Überblick

6 Fernaufrufe

6.1 Grundlagen

6.2 Konventioneller Aufruf vs. Fernaufruf

6.3 Marshalling und Unmarshalling

6.4 Transparenz

6.5 RPC-Semantiken

6.6 Behandlung von verwaisten Aufrufen

Aktions- vs. datenorientierte Kommunikation

- Bei klassischer Interprozesskommunikation steht der Austausch von Daten (Nachrichten, Datenströme) im Vordergrund
- Alternative: Auslösen einer *Aktivität* bei dem anderen Prozess: **aktionsorientierte Kommunikation**

- Grundschema der Interaktion zwischen Auftraggeber (AG) und Auftragnehmer (AN):

1. AG sendet Anforderung zur Diensterbringung, die ein AN empfängt
2. währenddessen wartet der AG auf eine Rückmeldung
3. die Rückmeldung sendet der AN nach erfolgter Diensterbringung
4. mit Zustellung der Rückmeldung kann der AG weiterarbeiten

Eine aktionsorientierte Kommunikation erfordert damit zwei Vorgänge **datenorientierter Kommunikation**: *request* und *reply*

Kommunikationsmittel „Prozedur“

- Prozeduraufruf vs. aktionsorientierte Kommunikation
  - Gemeinsamkeit
    - der AG entspricht der Routine, die den Prozeduraufruf tätigt *Client*
    - der AN entspricht der aufgerufenen Prozedur *Server*
    - request/reply entsprechen Aufruf/Rücksprung
  - Unterschied: beim Prozeduraufruf ist der die Prozedur aufrufende *Prozess* mit dem die Prozedur ausführenden identisch

- *aktionsorientierte Kommunikation*: Aufruf einer entfernten Prozedur
  - „entfernt“ heißt
    - anderer Faden
    - anderer Adressraum
    - anderer Prozess und/oder
    - anderer Rechner
  - daher auch bezeichnet als **Prozedurfernaufruf**

Synchrone Kommunikation

- remote-invocation send-Semantik

  unterstützt Prozedurfernaufrufe, die *ein Ergebnis liefern*

  - **send** gibt Anforderung ab, blockiert den AG und deblockiert ggf. den AN
  - **receive** vom AN nimmt die Anforderung entgegen
  - **reply** übermittelt die Rückmeldung des AN und deblockiert den AG

- synchronization send-Semantik

  unterstützt alle *sonstigen* Prozedurfernaufrufe

  - **send** gibt Anforderung ab, blockiert den AG und deblockiert ggf. den AN
  - **receive** vom AN nimmt Anforderung entgegen und deblockiert den AG

Asynchrone Variante — *Promises*

- asynchroner Prozedurfernaufruf kehrt *sofort* zurück und liefert als Ergebnis ein **Promise-Objekt** (→ „Versprechen“ auf das Ergebnis)
  - Verfügbarkeitszeitpunkt des Ergebnisses nicht festgelegt
  - Promise-Objekt dient der Aufnahme des ihm zugeordneten Resultats
  - Promise-Zustand kann (mittels `ready()`) abgefragt werden:

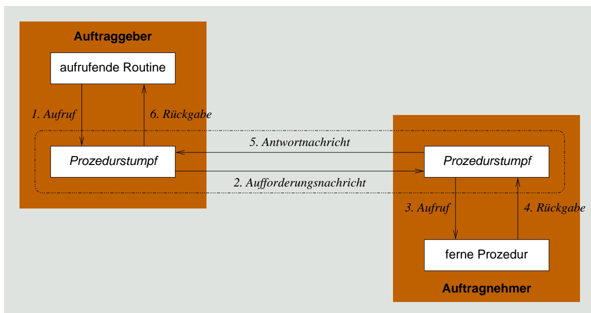
    blockiert ⇒ der Aufruf läuft, ein Resultat liegt noch nicht vor
    - ein `claim()`-Aufruf würde blockieren bis das Ergebnis vorliegt

    bereit ⇒ der Aufruf ist beendet, das Resultat liegt vor
    - in `claim()` ggf. blockierte Aufrufer werden deblockiert
    - der `claim()`-Aufruf liefert das Ergebnis zurürck
  - mit *no-wait send* wird die Promise-Implementierung ideal unterstützt
  - sprachl. Unterstützung im objektorientierten Sinn erleichtert Anwendung

