Verteilte Systeme - Prüfungsfragen

Was ist ein verteiltes System?

Ein VS ist mehr als die **Vernetzung** von Rechnern. Es besteht aus mehreren **unabhängigen** Knoten, die natürlich auch unabhängig voneinander ausfallen können. Ein VS ist durch ein **Netzwerk** verbunden, über das eine Interaktion in Form von **Nachrichtenaustausch** möglich ist. Das Netzwerk kann unzuverlässig sein, variable Nachrichtenverzögerungen aufweisen und ist im Vergleich zu Multiprozessorsystemen wesentlich langsamer. Ein weiteres Merkmal, durch das sich ein VS auszeichnet, ist die **Kooperation der Knoten** untereinander. Die Knoten (Prozesse) interagieren, um eine gemeinsame Aufgabe zu lösen oder einen Dienst anzubieten.

Welche Probleme gibt es bei VS?

1. lokal → entfernt: **mehr Fehlerarten** im Falle entfernt ausgelegter Interaktionen
2. direkte → indirekte **Bindung**: Konfigurierung wird zu einem dynamischen Vorgang, erfordert Bindungsunterstützung zur **Laufzeit**
3. sequentielle → **nebenläufige Ausführung**: Nebenläufigkeit durch Parallelität erfordert Mechanismus zur **Koordinierung** der Aktivitäten
4. synchrone → **asynchrone Interaktion**: Verzögerungen durch den Kommunikationskanal erfordern Unterstützung für asynchrone Interaktionen und zur Fließbandverarbeitung
5. homogene → **heterogene Umgebung**: Interaktion zwischen entfernten Systemen erfordern eine **gemeinsame Datenrepräsentation**
6. einzelne Instanz → **replizierte Gruppe**: Replikation kann Verfügbarkeit oder Zuverlässigkeit bereitstellen, erfordert aber auch **Maßnahmen zur Konsistenzwahrung**
7. fester Platz → Wanderung: **Lage** entfernter Schnittstellen kann sich zur Laufzeit ändern
8. einheitlicher → **zusammengeschlossener Namensraum**: **Namensauflösung** muss Verwaltungsgrenzen zwischen verschiedenen entfernten Systemen reflektieren
9. gemeinsamer → **zusammenhangloser Speicher**: Mechanismen des gemeinsamen Speichers sind nicht im großen Maßstab anwendbar

Welche Heterogenitäten gibt es?

1. Netzwerke: Anschlusstyp, Medium, Technik, Topographie
2. **Prozessoren**: Informationsdarstellung, **Byte Sex**
3. **Betriebssysteme**: Ausführungsumgebung, API
4. **Programmiersprachen**: Semantik, Pragmatik
5. Implementierungen durch verschiedene Personen

⇒ Was für Probleme gibt es mit unterschiedlicher Hardware?

Nachrichten müssen zur Übertragung ver- und wieder entpackt werden, was als (Un) Marshalling bezeichnet wird. Falls unterschiedliche Hardware verwendet wird, ist es oft so, dass Daten in anderer Art und Weise repräsentiert werden (Byte Sex: little/big endian).

Zur Lösung dieses Problems gibt es verschiedene Möglichkeiten. Bei Verwendung von **XDR** (external data representation) werden die Daten vom versendenden System in eine festgelegte **kanonische Repräsentation** überführt, dann versandt und am Ende von vom Empfänger in seine eigene Repräsentation umgewandelt. Das hat zur Folge, dass z.B. in homogenen VS unnötiger Aufwand betrieben werden muss.
Beim sogenannten "**Sender makes it right**" kennt der Sender die gewünschte Datenrepräsentation des Empfängers und wandelt sie dementsprechend um. Probleme treten allerdings bei Mehrteilnehmerkommunikation auf. "**Receiver makes it right**" bedeutet, dass der Sender der Nachricht Informationen über die Datenrepräsentation mitgibt, sodass der Empfänger anhand dieser Information die Daten selbst in seine gewünschte Form umwandeln kann.

⇒ Wie löst man das Problem der unterschiedlichen Programmiersprachen?

Mithilfe einer **Middleware**, also eine Softwareschicht zur Abstraktion von den jeweiligen Systemeigenheiten. Ein **programmiersprachenabhängiger** Vertreter für solch eine Middleware ist **Java RMI**. **Programmiersprachenunabhängig** hingegen sind **CORBA** und **Web Services**.

Welche Fehlerarten gibt es?

Ein **Error** kann durch einen **Fault** in einem System entstehen, was am Ende zu einem System **Failure** führen kann.

Welche Fehlermodelle?

Fehler werden in gutmütig (**benign**) und bösartig (**malicious**) eingeteilt. Zur ersten Gruppe zählt die Situation, in der ein Knoten komplett ausfällt (**Crash Stop**) - wobei eine feingranularere Unterteilung durch **Fail Stop** (jeder korrekte Knoten erfährt in endlicher Zeit vom Ausfall eines Knotens) und **Fail Silent** (keiner perfekte Ausfallerkennung möglich → asynchrones/partiell synchrones Modell) gegeben ist. **Crash Recovery** ist ebenfalls eine Art gutmütiger Fehler. Hierbei kann ein korrekter Knoten endlich oft ausfallen und wieder anlaufen.
Bösartige Fehler, auch byzantinische Fehler genannt, umfassen auch bösartige Angriffe von außen auf das System oder fehlerhafte Prozesse, die beliebige Aktionen - auch koordiniert untereinander - ausführen können.

