Verteilte Systeme

Jürgen Kleinöder

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.cs.fau.de

Sommersemester 2013

http://www4.cs.fau.de/Lehre/SS13/V_VS

Überblick

Fernaufrufe – Überblick

- IPC-Semantiken
- IPC und Fernaufrufe
- Prozeduraufruf und aktionsorientierte Kommunikation
- Semantikaspekte
  - Parameterarten, Parameterübergabe, Gültigkeitsbereiche, Speicheradressen
- Nachrichten zusammenstellen/auseinandernehmen
- Zustellungsgarantien, Aufrufsemantiken, Fehlermodell, Waisen

IPC-Semantiken

Grundkonzepte für die Kommunikation zwischen Prozessen durch Nachrichtenaustauasch

- **no-wait send** der Sendeprozess wartet, bis die Nachricht im Transportsystem zum Absenden bereitgestellt worden ist
  - *Pufferung* oder *Signalisierung* (dass der übergebene Puffer wieder frei ist)
- **synchronization send** der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess angenommen worden ist
  - *Rendezvous* zwischen Sende- und Empfangsprozess
- **remote-invocation send** der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess verarbeitet und beantwortet worden ist
  - *Fernaufruf* einer vom Empfangsprozess auszuführenden Funktion

IPC und Fernaufrufe

- IPC bildet die Grundlage für den Austausch von Nachrichten zwischen Prozessen
- die Nutzung der TCP/IP-Protokolle bietet zwei Alternativen
  - Verpacken von Nachrichten in UDP-Paketen
    - Ein Paket pro Nachricht
    - Nachrichtenzustellung ist znächst genauso unzuverlässig wie UDP
  - Versenden von Nachrichten über TCP-Verbindungen
    - Nachrichten müssen im Datenstrom der Verbindung kodiert werden (Anfang und Ende kenntlich machen)
    - Nachrichtenübertragung ist zuverlässig – *so lange die Verbindung steht!*

IPC und Fernaufrufe (2)

- IPC bildet die Basis, in der „darüberliegenden“ Ebene steht die *Bedeutung* der Nachrichten im Mittelpunkt
  - die Bedeutung kann sich *implizit* durch den Verarbeitungsalgorithmus (das Programm) ergeben oder
  - die Prozesse machen sie sich gegenseitig *explizit* über „Anweisungen“ bekannt
- die Nachrichten enthalten (problemspezifische) Daten und/oder Code:
  function shipping der Empfangsprozess interpretiert Programme
  - mobiler Code (Java Bytecode, PostScript) ggf. mit Daten unterfüttert

  data shipping der Empfangsprozess interpretiert Daten
- im „Normalfall“ bewirken Nachrichten die Ausführung entfernter Routinen
  - die aufzurufenden Prozeduren/Funktionen sind implizit oder explizit kodiert

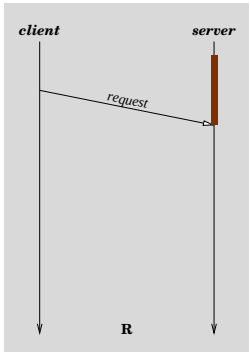
IPC-Protokolle für Fernaufrufe

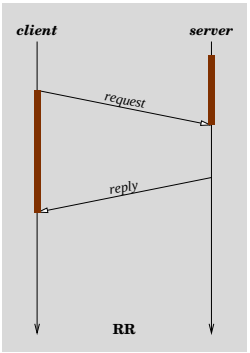
request (**R**) kann genutzt werden, wenn die entfernte Prozedur/Funktion keinen Rückgabewert liefert und der Sendeprozess keine Bestätigung für die erfolgte Ausführung benötigt . . . . . 1 Nachricht

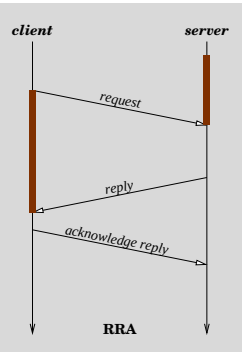
request-reply (**RR**) ist das geläufige Verfahren, da die Antwortnachricht implizit die Anforderungsnachricht bestätigt und dadurch explizite Bestätigungen entfallen . . . . . 2 Nachrichten

request-reply-acknowledge reply (**RRA**) gestattet es, die zum Zwecke der *Fehlermaskierung* (beim Server) gespeicherten Antwortnachrichten zu verwerfen, wenn (vom Client) keine weitere Anforderungsnachricht gesendet wird . . . . . 3 Nachrichten