- Alternative: asynchrone Aufrufe mit *wait by necessity*
  - keine expliziten Promise-Objekte
  - stattdessen: implizites Blockieren bei Benutzung des Ergebnisses

Prozeduraufruf ⇒ Nachrichtenaustausch

- Client Stub: Prozedurstumpf auf Seite des Auftraggebers
  - abstrahiert von der Örtlichkeit der entfernten, aufgerufenen Prozedur
  - setzt den Prozeduraufruf in einen Nachrichtenaustausch um
  - verpackt tatsächliche Aufrufparameter und entpackt Rückgabewerte
- Server Stub: Prozedurstumpf auf Seite des Auftragnehmers
  - abstrahiert von der Örtlichkeit der entfernten, aufrufenden Prozedur
  - setzt die Prozedurrückkehr in einen Nachrichtenaustausch um
  - entpackt die Aufrufparameter und verpackt Rückgabewerte



Konventioneller Aufruf vs. Fernaufruf

- Ziel eines Fernaufrufmechanismus ist es, die bekannte Semantik konventioneller Prozeduraufrufe aufrecht zu erhalten, obwohl sich die Ausführungsumgebung radikal anders gestaltet
  - aufrufende und aufgerufene Seite sind örtlich voneinander getrennt
    - Code und Daten teilen nicht denselben (physikalischen) Arbeitsspeicher
  - beide Seiten arbeiten weitestgehend autonom
    - sie werden von verschiedenen Prozessen/Prozessoren ausgeführt
  - beide Seiten können unabhängig voneinander ausfallen

⇒ *Die Semantik konventioneller, lokaler Prozeduraufrufe ist nur* **teilweise** *erreichbar*

Semantikaspekte

- Parameterarten sind zu unterscheiden, um Aufwand zu minimieren
  - Eingabe- vs. Ausgabe- vs. Ein-/Ausgabeparameter
- Parameterübergabe ist ggf. explizit zu spezifizieren
  - *call-by-reference* vs. *call-by-value/result*
- Gültigkeits-/Sichtbarkeitsbereiche sind ggf. massiv eingeschränkt
  - Variablen des entfernten umfassenden *Scopes* sind meist nicht gültig/sichtbar
- Speicheradressen sind im Regelfall nicht systemweit eindeutig
  - Zeiger in Nachrichten zu versenden, ist (nahezu) sinnlos

Semantikaspekte (2)

- die Auslegung der (formalen) Parameter bestimmt unter anderem den IPC- Mehraufwand:

  Eingabeparameter sind **nur** Bestandteil der *Anforderungsnachricht*

  Ausgabeparameter sind **nur** Bestandteil der *Antwortnachricht*

  Ein-/Ausgabeparameter sind Bestandteil **beider** Nachrichten

- nicht immer liefern Programmiersprachen passende Auslegungshinweise, z.B.

  C/C++ { *Zeiger* `char*`, `struct Foo*`; *Referenz* `struct Foo&`; *Feld* `char foo[4]` } welche Parameterart?

- die Art eines jeden Parameters müsste in der Schnittstelle spezifiziert sein

- *call-by-value/result* sind „geradlinig“ und einfach zu behandeln
  - die tatsächlichen Parameter werden in die/aus den jeweiligen Nachrichten kopiert
- *call-by-reference* wird je nach Parameterart abgebildet wie folgt

Eingabeparameter	→	*call-by-value*	dereferenzieren
Ausgabeparameter	→	*call-by-result*	dereferenzieren
Ein-/Ausgabeparameter	→	*call-by-value-result*	dereferenzieren
unspezifiziert	→	*call-by-value*	Speicheradresse

- *call-by-name* ggf. nur auf Basis von *function shipping*
  - eine Funktion wird mitgeliefert, die den tatsächlichen Parameter berechnet

Semantikaspekte (4)

- jeder Name (einer Variablen) ist mit einem Platzhalter assoziiert
  - die Variable belegt einen Speicherplatz an einer bestimmten Adresse
  - den Namen/Platzhaltern sind Gültigkeitsbereiche („Blöcke") zugeordnet
- die Bindung eines Namens an seinen Block (*Scope*) ist statisch oder dynamisch
  - statischer Scope
    - ist bereits zur *Übersetzungszeit* bekannt
    - ändert sich nur bei Quelltextänderungen am Programm
  - dynamischer Scope
    - ist erst zur *Laufzeit* bekannt
    - ändert sich mit Eintritt in/Verlassen von Prozeduren bzw. Funktionen
- der „umfassende Block" ist für eine entfernte Prozedur nicht oder nur bedingt zugänglich

