H 8. Übung

H.1 Überblick

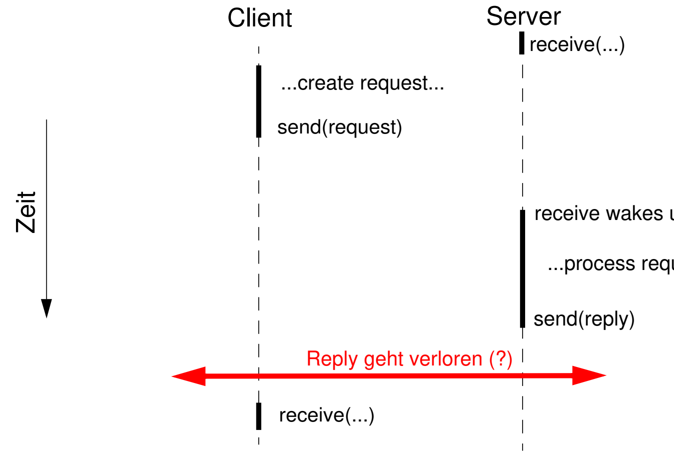
- Besprechung Aufgabe 4
- Ausgewählte Aspekte eines transparenten Fernaufrufmechanismus
  - ◆ Verklemmungsgefahr
    - Beispiel 1: Türme von Hanoi
    - Beispiel 2: Verteilte Koordinierung (Semaphore via RPC)
  - ◆ Aufruftransparenz
    - Komplexere Datenstrukturen bei Call-by-Value
    - Call-by-Reference nochmals betrachtet
    - “Echtes” Call-by-Reference über Callback-Mechanismus
- Aufgabe 5
  - ◆ Erweiterung des eigenen RPC um transparenten Callback-Mechanismus

H.2 Besprechung Aufgabe 4

- Aufgabe 4
  - ◆ "Call-by-Value-Result" (war hoffentlich einfach?)
  - ◆ automatische Stub-/Skeleton-Erzeugung mit IDLflex
- Fragen oder Probleme?

H.2 Besprechung Aufgabe 4

- Timing-Probleme beim Fernaufruf



H.2 Besprechung Aufgabe 4

- Lösungsmöglichkeiten
  - ◆ Kein Problem bei Unix-Sockets, wenn Empfangs-Socket bereits vor dem **`send`** geöffnet ist
    - Daten werden vom Kern automatisch gepuffert (i.d.R. max. 64kB)
  - ◆ Fast kein Problem bei echter Verteilung
    - Größerer Zeitabstand zwischen Senden des Request und Empfangen des Reply. Höchstwahrscheinlich hat Client bereits **`receive`** ausgeführt
  - ◆ Problem vermeidbar mit mehreren Threads auf Seite des Clients
    - **`receive`** wird von einem Dispatcher-Thread aufgerufen, bevor **`send`** vom eigentlichen Aufrufer ausgeführt wird
- Schlussfolgerung
  - ◆ Für ein *immer* korrekt funktionierendes System muss man sich überlegen:
    - Kann eine Antwort ankommen, bevor die RPC-Schicht bereit ist, diese zu empfangen?
    - Falls ja, wird diese dann vom Betriebssystem (oder sonst wo) gepuffert?

H.3 Transparenter RPC

1 Überblick

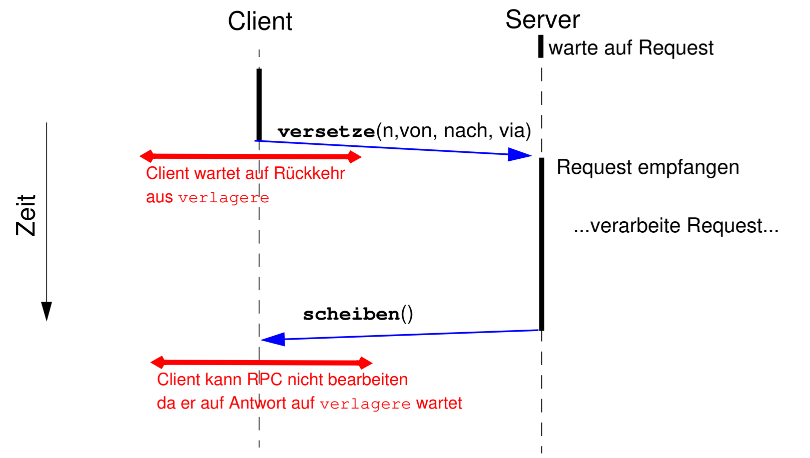
- Rückblick: Transparenz beim RPC
- Verklemmungsgefahr bei naiver Implementierung
- Komplexe Datenstrukturen bei "Call-by-Value"
- "Call-by-Reference" genauer betrachtet

