Aufgabe 3: RPC Grundlagen, Algorithmus von Lampson

Literatur:

[1] Nelson, B.J.:
Remote procedure Call. Tech. Rep. CLS-81-9, Xerox Palo Alto Research Center, Palo Alto, CA, 1981

[2] Kaiserswerth, M.:
Der Fernaufruf als Betriebssystemdienst. Arbeitsbericht des IMMD, Band 22, Nummer 1 Januar 1989

[3] Tanenbaum, A.
Modern Operating Systems
Prentice Hall Int., 1992

a) Welche Grundmechanismen werden bei RPCs unterschieden ?

b) Welche Aufrufsemantiken bei RPCs gibt es?
Welche Vor- und Nachteile beinhalten Sie?

c) Wie würde ein Aufruf des Kommandos *rusers* nach dem Algorithmus von Lampson ablaufen?

d) Wie kann der Algorithmus von Lampson verbessert werden, um die Mehrfachausführungen von Aufrufen zu verringern?

Lösungsvorschlag Aufgabe 3: RPC - Grundlagen

a) Welche Grundmechanismen werden bei RPCs unterschieden ?

Transfermechanismus

Betriebssystem und Compiler zusammen bieten von vorne herein einen Mechanismus an, der es (umgangssprachlich ausgedrückt) erlaubt, den Unterprogramm-Stack auf einem anderen Rechner fortzusetzen. Ein transparenter Zugriff auf die Daten anderer Rechner/Prozessoren muß möglich sein.
Übertragen werden **activation records**, die Parameter, Rücksprungadresse usw. enthalten (Marshalling).
Die Implementierung eines RPC ist einfach.

Stubs [3]

Stubs sind Platzhalter oder Stellvertreter für Prozeduren. Statt eines lokalen Prozeduraufrufs werden vom Client-Stub die Parameter in eine Nachricht verpackt und zum Server gesendet. Der Client-Stub wartet, bis er eine Nachricht mit dem Ergebnis erhält. Server analog.
Stubs verbergen das Senden von Nachrichten vor dem Benutzer.
Stubs werden bei vielen RPC-Implementierungen verwendet.

Probleme bei Stubs sind globale Daten, die Übergabe von Zeigern und die (unterschiedliche) Datenrepräsentationsform.
So ist es z.B. nicht möglich, einen Zeiger auf die Wurzel eines Baumes zu übergeben. Es gibt im Prinzip drei Möglichkeiten, um die Daten eines Baumes bei RPCs zu übertragen. Erstens kann der gesamte Baum in eine flache, geklammerte Darstellung umgewandelt werden und diese Werte können beim Aufruf übertragen werden. Zweitens kann bei dem RPC ein Nachfordern von Parametern beim Client implementiert werden. Damit müssen nur die angesprochenen Knoten übertragen werden. Eine dritte Möglichkeit, die sehr oft bei objektorientierten Systemen Anwendung findet, besteht darin, daß der Zeiger auf der Klientenseite einen weiteren Stub bildet. Da bei objektorientierten Sprachen der Client die Daten nicht direkt manipuliert, sondern durch Methodenaufrufe an dem Zeiger die Manipulation "durchführen läßt", kann man diese Manipulation wieder als RPC modellieren. Bei prozeduralen RPCs ist die erste Vorgehensweise üblich.

b) Welche Aufrufsemantiken bei RPCs gibt es? Welche Vor- und Nachteile beinhalten Sie?

Die wichtigsten Aufrufsemantiken im Bezug auf RPC's sind:

<u>exactly-once</u>

- Vorteil:
  Der optimale Fall. Keine Maßnahmen im Benutzerprogramm für Fehlerbehandlungen vorzusehen. Einfache Anwendbarkeit.

- Nachteil:
  So gut wie nicht zu realisieren. Erheblicher Aufwand selbst bei direkt erreichbarem Zielrechner. Bei Ausfall von "Gateways" und automatischer Alternativwegewahl und weiteren "Unter-"RPCs zur Aufgabenbewältigung nicht mehr effizient realisierbar.

<u>at-least-once</u>

- Vorteil:
  Gut geeignet für "abfragende" RPCs. Die Ausführung und damit das Ergebnis wird immer erreicht. Einfach zu realisieren.