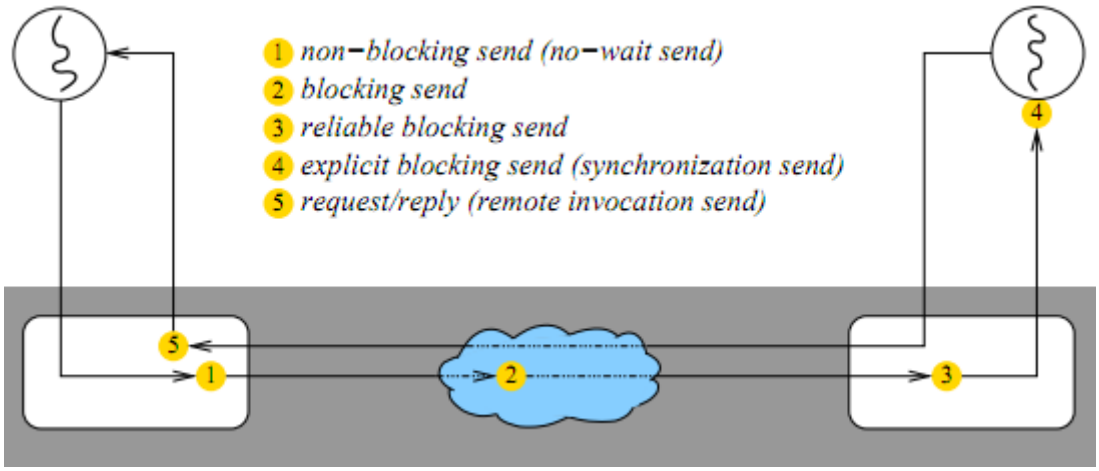
Welche Transparenzen gibt es?

1. **Netzwerktransparenz**
    a. **Zugriffstransparenz**: ermöglicht Zugriff auf lokale und entfernte Betriebsmittel unter Verwendung identischer Operationen
    b. **Ortstransparenz**: erlaubt Zugriff auf Betriebsmittel, ohne ihren Ort kennen zu müssen (DNS)
2. **Nebenläufigkeitstransparenz**: erlaubt mehreren Prozessen gleichzeitiges und konfliktfreies Arbeiten mit denselben gemeinsam genutzten Betriebsmitteln
3. **Replikationstransparenz**: erlaubt die Verwendung mehrerer Betriebsmittelinstanzen (Replikate), um Zuverlässigkeit und Leistung zu verbessern
4. **Fehlertransparenz**: erlaubt kontinuierlichen Rechnereinsatz trotz des möglichen Ausfalls von Hard- oder Softwarekomponenten

5. Migrationstransparenz: Verschieben/Wandern von Betriebsmitteln und Prozessen innerhalb eines Systems ohne Beeinträchtigung der laufenden Arbeit von Benutzern oder Programmen
6. Leistungstransparenz: erlaubt die lastabhängige Neukonfiguration des Systems zum Zweck der Leistungssteigerung
7. Skalierungstransparenz: erlaubt die Vergrößerung des Systems ohne Auswirkungen auf die realisierte Struktur und die zum Einsatz gebrachten Algorithmen

Welche IPC Semantiken gibt es?

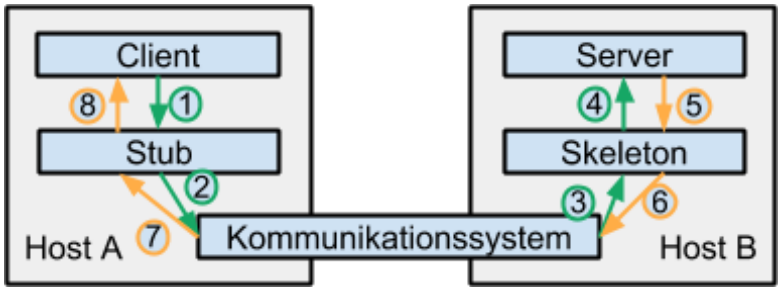
1. **no-wait send**: der Sendeprozess wartet, bis die Nachricht im Transportsystem zum Absenden bereitgestellt worden ist
2. **synchronization send**: der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess angenommen worden ist
3. **remote-invocation send**: der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess verarbeitet und beantwortet worden ist



Welche IPC Protokolle für Fernaufrufe gibt es?

1. **request**: falls entfernte Prozedur keinen Rückgabewert liefert → 1 Nachricht
2. **request-reply**: Antwort impliziert Bestätigung → 2 Nachrichten
3. **request-reply-ack reply**: gestattet Verwerfung der gespeicherten Antworten auf Server, wenn keine weitere Anforderungsnachricht gesendet wird → 3 Nachrichten

Wie funktioniert ein Fernaufruf (Java RMI)?



Der Client Stub ist der Prozedurstumpf auf Seite des Auftraggebers. Er abstrahiert von der Örtlichkeit der fernen, aufgerufenen Prozedur (Ortstransparenz) und setzt dazu den eigentlichen Prozeduraufruf in einen Nachrichtenaustausch um, indem er Aufrufparameter verpackt und Rückgabewerte wieder entpackt.

Der Server Stub (Skeleton) ist der Prozedurstumpf auf der Seite des Auftragnehmers und abstrahiert von der Örtlichkeit der fernen, aufrufenden Routine.

Probleme bei Rückruf und Vermeidungsmöglichkeiten?