Verteilter gemeinsamer Speicher

- Multiprozessor
  - eng gekoppelte Prozessoren
  - gemeinsamer und kohärenter Speicher
- Multicomputer und Verteilte Systeme
  - lose gekoppelte Prozessoren
  - nur privater Speicher verfügbar
- verteilter gemeinsamer Speicher
  - Illusion eines gemeinsamen Speichers

Verteilter gemeinsamer Speicher (2)

- Mögliche Realisierung
  - seitenbasierte Speicherverwaltung
  - Seitenadressierung, unterstützt durch MMU
  - entspricht logischem/virtuellem Speicher
  - ein virtueller Adressraum über alle beteiligten Rechner
  - eine Seite residiert zu einem Zeitpunkt immer nur auf genau einem der beteiligten Rechner
- Lokaler Zugriff
  - Seite ist in der lokalen Seitentabelle eingetragen, Präsenzbit ist gesetzt $\Rightarrow$ Zugriff ist möglich
  - Zugriff erfolgt auf lokalen Speicher

Verteilter gemeinsamer Speicher (3)

- Entfernter Zugriff
  - Seiteneintrag fehlt in lokaler Seitentabelle, Präsenzbit ist nicht gesetzt
  - lokaler Zugriff löste einen Seitenfehler (page fault) aus
    $\Rightarrow$ Unterbrechungsbehandlung durch das Betriebssystem
  - Betriebssystem holt die Seite von dem entfernten Rechner
    (hierbei wird die Seite dort ausgetragen!)
  - Seite wird in lokale Seitentabelle eingetragen, Präsenzbit wird gesetzt
  - Speicherzugriff wird wiederholt

- Kohärenter Speicher
  (Lese-Operation liefert immer den zuletzt geschriebenen Wert)
  - falls Hole- und Weitergabe-Operationen für Seiten atomar
  - falls keine Ausfälle auftreten

Verteilter gemeinsamer Speicher (4)

- Nebenläufige Zugriffe auf eine Seite sind sehr ineffizient
  - Seitenflattern zwischen Rechnern
- False-Sharing
  - zwei Datenobjekte liegen in derselben Seite, werden aber von unterschiedlichen Programmbereichen benutzt
  - die unterschiedlichen Programmbereiche werden parallel auf verschiedenen Rechnern ausgeführt
  - Ergebnis: gegenseitiges „Stehlen“ der Seite bei Datenzugriffen
- Maßnahmen zur Effizienzsteigerung erforderlich
  - ausgefeiltere Kohärenzprotokolle
  - paralleles Lesen erlauben
  - Abschwächung der Speicherkonsistenz-Anforderungen

Verteilter gemeinsamer Speicher (5)

- Fazit
  - Verteilter gemeinsamer Speicher ist nur bei sehr enger Rechnerkopplung sinnvoll
    - spezielle Hardware-Unterstützung
    - kurze Latenzen
    - schnelle Datenübertragung
    - ⇒ Höchstleistungsrechner
  - In lose gekoppelten Systemen ist reine Nachrichtenübertragung vorzuziehen
    - Kommunikation nicht über gemeinsamen Speicher, sondern über Prozeduraufrufe

Nachrichten zusammenstellen / auspacken

- *marshalling*

  to arrange (troops, things, ideas, etc.) in order; array; dispose

  - die einzelnen Datenelemente in einen Nachrichtenpuffer gepackt anordnen:
    1. für die **Serialisierung** verstreut vorliegender Daten sorgen
    2. die vereinbarte **Repräsentation** (Typ) der Daten gewährleisten
  - je nach Herangehensweise sind Referenzen aufzulösen und umzuwandeln
    - wenn die referenzierten Strukturen *call-by-value* zu übertragen sind
  - die so zusammengestellte Nachricht geht per IPC an den Empfangsprozess

- *unmarshalling*
  - die *inverse Funktion* auf Empfangsseite
  - Nachricht entpacken und die ursprünglichen Datenelemente wieder herstellen
  - hierbei ggf. eine andersartige Repräsentation auf der Zielseite berücksichtigen

