VO Verteilte Systeme
Fragenkatalog WS 2008/09
v1 | 27.04.2009

Ausarbeitung:
Matthias Wallner, TU Wien, April 2009. E-mail: hiace@gmx.net
Einige Passagen sind dem Wiki-Eintrag zur VO-Prüfung entnommen, danke den Postern!

Grafiken:
Distributed Systems. Tanenbaum A., Van Steen M.. Pearson International, 2007

Hinweis:
Kein Anspruch auf Vollständigkeit, Nutzung nur als Lernbehelf für Studenten der TU Wien

Links zur VO:
http://vowi.fsinf.at/wiki/TU_Wien:Verteilte_Systeme_VO_(Göschka)
http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/

Grundlagen und Konzepte

Buch: Kap. 1 und 2 bis inkl. 2.2

Fragen:

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.**

Unter einem Verteilten System versteht man eine Menge von unabhängigen Rechnern, die von den Usern als einziges, zusammenhängendes System wahrgenommen werden. Charakteristisch dafür ist, dass strukturelle Unterschiede zwischen den Computern oder die Details der Kommunikation für die Benutzer nicht sichtbar, also transparent, sind. In der Interaktion mit dem System ist es weiters irrelevant, zu welcher Zeit oder an welchem Ort die Benutzung erfolgt. Weiters sollen Verteilte Systeme leicht erweiterbar sowie permanent verfügbar sein, auch im Fall eines Ausfalls einzelner Komponenten. Die genannten Anforderungen werden durch eine Schichten-Organisation erfüllt, die einen eigenen Middleware-Layer bereitstellt. Dieser erstreckt sich über eine Vielzahl an Maschinen, und bietet, unabhängig von der zugrundeliegenden Hardware für jede Applikation das selbe Interface.

Beim Entwurf Verteilter Systeme sollte explizit auf die Verteilung der Komponenten eingegangen werden, wobei einige Problemquellen einzubeziehen sind. Etwa ist eine vollständige Fehlertransparenz aufgrund der inhärenten Unzuverlässigkeit von Netzwerken kaum zu erreichen. Auch wird man mit den Nachteilen der örtlichen Verteilung, insbesondere mit unterschiedlichen Zugriffszeiten oder mit Beschränkungen der Bandbreite konfrontiert werden.

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 ist eines der wichtigsten Ziele von Verteilten Systemen. Sie soll den Benutzern ein einheitliches Interface bieten und so die eigentliche Verteilung der Ressourcen verbergen. *Access Transparency* ist dafür verantwortlich, dass Unterschiede in der Darstellung von Daten, wie sie auf verschiedenen Systemen vorherrschen, für den User nicht sichtbar sind. *Location Transparency* ermöglicht ein einfaches Auffinden von Rechnern, unabhängig von ihrer physischen Position. Dies wird u.a. durch Naming erreicht. *Migration Transparency* bedeutet, dass örtliche Veränderungen keinen Einfluss auf die Art des Zugriffs haben, während *Relocation Transparency* einen fehlerfreien Zugriff explizit während eines Ortswechsels ermöglicht (zB W-LAN). Weiters können in Verteilten Systemen Kopien von Daten zur Steigerung von Verfügbarkeit oder Performance angelegt werden, worauf sich die *Replication Transparency* bezieht. In diesem Fall sollte auch eine *Location Transparency* gegeben sein, um auf unterschiedliche Orte der Kopien verweisen zu könenn. *Concurrency Transparency* ermöglicht einen gleichzeitigen Zugriff auf bestimmte Ressourcen, wobei ein konsistenter Zustand der Daten, etwa durch Locking-Mechanismen gewährleistet ist. *Failure Transparency* stellt dem User im Fall eines Fehlers eine volle Funktionalität zur Verfügung, wobei eine Unterscheidung zwischen einer schlechten Performance und einem Fehler allerdings nicht immer getroffen werden kann.

Nicht immer ist eine totale Transparenz erstrebenswert. In manchen Situationen soll etwa aus rein praktischen Gründen auf den nächst gelegenen Rechner zugegriffen werden (zB Print-Server) was einen expliziten Bezug auf die Position des Anwenders voraussetzt. Ubiquitous Computing zielt als solches überhaupt auf die völlige Sichtbarmachung der Verteilung ab. In Bezug auf das Internet ist auch eine Fehlertransparenz nicht immer wünschenswert, etwa dann wenn wiederholt auf einen möglicherweise nicht erreichbaren Server zugegriffen wird, bevor ein anderer kontaktiert wird. Ein Faktum bei großen Netzwerken ist die Zunahme der Zugriffszeit, was sich bei entfernt positionierten Einheiten eines Systems negativ auf die Performance auswirkt. Generell ist zu erwähnen, dass Transparenz meist nur mit einer verschlechterten Performance erkauft werden kann.

3. **Was versteht man unter "Openness"?**

Openness ist ein weiteres wichtiges Ziel von Verteilten Systemen. Es sorgt dafür, dass Services unter Rücksichtnahme auf Standards, die in Form von Protokollen Syntax und Semantik regeln, erstellt bzw. angeboten werden. Die Spezifikation erfolgt mittels Interfaces, die meist durch eine Interface Definition Language beschrieben werden und Bezug auf die Syntax des Systems, also Funktionen, Parameter, Exceptions, etc., nehmen. Damit wird ermöglicht, dass sich unterschiedliche Prozesse über ihre Interfaces unterhalten können, unabhängig von ihrer tatsächlichen Implementierung. Interfaces sollten *komplett* und *neutral* sein, d.h. alles zu beinhalten, was für eine spätere Implementierung vorausgesetzt wird, sowie nicht

eine bestimmte Umsetzung vorschreiben. Zwei wichtige Parameter für Openness sind *Interoperabilität* sowie Portabilität. Ersteres bezeichnet, in welchem Ausmaß zwei verschiedene Implementierungen unter Bezug auf die Services des jeweils anderen nebeneinander existieren oder zusammen arbeiten können. *Portabilität* bezieht sich auf die Möglichkeit der Ausführung eines für System A entworfenen Programms auf System B ohne Modifikation, sofern die selben Interfaces zur Verfügung stehen. Schlussendlich muss es in einem Verteilen System auch möglich sein, Funktionalität problemlos hinzuzufügen bzw. existierende Features zu ersetzen (*Erweiterbarkeit*).

4. Erläutern Sie Probleme und Lösungsansätze für "Scalability".

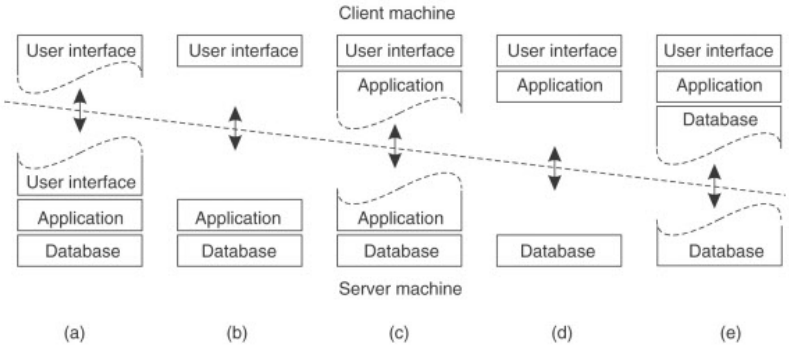
Eine Skalierung kann zum einen in Bezug auf die Größe des Systems und die Zahl der Anwender erfolgen. Dabei gibt es gewisse Limitierungen, die etwa bei zentralisierten Server auftreten. Problem ist hier weniger die Rechenleistung oder Speicherkapazität, als vielmehr die *Kommunikation*, die ein weiteres Wachstum verhindern könnte. Hingegen ist in manchen Fällen gerade eine solche Zentralisierung wünschenswert, etwa im Fall von sicherheitskritischen Applikationen wie Banken-Server.

Ein wesentliches Problem in Bezug auf eine geographische Skalierung von existierenden Systemen in lokalen Netzwerken ist, dass diese auf synchroner Kommunikation basieren. Das bedeutet, dass der Client so lange blockiert, bis er eine Anwort vom Server erhält. In wide-area Netzwerken führt diese Vorgangsweise allerdings zu hohen Latenzen. Während LANs durch eine zuverlässige Kommunikation, gekennzeichnet sind, die vornehmlich auf Broadcasting (Adressaten sind alle Maschinen!) basiert, ist diese Art der Lokalisierung von Services in globaleren Netzwerken mit ursprünglich Punkt-zu-Punkt-Verbindungen alles andere als sinnvoll. Ein weiteres Problem der Skalierung liegt in den unterschiedlichen *administrativen Richtlinien* betreffend Nutzung der Ressourcen (Entgelt), Management oder Security. Bei einer Erweiterung des Systems in neue Domains muss ersteres vor unberechtigten und möglicherweise gefährlichen Zugriffen geschützt werden, etwa durch Zugriffsrechte für fremden Code.

Um die genannten Nachteile zu umgehen, gibt es einige Lösungsansätze: Um die mit der geographischen Skalierung einhergehenden Latenzen zu verdecken, sollte ein Warten des Prozesses auf Antwort vermieden und stattdessen eine für den Anwender sinnvolle Aktion ausgeführt werden (*asynchrone Kommunikation*). Beim Eintreffen der Rückmeldung wird das laufende Programm unterbrochen, bzw. wird ein eigener Thread erstellt, der sich um die Bearbeitung der Antwort kümmert. In manchen Fällen wie zB interaktiven Anwendungen ist es hingegen effektiver, die Kommunikation auf das Wesentlichste zu reduzieren, indem ein Teil der Rechenarbeit des Servers auf den Client übertragen wird (zB Javascript). Eine weitere Technik zur Skalierung ist *Verteilung*, wo wie im Fall des DNS Naming Service Information aufgeteilt und gezielt im System verschickt wird. Auch die *Replikation* von Komponenten ist in der Lage, die Probleme der Skalierung zu vermindern. Durch eine Verteilung der Replikate wird sowohl die Verfügbarkeit, als auch die Verteilung der Systemlast verbessert, was sich im Fall einer nahe gelegenen Kopie positiv auf die Latenzzeit auswirkt. Ein Spezialfall der Replikation ist *Caching*, das explizit vom Client veranlasst wird und nicht geplant, sondern ad-hoc erfolgt. Die beiden letztgenannten Verfahren bringen allerdings Konsistenzprobleme mit sich, nämlich dann, wenn eine Kopie verändert wird.

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?

Anwendungen können logisch in 3 Schichten, User-Interface, Prozess-Komponente und Datenschicht, unterteilt werden. Auf ähnliche Weise können im Rahmen von *Multitiered Architectures* Client-Server Systeme strukturiert werden, wobei jede Einheit (Tier) einer Programmschicht einer vertikalen Verteilung entspricht. Logisch unterschiedliche Komponenten werden somit auch physisch, auf verschiedene Maschinen, verteilt. Dabe gibt es eine Vielzahl von Möglichkeiten, wie die Verteilung erfolgen kann. Eine mögliche *two-tiered* Architektur ist zB durch ein User-Interface auf Client-Seite, sowie die Verarbeitungsebene und Datenbank auf Server-Seite gekennzeichnet. Generell handelt es sich dabei um eine Ausprägung eines *thin client*, was bedeutet, dass der Client lokal lediglich ein Terminal/GUI vorsieht, alle rechenrelevanten Dinge aber am Server passieren. Nachteil solcher Systeme ist eine vom User empfundene schlechtere Performance. Beispiel für einen *fat client* wäre eine Client-Workstation, die über Netzwerk mit einer Datenbank verbunden ist, also der Client den größten Anteil der Rechenlast zu tragen hat.



Ein Java Applet stellt nach obiger Definition eher ein Beispiel eines Thick Client dar, da der Programmcode vom Server and den Client geschickt und dort zur Ausführung gebracht wird.

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?**

Von einer *horizontalen Verteilung (auch Peer-to-Peer System)* spricht man dann, wenn Client oder Server physisch in logisch gleichwertige Teile aufgeteilt werden, wobei jeder Teil einen eigenen Bereich des Gesamtset an Daten behandelt. Auf diese Weise kann die Systemlast effizient ausbalanciert werden. Eine Besonderheit ist, dass dabei jeder Prozess gleichzeitig als Client und als Server agiert und somit auch als *Servent* bezeichnet wird. Die Interaktion unter den Prozessen ist daher symmetrisch.
Derartige P2P-Netzwerke sind in einem Overlay Network organisiert, in dem Prozesse durch Nodes und Kommunikationswege durch Links dargestellt werden. Dabei wird zwischen strukturierten und unstrukturierten Architekturen unterschieden.

Horizontale Verteilung wird häufig mit einer vertikalen Verteilung kombiniert.

Communication (1) - Middleware und RPC

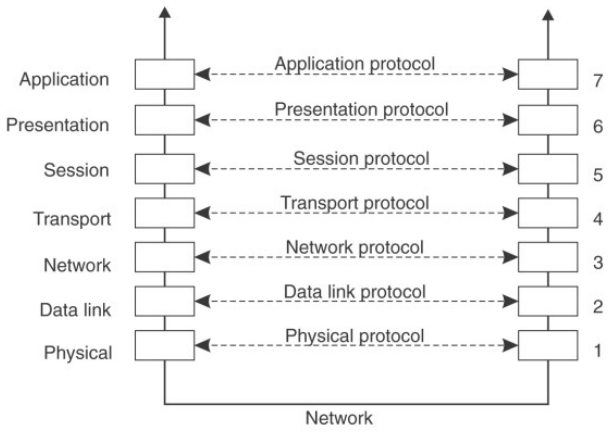
Buch: 2.1 (w.h.), 2.3, 2.4, 4.1, 4.2

Fragen:

1. **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?**

Das Open Systems Interconnection Reference Model erlaubt es Systemen, unter Rücksichtnahme auf bestimmte Regeln (Protokolle) zu kommunizieren. Das Modell ist in 7 Schichten unterteilt, die sich mit verschiedenen Aspekten der Kommunikation befassen. Will ein Prozess A mit Prozess B, der sich auf einem anderen Rechner befindet, kommunizieren, wird die Nachricht zuerst an den Application Layer geleitet. Dieser fügt der Nachricht Header-Informationen hinzu und leitet sie an den darunter liegenden Presentation Layer weiter. Das selbe Schema wird hinab bis zum Physical Layer fortgesetzt, wobei jeder Layer die Mitteilung um seinen eigenen Header oder Trailer ergänzt. Schlussendlich wird die Nachricht über das den Rechnern zugrundeliegende Netzwerk an den Empfänger geschickt, der wiederum das Schichtenmodell von unten nach oben durchläuft und dabei Header- und Trailer-Informationen ausliest.

Die TCP/IP Protocol Suite stellt eine Variante des ISO/OSI Modells dar, die auf die Datenübertragung im Internet abzielt. Aufgabe von TCP/IP ist es, eine Nachricht in Datenpakete zu zerkleinern (TCP) und diese über mehrere Stationen (Hops) zum Receiver zu routen (IP). Generell muss zwischen verbindungsorientierten und verbindungslosen Protokollen unterschieden werden. Während bei ersteren die Pakete in der korrekten Sequenz am Empfänger ankommen, können bei verbindungslosen Protokollen wie UDP die Pakete jeweils über verschiedene Wege geroutet werden und werden auf diese Weise unsortiert empfangen.



Transport-Layer Protokolle sind alleine für Verteilte Systeme nicht ausreichend. So können etwa zuverlässige Multicasting-Services nur unter Rücksichtnahme auf die Anforderungen der Anwendung implementiert werden. Aus diesem Grund werden der Middleware eigene, modifizierbare Kommunikations-Protokolle zur Verfügung gestellt, die u.a. die Grundlage für RMI darstellen. Presentation und Session Layer werden dabei durch einen Middleware Layer ersetzt.

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

Middleware ist eine Softwarekomponente die viele oft von verteilten Systemen benötigte Services zusammenfasst und in parametrisierbarer Form anbietet. Man muss daher gleiche Funktionalitäten nicht immer neu implementieren. Die Schicht ist logisch zwischen Application- und Transportlayer einzuordnen und stellt auf diese Weise einen Grad von Distribution Transparency bereit. Services von Middleware umfassen zB Verschlüsselung, Authentifizierung, Security, Replikation, Access-Transparenz und Naming.

In der Praxis sind Middleware Systeme nach gewissen Architekturschemata aufgebaut (Schichtenmodell, objektorientiert [CORBA], event-basierend, datenzentriert). Dies hat den Nachteil, dass Middleware nicht uneingeschränkt out-of-the-box von Anwendungen genutzt werden kann und an die spezifischen Anforderungen angepasst werden muss. Eine Lösung für dieses Problem wäre, unterschiedliche Middleware-Versionen zu entwerfen bzw. Middlewaresysteme leicht konfigurierbar und adaptierbar zu gestalten.

3. **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.**

Um das Zusammenspiel von Middleware und Anwendung zu verbessern, wird bei der Entwicklung neuer Systeme auf eine striktere Trennung von Richtlinien und Mechanismen geachtet wodurch das Verhalten von Middleware besser modifiziert werden kann. Ein solcher Mechanismus ist ein *Interceptor*. Dieses Softwarekonstrukt unterbricht den aktuellen Kontrollfluss, um anderen, anwendungsspezifischen Code einzufügen/auszuführen. Als Beispiel führt Objekt A eine Methode von Objekt B aus, das sich auf einer anderen Maschine befindet. Diese Remote Object Invocation wird in 3 Schritten ausgeführt:

- Objekt A wird lokal ein Interface zur Verfügung gestellt, das exakt jenem von Objekt B entspricht. A ruft die Methode auf.
- Der Aufruf wird von der Middleware mittels eines eigenen Object-Invocation-Interfaces auf Maschine A in einen generischen Objektaufruf umgewandelt.
- Der generische Objektaufruf wird in eine Nachricht umgewandelt und über das vom lokalen OS angebotene Transport-Level Netzwerk-Interface versendet.