Durch einen Rückruf an einen Aufrufer kann es (ähnlich zu prozessorientierten Programmen) zu einer **Verklemmung** kommen. Eine remote-invocation send-Umsetzung bewirkt dabei nämlich **zyklisches Warten**.
Verklemmungsfreie Fernrückrufe sind auf zwei Arten realisierbar:

1. **Zustandsmaschine**: Hier wird die Synchronität durch **synchronization send** oder **no-wait send** gelockert (statt remote-invocation send). Nach dem Senden wird im receive auf beides gewartet, Antwort oder Rückruf. Der Faden muss sich zwischen verschiedenen Arbeitsphasen multiplexen.
2. **Rückrufanbieter**: Wenn man das remote-invocation send-Modell als Fernaufrufgrundlage beibehalten will, muss man zur Rückrufverarbeitung einen **eigenen Faden** vorsehen. Der Rückruf wird faktisch als ganz normale **Dienstleistung** verstanden. Diese Lösung ist weniger fehleranfällig.

Können Stubs automatisch erzeugt werden, z.B. für C/C++?

Nicht so einfach, da Probleme mit Parameterübergabe. Bei Pointern nicht automatisch erkennbar, was dahinter steckt (Größe, Parameterart), Probleme durch Objektorientierung (Kassenhierarchie, abstrakte Klasse). Automatische Generierung möglich, wenn IDL (kurz erklären) verwendet wird.

Semantikaspekte?

In VS ist die Semantik konventioneller, lokaler Prozeduraufrufe nur teilweise erreichbar. Durch die örtliche Trennung teilen sich Code und Daten nicht den selben physikalischen Arbeitsspeicher, beide Seiten arbeiten weitestgehend autonom und können auch unabhängig voneinander ausfallen.
Um Aufwand zu minimieren, müssen **Parameterarten** unterschieden werden. Zu den Parameterarten zählen Eingabe-, Ausgabe- und Ein-/Ausgabeparameter. Auch die **Parameterübergabe** muss ggf. explizit spezifiziert werden. Call-by-reference unterscheidet sich z.B. erheblich von call-by-value/result.
Eingabeparameter sind nur Bestandteil der Aufforderungsnachricht, Ausgabeparameter Bestandteil der Antwortnachricht und Ein-/Ausgabeparameter Bestandteil beider Nachrichten.
Die Art (Ein- vs. Aus- vs. Ein-/Aus-) eines jeden Parameters müsste in der Schnittstelle spezifiziert sein, da z.B. in C/C++ bei Zeigern, Referenzen oder Feldern dies nicht eindeutig ersichtlich ist, ohne tiefgründige Kenntnisse über den Code zu haben.
Call-by-value/result ist einfach zu behandeln, da die aktuellen Parameter jeweils in die Anforderungs- und auch in die Antwortnachricht kopiert werden. **Call-by-reference** hat jedoch je nach Parameterart eine unterschiedliche Abbildung:

1. Eingabeparameter → call-by-value
2. Ausgabeparameter → call-by-result
3. Ein-/Ausgabeparameter → call-by-value/result

Gültigkeitsbereiche von Variablen sind anders als im lokalen Fall (z.B. statische Variablen sind zwar in einem Prozess eindeutig, jedoch nicht in VS). Allgemein sind Adressen abhängig vom Kontext, in dem sie definiert sind. Sie beziehen sich auf ein Programm in einem Adressraum auf einem Rechner. Adressen verteilter Programme müssen daher

eine **geographische Komponente** besitzen. Die Verwendung von **logischen Adressen** im Kontext VS bringt einen beträchtlichen Aufwand mit sich und setzt entsprechende Unterstützung voraus. Abhilfe schafft hier der **Smart Pointer**, womit eine ferne Adresse als zweiteiliger "geschickter" Zeiger ausgelegt wird:

1. ein systemweit eindeutiger **Kontextbezeichner** (z.B. Prozessidentifikation):
    a. durch Rückruf wird das referenzierte ferne Objekt herangeholt
    b. der Platzhalter zur Aufnahme der lokalen Kopie wird dynamisch angelegt
    c. die Adresse des Platzhalters wird dann als Zeigerkomponente übernommen
2. **Zeiger**: Adresse auf das ferne Objekt / den lokalen Platzhalter

Diese Technik wird durch linguistische Unterstützung praktikabler. In C++ könnte man z.B. ferne Adressen als Instanzen von Klassenschablonen realisieren und überlagerte Operatoren dazu verwenden um bei Bedarf Rückrufe abzusetzen und den Zugriff zu regeln. Unter Umständen sind lokale Kopien nicht zwingend und alle Zugriffe können Rückrufe sein. "Pure" Zeiger in Nachrichten zu versenden ist im Regelfall sinnlos. Der Unterschied zwischen statischem und dynamischen Scope ist, dass der statische Scope bereits zur Übersetzungszeit bekannt ist und sich der dynamische Scope mit Betreten/Verlassen von Funktionen und Prozeduren ändert.