Serialisierung/Deserialisierung

- geläufig ist, **verzeigerte Datenstrukturen** *call-by-value* (*-result*) zu übergeben
  - zum Marshalling wird eine Beschreibung des logischen Aufbaus strukturierter/dynamischer Daten benötigt (= Typbeschreibung)
  - alle Referenzen werden aufgelöst und die referenzierten Objekte kopiert
  - nach dem Empfang werden die Datenstrukturen wieder rekonstruiert

- zwei Sorten von Zeigern sind dabei zu unterscheiden:
  innere Referenzen: die Verkettungszeiger rekursiver Datenstrukturen
  - sind vergleichsweise unproblematisch: „logische Zeiger“ vergeben

  äußere Referenzen: Zeiger von außen hinein auf einzelne Verbundelemente
  - die relative Position der Verbundelemente kann sich ändern: Relokation

- der Aufwand kann je nach Art/Aufbau der Datenstrukturen beträchtlich sein

Packen/Entpacken strukturierter Daten

```
list.add("barfoo");
```

Prozedurfernaufruf mit zwei Referenzparameter

Bytestrom

6	3	3	'f'	'o'	'o'	12	9	3	'b'	'a'	'r'	0	15	6	'f'	'o'	'o'	'b'	'a'	'r'	...
0			3			6			9			12			15						

Objektorientierung „erschwert“ Automatisierung

- Referenzen vom Typ einer Oberklasse zeigen ggf. auf Instanzen von Unterklassen
  - eine *abstrakte Oberklasse* sagt wenig/nichts aus über die Objektstruktur

    ⇒ in der Fernaufrufschnittstelle ist die konkrete Ausprägung des zu übertragenden Objekts unbekannt
  - erst die spezialisierende Unterklasse enthält die erforderlichen Strukturinformationen

- der *tatsächliche Typ* des Parameters ist erst zur Laufzeit bekannt
  - automatische Generierung des Stubs mit der Marshalling-Funktion ist zur Übersetzungszeit nicht möglich
  - statt dessen „spezialisierte Zusammenstellung“ der Fernaufrufnachrichten
    - erfordert eine vollständige Analyse des (zu verteilenden) Quellprogramms
  - Alternative: Analyse der Parametertypen zum Zeitpunkt des Aufrufs — *Reflection*

Datenrepräsentation

Heterogenität – die einzelnen in Nachrichten übertragenen Elemente können Werte unterschiedlichster (elementarer) Datentypen repräsentieren.

- *Speicherung* wie auch *Darstellung* von Instanzen dieser Typen ist nicht in allen Rechnern identisch:
  - natürliche/ganze Zahlen: vorzeichenbehaftet, {1,2}er-Komplement
  - Fließkommazahlen: Basis, Mantisse, Exponent
  - Zeichensätze: ISO-8859-Familie (ASCII), BCD, EBCDIC, Unicode
  - Speicherreihenfolge: *big endian* vs. *little endian*

$\Longrightarrow$ Daten sind ggf. zu konvertieren, damit kooperierende Prozesse funktionieren!

Konvertierungsoptionen

beidseitig *external data representation* (XDR)

- zu sendende Daten in eine **kanonische Darstellung** umkodieren *und*
- empfangene („kanonische“) Daten in die lokale Darstellung umkodieren
- Problem: nutzloser Mehraufwand im Falle „gleichartiger“ Rechner

sendeseitig „*sender makes it right*“

- zu sendende Daten in die empfangsseitige Darstellung ggf. umkodieren
- Problem: Mehrteilnehmerkommunikation, Weiterleiten von Nachrichten

empfangsseitig „*receiver makes it right*“

- empfangene Daten in die lokale Darstellung ggf. umkodieren → *endian tag*

External Data Representation — XDR

entwickelt von Sun Microsystems, *RFC 1014*, 1987

- sprachbasierter Standard zur Beschreibung und Kodierung von Daten
  - der ISO/OSI **Präsentationsschicht** (*presentation layer*, 6) zugeordnet
  - **implizite Typung**, nicht explizit wie bei ASN.1
    - die Typen der einzelnen Daten in der nachricht ergeben sich implizit aus dem Typ der aufgerufenen Schnittstelle
    - bei ASN.1 werden sie explizit in der Nachricht mitgeschickt
  - Annahme: Bytes bzw. Oktets (d.h. Einheiten von 8 Bits) sind portabel

- Daten werden als „Vielfaches von vier Bytes“ (32 Bits) repräsentiert
  (*Tradeoff* „Vier“ — Groß genug als effiziente Lösung für die meisten Maschinen, mit Ausnahme von 64-Bit Architekturen, und klein genug als vertretbarer Mehraufwand zur Repräsentation der kodierten Daten.)
  - Füllbytes (0 – 3) ergänzen den Datenstrom immer zum Vielfachen von 4
- die Reihenfolge (der „Byte-Sex“) ist *big endian*

XDR „*Considered Harmful*"?