2 Transparenz beim RPC

- Intention beim RPC (Idealvorstellung)
  - ◆ Der Anwendungsentwickler soll vom Fernaufruf nur den Aufruf selbst wahrnehmen, nicht die zusätzlichen Probleme, die sich durch eine Verteilung ergeben können
  - ◆ An erster Stelle: **Netzwerktransparenz**
    - *Zugriffstransparenz:* Gleicher Mechanismus im lokalen wie im verteilten Fall.
      Erfordert bei heterogenen System *Heterogenitätstransparenz*
    - *Ortstransparenz:* Zugriff auf entfernten Dienst, ohne den Ort des Dienstes explizit zu kennen
  - ◆ Weitere: Nebenläufigkeits-, Replikations-, Fehler-, Migrations-, ...-Transparenz
- *Völlige Transparenz wird in Realität kaum erreicht!*

3 Verklemmungsgefahr beim RPC

- Beispiel 1: Türme von Hanoi (siehe Vorlesung)



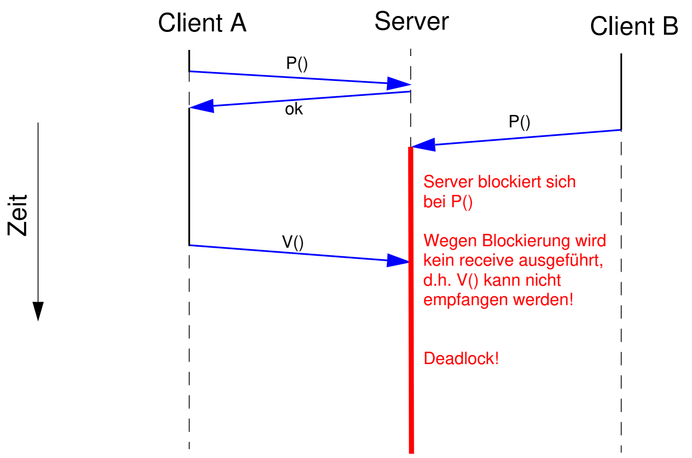
3 Verklemmungsgefahr beim RPC

- Implementierung einer Semaphore zur Koordinierung in verteilten Systemen
- Lokale Implementierung

```
class Semaphore {
    public:
    void P() {
        ...     // Lock ressource
                // If already locked, block until unlock
    }
    void V() {
        ...     // Unlock ressource
    }
}
```

3 Verklemmungsgefahr beim RPC

- Direkte Verwendung mit bisherigem RPC-System



3 Verklemmungsgefahr beim RPC

- Verhinderung einer Verklemmung
  - ◆ Verklemmung kann a priori ausgeschlossen werden
    - Es ist sichergestellt, dass es keine zyklischen Aufrufketten geben kann: z.B.: Ein Teilnehmer kann nie sowohl Client als auf Server sein
    - Die über RPC aufgerufene Prozedur kann nicht blockieren (beispielsweise keine Semaphore möglich)
  - ◆ Verklemmung kann durch geeignete RPC-Mechanismen vermieden werden
    - Mehrere Aktivitätsträger!
      Es gibt stets einen Aktivitätsträger, der Anforderungen entgegennehmen und verarbeiten kann

H.4 Komplexe Datenstrukturen beim RPC

1 Überblick

- Strukturen, Arrays
- Verzeigerte Datenstrukturen

2 Strukturen, Arrays

- Sequentielle Übertragung der einzelnen Elemente
- Beispiel

```
struct Person {
    char lastname[40];
    char firstname[40];
    int age;
}

class AdressBook {
    void addPerson(const Person p);
    ...
}
```