Desweiteren unterscheidet man verschiedene **Fernaufrufsemantiken**:

1. **maybe**: der Client hat im Fehlerfall keine Gewissheit über korrekte Ausführung
2. **at-least-once**: Mehrfachausführung möglich. Client wartet eine vorgegebene Zeitspanne und wiederholt nach Ablauf den Aufruf (im Normalfall ist die Zahl der Wiederholungen begrenzt, sodass nach dem n'ten mal eine Ausnahmesituation eintritt). Der Server muss hierfür **idempotente Operationen** anbieten.
3. **at-most-once**: Mehrfachausführung ist ausgeschlossen, dadurch ist diese Semantik auf dem Server vergleichsweise aufwendig und speicherintensiv. Eine Anfrage hat eine Kennung. Erhält der Server eine Anfrage mit einer Kennung, die er bereits erhalten hat, so führt er die Anfrage nicht erneut aus, sondern antwortet mit der bereits berechneten Antwort. Anfragen vom selben Client mit neuer Kennung erlaubt Löschen der vorherigen gespeicherten Antworten auf dem Server. Die Ausführung erfolgt allerdings nur, wenn keine Rechnerausfälle vorliegen.
4. **last-of-many**: akzeptiert nur die Antwort zum jüngst zurückliegenden Aufruf. Jeder (auch jeder wiederholte) Aufruf trägt eine eindeutige Kennung. Die Antwort trägt die selbe Kennung. Der Client verwirft (verzögerte) Antworten mit veralteter Kennung. last-of-many kann als Variante von at-least-once gesehen werden, jedoch ist die wiederholte Ausführung von Operationen weiterhin problematisch. Zumindest können die Clients aber mit aktuellen Daten arbeiten.
5. **exactly-once**: entspricht der Semantik lokaler Aufrufe, erfordert allerdings Transaktionskonzepte mit Wiederanlauf von Komponenten. Dies ist die wünschenswerte, ideale Semantik → leider in "Reinform" unerreichbar.

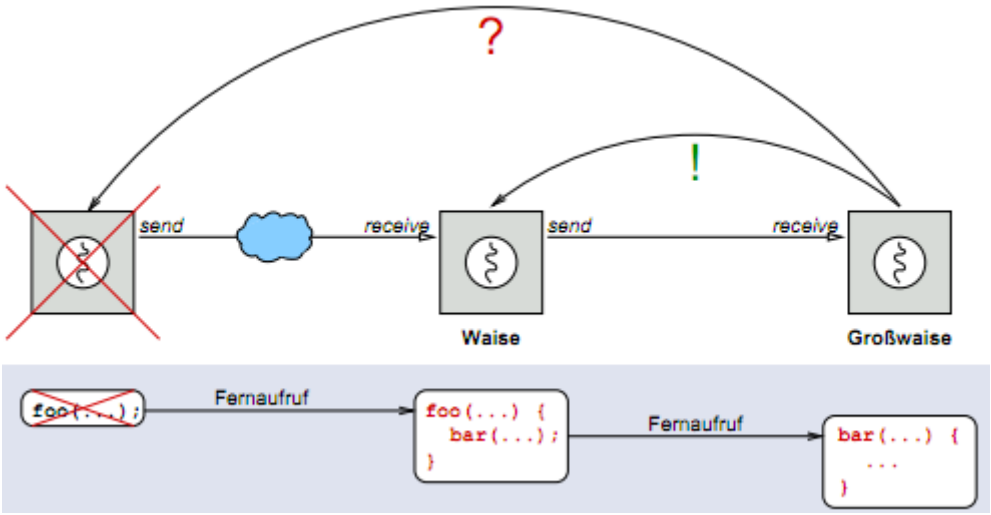
Was versteht man unter idempotenten Operationen?

Eine Operation wird als idempotent bezeichnet, wenn die wiederholte Ausführung derselben Operation (identische Parameterwerte) durch den Dienstanbieter immer den Effekt einer einmaligen Ausführung besitzt. Dazu ist ein **zustandsfreier Dienstanbieter** erforderlich (Bsp. NFS). Um Idempotenz bei zustandsbehafteten Diensten zu erreichen ist **Duplikateliminierung** notwendig. Duplikate können sonst zu wiederholten Ausführungen führen, was z.B. bei write-Operationen zur "Verfälschung" des Zustands führen würde.

Ein Beispiel für eine nicht idempotente Operation ist das Abheben am Geldautomaten. Wird hier z.B. aufgrund einer Verzögerung eine Anfrage wiederholt, empfängt der Dienstanbieter ein Duplikat der Anfrage, was zu einer wiederholten Abhebung führt, wenn der Dienstanbieter dieses Duplikat nicht erkennt. Der Abhebevorgang lässt sich allerdings in eine idempotente Operation überführen, indem zunächst der aktuelle Kontostand gelesen wird, lokal der abzuhebende Betrag subtrahiert wird und das Ergebnis als setze-Operation (setze Kontostand = x Euro) an den Dienstanbieter übertragen wird.

Was ist ein verwaister Aufruf und wie kann man Waisen behandeln?

Ein Waise (Orphan) ist ein Client, der den Aufruf (z.B. wegen eines zu kurzen Timeouts) abgebrochen hat und **nicht mehr am Ergebnis interessiert** ist oder mittlerweile überhaupt nicht mehr zur Verfügung steht.
Maßnahmen um unnötige Arbeit des Dienstanbieters möglichst zu vermeiden sind eine zusätzliche Überwachung des Clients durch den Server mithilfe eines **Timeouts**, ein **Ausfallzähler** pro Client oder der Versand von zusätzlichen **Statusanfragen** an den Client. Besondere Schwierigkeiten bereiten sogenannte Fernaufrufketten → **Transitivität**:

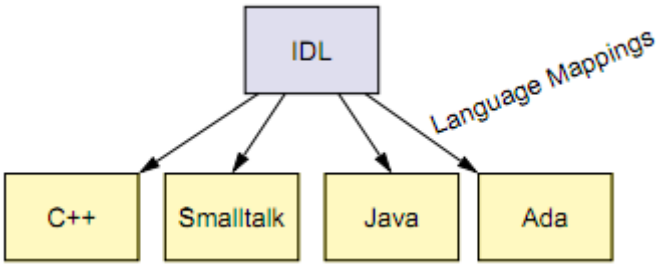


Es gibt verschiedene Ansätze zur **Waisenbehandlung**:

1. **extermination**: nach Wiederanlauf wird beim Client auf ausstehende Fernaufrufe geprüft. Den betreffenden Servern gehen dann Abbruchanforderungen zu. Es muss auch rekursiv vorgegangen werden (“Großwaisen”). Client Stubs müssen hierzu allerdings Logs vor der Anforderung anlegen und nach Empfang der Antwort löschen.
2. **expiration**: Server gibt dem Fernaufruf ein **Zeitquantum** zur Ausführung. Nach Ablauf erbittet er neues Zeitquantum vom Client → Fehlerfall führt zu Abbruch.
3. **reincarnation**: bei Wiederanlauf eines Client startet eine neue **Epoche**, was allen Maschinen mitgeteilt wird (broadcast), woraufhin auf den Maschinen alle Anbieter (Prozesse) terminiert werden. Jede Anforderungs- und Antwortnachricht enthält die Epochenkennung, womit unerwartete Antworten von Waisen herausgefiltert werden können.
4. **gentle reincarnation**: zum Epochenbeginn versucht jeder Server seine Clients zu lokalisieren. Erst wenn dies fehlschlägt wird der Prozesse terminiert.

Was ist die IDL?

Die Interface Definition Language (IDL) ist eine Sprache zur Beschreibung von Objekt-Schnittstellen, die unabhängig von der Implementierungssprache des Objekts ist. Es ist eine Sprachabbildung definiert, wie IDL-Konstrukte in Konzepte einer bestimmten Programmiersprache abgebildet werden. Diese Sprachabbildung ist Teil des CORBA Standards.



IDL ist angelehnt an C++. Es gibt eigene Datentypen, Module, Schnittstellen und Exceptions.

Was ist und wie funktioniert CORBA?

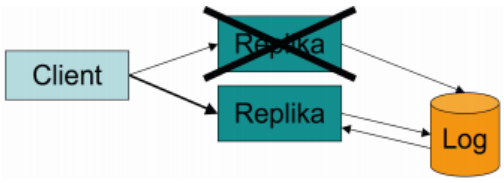
Die Common Object Request Broker Architecture ist eine **plattformunabhängige Middleware**-Architektur für verteilte Objekte. Es soll damit eine verteilte objektbasierte Programmierung ermöglicht werden. Durch CORBA wird **Heterogenitätstransparenz** erreicht:

1. verschiedene **Hardware-Architekturen**
2. verschiedene **Betriebssysteme** (und Betriebssystem-Architekturen)
3. verschiedene **Programmiersprachen** (→ in Java RMI nicht möglich)

Die Objekterzeugung auf Serverseite beginnt mit der Beschreibung der Objektschnittstelle in IDL. Anschließend wird das Server-Objekt in der Implementierungssprache programmiert und Stub/Skeleton erzeugt.
Ein Objekt wird am **ORB** registriert (bzw. am Portable-Object-Adapter (POA)), welcher eine Interoperable Object Reference (IOR) erzeugt. Der Client bindet sich an das Server-Objekt indem er sich eine **Referenz auf das Server-Objekt** besorgt (Rückgabewert eines Methodenaufrufs, Ergebnis einer Namensdienstanfrage oder durch statische Referenz (Benutzer “kennt” Objektreferenz)). Der **Client-Stub** wird erzeugt und Methodenaufrufe am Server-Objekt darüber getätigt. Der ORB vermittelt Methodenaufrufe von einem zum anderen Objekt.
Der POA generiert Objektreferenzen, nimmt eingehende Methodenaufrufanfragen entgegen, leitet diese an den entsprechenden Servant weiter, sorgt für Sicherheit (Authentifizierung) und (de)aktiviert Servants. Der POA trennt das Objekt (CORBA-Objekt mit seiner Identität) von der Objektimplementierung, die mehrere CORBA-Objekte realisieren kann. Objektimplementierungen können bei Bedarf dynamisch aktiviert werden.

Was muss eine Fehlertoleranzinfrastruktur leisten?

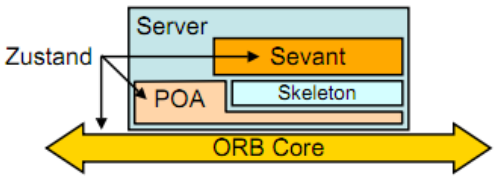
1. **Replikation von Objekten**
2. **Erkennung von Ausfällen**
3. **Logging** von Anfragen und **Zustandssicherung** von Objektzuständen
4. **Zustandstransfer**



5. **Transparente Verschattung von Ausfällen**

Grundlagen:

1. **Redundanz**: **Objekte** bilden die Einheit der **Replikation**. Mit **starker Konsistenz** haben alle Replikate den gleichen Zustand, was eine einfache Anwendungsentwicklung ermöglicht, aber hohe Anforderungen an die Mechanismen der Infrastruktur stellt, welche durch die Middleware bereitgestellt wird. Die Menge aller Replikate eines Objekts bilden eine Objektgruppe, welche über eine **Object Group Reference** (OGR) verfügt. Ziel ist **Replikations-**, sowie **Fehlertransparenz**. CORBA-Objekte werden durch ihren Ausführungsort identifiziert (schwache Form von Identität), **FT CORBA-Objekte benötigen eindeutige und ortsunabhängige Identifikation**.
2. **Zustand**: Der Zustand bildet die Menge an Informationen, die zur Erhaltung von konsistenten Replikaten nötig ist. Er ergibt sich durch das replizierte Objekt und die Infrastruktur (POA und ORB).