- Virtualisierung:
  Jede Maschine, deren physikalische Darstellung mit der durch XDR vorgegebenen virtuellen übereinstimmt, wird effizienter arbeiten als jene, bei der die physikalische von der virtuellen Darstellungsform stark abweicht:
  - Speicherreihenfolge („Byte-Sex")

	endian		endian		
Sun	*big*	↔	*big*	m68K	„optimal"
IBM 370	*big*	↔	*little*	Z80	
Alpha	*little*	↔	*big*	R10000	
VAX	*little*	↔	*little*	x86	„suboptimal"

  - Verschnitt („interne Fragmentierung")

Sprachintegration von Fernaufrufen

- Fernaufrufsysteme lassen sich in zwei Hauptkategorien einteilen:
  1. in einer Programmiersprache **integriertes Konzept** . . . . Argus
     - internes Wissen über Datentyp- und Laufzeitmodell ist verfügbar
     - der Übersetzer agiert gleichzeitig als *Stub-Generator*
     - umfassende Analysen, Optimierungen und Automatismen werden möglich
  2. von einer Programmiersprache **separiertes Konzept** . . . . CORBA
     - das Schnittstellenverhalten entfernter Prozeduren wird in einer eigenen „*Interface Definition Language*" explizit beschrieben (IDL)
     - fehlendes internes Wissen schränkt Analysen, Optimierungen etc. ein
  3. **teilweise integriertes Konzept** . . . . Java RMI
     - der Compiler kennt Konzepte des Fernaufrufs (Remote-Schnittstellen, etc.) nicht
     - in den Java-Standards sind die Konzepte definiert
     - Unterstützung für RMI integraler Bestandteil der Laufzeitumgebung
- nur das integrierte Konzept definiert eine einheitliche Semantik

Entlastung von Routineaufgaben

- Namensdienst
  - Server registriert Dienstanbieter (Ort, Referenz, Stub)
  - Client fragt ab
- die Stubs sorgen für eine **lose Kopplung** zweier Ausführungsumgebungen:

  client stub:

  1. *Marshalling* und Versenden der Anforderungsnachricht

     ⋮ die Durchführung der angeforderten Operation abwarten
  2. Empfangen und *Unmarshalling* der Antwortnachricht

  server stub:

  1. Empfangen und *Unmarshalling* der Anforderungsnachricht
  2. Durchführen der angeforderten Operation (lokaler Aufruf)
  3. *Marshalling* und Versenden der Antwortnachricht
- **Stub-Generatoren** erzeugen die dafür notwendigen Programmsequenzen

Transparenz von Fernaufrufen

- Verteilungstransparenz ist nur bedingt erreichbar, trotz integriertem Konzept:
  - syntaktische Unterschiede (in der Schnittstelle) werden vermieden
  - Parameterübergabe, *Marshalling* und *Unmarshalling* wird verborgen
  - Nachrichtenpuffer und Fäden werden erzeugt, verwaltet und entsorgt
  - Stubs werden automatisch erzeugt

    — was ist denn sonst noch nötig?

- Fernaufrufe sind fehleranfälliger als konventionelle lokale Prozeduraufrufe

entfernt	ein Netzwerk, ein anderer Rechner und ein anderer Prozess
$\neg$ entfernt	kein Netzwerk, kein anderer Rechner und kein anderer Prozess

- verteilte Systeme unterliegen einem radikal anderem **Fehlermodell**

Zustellungsgarantien

- *request-reply*-Protokolle eröffnen Optionen für Fehlertoleranzmaßnahmen:

  1. Wiederholung der Anforderungsnachricht . . . . *request retry*
     - bis die Antwort eintrifft oder ein Anbieterausfall angenommen wird
  2. Filterung der Anforderungsduplikate . . . . *duplicate suppression*
     - wenn die Anforderung empfangen wurde und noch in Arbeit ist
  3. Wiederholung der Antwortnachricht . . . . *reply retry*
     - bis die nächste Anforderung oder eine Bestätigung eintrifft

