D Verteilte Objekte und CORBA

D.1 Überblick

- Verteilte Systeme
- OOP und Verteilung
- Java RMI
- CORBA
  - Architekturüberblick
  - Der Object Request Broker (ORB)
    (Interface Description Language (IDL), Remote invocation, Dynamic Invocation, Komponenten des ORB)
  - Portable Object Adaptor
  - CORBA Services
- Design Patterns für verteilte Anwendungen

D.2 Verteilte Systeme

- *"Distributed System"*
  Definition nach Tanenbaum und van Renesse
  - It looks like an ordinary centralized system.
  - It runs on multiple, independent CPUs.
  - The use of multiple processors should be invisible (transparent).
- *"Distributed System"*
  Definition nach Mullender
  - Zusätzlich: Not any single points of failures
- Definitionen sind nicht präzise
  - Manchmal ist es schwierig, ein lokales bzw. verteiltes System zu identifizieren
  - Definitionen basieren oft auf bestimmten Eigenschaften, die gerade wichtig erscheinen

Literatur

NeS98. J. Nehmer, P. Sturm: *Systemsoftware, Grundlagen moderner Betriebssysteme*. dpunkt, 1998.

Mul89. S. Mullender (Ed.): *Distributed Systems*. ACM Press, 1989.

Tan94. A. S. Tanenbaum, M. van Steen: *Distributed Systems*. Prentice Hall, 2002.

Tan95. A. S. Tanenbaum: *Verteilte Betriebssysteme*. Prentice Hall, 1995.

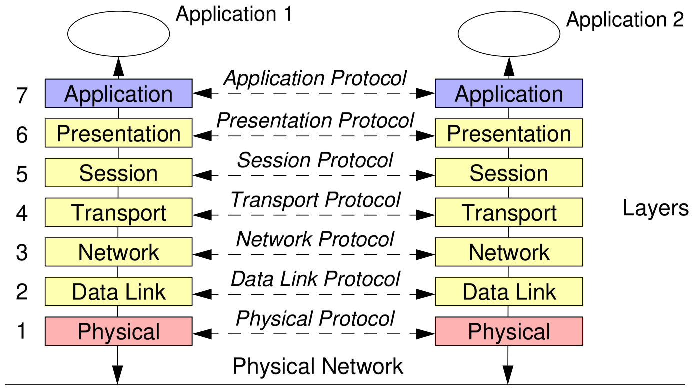
BiN84. A. D. Birrel, B. J. Nelson: "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* **2**(1), Feb. 1984, pp. 39–59.

Flyn72. M. J. Flynn: "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers*, **C-21**, Sept. 1992, pp. 948–960.

D.3 Kommunikationsmodelle

- Kommunikation erfordert gemeinsame Sprache
  - Protokolle

1 Protokollschichten nach dem ISO OSI Referenzmodell



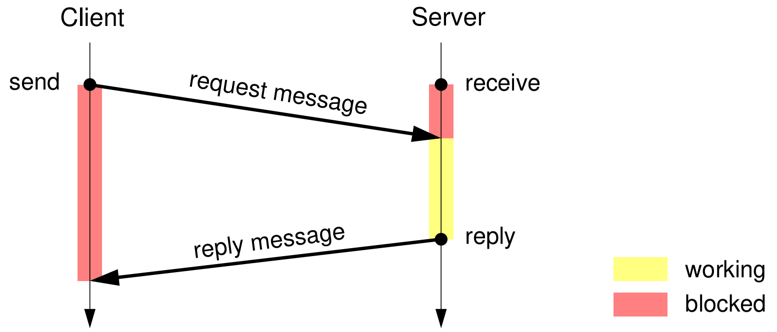
- Physical Layer (Physikalische Schicht)
  - Übertragung von 0-en und 1-en über ein Kabel
- Data Link Layer (Verbindungsschicht)
  - Senden von Bits; Trennung von Rahmen oder Paketen; Fehlerprüfung
- Network Layer (Netzwerkschicht)
  - Nachrichten durch größere Netze leiten (Routing)
- Transport Layer (Transportschicht)
  - Implementation von zuverlässigen Verbindungen
  - Fragementierung und Zusammensetzen von Paketen
- Session Layer (Sitzungsschicht)
  - Dialogkontrolle, Synchronisation
- Presentation Layer (Präsentationsschicht)
  - Transparenz von unterschiedlicher interner Datendarstellung
- Application Layer (Anwendungsschicht)
  - Menge von Anwendungsprotokollen
  - Electronic mail protocol
  - File transfer protocol
  - HTTP
  - etc.