2 Strukturen, Arrays

- Passender Stub für das Beispiel

```
struct Person {
        char lastname[40];
        char firstname[40];
        int age;
}
class AdressBook {
        void addPerson(const Person p);
        ...
}
```

```
class AdressBookStub {
    void addPersion(const Person p) {
        ...
        buffer = new Request(...)
        buffer.writeArray(40, p.lastname);
        buffer.writeArray(40, p.firstname);
        buffer.write(p.age);
        comm.send(buffer);
        ...
    }
    ...
}
```


2 Strukturen, Arrays

- Oder Hilfsklasse generieren (so z.B. in CORBA)

```
struct Person {
        char lastname[40];
        char firstname[40];
        int age;
}
class AdressBook {
        void addPerson(const Person p);
        ...
}
```

```
class AdressBookStub {
    void addPersion(const Person p) {
        ...
        buffer = new Request(...)
        PersonHelper.write(buffer, p);
        comm.send(buffer);
    }
}
class PersonHelper {
    static void write(Request req, Person &p) {
        req.writeArray(40, p.lastname);
        req.writeArray(40, p.firstname);
        req.write(p.age);
    }
}
```


3 Verzeigerte Datenstrukturen

- Im lokalen Fall wird hier normalerweise Call-by-Reference verwendet

```
typedef struct NodeStruct TreeNode
struct NodeStruct {
    TreeNode *left;
    TreeNode *right;
};

TreeNode *root = ...;
```

- Für verteiltes Call-by-Reference siehe nächster Abschnitt
- Variante 2: Daten rekursiv in Datenstrom schreiben
  - ◆ Client und Server müssen beide die Kodierung der Daten kennen (was kommt in welcher Reihenfolge)
  - ◆ Nachteil: Oft große Datenmenge, zu Übertragende Datenmenge nicht a priori bekannt

3 Verzeigerte Datenstrukturen

- Weitere Probleme: Zyklen in einem verzeigerten Graphen (Zum Beispiel verkettete Liste)
  - ◆ Gefahr einer Endlosschleife!
  - ◆ Unterscheidung notwendig:
    - *Zeiger zum ersten mal vom Marshalling betrachtet:* Übertragung der referenzierten Daten
    - *Weiteres Vorkommen des selben Zeigers:* Übertragung der Information, dass hier bereits zuvor übertragene Daten referenziert werden

H.5 Call-by-Reference genauer betrachtet

1 Eigenschaften und Probleme

- Call-by-Reference:
  - ◆ Es wird lediglich eine Referenz als Parameter übergeben
  - ◆ Vorteilte/Ziele:
    - Aufgerufene Prozedur kann Daten des Aufrufers verändern
    - Ggf. umfangreiche Daten müssen nicht kopiert werden
- Probleme von Referenzen im verteilten System:
  - ◆ Normalerweise sind Zeiger nur jeweils lokal gültig (Ausnahmen siehe Vorlesung; z.B. DSM-Systeme)
  - ◆ Eine einfache Übertragung eines Zeigers ist daher meist nicht sinnvoll möglich

2 Call-by-Value-Result

- Siehe vorherige Übung
- Eigenschaften
  - ◆ Aufgerufene Prozedur kann Daten des Aufrufers verändern

    (Einschränkung: nur während der Prozedurausführung möglich!
    Es ist nicht möglich, die Referenz ausserhalb des Prozedurkontextes zu speichern und später nach Rückkehr zu verändern!
    Häufig ist dies keine problematische Einschränkung.)
  - ◆ Komplette Daten werden sogar doppelt übertragen

    (sicher kein Problem bei primitiven Datentypen, aber bei komplexeren Strukturen kann dies erheblichen Aufwand bedeuten!)