- Kombinationen dieser Optionen führen zu verschiedenen Aufrufsemantiken

Fehlertoleranzmaßnahme vs. Aufrufsemantik

Option			**Semantik**	
request retry	*dupl. supr.*	*re-execute* *reply retry*		
Nein	—	—	„kann sein“	*maybe*
Ja	Nein	*re-execute*	„wenigstens einmal“	*at-least-once*
Ja	Ja	*reply retry*	„höchstens einmal“	*at-most-once*

Aufrufsemantiken (1)

- *maybe*: kann sein, der Aufruf wurde ausgeführt oder auch nicht
  - wenn der Aufrufer innerhalb einer vorgegebenen Zeitspanne (*Timeout*) keine Antwort erhält, hat er keine Gewissheit, ob die Operation ausgeführt wurde oder nicht

- *at-least-once*: mindestens einmal, die mehrfache Ausführung ist möglich
  - der Klient erwartet die Antwort innerhalb einer vorgegebenen Zeitspanne
    - der Fernaufruf wird mit einem *Timeout* versehen
    - nach der dadurch definierten Pause wird die Anfrage wiederholt
    - die Anzahl der Wiederholungen ist (üblicherweise) begrenzt
    - nach Erreichen der max. Anzahl wird ein Fehler signalisiert (*Exception*)
  - der Anbieter muss **idempotente Operationen** exportieren
  - wenn der Aufrufer ein Ergebnis erhält, weiß er, dass der Aufruf *mindestens einmal* ausgeführt wurde
  - im Fehlerfall hat der Aufrufer keinerlei Informationen über die Ausführung

Aufrufsemantiken (2)

- *at-most-once*: höchstens einmal, mehrfache Ausführung ist ausgeschlossen
  - vergleichsweise aufwändiges, speicherintensives Verfahren:
    - jedem neuen Aufruf wird eine eindeutige *Aufrufkennung* gegeben
    - Anfragewiederholungen kommen mit derselben Kennung und werden vom Server verworfen
      — Fehlersignalisierung analog zu *at-least-once* möglich
    - Antworten werden vom Server gespeichert und bei Anfragewiederholungen erneut geliefert
    - ein neuer Aufruf ist Anlass, gespeicherte alte Antworten zu entsorgen
  - wenn der Aufrufer ein Ergebnis erhält, weiß er, dass der Aufruf *genau einmal* ($\rightarrow$ *exactly once*) ausgeführt wurde
  - im Fehlerfall weiß der Aufrufer, dass der Aufruf *höchstens einmal* ausgeführt wurde

    (Während *at-least-once* also die Semantik im Erfolgsfall beschreibt, steht *at-most-once* tatsächlich für die Semantik im Fehlerfall!)

Aufrufsemantiken (3)

- *last-of-many*: analog zu *at-least-once*, der Aufrufer akzeptiert aber nur die Antwort zur letzten Anfrage
  - jede Anfrage, auch eine Wiederholte, erhält eine eindeutige Kennung
  - jede Antwort trägt die Kennung der zugehörigen Anfrage
    - der Klient verwirft Antworten mit nicht mehr aktueller Anfragekennung
    - eine Alternative zu *at-least-once*, falls mehrfache Anfragen nicht dieselben Antworten liefern (z. B. Zeitabfrage)

Aufrufsemantiken (4)

- *exactly-once*: genau einmal, entspricht der Semantik lokaler Aufrufe
  - erfordert Transaktionskonzepte mit Wiederanlauf von Komponenten
    - gibt damit (eine gewisse) Garantie bei Systemfehlern bzw. -ausfällen
  - nach Wiederanlauf sind Aussagen zur Operationsdurchführung „unscharf“:

    Imagine, for example, a chocolate factory, in which vats of liquid chocolate are filled by having a computer set a bit in some device register to open a valve. After recovering from a crash, there is no way for the chocolate server to see if the crash happened one microsecond before or after the bit was set. (Tanenbaum, Computer Networks, 1989)