2 Klassifikation

- Synchronität
  - wird der Sender blockiert bis der Empfänger die Nachricht hat, oder nicht?
- Interaktionsmuster
  - Message Passing — eine Nachricht wird von einem Teilnehmer an einen anderen gesendet
  - Request-Reply (Client-Server) Interaktion —
    Nachricht an einen Empfänger und Nachricht zurück an den ursprünglichen Absender
- Adressaten
  - Ein Empfänger
  - Mehrere Empfänger (Gruppenkommunikation, Multicast, Broadcast)

3 Synchrones Request-Reply Modell

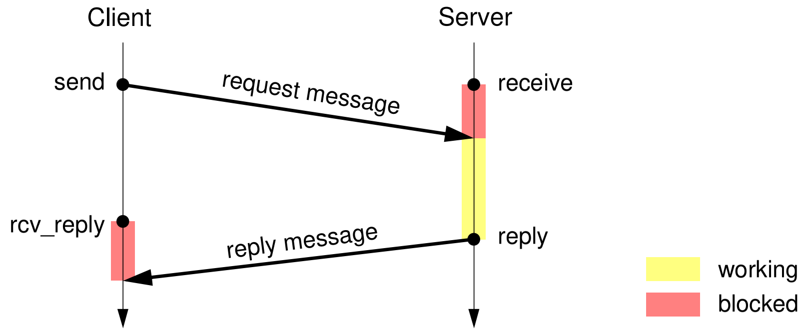
- Request-Reply Interaktion; synchrones Senden



- ◆ Sender wartet bis Antwort-Nachricht empfangen wurde
- ◆ Empfänger ist blockiert bis eine Nachricht eintrifft
- ◆ Client und Server arbeiten nicht nebenläufig
- ◆ bekannteste Realisierung ist RPC (Remote Procedure Call)

4 Asynchrones Request-Reply Modell

- Request-Reply Interaktion; asynchrones Senden



- ◆ Client und Server arbeiten nebenläufig
- ◆ Basis für Gruppenkommunikation

5 Zuverlässigkeit

- Nachrichten können verloren gehen wenn keine zuverlässige Verbindung benutzt wird
  - zuverlässige Verbindungen benutzen Acknowledge-Nachrichten (ACK)
  - für einfache Nachrichtenübertragung großer Overhead
- ★ Zuverlässigkeit mit Request-Reply Interaktionsmodell kombinieren
- Mögliche Fehler
  - Server Crash
Fehlermodell: Totalamnesie
(Server verliert alle Informationen über frühere Anfragen)
  - Anfrage-Nachricht geht verloren
  - Antwort-Nachricht geht verloren
- Ideale Semantik
  - *exactly-once*
die Anfrage wird genau einmal auf Serverseite bearbeitet

5 Zuverlässigkeit (2)

- **At-Least-Once Semantik**
  - Anfrage wird einmal oder mehrmals bearbeitet
  - Client bekommt nie eine Fehlermeldung aber er erkennt eventuell, dass die Anfrage mehrfach bearbeitet wurde: Operationen müssen *idempotent* sein!
- Implementation
  - wenn der Client nach einiger Zeit (time-out) keine Antwort erhält, wiederholt er die Anfrage
    - keine zusätzliche Funktionalität auf Server-Seite erforderlich
    - der Server darf wiederholte Anfrage aber ignorieren, wenn er sie erkennen kann