Aufgrund des sich ständig ändernden Umfeldes in Verteilten Systemen (Netzwerk-QoS, Hardware-Fehler, Mobilität) ist es nötig, den Programmablauf anzupassen, um eine sichere Ausführung gewährleisten zu können. Statt den Anwendungen selbst wird diese Aufgabe der Middleware überlassen. *Adaptive Software* wird durch 3 Grundtechniken charakterisiert:

- Separation of Concerns: Trennung des Systems in Basis- und Extra-Funktionalität (zB Sicherheit, Zuverlässigkeit) – schwierig
- Computational 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 (autonomic computing) bezeichnet high-level Feedback-Kontrollsysteme, die eine automatische Anpassung an Veränderungen ermöglichen. Eine Feedback-Schleife besteht aus drei wesentlichen Elementen. Zum einen muss das System überwacht werden, was eine Messung bzw. metrische Schätzung von bestimmten Parametern, etwa der Latenzzeit, erfordert. Die Messdaten werden anschließend einer Analyse unterzogen, wobei die Werte mit Referenzwerten verglichen und möglicherweise Maßnahmen zur Veränderung des Systems berechnet werden. Eine weitere Komponenete stellen die Veränderungs-Mechanismen dar, die ihrerseits etwa durch Replikation oder verändertes Scheduling die Leistung des Systems beeinflussen.

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

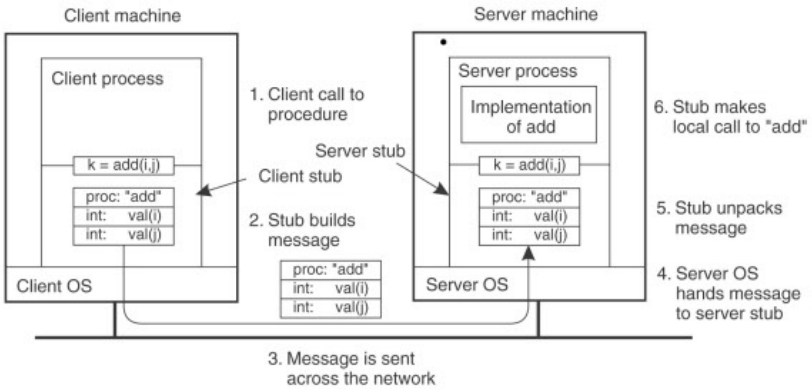
Ein Prozess auf Maschine A ruft eine Methode auf Maschine B auf und wird unterbrochen, worauf auf Maschine B die Methode ausgeführt wird. Information wird dabei in Form von Parametern von A nach B übertragen bzw. als Ergebnis der Prozedur an Prozess A zurück geführt. Charakteristisch für einen RPC ist, dass der gesamte Nachrichtenaustausch für den Programmierer unsichtbar abläuft. Grundidee ist also, den RPC wie einen gewöhnlichen, lokalen Aufruf aussehen zu lassen.

Für jede Remote-Prozedur wird ein sogenannter *Client Stub* angelegt, der auf gewöhnliche Weise aufgerufen werden kann. Allerdings wird im Gegensatz zu einem normalen Prozeduraufruf nicht das lokale OS aufgefordert, die Daten zu liefern, sondern die Parameter in eine Nachricht verpackt und das OS mittels SEND beauftragt, diese zum Server zu schicken. Daraufhin wird der Client durch RECEIVE geblockt, bis eine Anwort vom Server eintrifft. Nach der Übertragung wird die Nachricht vom OS des Servers an den *Server Stub* weitergeleitet, der die Parameter entpackt und den Code in einen lokalen Prozeduraufruf umwandelt. Nach der Ausführung durch den Server wird das Ergebnis an den Stub retourniert und die Nachricht analog zum oben angeführten Schema an den Client versandt bis schließlich das Resultat am Client-Prozess anliegt. Diese Vorgangsweise läuft komplett im Hintergrund ab, sodass seitens des Programmieres lediglich ein lokaler Prozeduraufruf zu machen ist.

5. **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"?**

Bei gewöhnlichen Prozeduraufrufen können Parameter in mehreren Varianten übergeben werden. Im Zuge von *call-by-value* wird der Parameter auf den Stack kopiert, für die aufgerufene Methode stellt der Wert eine

lokale Variable dar. Eine mögliche Modifikation durch die Prozedur hat für die Client-Seite keine Auswirkungen. Bei *call-by-reference* wird lediglich ein Zeiger auf eine Variable (d.h. auf eine Adresse) übergeben. Führt der Server eine Veränderung dieses Parameters durch, wird auch das Quell-Array in der aufrufenden Methode modifiziert. Mit *call-by-copy/restore* existiert ein weiterer Mechanismus. Hier wird die Variable wie bei call-by-value auf den Stack kopiert und in bearbeiteter Form, ähnlich call-by-reference, wieder zurück gegeben.



Handelt es sich um einen RPC, werden *Value Parameter* von Client Stubs, zusammen mit dem Namen der aufzurufenden Methode in eine Nachricht gepackt (*Parameter Marshaling*) und an den Server Stub weiter geleitet. Serverseitig überprüft der Stub die Nachricht auf die benötigte Prozedur und ruft diese lokal mit den initialisierten Variablen auf. Das Ergebnis der Prozedur wird wiederum vom Server Stub in die Nachricht gepackt und zurück zum Client geschickt, wo der Rückgabewert zur wartenden Client-Prozedur gelangt.

Probleme können in diesem Zusammenhang in großen heterogenen Systemen auftreten, wo verschiedene Arten von Rechnern zusammen arbeiten, die jeweils unterschiedliche Repräsentationen von Daten vorsehen. Ohne Vorkehrungen zu treffen würde es auf diese Weise zu falschen Interpretationen von Werten (zB Fließkommazahlen) kommen.

Die Übergabe von *Referenz Parametern* mittels RPC ist ein schwieriges Unterfangen. Da Pointer sich speziell auf den Adressraum der aufrufenden Methode beziehen, haben sie auf anderen Rechnern eine völlig andere Bedeutung. Eine Lösungsvariante ist, den Wert analog zu call-by-value in die Nachricht zu schreiben. Der Server Stub ruft nun die Methode mit einem Zeiger auf diesen Wert auf, wobei das Ergebnis diese Variable überschreibt und in der Nachricht dem Client übergeben wird. Dies entspricht in etwa einem call-by-copy/restore.

6. 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"?

Die Definition der Schnittstellen ist der wesentlichste Baustein von Client-Server Systemen. Sie wird mittels der Interface Definition Language in IDL Dateien spezifiziert, die Typdefinitionen, Konstantendeklarationen und Informationen beinhalten, die dem Marshaling von Parametern sowie dem Unmarshaling dienen. Generell kann man sagen, dass die IDL lediglich auf die Syntax der Aufrufe abzielt, semantische Informationen können als Kommentare ergänzend hinzugefügt werden. Jedes IDL File besitzt einen global eindeutigen Identifier für das spezifizierte Interface. Dieser wird vom uuidgen Programm generiert und im ersten RPC an den Server zur Verifizierung übergeben, um falsche Bindings zu vermeiden. Die IDL Datei wird nun um Name und Parameter der Remote-Prozeduren ergänzt und schlussendlich der IDL Compiler aufgerufen. Dessen Output umfasst:

- Header File: wird in Client- und Server-Code inkludiert, enthält UID, Typdefinitionen, etc.
- Client Stub: Prozeduren, die vom Client aufgerufen werden können (beinhaltet Marshaling).
- Server Stub: vom Server bei eintreffen des Aufrufs ausgeführte Methoden, die ihrerseits die tatsächlichen Methoden am Server aufrufen.

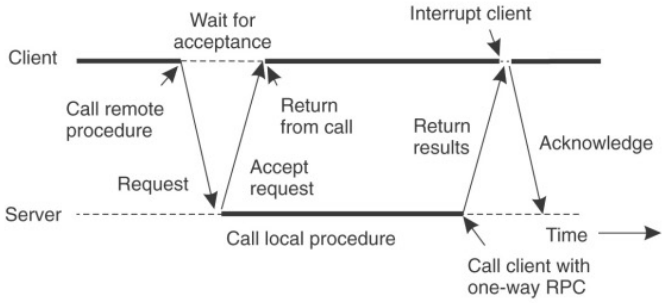
Schließlich müssen vom Programmierer Client- und Server-Code geschrieben werden. Dieser wird zusammen mit den 2 Stub-Prozeduren compiliert und nach Durchlaufen der Runtime Library zur Ausführung gebracht.

Um den Server von einem Client aus aufrufen zu können, muss ersterer registriert sein und mögliche

hereinkommende Aufrufe ausdrücklich zulassen. Ist der Server registriert, können er und in weiterer Folge der gewünschte Prozess vom Client über ein Location Service geortet werden. Dabei ist es für das Versenden von Nachrichten notwendig, dass dem Client der Endpunkt (Port) am Server bekannt ist. Dieser dient dem Server dazu, um hereinkommende Nachrichten, gerichtet an unterschiedliche Methoden, unterscheiden zu können. Hat der Server von seinem OS einen Port erhalten, registriert er diesen am DCE Daemon, der für die Verwaltung des Server-Endpunkt Mappings zuständig ist. Um im Netzwerk gefunden werden zu können, muss sich der Server beim Directory Service mit seiner Netzwerkadresse und einem Namen registrieren. Möchte ein Client nun ein *Binding* mit einem Server durchführen, fragt er zunächst beim Directory Server unter dem Namen des Servers nach und bekommt im Falle eines Treffers die dazugehörige Netzwerkadresse zurück. Danach kontaktiert er den DCE Daemon des Servers, der an einem Standard-Port läuft und beauftragt diesen mit einem Lookup der Port-Nummer des gewünschten Services. Der RPC kann somit durchgeführt werden, ein weiterer Lookup ist im späteren Verlauf der Kommunikation nicht erforderlich.

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

Persistente Kommunikation bedeutet, dass eine verschickte Nachricht von der Kommunikations-Middleware in einer ihrer Storage-Facilities so lange aufbewahrt wird, bis sie dem Empfänger zugestellt werden kann. Die Sender-Applikation kann somit nach dem Abschicken der Nachricht beendet werden. Dagegen sieht *transiente Kommunikation* lediglich eine Speicherung für jenen Zeitraum vor, in dem sowohl die Sender- als auch die Empfänger-Applikation laufen. Im Fall eines Übertragungsfehlers oder wenn der Empfänger nicht erreichbar ist, wird die Nachricht verworfen. Eine weitere Unterscheidung muss mit asynchroner bzw. synchroner Kommunikation werden. Asynchrone Kommunikation bedeutet, dass die Nachricht automatisch von der Middleware zwischengespeichert wird, sodass der Sender sofort nach Absenden der Nachricht mit der Ausführung seines übrigen Programmes fortfahren kann. Im Zuge synchroner Kommunikation blockiert der Sender so lange, bis seine Anfrage vom System bestätigt wird. Das kann jener Zeitpunkt sein, wo die Nachricht das erste Mal an der Middleware anliegt, dann, wenn die Nachricht von der Middleware an den Empfänger weitergeleitet wurde, oder, spätestens, mit der Antwort des Empfängers auf die Nachricht.



Es gibt Situationen, wo es für einen Client nicht notwendig ist, auf das Eintreffen einer Antwort zu warten, etwa wenn er keinen Rückgabewert erwartet. Dies wird im Rahmen von *asynchronen RPCs* unterstützt, indem der Server sofort bei Erhalt einer Nachricht eine Anwort an den Client schickt, um das Eintreffen zu bestätigen. Der Client fährt daraufhin mit seinem Programm fort, ohne zu blockieren. Obige Abbildung zeigt eine Kombination von 2 asynchronen RPCs (*deferred synchronous RPC*). Dies ist dann sinnvoll, wenn der Client eine Antwort vom Server erwartet, aber nicht bis zu ihrem Eintreffen warten will. Entscheidend ist, dass der zweite Aufruf (beinhaltet die Rückgabewerte der Methode) vom Server durchgeführt wird. Eine spezielle Variante von asynchronen RPCs sind die *one-way RPCs,* bei welchen der Client nicht einmal die Bestätigung durch den Server abwartet. Damit hat er keine Sicherheit, dass die Nachricht tatsächlich den Server erreicht hat.

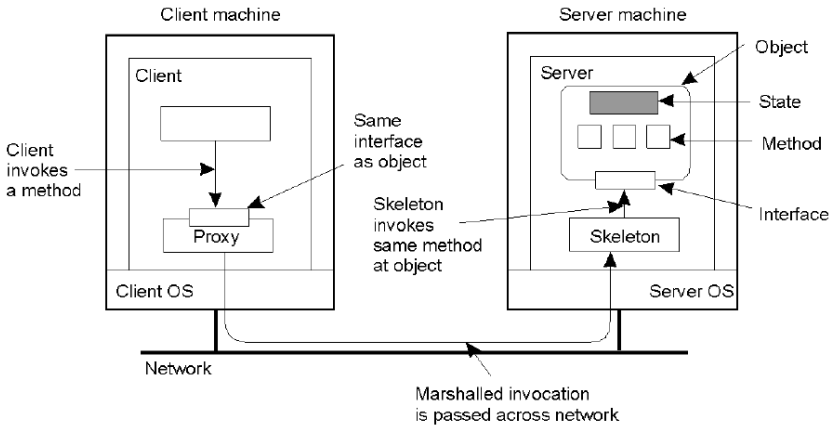
Communication (2) - RMI und Messaging

Buch: 4.3, 10.1.1, 10.3, 10.4

Fragen:

1. **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.**

Unter einem verteilten Objekt versteht man ein Objekt, wobei Interface und dessen Implementation auf verschiedenen Rechnern laufen. Führt ein Client ein Binding mit einem verteilten Objekt durch, wird eine Implementation des dazugehörigen Interfaces (= *Proxy*) in seinen Adressraum geladen. Proxies führen Marshaling/Unmarshaling durch und sind somit analog zu Client Stubs bei RPCs zu sehen. Das Objekt selbst liegt am Server, mit dem selben Interface wie beim Client. Der Server Stub (= *Skeleton*) ist nach Erhalt der Nachricht für das Unmarshaling sowie Methodenaufrufe über das Interface und das abschließende Marshaling verantwortlich. Er dient somit der Kommunikation der Server-Middleware mit den jeweiligen Objekten. Charakteristisch für die meisten verteilten Objekte ist, dass der Zustand auf *einer* Maschine verbleibt, lediglich die Interfaces werden für andere Maschinen zur Verfügung gestellt. Solche Objekte werden als *remote objects* bezeichnet.



Remote Method Invocations (RMI) entsprechen in ihrem Grundprinzip den RPCs, mit jenem Unterschied, dass Aufrufe auf entfernte Objekte erfolgen. Voraussetzung, um Methoden aufrufen zu können ist, dass der Client ein Binding zum entsprechenden Objekt hat.

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 instanziert werden.

- Vorteil: Aus einer Klassendefinitionen können Client & Server Stubs abgeleitet werden (IDL)
- Nachteil: weniger flexibel als reine Laufzeitobjekte. Sprachabhängig

Laufzeitobjekte werden zur Laufzeit konstruiert. D.h. Die Implementierung ist weitgehend offen.

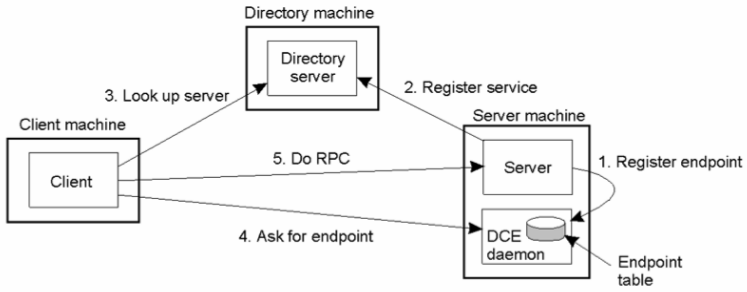
- Vorteil: verschiedene Programmiersprachen lassen sich leichter verbinden
- Nachteil: automatische Generierung von Stubs nicht möglich

Persistente Objekte sind unabhängig vom Zustand des Servers. Das bedeutet, dass sie auch dann weiter existieren, wenn sie von keinem Server-Prozess in dessen Adressraum enthalten sind. Der Zustand des Objekts wird vom Server vor dessen Beenden auf einem persistenten Speicher gesichert und kann später von dort wieder abgerufen werden.

Transiente Objekte existieren nur im flüchtigen Speicher des Servers, mit dem Beenden des Servers verschwindet daher auch das transiente Objekt.

2. **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.**

Systeme mit verteilten Objekten bieten im Gegensatz zu RPCs systemweite Objektreferenzen, die zwischen verschiedenen Maschinen und Prozessen weitergegeben werden können. Bevor nun ein Prozess über eine solche Referenz Methoden aufrufen kann, muss er ein *Binding* mit dem referenzierten Objekt durchführen. Dadurch wird ein Proxy in den Adressraum des Prozesses geladen, der ein Interface mit jenen Methoden implementiert, die vom Prozess aufgerufen werden können. Ein Binding kann *explizit*, d.h. über einen eigenen Aufruf zum Binding, oder *implizit* erfolgen.



In Bezug auf *Objektreferenzen* muss zwischen 2 Arten der Anwendung unterschieden werden:

- Global References werden bei implizitem Binding verwendet. Hierbei kann der Client direkt auf einer Referenz des Objektes die Methoden aufrufen.
- Explizites Binding verwendet sowohl lokale als auch globale Referenzen. Hier muss das Verteilte Objekt manuell an den lokalen Proxy gebunden werden. Das bedeutet, dass zuerst explizit ein Binding durchgeführt werden muss, bevor Methoden aufgerufen werden können.

In *Java RMI* besteht eine Objektreferenz aus Netzwerkadresse und Port des Servers sowie einem lokalen Identifier für das gewünschte Objekt im Adressraum des Servers. Zusätzlich muss auch der Protokoll-Stack in die Referenz kodiert werden. Sämtliche Informationen, die der Client benötigt, um auf eine Methode am Server zugreifen zu können, werden im Proxy gespeichert. Der Proxy kann in Java serialisiert, d.h. Zustand und Code werden gespeichert, und zu einem anderen Prozess transferiert werden. Dieser kann nun den Proxy als Objektreferenz verwenden und seinerseits Methoden am remote object aufrufen.

In *CORBA* können Objektreferenzen in manchen Fällen nicht einfach von Prozess A zu Prozess B geschickt werden, da sie lediglich *sprachenspezifische* Zeiger zu einer lokalen Repräsentation des Objekts darstellen und so nur eine Bedeutung im Quellprozess haben. Darum muss der Zeiger erst mit Hilfe des Runtime-Systems in eine prozessunabhängige Darstellung gebracht werden (Marshaling). Dabei ist es unerheblich, ob Prozess A und B in verschiedenen Sprachen geschrieben wurden. Diese system- und sprachenunabhängige Darstellung wird durch die sogenannte *Interoperable Object Reference (IOR)* ermöglicht. Wird eine Referenz an ein unterschiedliches CORBA System weitergegeben, erfolgt dies als IOR.