3. **Determinismus**: Replikation von Objekten erfordert deterministisches Verhalten: Aufrufe in gleicher Reihenfolge, ausgehend von identischen Ausgangs- werden identische Folgezustände erreicht, Objekt entspricht somit einem **Zustandsautomat**. Probleme ergeben sich durch “objektexterne” Informationen wie Zeit, Zufallszahlen, etc. Außerdem muss eine Koordination stattfinden, wenn parallel mehrere Threads ein Objekt verändern (um Determinismus zu bewahren).
4. **Art der Replikation**: aktiv vs. passiv, “warm passive” vs. “cold passive”

Was ist der Vorteil/Mehrwert bei der Benutzung von FT CORBA?

FT CORBA ist eine **Middleware für Fernaufrufe mit Unterstützung von Fehlertoleranz**. CORBA spezifiziert keine Mechanismen für Redundanz, und zuverlässige Verbindungen basierend auf TCP/IP ermöglichen nur eine beschränkte Erkennung von Ausfällen. Ziel von FT CORBA ist die Unterstützung von Fehlertoleranz gegenüber **fail stop** (Ausfall erkennbar nach endlicher Zeit) Fehlern. Durch Middleware-Mechanismen und starke Konsistenz sind replizierte Objekte weitgehend transparent für Anwendungen. Die Objektimplementierungen müssen allerdings angepasst werden → Zustandstransfer, Determinismus.
Nachteile: Nichtdeterminismus ist Sache des Anwendungsentwicklers und die stark konsistente Replikation erfordert aufwendige Gruppenkommunikation. Außerdem werden malicious faults nicht toleriert. Desweiteren müssen alle Replikate eines replizierten Objekts die gleiche Infrastruktur nutzen.

⇒ Wie wird Fehlertoleranz erreicht?

Fault-Detector erkennt Fehler und erzeugt Fehlerberichte.
Fault-Notifier sammelt und verarbeitet Fehlerbereichte von Fault-Detectors und Fault-Analyzers und bildet eine Art Datenbank für Fehlerberichte.
Fault-Analyzers sind spezifisch für jede Anwendung, verarbeiten Fehlerberichte und fassen abhängige Fehler zusammen.

+ siehe Grundlagen Fehlertoleranzinfrastruktur.
Replikation, Zustandssicherung, Zustandstransfer zur (Re-)initialisierung von Objekten.

⇒ Was wird repliziert?

Objekte bilden die Einheit der Replikation.

⇒ Was für Arten von Replikation gibt es?

Zustandslose Dienstanbieter können problemlos repliziert werden. Sobald ein Zustand ins Spiel kommt, kann man zwischen passiver und aktiver Replikation unterscheiden:

1. **aktiv**: alle Replikate führen immer alle Anfragen aus → sehr **geringe Verzögerung** bei Ausfällen. Duplizierte Ergebnisse (Rückgabewerte) müssen auf Seite des Aufrufers oder des Servers eliminiert werden. Auch duplizierte Aufrufe (wenn repliziertes Objekt andere Objekte aufruft) sollten unterdrückt werden. Konsistenzansprüche erfordern **Gruppenkommunikation** und **total geordnete** Zustellung von Nachrichten.
2. **passiv**: ein Master und viele Slaves. Ausfall des Masters erfordert Neuwahl. Nach jedem Zustandstransfer haben alle Replikate den gleichen Zustand (Konsistenz)
    a. **warm passive**: Aktueller Zustand des primären Replikats wird periodisch an alle Replikate übertragen und Aufrufe werden zusätzlich protokolliert. Dadurch ist eine schnellere Recovery als bei cold passive möglich.
    b. **cold passive**: Wiederherstellung wird durch Sicherungspunkt und Protokollierung von Aufrufen erreicht. Ausfälle können nur langsam kompensiert werden. Sicherungspunkte werden nicht direkt in Replikate eingespielt.

Was muss eine Anwendung bereitstellen, um replizierbar zu sein?

Eine Schnittstelle zum Erzeugen von Sicherungspunkten (get_state, set_state) und/oder partiellen Sicherungspunkten (get_update, set_update).

Wie haben wir für Fehlertoleranz in der Übung gesorgt (2 Wege)?

Auf Kommunikationsebene durch Implementierung verschiedener **Fernaufrufsemantiken** (maybe, at-least-once, at-most-once, last-of-many) und durch **Replikate**.

⇒ Was ist dafür nötig, dass Replikate konsistent sind?

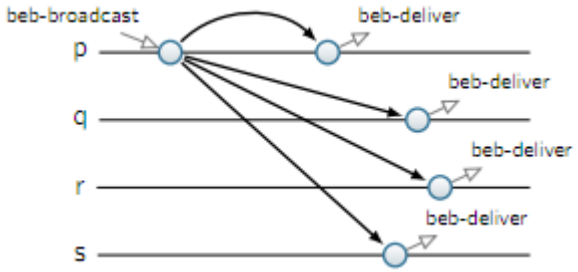
Die Nachrichten müssen **total geordnet** sein (FIFO reicht nicht aus). Anwendungen müssen deterministisch sein (keine Zufallszahlen, Zeit, …).

Gruppenkommunikation?