IPC-Protokolle für Fernaufrufe (2)







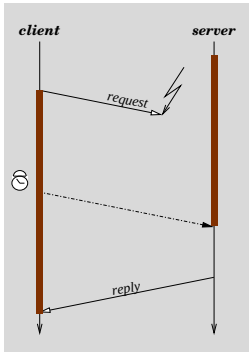
Zustellungsfehler

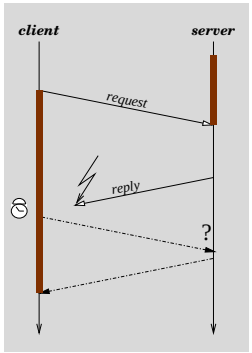
- Nachrichtenaustausch unterliegt bestimmten (typischen) *Fehlerannahmen*:
  1. Nachrichten können verloren gehen
     - beim Sender, beim Empfänger oder im Netz
  2. es können Netzwerk-Partitionierungen auftreten
     - ein oder mehrere Rechner (Knoten) werden „abgetrennt“
  3. Prozesse können scheitern (d.h. „abstürzen“)
     - Prozess-, Rechner- oder Netzwerkausfälle sind nicht unterscheidbar
  4. Daten können verfälscht werden
- als Folge sind unterschiedliche (typische) *Protokollvarianten* entstanden

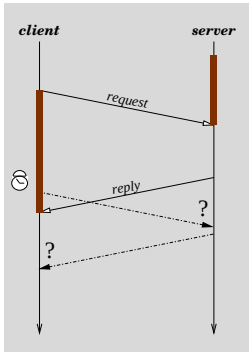
Fehlermaskierung

- Anforderungs- und ggf. auch Antwortnachrichten wiederholen
  - nach einer Pause (*time-out*) werden die Nachrichten erneut versendet
  - die „optimale“ Länge der Pause zu bestimmen ist äußerst schwierig
- eingetroffene Nachrichtenduplikate sind zu erkennen und zu ignorieren
  - ggf. bereits versandte Antwortnachrichten wiederholt versenden
  - auf Client- bzw. Server-Seite ist ggf. „*duplicate supression*“ anzuwenden
- *idempotente Operationen*/Zustandsfreiheit tolerieren Anforderungsduplikate

Fehlermaskierung (2)







Aktionsorientierte/datenorientierte Kommunikation

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

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

1. AG sendet die 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“

- Gemeinsamkeit von Prozeduraufruf und aktionsorientierter Kommunikation:
  - 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* ist der Aufruf einer entfernten Prozedur
  - „entfernt“ heißt „anderer Faden, Adressraum, Prozess und/oder Rechner“
  - => **Prozedurfernaufruf**

Synchrone Kommunikation

- die *remote-invocation send*-Semantik unterstützt Prozedurfernaufrufe, die *ein Ergebnis liefern*

  send gibt die 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

- die *synchronization send*-Semantik unterstützt alle *sonstigen* Prozedurfernaufrufe:

  send gibt die Anforderung ab, blockiert den AG und deblockiert ggf. den AN

  receive vom AN nimmt die Anforderung entgegen und deblockiert den AG

Asynchrone Variante — *Promises*

- asynchroner Prozedurfernaufruf kehrt *sofort* zurück und liefert als Ergebnis ein *promise*-Objekt (ein „Versprechen“ auf das Ergebnis, ohne jedoch den Verfügbarkeitszeitpunkt festzulegen)
  - das *promise*-Objekt dient der Aufnahme des ihm zugeordneten Resultats
  - der *promise*-Zustand kann (mittels `ready()`) abgefragt werden:

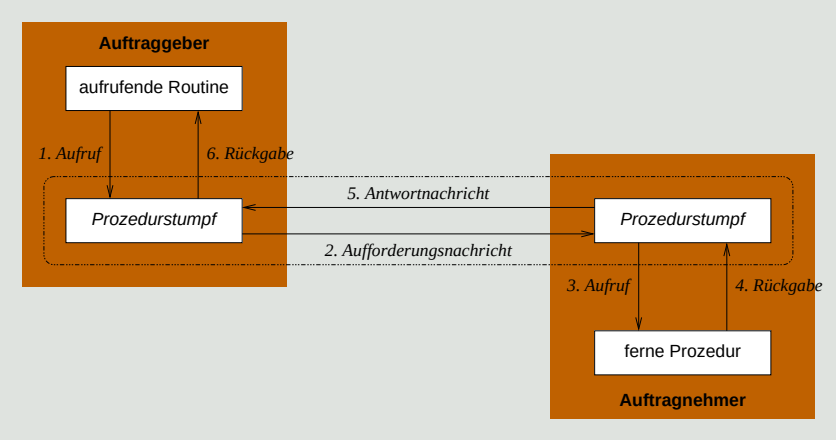
    blockiert ⇒ der Aufruf läuft, ein Resultat liegt noch nicht vor
    - ein `claim()`-Aufruf würde blockieren bis das Ergebnis vorliegt

    bereit ⇒ der Aufruf ist beendet, das Resultat liegt vor
    - in `claim()` ggf. blockierte Aufrufer werden deblockiert
    - der `claim()`-Aufruf liefert das Ergebnis zurück
  - eine sprachliche Unterstützung im objektorientierten Sinn erleichtert die Anwendung
  - Alternative für Sprachunterstützung: asynchrone Aufrufe mit *wait by necessity*
    - keine expliziten Promise-Objekte sondern implizites Blockieren bei Benutzung des Prozedurergebnisses
  - mit *no-wait send* wird die *promise*-Implementierung ideal unterstützt

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 die tatsächlichen 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

Prozedurstümpfe beim Prozedurfernaufruf



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) — Parameterarten

- die Auslegung der (formalen) Parameter bestimmt u.a. 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

Semantikaspekte (3) — Parameterübergabe

- *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:

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

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

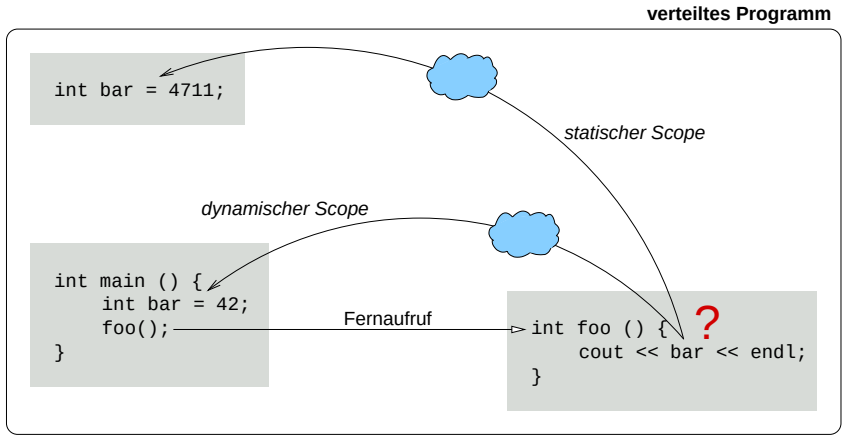
Semantikaspekte (4) — Gültigkeitsbereiche

- 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

Gültigkeitsbereiche von Programmvariablen



C/C++ bindet statisch, wie die meisten anderen Programmiersprachen auch. Das löst das Problem aber nicht.

Semantikaspekte (5) — Speicheradressen

- Programmadressen sind (im Regelfall) nicht systemweit eindeutig:
  - die fragliche Adresse kann beim Dienstanbieter bereits vergeben sein
  - nur im *single programm, multiple data* (SPMD) Modell wäre sie eindeutig (In dem Fall liegt auf den Rechnern des verteilten Systems dasselbe Programm. Demzufolge sind auf allen betrachteten Rechnern die Adressen der Variablen aber auch der Prozeduren/Funktionen identisch.)
  - im „Normalfall“ ist die Eindeutigkeit einer solchen Adresse eher zufällig
- allgemein sind Adressen abhängig von dem Kontext, in dem sie definiert sind
  - sie beziehen sich auf ein Programm in einem Adressraum auf einem Rechner
  - Adressen verteilter Programme besitzen eine **geographische Komponente**
- ferne Adressen entsprechen logischen Adressen — sie sind (ggf.) abbildbar