5 Zuverlässigkeit (3)

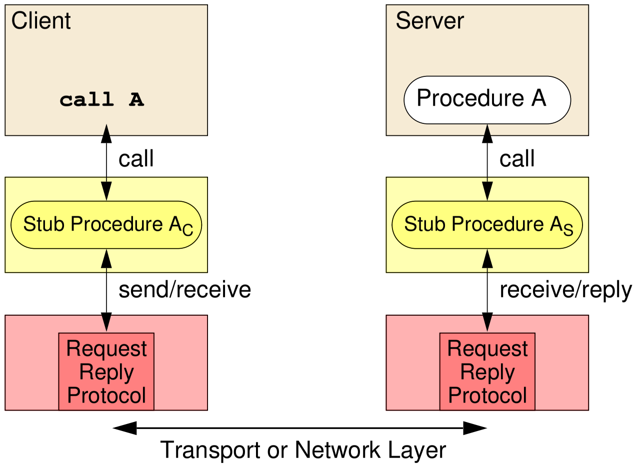
- **At-Most-Once Semantik**
  - Die Anfrage wird höchstens einmal (oder überhaupt nicht) bearbeitet
- Einfache Implementation (nur auf der Client-Seite)
  - wenn das Ergebnis innerhalb einer bestimmten Zeit nicht eintrifft wird dem Aufrufer eine Fehlermeldung zurückgegeben (at-most-once Semantik)
  - sonst wird das Ergebnis zurückgegeben (exactly-once Semantik)
- Komplexere Implementation
  - Client wiederholt Nachricht nach einem Time-out (verdeckt Nachrichtenverlust auf der Verbindung)
  - Client muss Server-Abstürze erkennen (Fehlermeldung an den Aufrufer, at-most-once Semantik)
  - Server hebt Antworten auf (Wiederholung bei Verlust der Nachricht)
  - Server muss alte Anfragen nach einem Absturz erkennen und ignorieren
  - wenn das Ergebnis zurückgegeben wird, haben wir exactly-once Semantik

6 Remote Procedure Calls

- Request-Reply-Modell kann für die Implementierung von RPCs genutzt werden
  [Birrell and Nelson 1984]
  - statt eine Anfrage zu senden wird eine Remote-Prozedur aufgerufen
  - statt einer Antwort erhält man die Ergebnisse des Aufrufs
- ★ Aufruf einer Prozedur ist ortstransparent
  - Syntax für lokale und remote Aufrufe kann identisch sein
  - sehr intuitiv
  - keine expliziten send- und receive-Anweisungen erforderlich
- Implementierung von RPCs
  - Stub-Prozeduren auf Client- und Server-Seite

6 Remote Procedure Calls (2)

- Implementierung von RPCs mit Stub-Prozeduren



nach Nehmer 1995

6 Remote Procedure Calls (3)

- Client-Stub
  - Marshalling der Parameter (Zusammenstellen einer Anfrage-Nachricht)
  - Senden der Anfrage-Nachricht
  - Warten auf die Antwort-Nachricht
  - Unmarshalling des Ergebnisses
  - Implementierung der Zustell-Semantik
- Server-Stub
  - Empfangen der Anfrage-Nachricht
  - Unmarshalling der Parameter
  - Aufruf der Server-Prozedur
  - Marshalling der Ergebnisse
  - Senden der Antwort-Nachricht
  - Implementierung der Zustell-Semantik

6 Remote Procedure Calls (4)

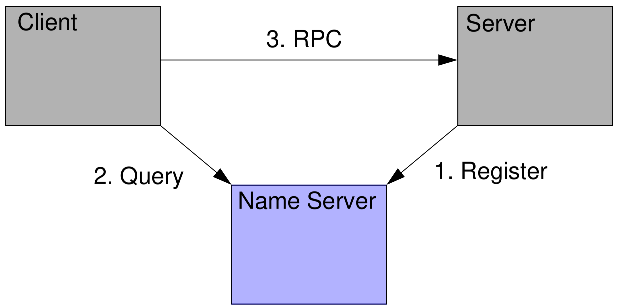
- Probleme bei RPCs
  - Marshalling der Parameter
    - Zahl und Typen müssen bekannt sein (vgl. mit C: `printf( "Count %d\n", count )`)
  - Parameter-Übergabe-Semantik
    - *Call-by-value:* unproblematisch
    - *Call-by-reference:* wie implementieren?
  - keine globalen Variablen
  - Semantik
    - Server-Absturz; keine exactly-once Semantik
  - Performance
    - keine Nebenkläufigkeit
    - große Parameter-Daten
    - kurze Prozeduren

6 Remote Procedure Calls (5)

- Automatische Erzeugung von Stub-Prozeduren
  - Werkzeug kann Code generieren für:
    - Parameter-Marshalling
    - Client-Stub Prozedur
    - Server-Stub Prozedur
    - Server-Schleife zum Warten auf Anfragen
- Binden von Client-Stubs an Server-Stubs
  - Server-Stub hat Netzwerk-Adresse, die Client-Stub kennen muss
  - Problem: woher kennt der Client den Server?
- Name-Server
  - Symbolische Namen werden in Netzwerk-Adressen umgesetzt