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

Statische Aufrufe setzen voraus, dass die Interfaces des Objekts bekannt sind, wenn das Client-System programmiert wird. Verändert sich das Interface, muss auch die Client-Anwendung neu kompiliert werden, da es sonst nicht verwendet werden kann. *Bsp: MyObject.doSomething(in1, in2, out1, out2)*

Dynamische Aufrufe ermöglichen es der Applikation, zur Laufzeit auszusuchen, welche Methode es am Remote Objekt ausführen möchte. *Bsp: invoke(MyObject, id(doSomething), in1, in2, out1, out2)*. id(...) bezeichnet hier den Lookup der Methode.

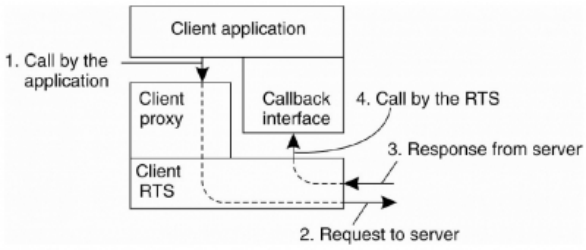
4. Wie funktioniert die Parameterübergabe bei RMI? Gehen Sie dabei auf jene Eigenschaften der Objektorientierung ein, welche den Vorteil von RMI gegenüber RPC bewirken.

Wird in RMI eine Methode mit einer Objektreferenz auf ein Remote Objekt als Parameter aufgerufen, wird diese Referenz kopiert und als Parameter übergeben (*passed by reference*). Handelt es sich hingegen um ein lokales Objekt, so wird dieses Objekt als Ganzes kopiert und mit dem Methodenaufruf weitergegeben (*passed by value*). Der grundlegende Vorteil gegenüber RPCs ist, dass systemweite Objektreferenzen nach Belieben von einer Maschine an die andere als Parameter weitergegeben werden können. Bei *RPCs* haben Referenzen lediglich lokale Bedeutung, d.h. im eigenen Adressraum des Prozesses und müssen per *copy/restore* übergeben werden.

5. **Erläutern Sie die Grundprinzipien 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.**

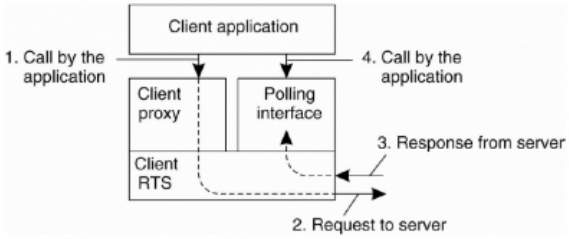
Message-orientierte Kommunikation findet vor allem in jenen Fällen Anwendung, wo RPCs und RMIs nur wenig geeignet sind. Etwa dann, wenn man nicht davon ausgehen kann, dass der Empfänger zum Zeitpunkt des Methodenaufrufes erreichbar ist. Auch bringt die synchrone Kommunikation (Client wartet solange bis Antwort eintrifft) in vielen Fällen erhebliche Nachteile mit sich.

Die objektorientierte Ausrichtung von CORBA hat Auswirkungen auf die Implementierung von Messaging, die sich in 2 Formen von asynchronen Methodenaufrufen manifestieren. Asynchronität bedeutet in diesem Zusammenhang, dass der aufrufende Prozess sofort nach dem Aufruf mit seiner Ausführung fortfährt, ohne auf eine Antwort zu warten. Das erste Modell basiert auf *Callbacks*. Der Client stellt ein Objekt zur Verfügung, das ein Interface mit Callback-Methoden implementiert. Über diese Methoden werden die Resultate bei Eintreffen an die Anwendungsebene des Client weiter gereicht, ein Callback wird ausgelöst. Dabei ist es die Aufgabe des Clients, den ursprünglichen synchronen Aufruf in einen asynchronen zu verwandeln – der Server wird lediglich mit einem gewöhnlichen synchronen Aufruf konfrontiert. 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).



CORBA's callback model for asynchronous method invocation.

Alternativ stellt CORBA ein *Polling Modell* zur Verfügung. Zu jeder ursprünglichen Schnittstelle werden 2 neue Schnittstellen erzeugt, wobei eine dem Aufruf dient (Parameter enthalten keine Return Werte). Die zweite Schnittstelle (Parameter sind Rückgabewerte der urspr. Schnittstelle) erlaubt es der Anwendungsebene, eingegangene Nachrichten (Resultate der Invocation) abzufragen. Wesentlicher Unterschied zum Callback Modell ist, dass diese Return-Methode ebenfalls vom RTS des Client implementiert wird.



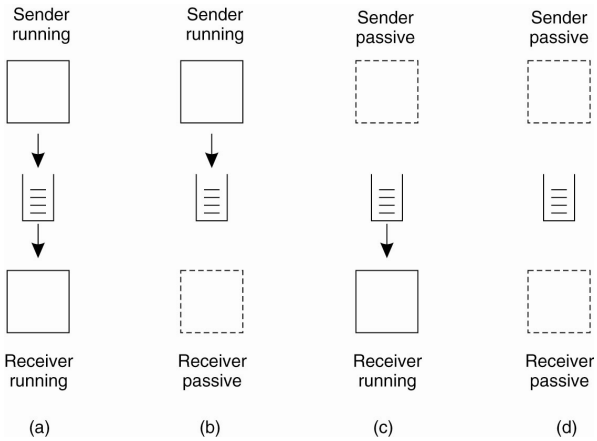
CORBA's polling model for asynchronous method invocation.

In beiden Fällen bleibt es dem Client überlassen, sich für synchrone oder asynchrone Kommunikation zu entscheiden und berühren den Server nicht. Die Repräsentation des Objekts beim Server bleibt unverändert.

6. **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 stellt umfassenden Support für persistente asynchrone Kommunikation zur Verfügung, d.h. bietet mittelfristig Speicherplatz für Nachrichten. Ein solches Modell stellt das *Message-Queuing* dar, bei dem Anwendungen miteinander kommunizieren, indem sie Nachrichten in bestimmte Queues eintragen. Sobald

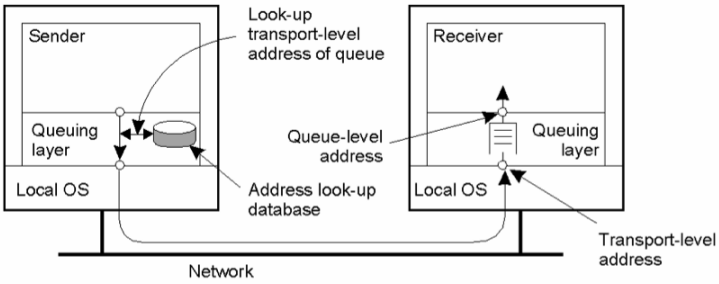
der Empfänger „online“ ist, wird ihm die Nachricht von seiner lokalen Queue zugestellt. Der Sender hat in einem solchen System keine Garantie darüber, wann und ob überhaupt die Nachricht gelesen wird. Die Besonderheit dabei ist, dass der Empfänger passiv sein kann, während der Sender sendet und umgekehrt (loosly-coupled communication). Demnach gibt es 4 mögliche Kombinationen:



Ein Message-Queuing System sieht die folgenden *Primitive* zur Steuerung vor:

- *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.

Charakteristisch für ein MQS ist, dass Sender bzw. Receiver nur Zugriff auf ihre jeweils lokale Warteschlange haben (source queue bzw. destination queue). Hat der Sender eine Nachricht auf seine Queue gelegt, ist es die Aufgabe des Queuing Layers, die Nachricht zur Empfänger-Queue zu schicken. Dabei muss zuvor der Name der Queue auf die Netzwerkadresse gemappt werden.



Message Brokers agieren in einem MQS als Gateways auf Anwendungsebene, deren Hauptzweck es ist, eingehende Nachrichten so zu formatieren, dass sie von der Applikation beim Empfänger verstanden werden. Eine wesentlich größere Rolle spielen Message Brokers aber in Zusammenhang mit *Enterprise Application Integration (EAI)*. Hier sind sie dafür verantwortlich, je nach ausgetauschter Nachricht den „richtigen“ Empfänger zu verständigen. Ein Beispiel dafür sind *publish/subscribe* Systeme. Postet etwa eine Anwendung eine Nachricht zu Thema X, wird diese vom Broker übernommen und an all jene Anwendungen weitergeleitet, die zuvor ihr Interesse bekundet haben, Nachrichten zu Thema X zu erhalten (subscription).

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

Um zeitabhängige Informationen ordnungsgemäß austauschen zu können, unterstützen Verteilte Systeme *Datenstreams,* also Folgen von Dateneinheiten. Diese können sowohl auf kontinuierliche (Audio, Video), als auch auf diskrete Medien angewendet werden. Die zeitliche Abfolge ist dabei für das Streaming von größter Bedeutung, damit die Daten rechtzeitig beim Client eintreffen.

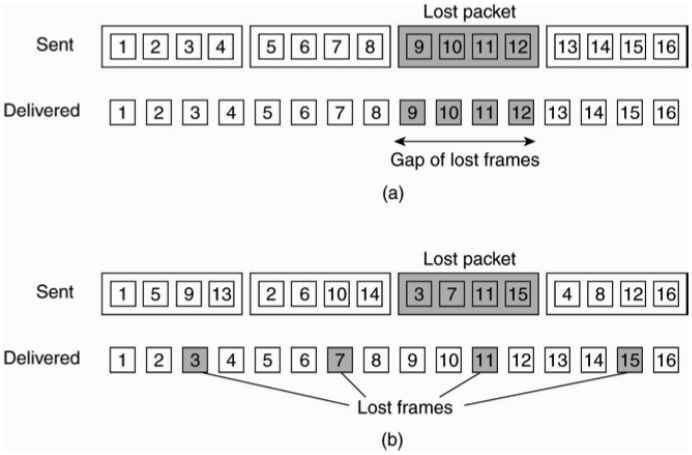
In Bezug auf das Timing wird zwischen 3 Arten des streamorientierten Kommunikation unterschieden. Bei *asynchroner Übertragung* werden die Datenelemente eines nach dem anderen übertragen, allerdings ohne zeitliche Einschränkungen (zB Download aus dem Internet). *Synchrone Übertragung* sieht eine maximale Verzögerung vor, in der die Daten den Empfänger erreichen müssen – die tatsächliche

Übertragungsverzögerung ist dabei nicht relevant, so lange sie darunter liegt (zB Sensor-Netzwerke). *Isochrone Übertragung* bedeutet, dass Dateneinheiten genau zum definierten Zeitpunkt ankommen müssen, d.h. es sind sowohl die Minimal- als auch die Maximal-Verzögerung festgelegt (zB Audio/Video). Daneben müssen wir einfache Streams von komplexen unterscheiden, wobei letztere aus mehreren einfachen Streams (substreams) bestehen.

Unter *Quality of Service (QoS)* werden im Allgemeinen Anforderungen an die zeitliche Steuerung zusammen gefasst. Diese beschreiben, was vom Verteiltes System bzw. 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

Das Auftreten von Fehlern in Zusammenhang mit der Übertragung von Streams kann durch einige Mechanismen von Verteilten Systemen minimiert werden. So werden Buffer dafür genutzt, um Jitter zu reduzieren. Dadurch kann ein unregelmäßiger Datenfluss vom System zu einem gewissen Maß (je nach Größe des Buffers) ausgeglichen werden. Eine andere Problematik ist der Paketverlust. Enthält ein Paket etwa mehrere Video Frames, führt dies zu einer großen Lücke bei der späteren Wiedergabe. Um dem vorzubeugen, kann das System die Frames *interleaved* verschicken, wobei das Zusammensetzen beim Empfänger erfolgt. Der Verlust von einem Paket hat dann nur den zwischenzeitlichen Ausfall von einzelnen Frames zur Folge.



Operating System Support

Buch: 10.2.1, Kap. 3

Fragen:

1. **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?**

Unter einem *Prozess* versteht man ein Programm, das auf einem der virtuellen Prozessoren des OS ausgeführt wird, welche in einer Prozesstabelle gemanaged werden. Dabei wird unter vergleichsweise hohem Aufwand sichergestellt, dass einzelne Prozesse von anderen Prozessen aufgrund der „geteilten" CPU nicht in ihrer Korrektheit beeinträchtigt werden (*concurrency transparency*). Wird ein neuer Prozess gestartet, muss vom OS ein völlig neuer Adressraum zur Verfügung gestellt werden. Zudem werden beim Wechseln von einem Prozess zum nächsten u.a. der CPU Kontext gespeichert oder Register modifiziert.

Im Unterschied zu Prozessen spielen sich *Threads* auf einer feineren Ebene ab. Ein Prozess kann aus mehreren Threads bestehen, die sich den selben Adressraum teilen. Hierbei muss der Programmierer dafür sorgen, dass ein Thread keinen Schaden/unerwünschte Zustände bei konkurierenden Speicherzugriffen erzeugt. Der Wechsel zwischen Threads ist weniger Aufwändig als bei Prozessen, es muss lediglich der CPU Kontext gewechselt werden. Aus diesem Grund ist die Performance einer Multithread-Applikation kaum schlechter, zumeist sogar besser als bei einem einzigen Thread. Dennoch muss beim Design eines solchen Systems sorgfältig vorgegangen werden, da mehrere Threads nicht automatisch voneinander geschützt sind, wie dies bei Prozessen der Fall ist.

Vorteile von Threads:

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

Bei der Verwendung von Threads muss jedoch darauf geachtet werden, dass ein blockierender Thread nicht seinen ganzen Prozess stoppt. Dies kann man mittels *Lightweight Processes (LWP)* erreichen, die nicht wie Threads im User-Space sondern im Kernel Space beheimatet sind und die eine Scheduling Routine für die Threads durchführen.

Für Verteilte Systeme sind Threads von großer Bedeutung, da sie eine Kommunikation zwischen Einheiten durch mehrere gleichzeitige Verbindungen ermöglichen. Ein typisches Beispiel für einen *MT-Client* ist ein Web Browser. Nachdem die HTML-Datei geladen wurde, kümmern sich mehrere Threads um die restlichen zu ladenden Daten wie Grafiken, Applets, usw. und ermöglichen so eine Darstellung am Rechner, obwohl im Hintergrund noch Teile geladen werden. Wenn die Daten der Website auf mehrere Server verteilt (repliziert) sind, ermöglicht dies ein paralleles Laden der Threads, wodurch sich die Ladezeit markant verringert.

Von noch größerer Bedeutung sind Threads auf *Server*-Seite, wo sie einen hohen Grad an Parallelismus ermöglichen. Eine Möglichkeit wäre die Verwendung eines *Dispatcher Threads*, der einkommende Anfragen übernimmt und an einen freien (Worker-) Thread übergibt. Wenn dieser, beispielsweise aufgrund eines System Call wie *blocking read* blockiert, wird vom Dispatcher bei neuen Anfragen einfach ein anderer, Thread ausgewählt. Dies führt im Vergleich zu einer 1-Thread-Lösung zu einer hohen Leistungssteigerung, da mehr Anfragen/Sekunde bearbeitet werden können.

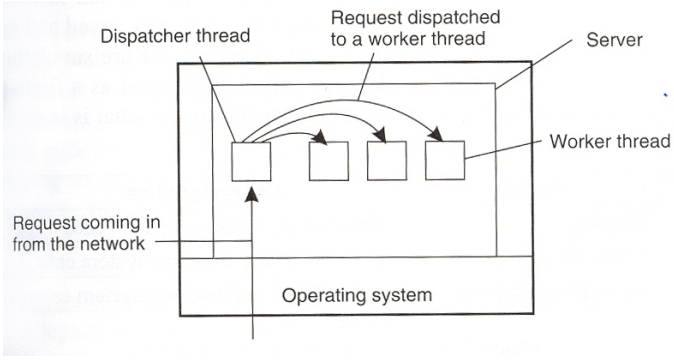
2. **Welche Aspekte Verteilter Systeme sind auf Client-Seite zu berücksichtigen? Wie werden User Interfaces in die Architektur Verteilter Systeme eingebunden? Wie können dabei verschiedene Arten der Transparenz unterstützt werden?**

Clients müssen in verteilten Systemem 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 (siehe auch 1). Auch die lokale Abarbeitung von Daten steigert die Performance, da zB der Netzwerktransfer minimiert werden kann (zB durch Formularkontrolle am Client anstatt am Server).

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 *Zugriffstranzperenz* 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 Kommuniktion.
Es gibt unterschiedliche Möglichkeiten, *Orts-, Migrations- und Relokationstransparenz* zu realisieren. Die Verwendung eines praktischen Namenssystem ist dabei wesentlich. In vielen Fällen ist auch die Zusammenarbeit mit clientseitiger Software wichtig. Hat beispielweise ein Client bereits ein Bind zu einem Server, 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, mit Hilfe clientseitiger Lösungen, wird von vielen Verteilten Systemen die *Replikationstransparenz* implementiert.
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 kontaktiert *(Fehlertransparenz).*

3. **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?**

Zunächst ist zwischen einem *iterativen* Server, der eine Anfrage selbst behandelt und, wenn nötig eine Antwort an den Client schickt, sowie einem *Concurrent* Server, der die Anfrage an einen Dispatcher Thread weiterleitet (bzw. einen neuen Prozess startet) und auf die nächste Anfrage wartet, zu unterscheiden.



Weiters stellt sich die Frage, wie der Server kontaktiert werden kann. Grundsätzlich handelt es sich bei der *Schnittstelle* um einen Endpunkt (Port) auf der Maschine des Servers. Dieser ist entweder als „well-known“ Port dem Client automatisch bekannt (zB HTML, FTP, etc.), oder muss vom Client, etwa über einen am Server laufenden Daemon (hängt selbst an einem WKP) identifiziert werden. Von Bedeutung im Design ist auch, wie *Interrupts,* etwa im Fall eines Download-Abbruchs seitens des Client von einem FTP-Server, behandelt werden. Dies kann entweder durch clientseitiges Schließen der Anwendung (Server beendet ebenfalls die Verbindung) oder mittles *out-of-band Daten* geschehen, die einen Interrupt beim Server auslösen.

