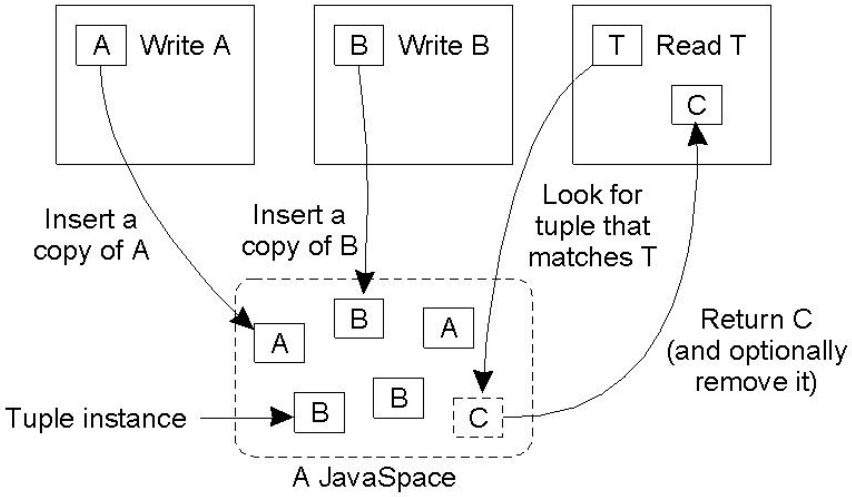
		Temporal	
		Coupled	Uncoupled
Referential	Coupled	(a) Direct	(b) Mailbox
	Uncoupled	(c) Meeting oriented	(d) Generative Communication

a) Direct: Telefon, TCP; RPC/RMI/Transient Message Passing
b) Mailbox. Mail/Brief/SMS; Peristant message passing
c) Meeting orented: TIB/Rendevous, Publish/Subscribe Systeme, event & subject-based, "ad hoc" communication
d) Generative Communiation: Jini/Linda; tuple spaces; peristant DSM (Distributed Shared Memory); associative; JavaSpaces

61. Erläutern Sie das Grundprinzip von publish/subscribe als Kommunikations-/Koordinationsmechanismus. Worin bestehen die Möglichkeiten aber auch die Probleme dieses Mechanismus (v.a. dessen Implementierung). Erläutern Sie weiters JINI und JavaSpaces als konkrete Technologien.

Publish/Subscribe-Systeme, bei denen Nachrichten mit einem gewissen Typ versehen werden. Diese werden an alle Prozesse zugestellt, die sich dafür interessieren (indem sie sich vorher dafür registriert haben). Funktioniert nur wenn beide Prozesse aktive sind. Dieser Ansatz wird als Meeting-based bezeichnet. [vor-Nachteile]

Jini setzt Generative Communication um. Es speichert Tupel von Java-Objekten in einem Shared Dataspace (**JavaSpace**), aus dem sie von interessierten Prozessen durch Matching mit einem Template (ebenfalls ein Tupel aus Objekten, das aber nicht vollständig ausgefüllt sein muss) wieder abgefragt werden können. Das Abfragen kann dabei auch das gelesene Tupel auch gleich aus dem Datastore entfernen (take-Operation). Neben dem Zugriff auf JavaSpaces unterstützt Jini auch noch Naming und Security (Authentification, so dass nur berechtigte Prozesse auf den Datastore zugreifen können).



62. Was ist ein Web-Server? Wozu dient das Hypertext Transfer Protokoll HTTP? Nennen Sie zwei HTTP Operationen. Diskutieren Sie die Funktionsweise eines CGI-Programms.

Ein Webserver ist ein Programm das eingehende HTTP Anforderungen erfüllt, indem er das angeforderte Dokument lädt und es an den Client schickt.

HTTP ist ein allgemeines Client-Server-Protokoll, das für viele dokumentenbasierte Übertragungen im Internet verwendet wird. Es ist nicht für eine bestimmte Anwendung spezialisiert und kann Dokumente in beide Richtungen übertragen. Am bekanntesten ist die Übertragung gewöhnlicher Webseiten, es können aber auch ganze Anwendungen auf http aufgebaut werden. Obwohl http selbst zustandslos ist kann man mit Hilfe von Cookies ein zustandsbehaftetes Protokoll simulieren, was für moderne Webabwendungen notwendig ist. Insgesamt wird http verwendet um von Clients aus bestimmte Services in Anspruch nehmen zu können.

HTTP-Operationen:

Operation	Beschreibung	
Get	Dokument an den Client schicken	Mit "Get" bekommt ein Client ein Dokument vom Server.
Put	Dokument speichern	Mit "Put" fragt der Client den Server, ob er ein Dokument unter einem bestimmten Namen speichern kann. (Der Server weist zwar nicht einfach Anfragen zurück, akzeptiert aber nur jene von Authorisierten Clients).
Post	Bereitstellung von Daten die einem Dokument hinzugefügt werden soll	"Post" ist ähnlich dem Speichern eines Dokumentes, nur dass der Client eine Anfrage schickt, um Daten zu einem Dokument oder einer Sammlung von Dokumenten hinzuzufügen. (Bsp.: Artikel zu einer Newsgroup hinzufügen).
Delete	Dokument löschen	"Delete" ist eine Anfrage um ein Dokument mit einem bestimmten Namen zu löschen.
Head	Anforderungen, Head eines Dokument zurückzugeben	"Head" wird verwendet am Server, wenn der Client nicht das eigentliche Dokument will, aber die dazugehörigen Metadaten

CGI ist eine der ersten Erweiterungen der HTML Basis-Architektur und dient der Unterstützung einfacher Benutzer-Interaktion. Es definiert eine Standardmethode, wie der Webserver ein Programm ausführen kann mit Benutzerdaten als Input. CGI Programme arbeiten meist mit der lokal auf dem WEB Server gespeicherten Datenbank. Nach Verarbeitung der Daten erzeugt das Programm ein html-Dokument und gibt dies dem Server zurück, der es dem Client weiter gibt (generierte Dokumente).