7 Name-Server und Binden

- Bekannter Name-Server wandelt Namen in Adressen um
  - ◆ Client kennt Namen seines Servers und die Adresse eines Name-Servers
  - ◆ Name-Server wandelt Namen in eine (dynamische) Netzwerk-Adresse um
  - ◆ Client kann sich immer an Server binden
  - ◆ Server muss seine Netzwerkadresse beim Name-Server registrieren



D.4 OO Verteilte Anwendungen

- Objektorientierte Anwendungen im verteilten System
- Verteilung auf
  - ➤ verschiedene Rechner
  - ➤ verschiedene Prozessoren
  - ➤ verschiedene virtuelle Maschinen
  - ➤ verschiedene Adressräume
    - Auch wenn nur die ersten beiden Punkte die Definitionen von Verteilten Systemen treffen, treten auch bei einer Verteilung auf verschiedene virtuelle Maschinen oder verschiedene virtuelle Adressräume auf einer CPU ähnliche Probleme auf: Es gibt eine Grenze zwischen Objekten, die von der "normalen Objektinteraktion" (=Methodenaufruf) nicht ohne besondere Vorkehrungen überwunden werden kann.
- Objektinteraktionen zwischen objektorientierten Programmen *oder* Aufteilung eines Programmes in interagierende Programmteile?
  - In beiden Fällen interagieren Objekte über Grenzen hinweg. Meist ist eine solche Interaktion nicht von der jeweiligen Programmiersprache direkt unterstützt, so dass zusätzliche Mechanismen ins Spiel kommen müssen.

D.5 OOP und Verteilung

1 Klassifikation von Interaktionsformen

- Formen/Ausprägungen von Objektinteraktion im Verteilten System
  - ◆ explizit
    - der Anwendungscode beschreibt die Interaktion im verteilten System explizit (z.B. durch Aufbau einer Verbindung zu einem anderen Rechner)
  - ◆ implizit
    - die Interaktion im verteilten System erfolgt bei Bedarf implizit (z. B. wenn eine Objektreferenz eine Remote-Referenz ist)
  - ◆ orthogonal
    - die Programmiersprache enthält keine Sprachkonzepte zur Unterstützung verteilter Anwendungen. Die Interaktion zwischen verteilten Objekten erfolgt mit Hilfe von Klassen- oder Funktionsbibliotheken, die Kommunikationsmechanismen bereitstellen
  - ◆ nicht-orthogonal
    - die Programmiersprache enthält Sprachkonstrukte, die eine Interaktion zwischen Objekten im verteilten System erlauben
  - ◆ uniform
    - lokale und verteilte Objektinteraktion unterscheiden sich auf programmiersprachlicher Ebene nicht
  - ◆ nicht-uniform
    - die Programmiersprache unterscheidet lokale und verteilte Objektinteraktion
  - ◆ transparent
    - die Verteilung ist für den Anwendung transparent, bei der Programmierung muss in Bezug auf Verteilung nichts berücksichtigt werden
  - ◆ nicht-transparent
    - die Verteilung ist nicht transparent, selbst bei uniformer und impliziter Objektinteraktion im verteilten System muss der Programmierer Vorkehrungen getroffen haben (z.B. Abfangen von Remote-Exceptions)

2 explizite, orthogonale Interaktion

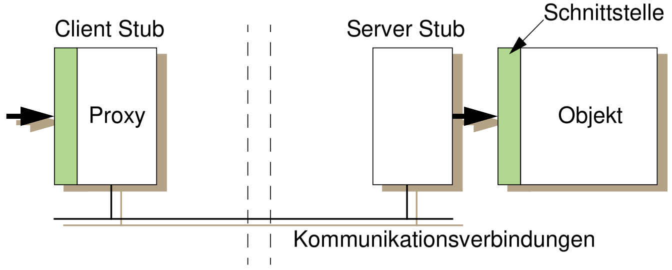
- weit verbreitete Vorgehensweise
- von klassischen Interprozesskommunikations-Mechanismen geprägt
  - ➤ Nachrichten (Datagramm-Sockets, Messages, …)
  - ➤ Verbindungen (Stream-Sockets, Pipes, …)

\+ Vorteile
- ◆ meist weit verbreitete, etablierte Infrastruktur vorhanden
- ◆ kein "unsichtbarer" Overhead