  - wünschenswerte, ideale Semantik, die in „Reinform“ jedoch unerreichbar ist

Idempotente Operationen

- Idempotenz („idem" (*lat.*) dasselbe)
  wenn die wiederholte Ausführung derselben Operation (mit denselben Parameterwerten) durch den Dienstanbieter immer den Effekt einer einmaligen Ausführung besitzt
  - wiederholte Ausführungen sind möglich im Falle ungefilterter Duplikate
    - kritisch sind z.B. Schreiboperationen am Dateiende (Anhängen)
    - ebenso Operationen, die eine Folge (Liste) um Elemente erweitern (Im Gegensatz zu Operationen, die auf Mengen arbeiten.)
  - ein **zustandsfreier Dienstanbieter** (*stateless server*) ist erforderlich (keine Informationen über den aktuellen Zustand der Interaktion mit dem Klienten)
    - bekannter Vertreter ist das Sun *Network File System* (NFS)
  - bei zustandsbehafteten Dienstanbietern ist Duplikatelimination notwendig

Fehlersemantiken

Aufrufsemantik	**Fehlerart**							
	fehlerfreier Ablauf		Verlust von Nachrichten		zusätzlicher Ausfall von Anbieter		Klient	
	Aus.	Erg.	Aus.	Erg.	Aus.	Erg.	Aus.	Erg.
maybe	1	1	0/1	0/1	0/1	0/1	0/1	0/1
at-least-once	1	1	≥ 1	≥ 1	≥ 0	≥ 0	≥ 0	0
at-most-once	1	1	1	1	0/1	0/1	0/1	0
exactly-once	1	1	1	1	1	1	1	1

Aus.= Ausführung, Erg.= Ergebnis; die jeweilige Anzahl ist angegeben

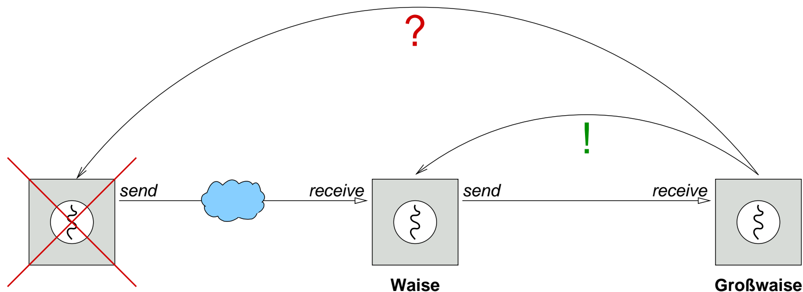
Verwaister Aufruf (1)

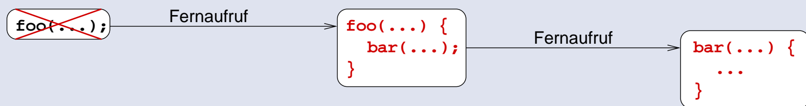
- *Orphan*
  eine Ausführung auf Serverseite, deren Klient nicht mehr existiert (z. B. weil der Klient den Aufruf wegen eines *Timeout* abgebrochen hat und nicht mehr am Ergebnis interessiert ist oder weil der Klient abgestürzt ist)

- Maßnahmen, um unnötige Arbeit des Dienstanbieters zu vermeiden:
  1. zusätzliche Zeitüberwachung des Klienten durch den Dienstanbieter
     - die *Timeout*-Wahl ist klientenabhängig ($T_S \gg T_C$)
  2. pro Klienten einen Ausfallzähler beim Dienstanbieter verwalten
     - Abbruch aller alten Aufrufe nach Ausfall und Wiederanlauf des Klienten
  3. zusätzliche direkte Statusanfragen an den Klienten senden
     - schnelle Waisenerkennung, sofern der Ausfall diagnostizierbar ist

- besondere Schwierigkeiten bereiten „Fernaufrufketten“: **Transitivität**

Verwaister Aufruf (2)





Waisenbehandlung (1)

- *extermination* (Vertilgung, Ausrottung, Wegschaffung)
  - nach Wiederanlauf wird (beim Klienten) auf ausstehende Fernaufrufe geprüft
    - den betreffenden Anbietern gehen sodann Abbruchanforderungen zu
    - rekursives Vorgehen, wegen der Möglichkeit von „Großwaisen"
  - Client-Stubs müssen „Buch" führen über alle Fernaufrufe
    - *Log* anlegen vor der Anforderung und zerstören nach Empfang der Antwort