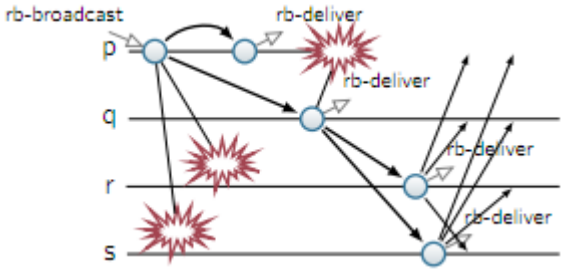
Multicast-Kommunikation bildet einen Basismechanismus zur Realisierung von komplexen verteilten Algorithmen (z.B. zur Bereitstellung fehlertoleranter verteilter Dienste wie im Rahmen von FT CORBA). Nachrichten werden hierbei innerhalb einer Gruppe von Prozessen ausgetauscht (→ Gruppenkommunikation). Es können verschiedene Zustellungsgarantien bezüglich **Zuverlässigkeit** und **Ordnung** (Nachrichtenreihenfolge) zugesichert werden:

1. Zuverlässigkeit

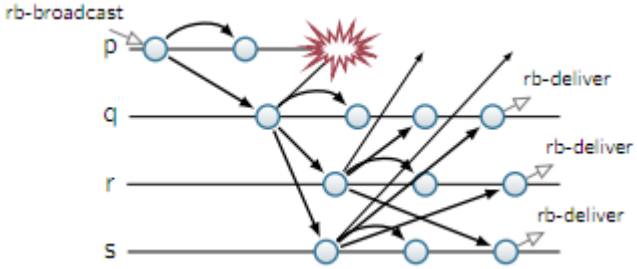
a. **best-effort Multicast**: schwächste Form.



b. **zuverlässiger Multicast**:



c. **uniformer zuverlässiger Multicast**:



2. Ordnung
    a. **ungeordnet**
    b. **FIFO-Ordnung**: Nachrichten, die von einem Knoten in einer bestimmten Reihenfolge erzeugt wurden, werden von anderen Knoten in genau dieser Reihenfolge empfangen und bearbeitet.
    c. **kausale Ordnung**
    d. **totale Ordnung**: FIFO++

Grundlegende Dienste zur Gruppenkommunikation sind ein Membership-Service, Total-Ordering-Multicast und Zustandstransfer-Mechanismen.
Virtual Synchrony: gemeinsame Sicht auf Gesamtsystem (View), Abfolge von Views als gemeinsame logische Zeitbasis.

(A)Synchronität im Zusammenhang mit Verteilten Systemen?

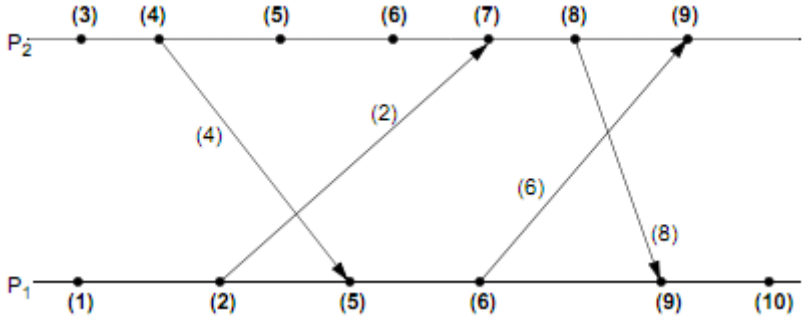
Bei der Ausführung von **verteilten Aktivitäten** bedeutet synchron “gleichzeitig”, also dass Aktivitäten mit Synchronisierung ausgeführt werden, was z.B. durch **gemeinsame Uhren** oder andere Synchronisationsmechanismen gesteuert werden kann. Asynchron bedeutet, dass solch eine Synchronisation nicht vorhanden ist und somit eine Zeitunabhängigkeit gegeben ist.
Bei **entfernten Methodenaufrufen** bedeutet synchron, dass der Aufrufer blockierend auf das Ergebnis wartet, asynchron, dass er nicht blockiert und sofort weiterrechnen kann.

Warum Zeitsynchronisation?

Die Notwendigkeit von Uhrensynchronisation ergibt sich aus der Tatsache, dass Zeit als Mittel zur Reihenfolgebestimmung verwendet wird. Jedoch ist in VS eine gemeinsame Uhr für alle Knoten kaum realisierbar und sogar “zentrale Referenzuhren” (GPS, …) liefern nur eine technisch beschränkte Genauigkeit.

Logische Uhren nach Lamport?

Idee: Aussagen zur Reihenfolge unterschiedlicher Ereignisse werden nur benötigt, wenn eine Interaktion zwischen einzelnen Komponenten des VS erfolgt ist. Eine Aktion b, die logisch durch die Interaktion bedingt “nach” einer Aktion a stattfindet, soll stets den größeren Zeitstempel tragen. Die Synchronisation erfolgt direkt bei der Kommunikation.
Jeder Prozess verfügt über einen Zähler (logische Uhr), der nach bestimmten Regeln erhöht wird.



Eigenschaften:

1. **Potentielle** kausale Abhängigkeit:
   $E_1 \rightarrow E_2 \Rightarrow t(E_1) < t(E_2)$ **gilt**, aber Umkehrung $t(E_1) < t(E_2) \Rightarrow E_1 \rightarrow E_2$ **gilt nicht**!
2. Zeitstempel erzeugen eine **partielle Ordnung** auf der Menge aller Ereignisse, d.h. es gibt Ereignisse, die “gleichzeitig” auftreten und somit nicht geordnet werden können.

Vektoruhren?