– Nachteile
- ◆ Programmierung aufwendig
- ◆ Bruch im objektorientierten Paradigma
  - ➤ Serialisierung von Parametern, Verlust von Typ-Information
- ◆ Verteilung wird durch die Programmierung "fest verdrahtet"
  - ➤ Software sehr unflexibel in Bezug auf Änderungen

3 implizite, nicht-orthogonale Interaktion

- Interaktion zwischen lokalen und verteilten Objekten unterscheidet sich prinzipiell nicht
  - nur ein Interaktionsmechanismus: Methodenaufruf
- grundlegendes Konzept: Remote Procedure Call
  - Verteilung wird durch Vermittler- oder Stellvertreterobjekte vor den Kommunikationspartnern (weitgehend) verborgen
  - Proxy/Stub-Prinzip



- Ein Stub ist ein Vermittler für Methodenaufrufe, ein Proxy ist ein Stellvertreter für ein Objekt (die Begriffe werden häufig synonym gebraucht, ganz genau genommen nehmen die Stubs die Funktion von Proxies wahr).
- Mit Hilfe von Stubs kann die Objektinteraktion über Adreßraumgrenzen realisiert werden. Auf der Client-Seite gibt es das Proxy- oder Client-Stub-Objekt. Es stellt die gleiche Schnittstelle wie das anzusprechende Objekt selbst bereit. Über einen Kommunikationskanal, der z.B. durch UNIX-Sockets oder ähnliches realisiert wird, kommuniziert der Client-Stub mit dem Server-Stub, der letztlich den am Proxy registrierten Methodenaufruf am richtigen Objekt durchführt.
- Der Client-Stub ist ein Stellvertreter für das eigentliche Objekt, der Server-Stub ein Stellvertreter für den Aufrufer. Der Server-Stub wird häufig auch als "Skeleton" bezeichnet.

4 uniforme / nicht-uniforme Interaktion

★ **nicht-uniforme Interaktion**

- unterschiedliche Methodenaufrufe für lokale und remote-Referenzen
- unterschiedliche Semantik bei der Parameterübergabe
  - by reference, by value
  - Problem: Übergabe von lokalen Objektreferenzen

★ **uniforme Interaktion**

- keinerlei Unterschied zwischen lokalen und remote-Aufrufen

5 transparente / nicht-transparente Verteilung

- volle Transparenz:
  Anwendungsentwickler sieht keinerlei Unterschied zwischen lokalen und verteilten Objekten
- Probleme:
  - im verteilten Fall können spezielle Fehler auftreten
    - ➤ unabhängiger Objektausfall →Remote Exception
  - verteilte Interaktion ist implizit signifikant teurer
    - ➤ Transparenz kann zu Ineffizienz führen
  - Verteilung ist häufig ein Entwurfskriterium
    - ➤ Verbergen der Verteilung in der Implementierung ist unsinnig
- Fazit:
  - bei der Programmierung sollte zwischen potentiell verteilten und definitiv lokalen Objekten unterschieden werden können

6 Herausforderungen

- Überwindung heterogener Hardware- und Softwarestrukturen
  - ➤ verschiedene Hardware
  - ➤ verschiedene Betriebssysteme
  - ➤ verschiedene Programmiersprachen
    - Durch die Verteilung von Programmteilen auf mehrere Rechner entsteht meist automatisch das Problem der Heterogenität der verwendeten Komponenten.
- Ortstransparenz
  - ➤ statische Konfiguration
  - ➤ Objektmigration
    - Ist es unwichtig, auf welchem Rechner ein Objekt nun wirklich implementiert ist, so lassen sich zum einen Konfigurationsänderungen bei der Verteilung ohne das Ändern der Applikation durchführen (statische Änderungen). Zum anderen ist es eventuell auch möglich Objekte zur Laufzeit von einem Rechner zum anderen wandern zu lassen (dynamische Änderung).
- Globale Dienste
  - – z.B. Namensdienste, Transaktionsdienst, Persistenz, Kontrolle der Nebenläufigkeit
    - Objekte müssen sich nun finden. Dazu wird ein übergreifender Namensdienst benötigt.
    - Andere höherwertige Dienste können für die verteilte Applikation nicht lokal erbracht werden, sondern müssen übergreifend von allen Rechnersystemen bereitgestellt werden.