- *expiration* (Verscheiden, Ablauf)
  - der Anbieter gibt dem (Fern-) Aufruf ein *Zeitquantum* zur Ausführung
    - mit Ablauf erbittet der Anbieter ein weiteres Zeitquantum vom Klienten
    - im Fehlerfall bricht der Anbieter die Ausführung des Aufrufs ab bzw. terminiert
  - nach Wiederanlauf wird der erste Fernaufruf um ein Zeitquantum verzögert
  - *Leasing*: zeitlich begrenzt gültige Referenzen - erstmalig realisiert in *Jini*

Waisenbehandlung (2)

- *reincarnation* (Wiederverkörperung, Wiedergeburt)
  - bei Wiederanlauf eines Klienten wird eine neue *Epoche* eröffnet (Realisierung durch Epochenzähler)
    - der Beginn einer neuen Epoche wird allen Maschinen mitgeteilt (*broadcast*)
    - auf den Maschinen werden daraufhin alle Anbieter (Prozesse) terminiert
  - jede Anforderungs- und Antwortnachricht enthält eine Epochenkennung
    - damit können unerwartete Antworten von Waisen herausgefiltert werden

- *gentle reincarnation* (sanfte Wiedergeburt)
  - wenn ein Epochen-Broadcast empfangen wird, überprüft ein Server, ob er gerade Fernaufrufe bearbeitet
  - ist dies der Fall, versucht er die zugehörigen Klienten zu lokalisieren
  - ist ein Klient nicht lokalisierbar, werden die zugehörigen Ausführungen abgebrochen

Ausnahmebehandlung — *Exception Handling*

- volle Verteilungstransparenz erreichen zu können, ist reine Illusion
  - auch *exactly-once* kann nur die erkennbaren Fehler maskieren
  - in den anderen Fällen ist die Rückkehr vom Aufruf keinesfalls garantiert

    (Es ist der Fernaufrufmechanismus selbst, der einen Klienten z.B. bleibend verklemmen kann. Sicherlich kann auch die Implementierung des per Fernaufruf in Anspruch genommenen Dienstes Verklemmungsursache sein, was aber hier nicht zur Debatte steht.)
- die Stubs (auf beiden Seiten) sollten Ausnahmesituationen anzeigen
  - Ausnahmen der Anbieterseite werden als Rückruf zum Klienten propagiert
  - Ausnahmen der Klientenseite werden konventionellerweise „hochgereicht"
- die Anwendungen sollten Maßnahmen zur Fehlertoleranz beinhalten

Zusammenfassung

- Fernaufrufe sind konventionellen Prozeduraufrufen nur ähnlich, nicht gleich:
  - Parameter{arten, Übergabe}, Gültigkeitsbereiche und Speicheradressen
  - Fehlermodell, Zustellungsgarantien, Aufrufsemantiken, verwaiste Aufrufe
  - die Latenz entfernter Aufrufe ist um Größenordnungen höher: *Promise*
- Verteilungstransparenz kann nicht wirklich (durchgängig) erzielt werden
  - *exactly-once* ist nicht (bzw. nur mit Abstrichen) zu verwirklichen
  - Ausnahmen sind zu behandeln, die nur in verteilten Systemen auftreten
  - Fernaufrufe machen fehlertolerante Anwendungssoftware keinesfalls obsolet
- der Unterschied zu konventionellen Aufrufen ist (doch) explizit zu machen

Referenzen

[1] ASN.1 Information site.
http://asn1.elibel.tm.fr, 2002.

[2] B. Liskov.
Distributed programming in argus.
Commun. ACM, 31(3):300–312, 1988.

[3] B. H. Liskov.
Primitives for Distributed Computing.
In *Proceedings of the Seventh ACM Symposium on Operating System Principles (SOSP)*, volume 13 of *SIGOPS Operating Systems Review*, pages 33–42, Pacific Grove, California, USA, Dec. 1979. ACM.

[4] B. H. Liskov and L. Shrira.
Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems.
In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23 of *SIGPLAN Notices*, pages 260–267, Atlanta, Georgia, USA, June 1988. ACM.

[5] B. J. Nelson.
Remote Procedure Call.
Technical Report CMU–81–119, Carnegie-Mellon University, 1982.

[6] R. Srinivasan.
XDR: External Data Representation Standard.
http://www.faqs.org/rfcs/rfc1832.html, 1995.

[7] P. Wegner.
Classification in Object-Oriented Systems.
ACM, SIGPLAN Notices, 21(10):173–182, 1986.