Ein anderes Kriterium ist, ob die Server Informationen über deren Clients persistent speichern. Im Fall von *stateless* Servern (zB Web-Server, der lediglich HTML-Anfragen bearbeitet) ist dies zumeist nicht der Fall, wogegen *stateful* Server wie etwa File-Server mit Schreibrechten für bestimmte User eine User-Tabelle führen müssen. Der Vorteil gegenüber stateless Servern liegt hier in einer höheren Performance, die allerdings mit dem großen Aufwand einer Wiederherstellung im Fall eines Server-Crashes erkauft wird. Manche stateless Server unterstützen einen sogenannten *Soft State*, wo Information für einen vom Client bestimmten Zeitraum gespeichert wird. Dies ist dann der Fall, wenn zB ein Client vom File-Server für kurze Zeit über Updates informiert werden will. Weiters: *session state/permanent state* (siehe Dependability and Fault Tolerance).

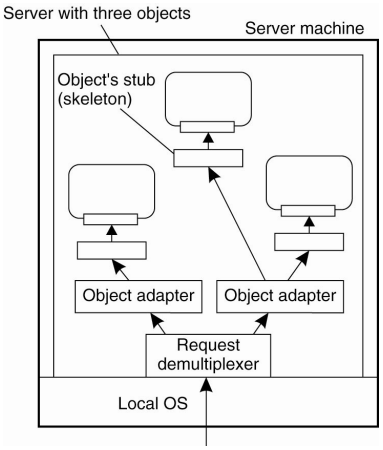
Beim *Einsatz von Multi-Threaded Servern* ist vom Entwickler besonders Wert auf die Nebenläufigkeit zu legen. Problem: Threads greifen alle auf den selben Speicherbereich des Prozesses zu und können so gegenseitig Werte überschreiben, was zu Inkonsistenzen führen kann. Auch muss bei der Verwendung von

Threads darauf geachtet werden, dass ein blockierender Thread nicht seinen ganzen Prozess stoppt. Dies kann man mittels *Lightweight Processes (LWP)* erreichen, die nicht wie Threads im User-Space sondern im Kernel Space beheimatet sind und die eine Scheduling Routine für die Threads durchführen.

4. **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?**

Aufgabe der Objekt-Server ist das Hosten von verteilten Objekten. Im Unterschied zu gewöhnlichen Servern bieten sie keine eigentlichen Services an, sondern dienen als Schnittstelle für den Aufruf von lokalen Objekten. Services können somit einfach durch den Austausch von Objekten geändert werden. Der Aufruf einzelner Objekte wird durch verschiedener *Policies* (Richtlinien) geregelt, die sich u.a. darauf beziehen, ob der Server lediglich einen einzigen *Thread* (nicht sofort bearbeitbare Anfragen kommen in eine Warteschlange), oder einen für jedes Objekt führt, wobei hier, wie in 3) besprochen, auf Concurrency geachtet werden muss. Auch die Speicherverwaltung ist Thema der Policies, d.h. ob jedes Objekt ein eigenes Speichersegment erhält und somit weder Code noch Daten mit anderen Objekten teilt, oder Codesharing betrieben wird. Im Fall einer Datenbank könnte so ein einziges Objekt von allen Anfragen verwendet werden, um auf die Daten zuzugreifen. Zuguterletzt stehen verschiedene Arten der Aktivierung von Objekten zur Verfügung: Sie können beim Start des Servers (alle zugleich) erstellt werden, oder beim ersten Aufruf (bleiben danach bis zur Beendigung des Servers bestehen bzw. werden gleich nach Ende der Anfrage zerstört).

Ein *Objekt Adapter* ist eine Software, die Mechanismen anbietet, um Objekte aufgrund der zuvor genannten Activation Policies zu gruppieren. Am Server können nun je nach Zahl der Richtlinien mehrere Objekt Adapter bereit stehen, wobei eine hereinkommende Anfrage dem passenden Adapter zugewiesen wird. Dabei sind ihnen die Interfaces der Objekte, die sie kontrollieren, nicht bekannt – sie sind somit generisch und müssen lediglich für eine gewünschte Policy (zur Laufzeit) konfiguriert werden. Der eigentliche Aufruf der Objekte vom Adapter erfolgt schließlich über deren Server Stubs (Skeletons).



5. **Erläutern Sie die wichtigsten Aspekte der Code Migration. Erklären Sie "strong mobility" und "weak mobility" und geben Sie für "weak mobility" ein Beispiel an.**

Code Migration, sei sie *sender- oder receiver-initiated (Java)*, ermöglicht eine Weitergabe von Programmteilen auch im laufenden Zustand. Zwei wichtige Gründe dafür sind eine verbesserte *Performance* sowie erhöhte *Flexibilität*. Im Fall von „teurer“ Kommunikation ist es günstig, einige Berechnungen vom Server zum Client zu verlagern und lediglich das Ergebnis zu übertragen (zB Formular-Eingaben). Generell ist es günstiger, Berechnungen in der Nähe der Daten durchzuführen. Flexibilität wird erhöht, wenn etwa ein Client für die Kommunikation mit einem speziellen Server eigene Software benötigt. Im Fall von Code Migration kann er diese dynamisch, also bei Bedarf, direkt vom Server beziehen und muss sie folglich nicht vorinstalliert haben. Allerdings hat Code Migration auch nachteilige Auswirkungen v.a. auf die Sicherheit (siehe Security).

Hinsichtlich des Umfangs der Datenmigration wird zwischen strong mobility und weak mobility unterschieden. *Weak mobility* ermöglicht den Transfer des Code Segments mit den zur Ausführung benötigten Programmteilen sowie von Daten zur Initialisierung. Es ist hier nur möglich das Programm an einer vordefinierten Position zu starten, wie zB bei *Java Applets* (Programm wird im Adressraum des Browsers ausgeführt) die immer beim Beginn gestartet werden. Der Vorteil dieser Methode ist die

Einfachheit, die einzige Anforderung ist, dass der Code vom Client ausgeführt werden kann. Strong mobility erlaubt dagegen auch den Austausch des Ausführungs-Segments, in welchem der aktuelle Programmstatus (private Daten, Stack, Program-Counter) abgespeichert wird. Charakteristisch ist, dass der laufende Prozess angehalten, auf eine andere Maschine übertragen, und dort fortgesetzt werden kann. Nachteilig ist hier die schwierige Implementierung.

6. **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. Auf einem Computer mit einem einzigen Prozessor ist diese Gleichzeitigkeit natürlich eine Illusion, die durch schnelles Umschalten zwischen beiden Prozessen erreicht wird (*resource virtualization*).

Im Fall von heterogenen Netzwerken, wo eine große Anzahl von Servern den Clients unterschiedliche Applikationen anbietet, kann Virtualisierung helfen, die Vielzahl an unterschiedlichen Plattformen zu reduzieren. Dazu lässt man alle Anwendungen auf eigenen virtuellen Maschinen laufen – u.U. Inklusive eigenem Betriebssystem und Bibliotheken – die ihrerseits auf einer einzigen Plattform zusammengefasst werden können. Auf ähnliche Weise ist es mittels Virtualisierung möglich, *Legacy Systeme* auf neuen Plattformen zu betreiben. Ein weiteres Argument für Virtualisierung liegt in der *Portabilität*. So können etwa Content Delivery Networks ganze Seiten inklusive ihrer Umgebungen replizieren.

Ziel der Virtualisierung ist es, die unterschiedlichen Interfaces eines Computer Systems (jeweils zwischen Anwendung, Bibliotheken, OS und Hardware) nachzuahmen. Eine Möglichkeit ist, mittels einer *Process Virtual Machine* einen einzigen Prozess zu unterstützen. Dabei stellt ein eigenes Runtime System ein Set an Instruktionen zur Verfügung, womit Anwendungen ausgeführt werden. Eine andere Variante wäre, ein System einzuführen, das als eine Schicht zwischen OS und Hardware implementiert wird. Der *Virtual Machine Monitor (VMM)* kann mehrere Programme bzw. verschiedene Betriebssysteme zur gleichen Zeit bedienen und sorgt durch seine Integration als Schicht über der Hardware dafür, dass Software unabhängig von der Hardware betrieben und auf diese Weise eine komplette Umgebung von einer Maschine auf die andere portiert werden kann.

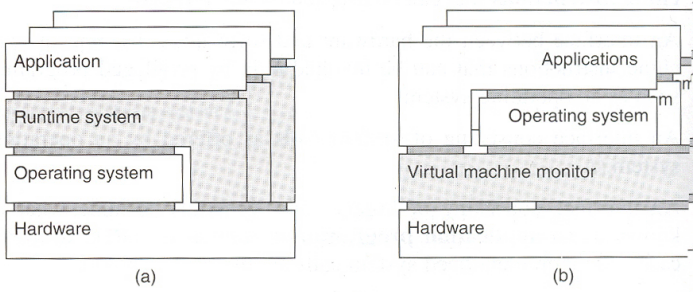


Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor, with multiple in-

Durch Virtualisierung bekommt man zusätzlich jene Probleme mit Heterogenität in Griff, die in *Zusammenhang mit Code Migration* zwischen unterschiedlichen Plattformen auftauchen. Eine Lösung ist der bereits angesprochene VMM, der ein Verschieben von Prozessen, zusammen mit deren OS, erlaubt. Die alternative Variante wird von Java gewählt, indem statt Maschinencode ein plattform-unabhängiger Code erzeugt wird, welcher wiederum von einer plattform-abhängigen Java Virtual Machine (JVM) interpretiert wird.

Naming and Discovery

Buch: Kap. 5

Fragen:

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

Unter *Name* versteht man in verteilten Systemen einen String, der auf eine Entität (zB Drucker, Hosts, Prozesse, etc.) verweist. Um mit einer Entität arbeiten zu können, benötigt man einen Zugangspunkt, bzw. eine *Adresse*, die auf einen solchen verweist. Dabei kann eine Einheit durchaus mehrere Adressen aufweisen und diese bei Bedarf wechseln. Beispiele für Name und Adresse sind etwa eine Kombination aus Hostname und IP-Adresse bzw. Name und Telefonnummer. Vorteil dieses Systems ist, dass ein Server, der auf eine andere Maschine transferiert wird, auch danach noch unter dem gleichen, *ortsunabhängigen* Namen erreichbar ist. Unter *Identifier* versteht man einen Namen, der sich auf genau eine Einheit bezieht (bijektiv) und nur einmal vergeben werden kann. Praktischen Zweck hat diese Konvention in Verteilten Systemen v.a. Für das Message Routing, wobei Namen bzw. Identifier zu Adressen aufgelöst werden (*lookup*).

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

Name Spaces sind Strukturen, die eine Gruppe von Namen umfassen. Sie können als gerichtete Graphen durch Blattknoten (enthält Informationen über eine Entität, zB Adresse, oder deren Zustand, zB Datei) und Verzeichnisknoten (enthält eine Tabelle von ausgehenden Kanten sowie deren Knoten). Jene(r) Knoten, der keine eingehenden Kanten aufweist, heißt *Root*. Eine Sequenz aus Kanten lässt sich zu einem *absoluten* (wenn er mit dem Root-Knoten beginnt) oder *relativen Pfad* zusammensetzen.

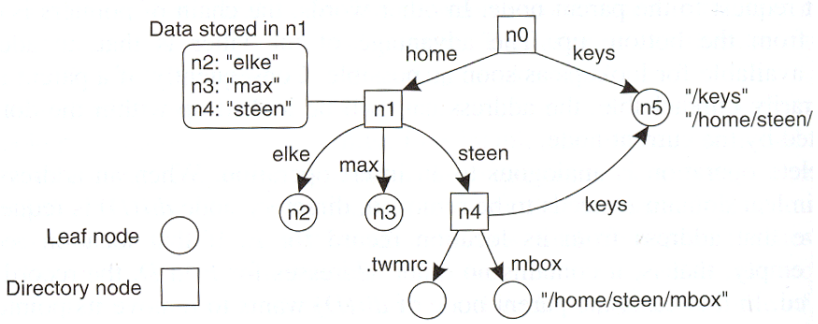


Figure 5-9. A general naming graph with a single root node.

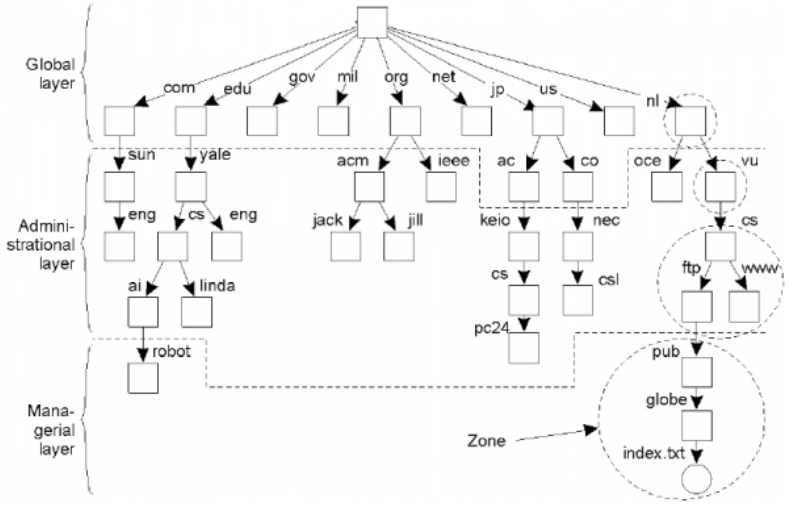
Mithilfe eines solchen Pfades kann man nun mittels *Name Resolution* (entspricht Lookup) auf Informationen des Nodes, auf den sich der Pfad bezieht, zugreifen. Zunächst muss man herausfinden, an welchem Knoten aus dem Name Space mit der Resolution begonnen werden soll *(Closure Mechanismus)*. Um in UNIX einen Dateinamen wie */home/student/uni* auflösen zu können, muss dem Dateisystem der Root-Knoten *„/“* bekannt sein. Der tatsächliche Offset des Root-Knotens ist im Superblock des logischen Laufwerks kodiert.

3. **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".**

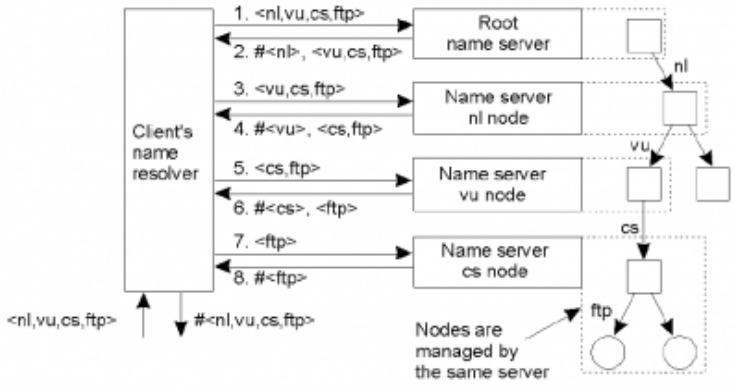
Name Spaces sind gewöhnlicherweise hierarchisch, in logischen Schichten, organisiert. Der g*lobal layer* besteht aus den highest-level Nodes, also dem Root-Node und seinen unmittelbaren Nachfolgern. Die Verzeichnistabellen der Knoten in diesem Layer verändern sich kaum und bilden Organisation oder Gruppen von Organisationen ab (zB .com, .org, .at). Der *administrational layer* enthält seinerseits die Verzeichnisknoten einer einzigen Organisation, während der *managerial layer* ständig wechselnde Knoten enthält (zB Hosts in einem lokalen Netzwerk, freigegebene Dateien). Im Gegensatz zu den beiden oberen Schichten wird der Inhalt des managerial layer weniger von Administratoren, sondern von den einzelnen Usern verändert.

Zusätzlichen Performance-Gewinn und erhöhte Verfügbarkeit im GL kann durch Replikation der Server sowie client-seitiges Caching effizient erreicht werden, da nur wenige Änderungen zu erwarten sind. Erreichbarkeit von Name Servern im AL ist nur dann von Bedeutung, wenn sich der Client in der selben

Organisation wie der Name Server befindet. In diesem Fall können Ressourcen aus der eigenen Organisation nicht gesucht werden. Ebenso wie im GL ist Caching ein probates Mittel, um die Performance zu steigern – jedoch sollten Updates schneller durchgeführt und Lookup-Resultate innerhalb weniger Millisekunden zurück gegeben werden was den Einsatz von Hochleistungsrechnern notwendig macht. Eine schnelle Reaktionsszeit ist auch im ML unabdingbar, ein Caching ist hier nicht sinnvoll, da mit regelmäßigen Änderungen zu rechnen ist.

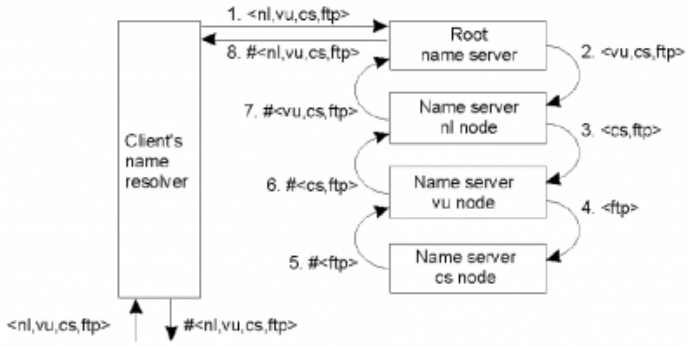


In einem Verteilten System hat jeder Client Zugang zu einem lokalen name resolver. Dieser kann die Namensauflösung in 2 verschiedenen Varianten durchführen: Im Zuge der *iterativen* Namensauflösung wird der gesamte Name dem Root-Name-Server übergeben, der nun, soweit ihm möglich, den Namen auflöst und das Resultat zurück gibt. Nun kontaktiert der Client diesen Name-Server, der wiederum einen Teil auflöst, usw. Abschließend löst der FTP Server den letzten Teil der ursprünglichen Pfad-Angabe auf (pub, globe) und gibt das HTML-File zurück. In der Praxis wird dies meist direkt vom Client übernommen und lediglich der Pfad *root:<nl, vu, cs, ftp>* übergeben.



ftp://ftp.cs.vu.nl/pub/globe/index.html

Alternativ dazu gibt es das Modell der *rekursiven Namensauflösung,* bei welcher die Zwischenresultate nicht zum Client gesendet, sondern an den nächsten Name Server weitergeleitet werden. Die Ergebnisse des Lookup werden schließlich über alle beteiligten Server zurück zum Root Name Server und schließlich zum Client übertragen, welcher den FTP Server für den Transfer kontaktiert. Nachteilig wirkt sich bei der rekursiven Methode der höhere Rechenaufwand an den Name Servern aus, die jeweils den gesamten verbleibenden Pfad zu verarbeiten haben. Aus diesem Grund unterstützen Server im Global Layer lediglich iterative Namensauflösung. Als Vorteile sind zu nennen, dass die Kosten der Kommunikation wesentlich geringer ausfallen sowie ein effektiveres Caching, das bei der iterativen Suche auf den Client Name Resolver beschränkt ist.



ftp://ftp.cs.vu.nl/pub/globe/index.html

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

Das DNS, das eines der größten verteilten Naming Services darstellt, hat die Aufgabe, IP Adressen von Hosts und Mail Servern zu finden (*mapping*). Mit dem DNS ist neben dem gewöhnlichen Lookup von Name zu IP auch eine umgekehrte Auflösung von IP-Adressen in Namen möglich *(reverse lookup)*. In Analogie zum Telefonbuch entspricht dies einer Suche nach dem Namen eines Teilnehmers zu einer bekannten Rufnummer.