3 Echtes Call-by-Reference

- Im allgemeinen Fall meist nur mit OS-Unterstützung möglich
- Spezieller Fall: Objektreferenzen in OO-Programmiersprachen
  - ◆ Objekt kapselt Daten, Zugriff (Idealfall) nur durch Methodenaufrufe
  - ◆ Übertragung einer Objektreferenz kann auf Server-Seite automatisch dazu führen, dass eine Objekt-Stub erzeugt wird, der es erlaubt, alle Methoden des Originalobjekts per RPC aufzurufen.
  - ◆ Server hat damit transparenten Zugriff über die Referenz auf das Original-Objekt. Verhalten entspricht daher "echtem" Call-by-Reference
  - ◆ Einschränkung: Kein direkter Zugriff auf Objektdaten
    (keine privaten Variablen)

3 Echtes Call-by-Reference

- Beispiel

```
typedef char message[256];
interface Printer {
    void write(in message msg);
};
interface HelloServer {
    void sayHello(in Printer prt, in message msg);
};
```

- ◆ Client besitzt lokale Implementierung von Printer

```
class PrinterImpl: public Printer {
    void write(char msg[256]) { printf(...); }
}
```

- ◆ Auf Serverseite existiert eine Implementierung von HelloServer

```
class HelloServerImpl: public HelloServer {
    void sayHello(Printer p, char msg[256]) { p.write(msg); }
}
```

3 Echtes Call-by-Reference

- Mögliches Vorgehensweise

Client | Server

1. Erzeuge Stub für HelloServer => Referenz auf Interface
2. Rufe sayHello(...) mit Referenz auf lokale Instanz von PrinterImpl auf
3. Stub von HelloServer erzeugt ggf. Skeleton für den Printer und registriert diesen am RPC-System
4. Remote-Adresse wird mit sayHello-Aufruf zum Server übertragen
5. Server-Skeleton wird aufgerufen und ließt Remote-Adresse; ein PrinterStub wird erzeugt und damit initialisiert
6. sayHello von HelloServerImpl wird mit Referenz auf den Stub aufgerufen
7. Server ruft write am Interface Printer auf
8. Transparenter Fernaufruf zum Client über PrinterStub

4 Verwaltung der Stubs und Skeletons

- Bei naiven Vorgehen:
  - Bei jedem Marshalling einer Objekt-Referenz wird ein Stub und ein Skeleton erzeugt
  - Unnötig, falls die selbe Objektreferenz mehrfach übertragen wird!
- Übliches Verfahren in vielen RPC-Systemen:
  - Hashtabelle bei Client und Server
    - für Client: Abbildung lokaler Objektreferenzen auf Skeletons (oder Liste aller Skeletons, die durchsucht werden kann)
    - für Server: Abbildung von Remote-Referenzen auf Stubs
- Wann können Stub oder Skeleton aufgeräumt werden, d.h. wann werden sie nicht mehr benötigt?

4 Verwaltung der Stubs und Skeletons

- Stubs
  - In der aufgerufenen Server-Methode (explizit durch free, implizit)
  - Im Skeleton, der den Stub erzeugt hat, diesen auch wieder freigeben.
  - ???
  - Durch automatische Garbage Collection (Achtung, lokale Hashtabelle besitzt Referenzen auf Stubs, hier sind z.B. weak references (Java) zu verwenden!)
- Skeletons
  - 1. Möglichkeit: Skeleton erst dynamisch erzeugen, wenn eine Anfrage kommt, nach der Bearbeitung der Anfrage wieder löschen
    - Dazu muss z.B. die Remote-Referenz die lokale Adresse des Objekts beinhalten. Strenge Typprüfung kaum möglich
  - 2. Möglichkeit: Destruktor des Stubs benachrichtigt die Gegenseite davon, dass der Skeleton nicht mehr benötigt wird.

5 Aufrufketten

- Referenz wird über mehrere Fernaufrufe weitergereicht

Rechner 1	Rechner 2	Rechner 3
Objekt A (lokal)		
RPC: Skeleton für A =>	B := Stub für A	
	RPC: Skeleton für B =>	Stub für B

  - Aufruf von Rechner 3 wird über Rechner 2 zu Rechner 1 geschickt
- Besser: Aufrufoptimierung
  - Von Rechner 2 wird keine Skeleton und keine neue Remote-Adresse generiert, sondern die ursprüngliche Adresse von Rechner 1 weitergegeben

H.6 Aufgabe 5

- Implementiere im bisherigen RPC-System einen Call-by-Reference-Mechanismus
- einfache Beispielanwendung

```
typedef char message[256];

interface Printer {
    void write(in message msg);
};
interface HelloServer {
    void sayHello(in Printer prt, in message msg);
};
```