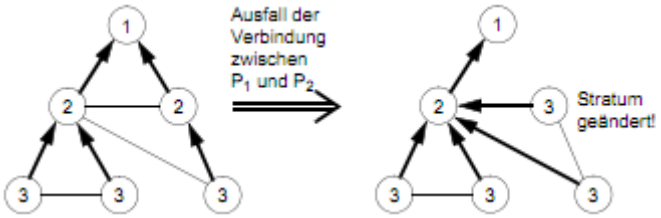
Vektoruhren ergänzen logische Uhren zu **vollständiger Ordnung**. Jeder Knoten besitzt nun eine lokale Uhr, die aus einem Vektor der Länge #Knoten besteht.
$E_1 \rightarrow E_2 \Rightarrow t(E_1) < t(E_2)$ **und** $t(E_1) < t(E_2) \Rightarrow E_1 \rightarrow E_2$ **gelten!**
Vorteil: exakte Aussage zum kausalen Zusammenhang von Ereignissen ist mit Hilfe dieses Zeitstempels möglich.
Nachteil: es entsteht ein hoher Kommunikationsaufwand (für große #Knoten).

Wie funktioniert der Konvergenzalgorithmus CNV?

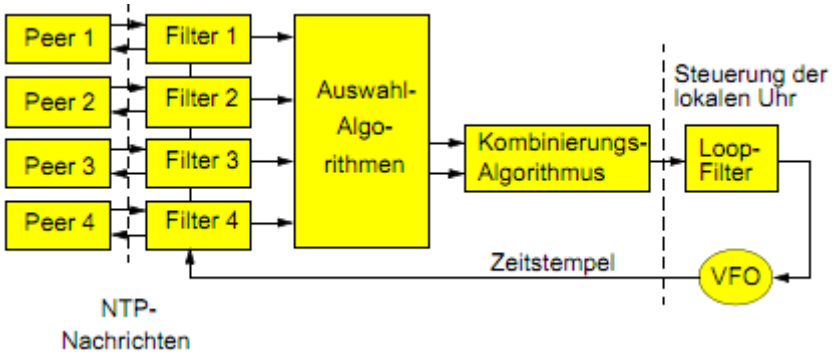
Aufgabe des Algorithmus ist die Vermeidung einer Akkumulation einer immer größer werdenden Abweichung zwischen mehreren Uhren. Dazu liest jeder Prozess die Uhr jedes anderen Prozesses, stellt seine eigene Uhr auf den Mittelwert, wobei für Uhren, die mehr als δ von der eigenen abweichen, der Wert der eigenen Uhr genommen wird. Konvergenz gegen gemeinsame Uhrzeit und Tolerierung von fehlerhaften Uhren werden erfüllt, wenn **weniger als ein Drittel der Uhren fehlerhaft** ist.

Was ist NTP und wie funktioniert es?

Das Network Time Protocol (NTP) bietet die Möglichkeit, lokale Uhren mit für viele Zwecke ausreichender Genauigkeit an die offizielle Zeit zu synchronisieren und verwendet dabei ein **hierarchisches Synchronisationsnetz** und ist **selbstorganisierend** und **fehlertolerant**. Primärer Server (**Stratum** 1) ist an amtliche Zeitstandards gebunden. Stratum kann sich dynamisch ändern:



Zuverlässigkeit ist durch mehrere redundante Server und Netzpfade gegeben. Die lokalen Uhren werden in **Zeit und Frequenz** durch einen adaptiven Algorithmus angepasst.
Die Architektur von NTP:

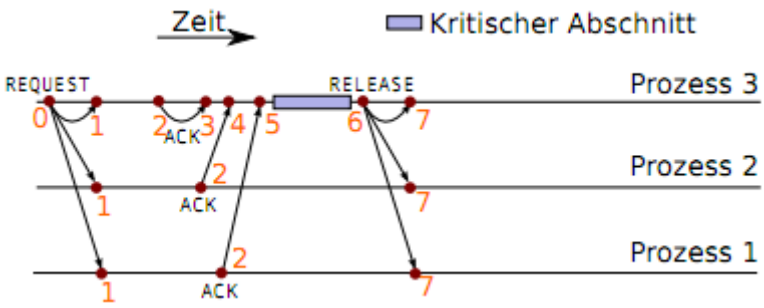


Es gibt mehrere Referenzserver (**Peers**) für Redundanz und zur Fehlerstreuung. Die **Peer-Filter** wählen pro Referenzserver den jeweils besten Wert aus den 8 letzten Offset-Messwerten. Die **Auswahlalgorithmen** versuchen zunächst richtig gehende Uhren zu erkennen und falsch gehende Uhren herauszufiltern und wählen dann möglichst genaue Referenzuhren aus. Der **Kombinationsalgorithmus** berechnet einen gewichteten Mittelwert der **Offset-Werte**. **Loop-Filter** und **VFO** bilden zusammen die geregelte lokale Uhr.

Lamport Lock?

Idee: Ausnutzen der totalen Ordnung über Zeitstempel von logischer Uhr bezüglich Lock-Anfragen. Vorausgesetzt wird das FIFO-Protokoll und ein zuverlässiger Nachrichtenkanal. Toleriert werden allerdings ohne weitere Maßnahmen keine Ausfälle.
Der Ablauf ist wie folgt:

1. REQUEST via Broadcast an alle Prozesse versenden
2. Warten, bis eigene Anfrage vorne in der Warteschlange steht und kein anderer Prozess sich vor dem eigenen Eintrag einreihen kann
3. Kritischen Abschnitt ausführen
4. Broadcast der RELEASE-Nachricht zum Freigeben des Locks


Zeit
Kritischer Abschnitt
REQUEST
RELEASE
Prozess 3
Prozess 2
Prozess 1
0
1
2
ACK
3
4
5
6
7
1
2
ACK
7
1
2
ACK
7