Der *DNS Name Space* ist hierarchisch als Wurzelbaum organisiert, dessen Labels aus Strings bestehen, die durch einen Punkt getrennt sind und zusammen einen Pfad ergeben. Eine Domain kann man sich als Unterbaum dieses Graphen vorstellen, ein Pfadname wird als *domain name* bezeichnet. Dabei besteht der Name Space lediglich aus einem Global und einem Administrational Layer – der Managerial Layer ist nicht Teil des DNS. Inhalte der Knoten setzen sich aus sogenannten *resource records* zusammen. Diese umfassen u.a. Informationen über die jeweilige Zone, Host IP-Adresse, Host-Name, etc.

Ein konkreter *Lookup* gestaltet sich wie folgt: Der Client stellt eine Anfrage für die IP Adresse von www.tuwien.ac.at an seinen lokalen DNS Server. Falls er einen entsprechenden Eintrag in seiner Datenbank findet, wird die IP an den Client (bzw. an dessen DNS) geschickt. Andernfalls teilt er den Domain-Namen immer kleinere Teile, die er wiederum aufzulösen versucht (tuwien.ac.at, .ac.at, .at), was einem rekursiven Lookup entspricht. Bringt dies keinen Erfolg, wird einer der Root Name Server kontaktiert. Da dem Root sämtliche Top-Level-Domains bekannt sind, schickt er dem lokalen Name Server am Client einen Verweis zu einem .at-Server. Im nächsten Schritt erfolgt die ursprüngliche Anfrage, diesmal aber an den .at-Server. Nachdem auch der .ac-Server kontaktiert wurde, wird im Idealfall ein DNS für die Domain tuwien.ac.at gefunden, an welchen der lokale DNS wiederum seine Anfrage sendet. Die Rückantwort sollte nun die IP Adresse von www.tuwien.ac.at enthalten und schlussendlich den Client erreichen.

Ein *reverse lookup* wird benötigt, um von einer IP auf einen Domainnamen zu schließen. Im ersten Schritt wird die IP-Adresse umgeformt, um sie von rechts nach links lesen zu können. In weiterer Folge wird die Domain *in-addr.arpa* hinzugefügt und für gegebene IP Adresse der Pointer Record (PTR) abgefragt, der auf den gesuchten Namen referenziert.

Um eine hohe Ausfallssicherheit zu gewährleisten, werden die Zonendaten nicht nur auf dem Primary Server gehalten, sondern auf Secondary Server repliziert. Bei Änderungen muss sichergestellt sein, dass alle Server den gleichen Datenbestand besitzen. Die Synchronisation zwischen den beteiligten Servern (*primary → secondary*) wird durch den *Zonentransfer* realisiert, der in der Regel auf Initiative der Secondary Server hin durchgeführt wird.

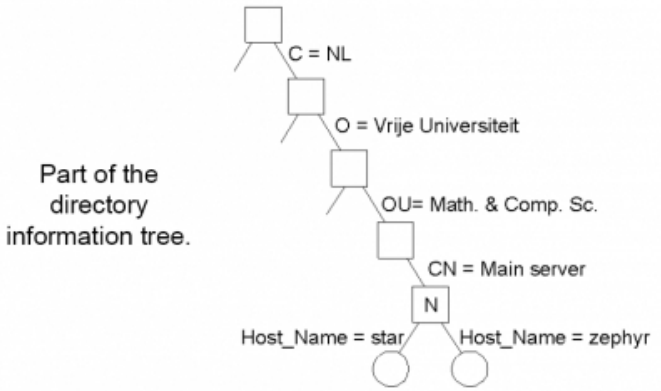
5. **Was ist Directory Service bzw. "Attribute-based naming"? Beschreiben Sie den prinzipiellen Aufbau des X.500 Name Space sowie dessen LDAP Implementierung.**

Bei *Attribute-based Naming* werden jeder Entität Paare (Attribut, Wert) zugeordnet wobei jedes Attribut etwas über die Entität aussagt. Ein User kann nun nach bestimmten Entitäten suchen, indem er Werte angibt, die die Attribute erfüllen müssen. Das Naming System gibt schließlich eine oder mehrere Entitäten zurück, die der Abfrage entsprechen.

Attributbasierte Services werden oft auch als *Directory Services* bezeichnet, wogegen *Naming Systems*

generell Systeme darstellen, die strukturiertes Naming unterstützen. Die Auswahl geeigneter Attribute ist jedoch schwierig, wenn spezifische Ergebnisse erwartet werden. Zudem ist die Suche rechnerisch aufwändig, da alle Deskriptoren untersucht werden müssen, vor allem bei verteilten Datenbanken.

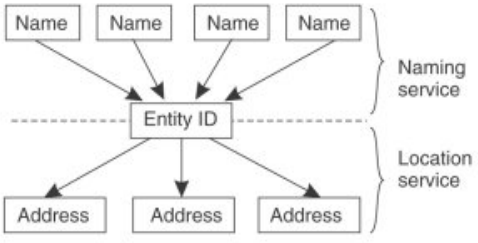
Bei *x.500* handelt es sich um einen OSI-Standard, der den Entwurf eines globalen Verzeichnisdienstes beschreibt, der auf strukturiertem und attributbasiertem Naming basiert. Der Aufbau von x.500 besteht aus Objekten, über die Informationen (Attributwerte) im Verzeichnis abgelegt werden, die notwendig oder optional sein können. Alle Objekte sind hierarchisch im *Directory Information Tree (DIT)* miteinander verbunden, wobei jede Ebene von Objekten für die Organisation der darunterliegenden Objekte verantwortlich ist. Jeder Eintrag hat einen eindeutigen Distinguished Name, der eine Kombination aus seinem eigenen RDN (Relative Distinguished Name) und den RDN all seiner übergeordneten Einträgen ist. Die Gesamtheit aller Informationen im Verzeichnis wird als Directory Information Base (DIB) bezeichnet.



Das *lightweight directory access protocol (LDAP)* ist eine effizientere Umsetzung von x.500, die auf dem Client/Server Prinzip basiert und direkt auf TCP aufsetzt. Es gilt als de Facto Standard für internet-basierte Verzeichnisdienste. Ein LDAP Verzeichnisdienst besteht aus verschiedenen Verzeichniseinträgen, die jeweils von mehreren (Attribut, Wert) Paaren beschrieben werden. Das Verzeichnis kann beispielsweise ein E-Mail-Adressbuch enthalten und die gewünschte Adresse zu einer gegebenen Person liefern.

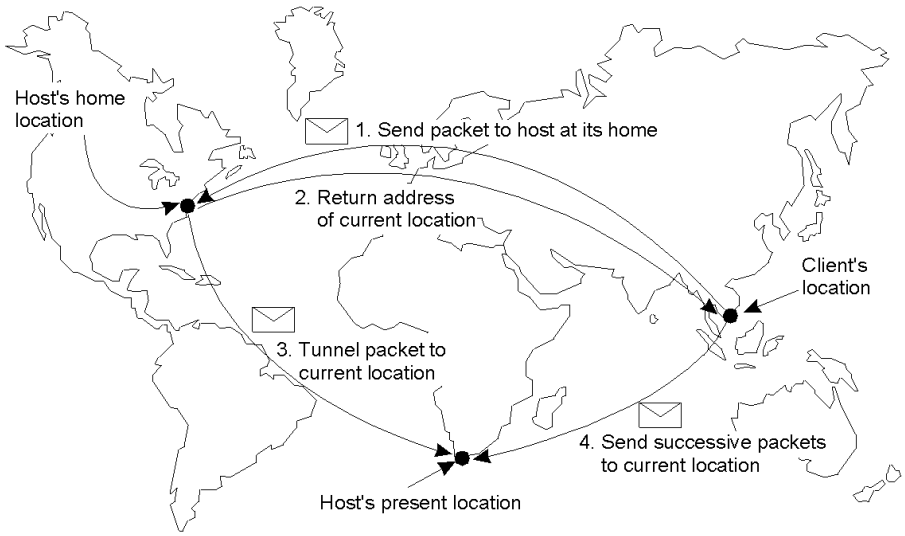
6. Wie funktioniert in flachen Namensräumen das Location Service? Geben Sie das Grundprinzip möglicher Lösungen an und gehen Sie dabei auch auf die Begriffe Mobility und Discovery ein. Erläutern Sie Vor- und Nachteile bei der Verwendung von "Forwarding Pointers". Wie funktionieren "Home-based approaches"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

Ein Location Service bezeichnet ein Two-Level mapping zum Übersetzen von Namen in Adressen über sogenannte Identities. Es hat in Ad-hoc Netzwerken (zB P2P), deren Ressourcen und Services sich dynamisch ändern, die Aufgabe, Services zu registrieren bzw. aufzufinden *(Discovery)*. Dadurch erreicht man einen hohen Grad an *Mobility*, da ein Host nicht mehr über eine fixe IP-Adresse angesprochen wird. Grundlegender Unterschied zu Name- und Directory Services ist, dass sich ein Location Service häufig und schnell anpassen muss.



Ein Ansatz, um Mobility zu erreichen, ist die Verwendung von *Forwarding Pointers*. Wechselt eine Entität ihren Ort, lässt sie eine Referenz *(symbolic link)* zurück, welche auf die neue Position zeigt. Dieses System ist sehr einfach zu implementieren, kann jedoch lange Ketten aufweisen, die viele Zwischenschritte notwendig machen. Kommt es bei einer dieser Schritte zu einer falschen Referenz *(broken link)*, wird die gesamte Kommunikation unterbrochen. Daher ist es nötig, die Ketten so kurz wie möglich zu halten. *Homebased Approaches* stellen Mobility in großen Netzwerken sicher. Die Home Location speichert dabei immer den aktuellen Standort einer Entität. Ändert die Entität ihre Position, registriert sie eine Care-Of-Addresse beim Home-Agent. Ein Client, der den aktuellen Standort nicht kennt, schickt eine Nachricht zur

bekannten Adresse des mobilen Client. Der dortige Home-Agent kennt dessen aktuelle Adresse und „tunnelt“ die Pakete an den Empfänger weiter. Gleichzeitig informiert er den Sender über die tatsächliche Adresse des mobilen Clients – nachfolgende Nachrichten werden somit direkt zwischen den beiden Clients ausgetauscht. Nachteil eines solchen Systems ist die hohe Latenz in der Kommunikation.



Clocks and Agreement

Buch: Kap. 6

Fragen:

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

Uhrensynchronisation wird in Verteilten Netzwerken benötigt, um eine zeitliche Abfolge von Operationen festzulegen. Unterscheiden sich die Timings, kann es zu einer Vielzahl von Fehlern, u.a. einer falschen Zuweisung von Werten, kommen. Aber auch in einem unvernetzten Rechner mit Multi-Prozessoren ist Synchronisation notwendig, da eine Synchronität von verschiedenen Hardware-Uhren in den Prozessoren nicht garantiert werden kann.

Ein Modell zum Zeitabgleich ist das *Network Time Protocol (NTP)*. Grundgedanke dabei ist, dass ein zentraler Time Server über die korrekte aktuelle Uhrzeit verfügt und die Zeit stets lokal korrigiert wird. Bei der Abfrage durch die Clients kommt es aber zu unterschiedlichen Message Delays, die die Information wieder obsolet machen. Eine geeignete Gegenmaßnahme ist, einen passenden Schätzwert für diese Delays zu finden. Sendet nun A eine Anfrage an den Time Server B, beinhaltet diese den Zeitstempel T1 des Sende-Zeitpunktes. Der Zeitpunkt T2, an dem die Nachricht B erreicht, wird von diesem notiert und zusammen mit dem Zeitstempel seiner Rückantwort T3 an Rechner A geschickt. Abschließend hält A den Zeitpunkt T4 des Eintreffens der Nachricht von B fest. Nun wird ein Offset, eine Art Zeitabweichung, sowie der Schätzwert berechnet, wobei anhand von 8 Messwerten der kleinste als bester Schätzwert ausgewählt wird.

Während in NTP der Time Server passiv agiert und lediglich Zeitanfragen beantwortet, sieht *Berkeley UNIX* einen Time Daemon vor, der jede Maschine von Zeit zu Zeit nach ihrer lokalen Zeit fragt. Basierend auf den Antworten wird eine Durchschnittszeit berechnet an die sich die Rechner angleichen müssen. Die Zeit am Time Daemon muss dabei manuell eingestellt werden. Dabei ist es irrelevant, ob diese gemeinsame Zeit mit der Real-Zeit übereinstimmt, solange kein Computer mit externen Rechnern kommuniziert.

2. **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"?**

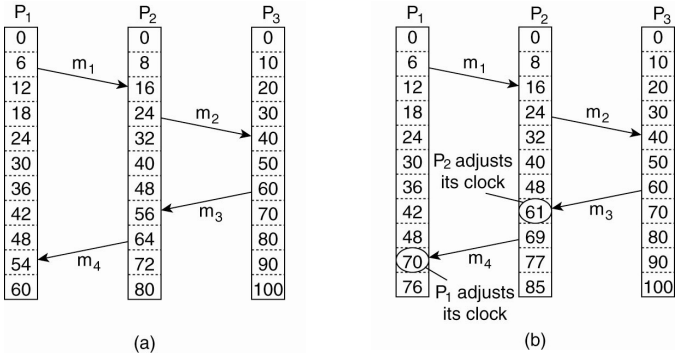
In vielen Fällen ist es ausreichen, dass sich die einzelnen Prozesse untereinander auf die Reihenfolge der Ausführung verständigen *(logical clocks)*, was dadurch sichergestellt werden kann, indem alle die selbe Zeit aufweisen. Diese muss nicht notwendigerweise mit der realen Zeit der physischen Uhr (zB Kristall-Oszillator, mittlere Sonnensekunde, Atomuhr, UTC (Universal Coordinated Time), übereinstimmen.

Die *happens-before* Beziehung dient der Synchronisation logischer Uhren. Relation a → b bedeutet etwa, dass sich alle Prozesse darauf einigen, dass b erst nach a ablaufen darf. Dabei kann es zu 2 Situationen kommmen:

- sind a,b Ereignisse innerhalb eines Prozesses und tritt a vor b auf, dann gilt a → b
- beschreibt a den Zeitpunkt des Sendens einer Nachricht von einem Prozess und b den Erhalt der Nachricht bei einem anderen Prozess, gilt ebenfalls a → b – eine Nachricht kann nicht zur selben Zeit oder bevor sie verschickt wurde, ankommen.

Für diese Implikation gilt die Transitivität. Wenn nun zwei Ereignisse, x und y, in verschiedenen Prozessen passieren, die keine Nachrichten untereinander austauschen, nennt man die Eregnisse *concurrent* (nebenläufig), d.h. man kann keine Aussage darüber treffen, wann x und y eingetreten sind und welches der beiden zuerst eingetreten ist.

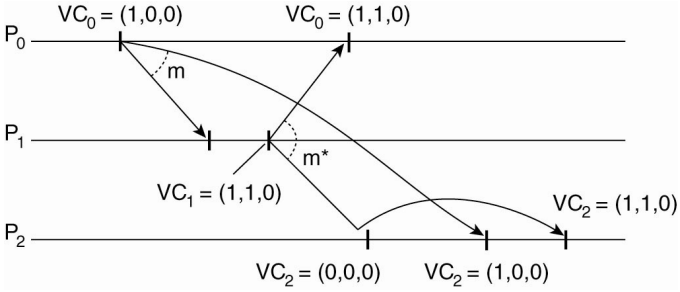
Idee der *Lamport-Timestamps* ist, dass Prozesse mit Zeitwerten *C* markiert werden. Dabei wird jedem abhängigen Ereignis einer größere Nummer zugewiesen, als der Ursache. Wenn also $a \rightarrow b$, dann gilt $C(a) < C(b)$. Außerdem darf C nie dekrementiert werden. Muss ein Zeitwert korrigiert werden, darf davon nichts abgezogen, sondern muss eine entsprechende Zahl addiert werden. Die untenstehenden Grafik links stellt 3 Prozesse dar, die auf verschiedenen Maschinen mit unterschiedlichen Uhren bzw. Geschwindigkeit laufen. Jeder Prozess hat seinen eigenen Counter. Bei jedem Sendeereignis wird dieser um 1 erhöht und hängt diese Zahl an die Nachricht an. Der Empfänger setzt seinen Counter auf das Maximum aus seinem aktuellen Counter und der mitgeschickten Zahl, erhöhte den Counter um 1 und sendet die Nachricht an die Applikation.



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

Lamport Clocks besitzen die Eigenschaft $a \rightarrow b \Rightarrow C(a) < C(b)$. Allerdings kann man anhand von Timestamps C(a) bzw. C(b) keine Aussage darüber treffen, welches Ereignis vorher eingetreten ist, da es sich ja auch um voneinander unabhängige Events verschiedener Prozesse handeln kann. Das bedeutet, dass *kausale Abhängigkeiten* nicht beachtet werden.

Dem kann man mit *Vector Clocks* begegnen, die die folgende Eigenschaft haben: Wird *VC(a)* dem Event *a* zugeordnet, dann gilt, wenn *VC(a) < VC(b)* für ein Event *b* ist, dass *a kausal vor b* ausgeführt worden ist. Jeder Prozess verwaltet einen Vektor VC, wobei *VC[i]* die logische Uhr, also die Anzahl der Events bis Zeitpunkt i, darstellt. Bevor nun eine Nachricht über das Netzwerk verschickt wird, wird dieser Zähler um 1 erhöht. Sendet Prozess P1 eine Nachricht m nach P2, wird der Timestamp ts(m) auf den Wert des Zählers VC[1] gesetzt. Beim Erhalt von m vergleicht P2 seinen bisherigen Vektor mit ts(m), behält den höheren Wert als Vektor bei und inkrementiert um 1. Schlussendlich wird eine Nachricht so lange verzögert, bis alle anderen Nachichten, die kausal vorher verschickt wurden, beim Empfänger eingegangen sind.



Nachteil dieses Verfahrens ist, dass alle *möglichen* kausalen Abhängigkeiten betrachtet werden. Dies kann zu Problemen in der Effizienz führen → höherer Speicher-/Kommunikationsaufwand.

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

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

- *token ring:* 1 Token wird zwischen den Prozessen weitergereicht, nur der Prozess, der das Token hält, ist zum Zugriff auf die Resource berechtigt. Benötigt ein Prozess keinen Zugriff, so reicht er einfach das Token zum nächsten Prozess weiter. Vorteile sind die einfache Implementierung, zudem treten keine Starvation, d.h. jeder Prozess kommt an die Reihe, sowie keine Deadlocks auf. Stürzt allerdings jener Prozess, der das Token hält, ab, geht es verloren. Es ist schwer zu erkennen, wann ein Token verloren gegangen ist. Weiters müssen Token ständig weiter gereicht werden, auch wenn kein Prozess auf die Resource zugreifen will.
- *centralized:* 1 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 gerade in Verwendung ist, kann der Koordinator die Anfrage in einer Warteschlange zwischenspeichern. Das System ist sehr effizient und einfach zu implementieren, wie beim token ring kommt es zu keiner Starvation, zudem werden

auch Deadlocks vermieden. Allerdings besteht hier die Gefahr eines Koordinator Crash (stellt Single-Point-of-Failure dar).

- *distributed:* Wenn ein Prozess exklusiven Zugriff auf eine Resource 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 Resource zu, so antworted 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.*

  Nachteilig ist, dass dezentralisierte Systeme zu einem hohen Kommunikationsaufwand führen. Wartet zudem gerade jeder Prozess auf ein OK, kann es zu einem kompletten Stillstand kommen, indem ein einziger der Prozesse crasht.

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1,2,...	2 m	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

5. 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?

angepasster centralized Mechanismus (siehe auch vowi):
Damit mehrere Anfragen gleichzeitig verarbeitet werden können, würde ich einen multi threaded server entwickeln. Bestehend aus:

- dispatcher thread (nimmt Anfragen entgegen, startet pro Anfrage einen worker thread)
- worker thread (bedient die Anfrage eines Clients)
- Liste aller Dateien (pro Datei wird vermerkt, welcher Thread wie (read/write) gerade darauf zugreift, in Form einer Queue).

Die Threads innerhalb des Server Prozesses treffen die Entscheidungen, wobei die Grundlage dafür ein zentraler Datenbestand innerhalb des Prozesses liefert. Vorteile: exklusiver Schreibzugriff, effizient, keine Starvation/Deadlocks. Eine Client Anfrage besteht aus einem Befehl (read/write) und dem Dateiname (Pfad). Somit gibt es 2 Möglichkeiten, wie der worker thread reagieren kann:

- *read*: worker thread trägt sich bei der Datei in die Warteliste als read ein; prüft ob in der Liste ein Thread die Datei gerade schreibt, wenn ja->wait; danach: senden der Datei an Client; austragen aus der Liste.
- *write*:worker thread trägt sich bei der Datei in die Warteliste als write ein;prüft ob in der Liste ein Thread die Datei gerade schreibt, wenn ja → wait; danach: alle thread in der Liste dieser Datei auf wait setzen; Datei von Client empfangen & schreiben; gibt es einen weiteren write thread in der Liste, dann diesen "aufwecken" (notify), sonst alle read threads aufwecken mit notify.

6. 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 (zB 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 die eigene ID. Wenn er keine Antwort erhält, so ist er der neue Koordinator (schickt COORDINATOR Nachricht an die anderen Prozesse), antwortet jedoch einer der Prozesse, so übernimmt dieser die Kontrolle und sendet eine ELECTION Nachricht. Das geschieht so lange, bis der Prozess mit der höchsten ID Koordinator geworden ist und eine COORDINATOR Nachricht an die anderen Prozesse aussendet.

Ring Algorithmus:
Die Prozesse sind logisch in Form eines Rings angeordnet. Bemerkt ein Prozess den Ausfall des Koordinators, sendet er eine ELECTION Nachricht an den Nachfolger (wenn dieser nicht antwortet an dessen Nachfolger, so lange bis einer antwortet). Jeder Prozess hängt dabei 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 nun seine Arbeit aufnehmen, die COORDINATOR Nachricht wird nach einem Durchlauf entfernt. Falls mehrere Prozesse gleichzeitig den Ausfall des Koordinators bemerken und eine ELECTION Nachricht aussenden, ändert dies nichts am Ergebnis, führt aber zu zusätzlichem Traffic.

Beide angeführten Algorithmen sind für ad-hoc Systeme geeignet und haben die Bestimmung, einen einzigen Koordinator (zB Kriterium = Bandbreite) zu wählen. Allerdings gibt es Situationen, in denen mehrere Koordinatoren Sinn machen (zB Superpeers in P2P-Netzwerken). In large-scale Netzwerken werden daher andere Algorithmen verwendet. Ziel ist, dass eine auf die Menge an Nodes abgestimmte Anzahl von Superpeers gleichmäßig im Netzwerk verteilt wird. Damit wird sichergestellt, dass auf einen Superpeer nur eine bestimmte Anzahl an Nodes kommt.

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

Der *global state* eines Verteilten Systems besteht per Definition aus den einzelnen local states sowie den im Umlauf und damit noch nicht zugestellten Nachrichten. Eine Variante der Ermittlung zu einem bestimmten Zeitpunkt funktioniert mittels eines *distributed snapshot,* bei dem der aktuelle Zustand alle Ereignisse bis zum *cut* beinhalten. Problematisch ist jedoch, dass es zu *inkonsistent cuts* kommen kann, nämlich dann, wenn zwar das Ereignis des Eintreffens einer Nachricht, nicht aber das Versenden im Snapshot aufscheint (Ereignis ohne Ursache).

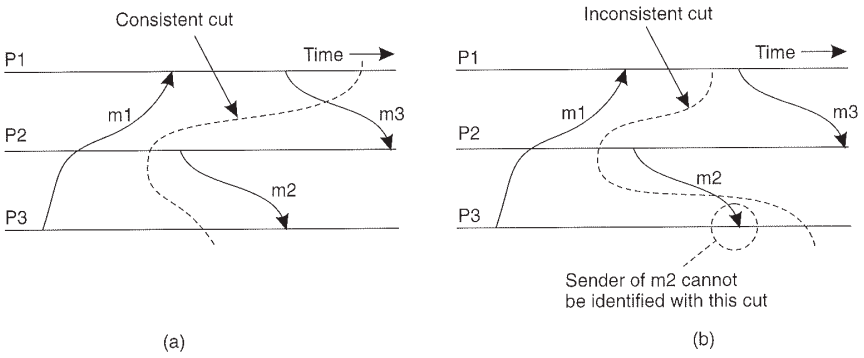
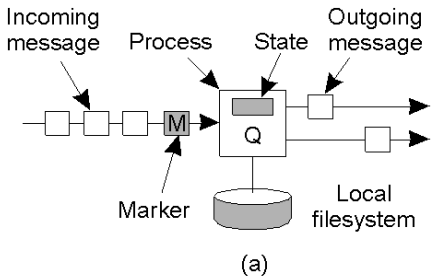


Figure 5-9. (a) A consistent cut. (b) An inconsistent cut.

Der *Snapshot-Algorithmus* nach Chandy & Lamport gestaltet sich wie folgt: 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. Bis zum Empfang des Markers zum zweiten Mal zeichnet jeder Prozess alle eingehenden Nachrichten auf. Beim Empfang des Markers zum zweiten Mal ist die Aufzeichnung abgeschlossen: 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*, allerdings müssen die lokalen Zustände in dieser Kombination nicht notwendigerweise jemals gemeinsam aufgetreten sein.



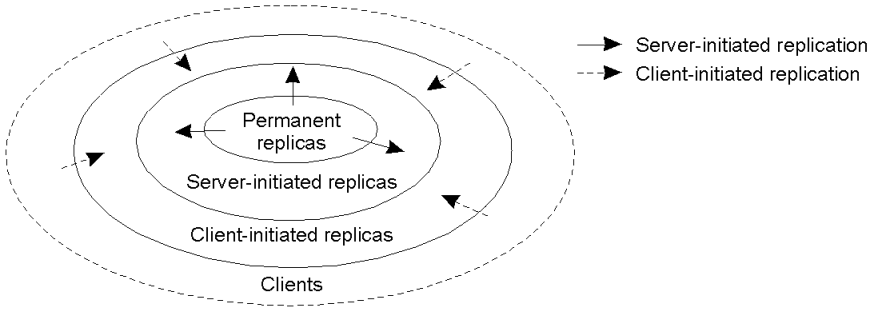
Consistency and Replication

Buch: 7.1, 7.4, 7.5, 10.6, 4.5.2

Fragen:

1. **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.**

Die Hauptgründe für den Einsatz von Replikation sind *Zuverlässigkeit* und *Leistung*. Wurde eine bestimmte Datei repliziert, ist es möglich, durch Wechseln auf eine Kopie die Arbeit fortzusetzen, wenn eines der Replikate abstürzt. Weiters bietet Replikation auch eine Möglichkeit, korrupte Dateien, wie sie etwa im Fall eines fehlerhaften Schreibzugriffes entstehten, zu erkennen. In Fällen, wo ein Server eine Vielzahl von Anfragen zu selben Zeit erfüllen muss, kann die Leistung gesteigert werden, indem der Server repliziert und die Arbeit in der Folge aufgeteilt wird (*Skalierung nach Größe*). Auch bei *geographischer Skalierung* wird auf Replikation zurück gegriffen. Grundidee ist, die Daten in die Nähe jener Prozesse zu platzieren, von denen sie benötigt werden, um so die Zugriffszeit zu verringern. Der *Nachteil* der Replikation ist jedoch, dass Replikate Mehraufwand bedeuten, da die Daten konsistent gehalten werden müssen, was sich insbesondere in jenen Fällen negativ auswirkt, wo die Zahl der Updatevorgänge die Zahl der Zugriffe übersteigt. Man muss also abschätzen, ob der Einsatz von Replikaten den zusätzlichen Aufwand (Traffic, Resourcen) rechtfertigt (trade-off).



Das Content Placement unterscheidet die 3 folgenden Arten von Replikas: permanent, Server-initiiert sowie Client-initiiert.

Permanente Replikas kann man sich als bereits von Anfang an vorhandenes, statisches kleines Set von Kopien vorstellen, die zusammen einen verteilten Speicher darstellen. Beispiele dafür sind das Mirroring von Webseiten, bei dem diese auf unterschiedliche Server (geographisch) skaliert werden, sowie die Replikation von Daten in Server-Clustern, wodurch jeder Rechner seinen eigenen Datenbestand erhält.
Server-initiierte Replikas dienen der Leistungssteigerung und werden vom Besitzer der Daten (Server) veranlasst. Beispielsweise hat ein Web-Server über einen bestimmten Zeitraum eine große Menge von Anfragen einer „exotischen" Location zu bedienen. Dann ist es sinnvoll, temporäre Replikas dynamisch in dieser Umgebung zu installieren.
Client-initiierte Replikas werden auch als *Caches* bezeichnet. Dabei wird vom Client *temporär* eine Kopie der von ihm zuvor abgefragten Daten angelegt, wodurch sich in vielen Fällen die Zugriffszeit reduziert. Gespeichert werden die replizierten Daten in einem Speicherplatz (Cache), der lokal zum Client ist. Das kann am selben Rechner oder aber auf einer anderen Maschine im lokalen Netzwerk sein. Somit können sich auch mehrere Rechner einen einzigen Cache teilen. Führt der Client eine Anfrage aus, und kann sie vom Cache erfüllt werden, bezeichnet man dies als *cache hit*.

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

Update Propagation bezieht sich darauf, wie replizierte Inhalte im Bedarfsfall upgedatet werden. Zunächst stellt sich die Frage *was* überhaupt weitergegeben werden soll.

- *Benachrichtigung*: In einem Invalidierungsprotokoll werden andere Kopien darüber informiert, dass eine Aktualisierung stattgefunden hat und die Daten somit nicht mehr gültig sind. Da keine Datensätze verschickt werden, wird nur wenig Bandbreite im Netzwerk benötigt. Dieses Modell ist dann sinnvoll, wenn viele Aktualisierungen wenigen Lese-Zugriffen gegenüber stehen.

- *Datentransfer*: Sendet die aktualisierten Daten an die Replika-Hosts. Einsatz findet diese Variante bei einem hohen lese-schreib Verhältnis. Andernfalls kann es sein, dass Updates von niemandem gelesen werden und somit obsolet sind. Alternativ ist es möglich, dass nicht Daten, sondern Logs verschickt werden, um Bandbreite zu sparen.
- *Operationstransfer*: Hier werden keine Daten verschickt, sondern jeder Replika mitgeteilt, welche Update-Operationen auf ihren Daten durchzuführen sind. Statt ganzen Dateien werden somit Parameter-Werte versendet. Dieses Modell der Content Distribution wird auch als *aktive Replikation* bezeichnet, da angenommen wird, dass jede Kopie über einen Prozess verfügt, der sie „aktiv“ auf dem aktuellen Stand halten kann. Vorteil ist hier eine geringe Bandbreiten-Anforderung, allerdings kann es bei komplexen Operationen zu einem hohen lokalen Rechenaufwand kommen.

Push/Pull Protokolle:
Dieser Entwurfsaskpekt unterscheidet, ob Aktualisierungen abgeholt (pull) oder automatisch verschickt werden (push). In *push*-basierten Ansätzen, auch als Server-basierte Protokolle bezeichnet, werden Aktualisierungen an andere Replikas weitergegeben, ohne dass diese welche angefordert haben. Dies wird besonders in Fällen eingesetzt, wo Replikas im Allgemeinen einen relativ hohen Konsistenzgrad aufweisen müssen, etwa beim Caching, wo v.a. Leseoperationen durchgeführt werden. Charakteristisch ist auch, dass am Server ein hoher Overhead entstehen kann, da alle zu aktualisierenden Client-Caches von ihm verwaltet werden müssen (*stateful Server*).

In einem *pull*-basierten Ansatz, auch als Client-basiert bezeichnet, fordert ein Server oder Client einen anderen Server auf, ihm Aktualisierungen zu senden, die 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. Cient-basierte Protokolle sind effizient, wenn das Lese/Aktualisierungs Verhältnis relativ niedrig ist. Der größte Nachteil ist, dass die Antwortzeit im Falle eines veralteten Cache-Eintrages steigt.

Issue	Push-based	Pull-based
State at server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

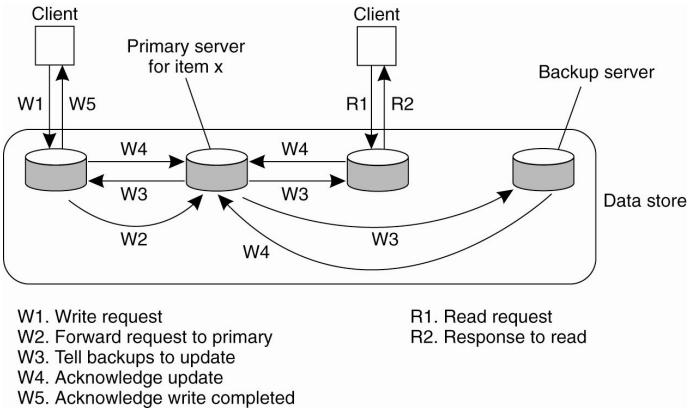
Uni-/Multicasting:
In Zusammenhang mit der Einscheidung nach push/pull Protokoll muss geklärt werden, ob *uni-* oder *multicasting* verwendet werden soll. Bei einer Unicast-Kommunikation sedet ein Server, der Teil des Datenspeichers ist, seine Aktualisierung an N andere Server, indem er N separate Nachrichten sendet, je eine an jeden Server. Beim Multicasting übernimmt das zugrunde liegende Netzwerk die Aufgabe, eine Nachricht effizient an mehrere Empfänger zu senden. Das ist dann von Vorteil, wenn alle Replikas im selben LAN hängen und somit auf das Hardware-Broadcasting zurück gegriffen werden kann. *Multicasting* kann häufig effizient mit einem *push*-basierten Ansatz kombiniert werden.In diesem Fall verwendet ein Server eine Multicast-Gruppe, um mehreren anderen 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.

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

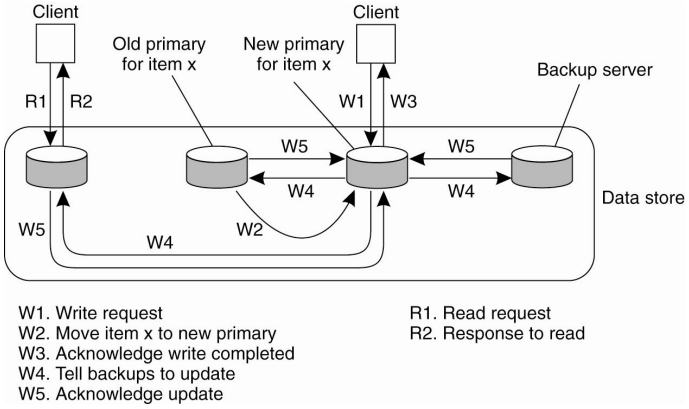
Primary-based Protokolle beschreiben eine Art der Implementierung von Konsistenz-Modellen. Dabei werden alle Update-Oparationen zu einer Primärkopie weitergeleitet, die ihrerseits sicherstellt, dass Updates korrekt weitergegeben und in der richtigen Reihenfolge durchgeführt werden. Im Gegensatz dazu werden bei Replikated-Write Protokollen Updates an mehrere Replikas zugleich gesendet. In primary-based Protokollen wird jedem Datensatz x ein Primary zugeordnet, der für die Koordination der Schreibvorgänge auf x zuständig ist.

Remote-write
Bei dem einfachsten Primary-based protokoll werden alle Schreib-Operationen zu einem bestimmten (fixierten) Primary Server weitergeleitet. Dieser führt daraufhin ein Update auf der lokalen Kopie des Items durch und leitet es an alle Backup-Server weiter, die ihrerseits lokale Updates durchführen und abschließend ein Acknowledge zurück an den Primary senden. Wenn alle Aktualisierungen durchgeführt, wird dies vom Primary dem initialisierenden Prozess bestätigt. 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, bei dem aber nicht sichergestellt werden kann, dass die Updates von den Backup Servern durchgeführt wurden.



Local-write
Um ein Update an einem Item durchzuführen, wird zuerst die primäre Kopie lokalisiert (wie zuvor) und das eigene File mit dieser Kopie überschrieben. Der Hauptvorteil dieser Vorgehensweise besteht darin, dass mehrere Schreiboperationen nacheinander lokal erfolgen können, während lesende Prozesse weiterhein auf ihre lokalen Kopien zugreifen können. Eine Verbesserung dieser Art kann jedoch nur erreicht werden, indem non-blocking Protokolle verwendet und Aktualisierungen an die Replikate weitergeleitet werden, nachdem der primäre Server das lokale Update beendet hat.

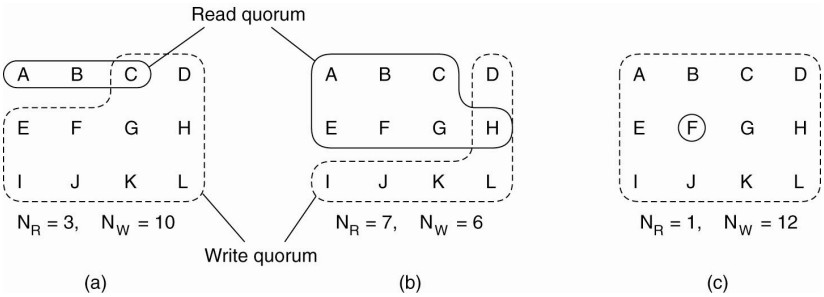


4. **Erläutern Sie die Funktionsweise der "replicated-write" Protokolle. Bewerten und vergleichen Sie die verschiedenen Arten. Welche Probleme können bei "Active Replication" auftreten? Erklären Sie "quorum-based" Replikationsprotokolle und geben Sie die Bedingungen an, um Read-Write und Write-Write Konflikte zu verhindern.**

Replicated-write Protokolle unterstützen parallele Schreib-Operationen an mehreren Kopien. Dabei wird zwischen aktiver Replikation, in der Operationen an alle Replikas weitergeleitet werden, und Konsistenz-Protokollen, die auf einer Mehrheitsentscheidung (Quorum) basieren, unterschieden.

Aktive Replikation
Jeder Kopie ist ein Prozess zugeordnet, der Updatevorgänge durchführt, indem er die benötigten Operationen (bzw. eine aktuelle Kopie) an alle Replikas sendet. Dabei muss aber sichergestellt werden, dass alle Operationen überall in der selben Reihenfolge ausgeführt werden (zB mit Lampert Clocks). Besser ist aber die Integration eines zentralen Koordinators, eines *Sequencers*, der allen Operationen eine unique ID zuweist, bevor sie nacheinander, entsprechend ihrer Numerierung, ausgeführt werden. Ein mögliches Problem ist, dass einzelne Requests von einem replizierten Objekt (d.h. von jedem Replikat) mehrfach durchgeführt werden könnten → siehe Replicated Invocations (Frage 5).

Quorum-basierte Replikation
Die Grundidee ist hier, dass Clients eine Erlaubnis für jeden Lese-/Schreibzugriff auf eine Kopie einholen müssen. Möchte der Client eine Kopie updaten, müssen dem mindestens die Hälfte aller Server zustimmen. Erst dann wird die Datei verändert und mit einer neuen Versionsnummer versehen. Damit können write-write Konflikte vermieden werden. Vor Lesezugriffen muss der Client ebenfalls zumindest N/2 + 1 Server kontaktieren. Verfügen alle über die selbe Versionsnummer, muss es sich um die aktuelle Version handeln.

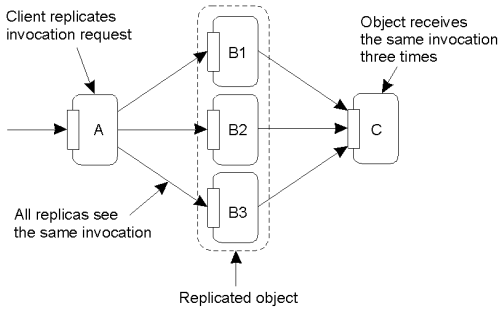


Bedingungen
1) $N_R + N_W > N$
2) $N_W > N/2$

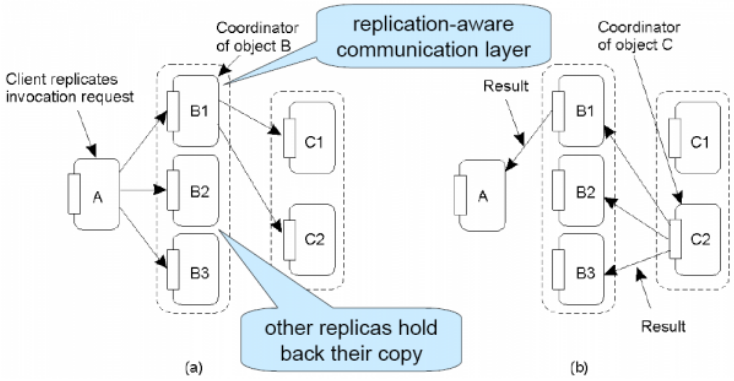
In obenstehender Grafik erzeugt (b) möglicherweise write-write Konflikte, da 2) verletzt ist.

5. **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").**

Wie in 4) erwähnt, kann es zum Problem von wiederholten Aufrufen kommen, wie in folgender Grafik dargestellt ist.



Eine Möglichkeit ist, Mehrfachaufrufe einfach zu verbieten, was jedoch zu Lasten der Fehlertoleranz geht. Stattdessen wird die folgende Methode gewählt: Nachdem alle Objekte B von A aufgerufen wurden, wird allen Requests von B nach C der selbe UID zugeordnet. Eine der Replikas von B, die als Koordinator fungiert, leitet ihre Anfrage nun an alle Kopien von C weiter, während die anderen Aufrufe von B zurückgehalten werden. Für die Rückleitung der Antwort der Objekte C wird der selbe Mechanismus angewendet. Ein Koordinator von C sendet die Antwort an alle B's, von wo das Resultat vom B-Koordinator nach A geschickt wird.



6. **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 also durchaus Inkonsistenzen während der Updates beinhalten. Gibt es keine Aktualisierung, muss nur sichergestellt sein, dass alle Repliken irgendwann identisch sind. Weiters sollen Aktualisierungen mit möglichst wenigen Nachrichten an alle Replikas weitergegeben werden. Dadurch wird ein schneller, dezentraler (auf lokalen Informationen basierender) Informationsaustausch innerhalb großer Netzwerke

sichergestellt. Vorteil ist neben der geringen Netzwerkbelastung vor allem die gute Skalierbarkeit. Schwierig ist hingegen das Löschen eines Datensatzes, das an alle übrigen Nodes weitergegeben werden muss. Auch wird, wie bereits erwähnt, keine totale Konsistenz garantiert.

Das *Anti-Entropie Modell* ist ein Weitergabemodell für Epidemische Protokolle welches per Zufall einen anderen Server auswählt und mit diesem dann Aktualisierungen austauscht (push bzw. pull). Sind die meisten Knoten „infiziert“, ist beispielsweise die Chance relativ gering über den push-Ansatz weitere Knoten für ein update zu finden.

Gossiping ist eine spezielle, effizientere, Form des Anti-Entropy Modells. 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, gibt es nur eine geringe Wahrscheinlichkeit der Weiterleitung. Zumeist wird die Nachricht in diesem Fall gelöscht. Gossiping ermöglicht ein rasches „Spreading“, allerdings kann nicht garantiert werden, dass alle Server benachrichtigt werden.

Dependability and Fault Tolerance

Buch: 8.1 - 8.4

Fragen:

1. **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.**

Eine Eigenschaft von verteilten System ist, dass es keinen single-point of failure gibt, fällt eine Komponente aus, so werden möglichweise einige andere Operationen beeinträchtigt, jedoch nicht das komplette System. Fehlertoleranz bedeutet, dass ein System auch im Falle eines Fehlers seine Dienste zur Verfügung stellt.

Attribute/Requirements

- *Availability*: wird als Wahrscheinlichkeit angegeben, mit welcher ein Service/System korrekt arbeitet zu einer beliebigen Zeit, sagt jedoch nichts darüber aus, wie lange ein System am Stück Verfügbar ist.
- *Reliability*: Definiert wie lange (Zeitspanne) ein System ohne Ausfall korrekt arbeiten kann. Wenn ein System jede Stunde für 1 ms ausfällt, hat es zwar eine sehr hohe Availability, ist jedoch höchst unzuverlässig.
- *Safety*: Definiert, wie sicher ein System im Falle eines Fehlers agiert, sodass keine größeren Schäden passieren.
- *Integrity*: Das System soll nicht in einen ungewünschten/nicht definierten Zustand gelangen.
- *Maintainability*: 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.

Klassifizierung von Faults

- *transient*: Treten zufällig, jedoch nicht wiederholt auf (zB Vogelschwarm stört Funkverbindung)
- *intermittent*: Treten zufällig auf, verschwinden, kehren aber wieder (zB Wackelkontakt)
- *permanent*: Treten solange dauerhaft auf, bis die fehlererzeugende Komponente ersetzt wurde (zB Hardware-Defekt)

2. **Geben Sie verschiedene Fehlermodelle ("failure models") an und diskutieren Sie diese. Wozu benötigt man überhaupt Fehlermodelle? Inwiefern ist es u.U. heikel zu spezifizieren, daß ein System "k-fault-tolerant" sein soll?**

Fehlermodelle werden dazu verwendet, um klassifizieren zu können, wie schwerwiegend ein bestimmter Fehler ist, d.h. um mögliche Auswirkungen besser einschätzen zu können.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

K-fault-tolerant
Diese Eigenschaft trifft auf Systeme zu, bei welchen bis zu k Komponenten ausfallen können, ohne dass das System davon beeinträchtigt wird und immer noch korrekte Ergebnisse geliefert werden. Das Problem bei der k-fault-tolerance ist, dass Prozesse auch weiterhin aktiv sein können, nachdem sie in einen fehlerhaften Zustand geraten sind. Gegeben der Fall, es tritt ein *byzantinischer Fehler* auf und liefern die fehlerhaften k Prozesse sowohl korrekte, als auch falsche Ergebnisse. Dann sind mindestens 2k+1 Prozesse nötig, um die k fehlerhaften Prozesse zu überstimmen und somit k-fault-tolerant zu sein. In der Realität ist es jedoch ohne statistische Analysen kaum möglich, exakt zu bestimmen, wieviele Prozesse fehlerhaft sind.

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

Um auch zum Zeitpunkt eines Fault ein fehlerfreies Verhalten garantieren zu können, sind Redundanzen notwendig. Diese können entweder *zeitlich* (zB mehrmalige Übertragung wie bei TCP), *datentechnisch* (Prüfsummen, Hamming-Distanzen, etc.) oder *physikalisch* (doppelte Ausführung eines Servers wie zB im DNS) realisiert sein.
Zeitliche Redundanz ist besonders bei transient oder intermittent Faults hilfreich. 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). Ein Beispiel für eine redundante Systemarchitektur ist Triple Modular Redundancy (TMR). Dabei wird jedes Element einer sequentiellen Aneinanderreihung von Komponenten 3 Mal ausgeführt. Als Input bekommt jedes Element das (von einem Voter ermittelte) Mehrheitsergebnis der vorherigen Komponente. Die Voter müssen, da sie selbst fehlerhaft sein können, ebenfalls repliziert sein.

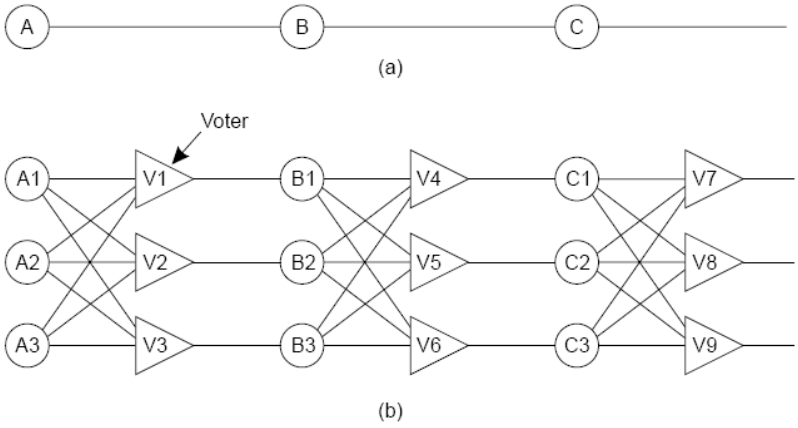


Figure 7-2. Triple modular redundancy.

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

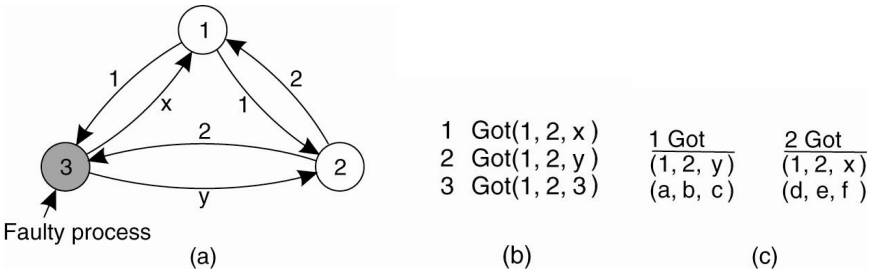
Im two-army Problem geht es darum, dass mit unzuverlässiger Kommunikation 2 Prozesse nicht zu einem gemeinsamen Konsens kommen können. Grundannahme ist, dass es eine rote und eine blaue Armee gibt. Die blaue Armee, bestehend aus 2 Generälen, an verschiedenen Orten, möchte sich einen Angriffzeitpunkt 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. Dies führt zu einem hohen Nachrichten-Aufwand und zu der Feststellung, dass es bei unsicheren Verbindungen keine Sicherheit über den Erhalt der Nachricht gibt. Damit ist es hier nicht möglich, zu einer Übereinkunft zu kommen.

5. Erläutern Sie die Aussage der "Byzantinischen Generäle".

Das Problem der *Byzantinischen Generäle* 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. 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 einen, der versucht die Kommunikation zu stören und bei jeder Nachricht eine

andere Truppenstärke liefert. Schließlich weiß jeder General über die Truppenstärke der anderen Bescheid und teilt den anderen seine gesammelten Informationen mit, wobei der falsche General wieder fehlerhafte Werte nennt. 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 (nach Lamport) 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. Die folgende Grafik verdeutlicht: Bei 3 Teilnehmern und einem fehlerhaften davon kann es zu keinem Konsens kommen, da keiner eine Mehrheit aus 2 gleichen Vektoren bilden kann. Die Werte a-f werden von General 3 „erfunden“.



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

Client kann Server nicht finden
Dies ist der Fall wenn zB alle Server down sind oder sich die Interfaces in der Zwischenzeit verändert haben.

Anfrage von Client ist verloren gegangen
Eine Möglichkeit um einem solchen Fehler zu begegnen ist, die Nachricht in einer gewissen Periode noch einmal zu senden, sollte keine Antwort vom Server eintreffen.

Server crasht nach Erhalt einer Anfrage
Hier sind 2 Fälle zu unterscheiden: Entweder kommt es vor oder nach der Bearbeitung der Anfrage zum Crash.

Antwort des Servers ist verloren gegangen (lost reply)
Falls der Client keine Antwort vom Server erhält, kann er kaum beurteilen, ob seine Anfrage bereits durchgeführt wurde oder nicht. So könnte seine ursprüngliche Anfrage den Server nie erreicht haben oder aber die Anfrage wurde bereits ausgeführt und nur die Bestätigung ist verloren gegangen. Auch kann die Anfrage den Server zwar erreicht haben, dieser aber gleich danach abgestürzt sein.
Eine Möglichkeit wäre es, die Anfrage erneut zu stellen, was aber nur bei idempotenten Operationen (Operationen welche wiederholt ausgeführt werden können, zB Daten lesen) sinnvoll ist. Man könnte nun versuchen, alle Request idempotent aufzubauen, was aber nicht immer möglich ist (zB Banküberweisung). Eine weitere Möglichkeit ist, 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, um zu markieren, ob es sich um eine neue Nachricht oder eine Retransmission handelt.

Client crasht nach Senden einer Anfrage
Dies hat ein Verwaisen der Rückantwort zur Folge, was zu unnötiger CPU Last oder im schlimmsten Fall zum Blockieren von Files oder Ressourcen führen kann.

7. 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").

Reliable Multicasting-Schemes bezeichnen Services, die garantieren, dass Nachrichten an alle gewünschten Prozesse verlässlich versendet werden bzw. dort ankommen. Eine sichere Übertragung an wenige Prozesse kann einfach durch point-to-point Verbindungen (TCP) bewerkstelligt werden. Schwieriger ist es allerdings, ein gesichertes Multicasting anzubieten, vor allem wenn sich die Zahl der Mitglieder einer Gruppe dynamisch ändert, etwa neue Mitglieder hinzukommen oder das Crashen eines Prozesses. Hier muss vor dem

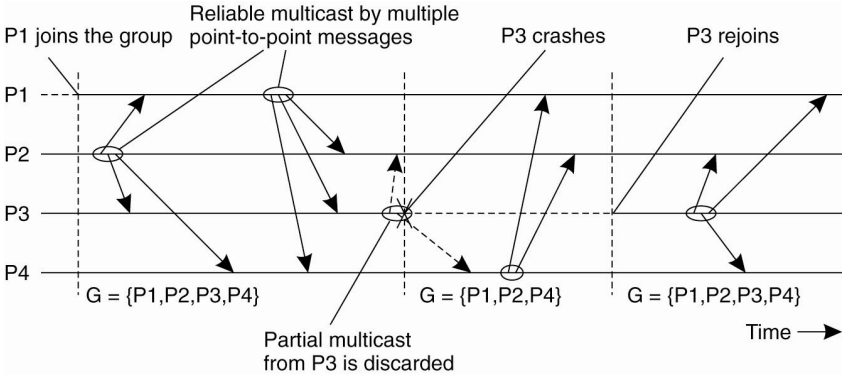
Versenden der Nachricht die Gruppen-Zusammensetzung geklärt sein (*siehe Atomic Multicast*). Zudem sollten die Mitglieder alle versendeten Nachrichten in der gleichen Reihenfolge erhalten. Dies kann man erreichen, indem der sendende Prozess jeder Nachricht eine Sequenz-Nummer anhängt. Ein Receiver kann somit leicht erkennen, ob eine an ihn adressierte Nachricht verloren gegangen ist und entweder den Erhalt bestätigen oder einen Verlust melden.

Ein Hauptproblem bei dieser Form von Multicasting ist die *schlechte Skalierbarkeit*. Gibt es N Empfänger, kommen in der Regel auch N Acknowledgements zurück, die den Server buchstäblich überschwemmen. Eine Möglichkeit ist, dass Empfänger den Sender nur in jenen Fällen informieren, wenn sie eine Nachricht nicht erhalten haben. Jedoch kann der Server aus Performancegründen nicht ewig auf eine solche Fehlermeldung warten sondern wird die ursprüngliche Nachricht nach einer bestimmten Zeit löschen. Dadurch kann es passieren, dass Nachrichten einzelne Empfänger nie erreichen.

Atomic Multicast
Ein Verteiltes System muss sicher stellen, dass alle Nachrichten die Prozesse in einer bestimmten, gleichen Reihenfolge erreichen. Außerdem muss garantiert werden, dass entweder alle Nachrichten oder keine einzige Nachricht versendet werden. Diese Anforderungen werden als *Atomic Multicast Problem* bezeichnet. Sei nun eine verteilte Datenbank gegeben, in der Prozesse zu Gruppen zusammengefasst sind. Somit erhält jede Replica einen eigenen Prozess. Update-Operationen werden nun an alle Prozesse weiter geleitet (Multicasting) und sofort durchgeführt. Wenn während dieses Update eine Kopie crasht, bleibt diese auf dem alten Stand während sich alle anderen Replikas korrekt upgedatet haben. Wird die eine Kopie wieder aktiv, kann es sein, dass sie mehrere Updates versäumt hat.

Im Gegensatz dazu sieht *Atomic Multicasting* vor, dass die restlichen Prozesse nur dann ein Update durchführen, wenn sie sich darauf geeinigt haben, dass der fehlerhafte Prozess nicht Teil ihrer Gruppe ist. Nach dessen Wiederherstellung (auf den alten Stand) muss er der Gruppe wieder beitreten. Dann wird sein Datenbestand dem der anderen Gruppenmitglieder angeglichen. Zusammenfassend wird mit diesem System sichergestellt, dass funktionierende Prozesse einen einheitlichen/konsistenten Eindruck von der Datenbank haben.

Virtual Synchrony
Diese (strengere) Ausprägung eines reliable Multicast garantiert, dass eine Nachricht, die an eine Gruppe gesendet wird, alle funktionierenden Prozesse erreicht. Crasht der sendende Prozess während des Sendevorganges, erhalten entweder alle übrigen Prozesse die Nachricht, oder sie wird, in den Fällen wo sie bereits eingegangen ist, vom Empfänger ignoriert. Virtual Synchrony sowie Atomic Multicast im Allgemeinen basieren auf einer sogenannten Group View - das ist eine Liste, die alle Prozesse einer bestimmten Gruppe zum Zeitpunkt des Multicast, basierend auf dem Kenntnisstand des Senders, beinhaltet. Verändert sich die Gruppenzusammensetzung, findet ein *view change* statt. Das Modell der virtual Synchrony bestimmt, dass ein Multicast, während dessen Übermittlung ein view change auftritt, fertig gestellt werden muss, bevor die Meldung eines veränderten Views an alle Prozesse übermittelt und der View Change statt findet.



Security

Buch: Kap. 9, 12.8

Fragen:

1. **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.**

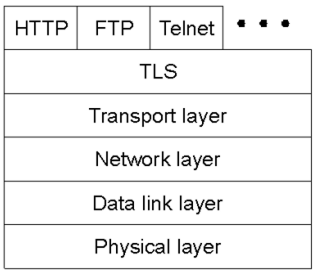
Availability: Eine Ressource sollte stets verfügbar sein, wenn sie benötigt wird.
Confidentiality: Nur berechtigte Nutzer dürfen Zugriff zu einem System bzw. Information haben.
Integrity: Veränderungen an Hardware, Software, Datenbestand dürfen nur nach erfolgter Authorisierung durchgeführt werden.

Generell steht Security in einem Naheverhältnis zur *Dependability*, die sich darauf bezieht, dass man sich darauf verlassen kann, dass die angebotenen Dienste auch verfügbar sind.

Security threats:
- *Interception*: Unauthorisierter Zugriff, zB Abhören von Kommunikation
- *Interruption*: Services oder Daten sind nicht mehr zugänglich bzw unbrauchbar, zB DoS-Attacke
- *Modification*: Unauthorisierte Änderung von Daten, zB Änderung einer Message vor Zustellung
- *Fabrication*: Zusätzliche Daten werden generiert, die normalerweise nicht existieren würden, zB Anlegen eines neuen Passwort-Eintrages) – Abhilfe durch Authorisierung

Security mechanisms:
- *Encryption*: Verändert Daten sodass sie von Eindringlen nicht verstanden werden, erlaubt Integritäts-Checks
- *Authentication*: Verifiziert die Identität von Einheiten (User, Host, etc.)
- *Authorization*: Stellt fest, ob eine Einheit bestimmte Aktionen durchführen darf
- *Auditing*: Sammelt Informationen über Aktionen in Logs, kein aktiver Schutz vor Eindringlingen

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



Zweck des SSL bzw. seines Nachfolgers, TLS, ist es, eine Verbindung über TCP zu verschlüsseln. Beide sind applikations-unabhängig und befinden sich im Internet Protokoll Stack zwischen dem Transport Protokoll und den higher-level Protokollen wie HTTP, FTP. Wichtigster Baustein von TLS ist der *record protocol layer,* der eine sichere Verbindung zwischen Client und Server gewährleistet.

Herstellen der Verbindung
1. Client informiert Server über unterstützte Cryptographie- und Kompressions-Methoden
2. Server wählt Methoden aus
3. Server sendet Zertifikat mit Public Key an Client
(4. Client sendet eigenes Zertifikat zurück)
(5. Client sendet Zufallszahl, verschlüsselt mit Server-Key, an Server)

3. **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.**

Cryptographie befasst sich mit der Ver-/Entschlüsselung von Daten, um sie vor sicherheitskritischen Aktionen zu schützen. Der Empfänger verschlüsselt eine Nachricht (*plaintext*) bevor er sie an den Empfänger schickt. Dieser muss, um den Inhalt lesen zu können, die verschlüsselte Nachricht (*ciphertext*)

entschlüsseln.

Symmetrische Verschlüsselung
Hier wird der selbe Schlüssel für En-/Decryption verwendet wobei Sender und Empfänger darüber verfügen müssen. Darum werden entsprechende Systeme als *shared-key* oder *secret-key* Systeme bezeichnet. Schlüssel und Nachricht müssen zusammen versendet werden - da der Schlüssel geheim gehalten werden muss, ist hierfür eine sichere Verbindung notwendig.

Vorteil:	Geschwindigkeit
Nachteile:	Key-Management ist sehr komplex → jeder Host verfügt über N-1 Schlüssel = N*(N-1)/2 gesamt
Beispiel:	Data Encryption Standard (DES)

Asymmetrische Verschlüsselung
Charakteristisch hierfür ist, dass es verschiedene Schlüssel zum Ent-/Verschlüsseln gibt, die aber zusammen ein eindeutiges Paar ergeben. Dabei ist ein Schlüssel geheim, der andere aber öffentlich – darum bezeichnet man dieses System als *public-key* System. Möchte nun Person A eine Nachricht an Person B schicken, verschlüsselt sie diese mit dem öffentlichen Schlüssel von B. Nachdem B die einzige Person ist, die über ihren (privaten) Schlüssel verfügt, kann sie die Nachricht dekodieren.

Vorteil:	Sicherheit (Entschlüsselung nur durch 1 Person), einfache Verteilung der Schlüssel (public key alleine ist nutzlos)
Nachteile:	langsam, jede Person muss über seine 2 eigenen Schlüssel, sowie die public keys aller N-1 übrigen Personen verfügen
Beispiel:	RSA (zB Digitale Signatur)

4. 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.

Bei einem Secure Channel handelt es sich um eine sichere Verbindung, um die Kommunikation zwischen zwei oder mehreren Teilnehmern vor Interception, Modification und Fabrication zu schützen. Dies wird durch Verschlüsselung (public- und private-key encryption) des Transfers erreicht.

Unerwünschte Effekte (Auszug):

- *Spoofing* (Vortäuschung einer bestimmten Identität)

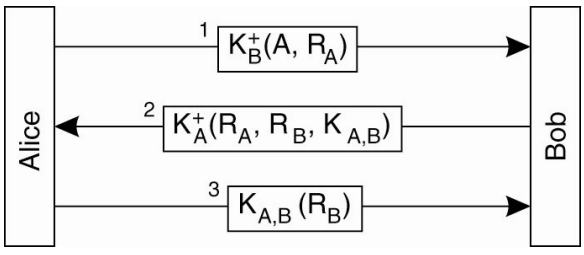
```
'A'-> B : "I want to buy 10 widgets."
 B -> A : "Here are your 10 widgets."
 A -> B : "I did not order widgets."
```

- *Message Tampering* (Änderung des Nachrichteninhaltes)

```
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!"
```

- *Eavesdropping* (Mithören des Nachrichtenaustausches)
- *Replaying* (Angreifer sendet zuvor mitaufgezeichnete Daten um Identität vorzugeben)

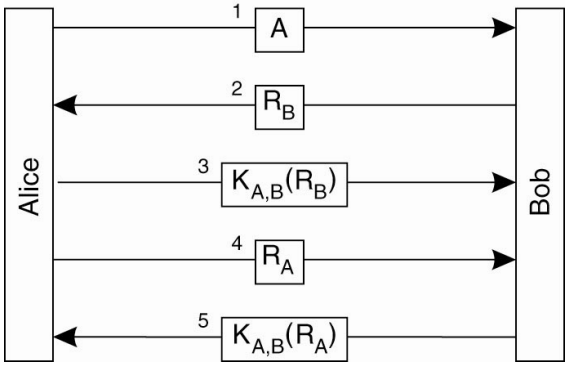
Authentifizierung mittels public-key Kryptographie:



0. 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 (zB Zufallszahl, nur 1x verwendet. 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 Ka,b, 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.

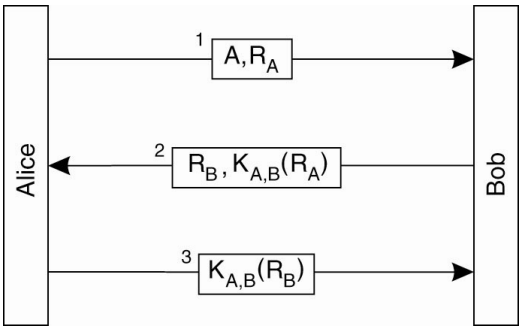
5. Erklären Sie, wie sich zwei Kommunikationsteilnehmer basierend auf symmetrischen Schlüsseln gegenseitig authentifizieren können. Zeigen Sie auch auf, welche Probleme entstehen können, wenn ein Protokoll falsch "optimiert" wird.

Wie im Fall von Kommunikation basierend auf public-key Kryptographie handelt es sich auch hier um ein *challenge-response Protokoll*. Das bedeutet, dass ein Teilnehmer den anderen zu einer bestimmten Antwort auffordert, die dieser nur geben kann, wenn er den shared key kennt.

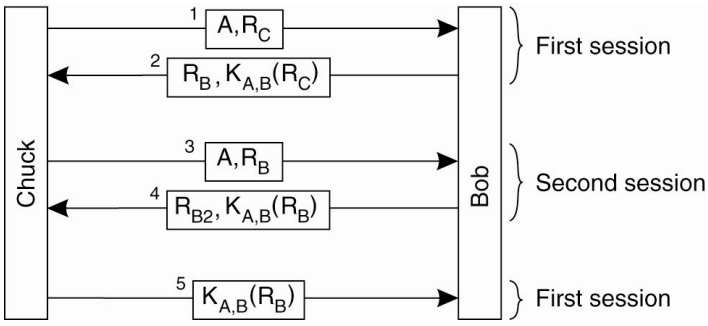


1. Alice sendet ihre Identität an Bob um einen Kommunikationskanal zu eröffnen.
2. Bob Sendet eine Challenge Rb an Alice (zB Zufallszahl)
3. Alice verschlüsselt diese mit dem secret Key Ka,b (den sie mit Bob teilt) und schickt die Nachricht zurück. Bob kann die Nachricht entschlüsseln und festzustellen, 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 beim Kommunikationspartner um Bob handelt.
4. Alice sendet Challenge Ra an Bob.
5. Bob retourniert die mit Ka,b verschlüsselte Challenge Ra.

Eine Optimierung zeigt das Modell mit 3 Nachrichten:



Hierbei kann es zu einer *Reflection Attack* kommen. Eindringling Chuck stellt eine Verbindung mit Bob unter Vortäuschung der Identität von Alice her.
1. Chuck sendet eine Nachricht mit der Identität von Alice sowie Rc.
2. Bob sendet eigene Challenge sowie die Antwort Ka,b(Rc) an Chuck.
X. Hier sollte Chuck eigentlich die mit dem shared Key verschlüsselte Rc retournieren.
X. Stattdessen versucht Chuck einen zweiten Kanal zu öffnen, sodass Bob die Verschlüsselung für ihn übernimmt!
3. Chuck sendet die Identität(A) mit Challenge Rb an Bob.
4. Bob weiß nicht, dass er Rb bereits verwendet hat und antwortet mit der verschlüsselten Rückantwort Ka,b(Rb) und Challenge Rb2.
5. Nun verfügt Chuck über Ka,b(Rb) und kann den ersten Kanal öffnen.



6. 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.

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. Gleichzeitig wird eine Veränderung der Nachricht verhindert, da die Signatur eindeutig an den Inhalt der Nachricht gebunden ist.

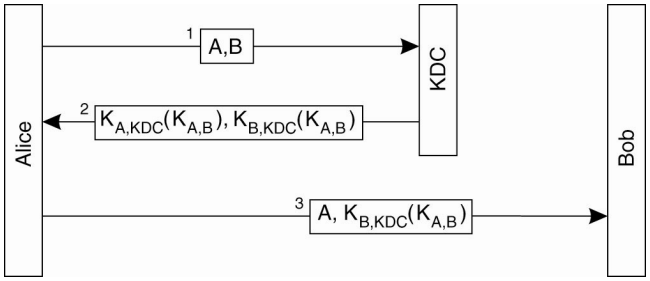
Zu Beginn wird der *Hashwert* der zu signierenden Nachricht gebildet. Dieser Hashwert wird mit dem privaten Schlüssel des Absenders A verschlüsselt (=Signatur) und an die Nachricht angehängt. Der Empfänger der Nachricht bildet wiederum den Hashwert und entschlüsselt die Signatur mit dem öffentlichen 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.

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.
- *Weak collision resistance*. Gegeben *x*, ist es schwer ein *y (!= x)* zu finden, für das gilt: *H(x) = H(y)*.
- *Strong collision resistance*. Es ist schwer, beliebige x und y zu finden für die gelten: *H(x) = H(y)*.

7. Was ist ein Key Distribution Centre (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, da N*(N-1)/2 Schlüssel gewartet werden müssen. Mittels eines KDC kann die Anzahl der Schlüssel auf N reduziert werden. Die Idee ist dabei, dass das KDC mit jedem Host genau einen Schlüssel teilt.



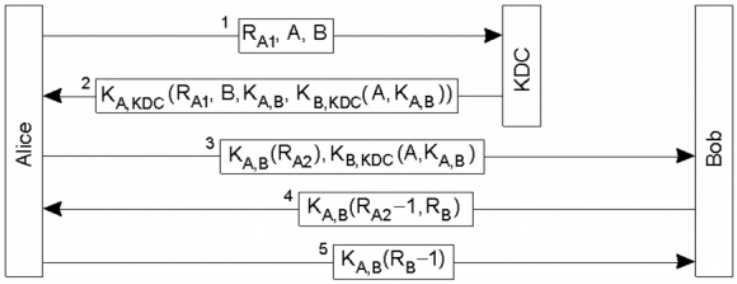
1. Alice teilt dem KDC mit, sich mit Bob unterhalten zu wollen.
2. KDC antwortet mit einer Nachricht, die mit dem *geheimen Key Ka,kdc* verschlüsselt ist und als Inhalt einen *shared-key Ka,b* hat und sendet Ka,kdc(Ka,b) mit, das für Bob bestimmt ist.
3. Alice übermittelt den Schlüssel (Ticket) aus 2) an Bob.

X. Würde KDC das Ticket direkt an Bob senden, könnte es passieren, dass Alice einen Kanal zu Bob aufmachen will, obwohl dieser den Schlüssel noch nicht hat.

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

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. Zuletzt 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



Technology Overview

Buch: 10.1.2, Überblick Kap. 12 und 13

Fragen:

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

Voraussetzungen für Kommunikation:
referential coupling: Prozess muss explizite Referenz auf einen anderen haben
temporal coupling: Die beteiligten Prozesse müssen aktiv sein

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

Antwort:

		Temporal	
		Coupled	Uncoupled
Referential	Coupled	(a) TCP/Telefon	(b) Mail/Brief/SMS
	Uncoupled	(c) CSS ???	(d) jini/Linda

(c) publish/subscribe Systeme

2. **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.**

In *Publish-subscribe* Systeme, die zu den event-basierten Systemen mit *loose coupling* gehören, können sich Prozesse für Nachrichten bestimmter Themen anmelden. Dann werden ihnen alle gewünschten Nachrichten, die von anderen Prozessen erstellt werden, automatisch zugestellt. PS-Systeme setzen voraus, dass beide Prozesse zum Zeitpunkt der Kommunikation aktiv sein müssen (*temporally coupled*), eine dezidierte Adressierung des Empfängers ist jedoch nicht nötig (*referentially uncoupled*).

Jini 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).

3. **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 es 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.

Operation	Beschreibung
`Get`	`Dokument an den Client schicken`
`Put`	`Dokument speichern`

Post	Bereitstellung von Daten die einem Dokument hinzugefügt werden soll
Delete	Dokument löschen
Head	Anforderungen, Head eines Dokument zurückzugeben

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