- Nachteil:
  Ausführung kann mehrfach erfolgen, was bei bestimmten Einsatzgebieten nicht tolerierbar sein kann (z.B. Buchungssysteme). Mehrfache Ergebnisse müssen im Benutzerprogramm erkannt und behandelt werden. Für "modifizierende" RPCs völlig ungeeignet.

<u>last-of-many</u>

- Vorteil:
  Gut geeignet für "abfragende" RPCs. Liefert das aktuellste Ergebnis. Die Ausführung und damit das Ergebnis wird immer erreicht. Einfache Anwendbarkeit. Auch noch einfach realisierbar. Ist mit Einschränkungen auch bei "modifizierenden" RPCs einsetzbar.

- Nachteil:
  Ausführung kann mehrfach erfolgen, was bei bestimmten Einsatzgebieten nicht tolerierbar sein kann (z.B. Buchungssysteme). Die Zustellung des Ergebnisses an den Aufrufer erfolgt u.U. später als das "erste" Ergebnis vorliegt. Neigt bei falscher Parametrierung zu schlechtem Durchsatz (Timeout zu kurz - häufiges erneutes Versuchen, bis der Zielrechner schnell genug war / Timeout zu lang - lange Antwortzeiten bei Verlust des Aufrufs).

c) Wie würde ein Aufruf des Kommandos *rusers* nach dem Algorithmus ablaufen?

Das UNIX-Kommando '*rusers faui45s*' gibt die Namen der Benutzer aus, die auf dem Rechner '*faui45s*' arbeiten.
Es wird über einen RPC abgewickelt. Es gibt keine Eingabeparameter und als Ausgabeparameter wird eine Liste mit Benutzernamen zurückgeliefert.

Aufbau einer Nachricht:

Name	Typ	Bemerkung
state	call/return	Zustand der Nachricht, Anfrage oder Antwort
source	Processor	Port des aufrufenden Prozesses (Client)
dest	Processor	Port des aufgerufenen Prozesses (Server bzw. Portmapper)
id	ID	Sequenznummer für Wiederholungen
request	ID	Sequenznummer für diesen speziellen Fernaufruf
action	procedure	Name des gewünschten Fernaufrufs (RPC)
val	Value	Ein- bzw. Ausgabeparameter für die Prozedur

Es ist dabei zu beachten, daß die Sequenznummer für die Wiederholungen nicht bei jedem neuen Fernaufruf (request) auf 0 oder 1 zurückgesetzt wird, sondern daß sie für alle Aufrufe eindeutig und streng monoton wachsend ist (Funktion UniqueID()).

Beispiel: Startnachricht für *rusers*

Name	Wert	Bemerkung
state	call	Zustand der Nachricht: Anfrage
source	131.188.44.84:4711	Port des Client *rusers*
dest	131.188.44.55:111	Port des Server (Portmapper)
id	3501	Sequenznummer für Wiederholungen
request	3501	Sequenznummer für '*rusers faui45s*'
action	get_service	Name des gewünschten Fernaufrufs
val	rusers	Parameter - hier Name des Services

Bei Wiederholungen würde sich nur die Sequenznummer *id* ändern, z.B. 4008, 4057, 4259, ...

Beispielablauf:

Eine Nachricht mit der ID=4003 wird erzeugt durch

Client auf faui47e:

- DoCall (faui45s:1702, rusers, NIL);
    - ID=4003
    - Send (faui45s, Message)
    - Wait (received, timeout)

Distributor auf faui45s

- Receive (status, Message)
- OKtoAccept (Message)
- StartCall (Message)
    - Nachricht ablegen
    - Signal (w->work)

Worker bzw. Server (deblockiert an w->work)
- user-list = rusers ();

Timeout bei DoCall läuft ab.

Client auf faui47e:

    - ID=4008
    - Send (faui45s, Message)
    - Wait (received, timeout)

Distributor af faui45s

- Receive (status, Message)
- OKtoAccept (Message)
- StartCall (Message)
    - ID=4008 /* Aktion hiermit beendet */

Timeout bei DoCall läuft noch einmal ab.

Client auf faui47e:

    - ID=4057
    - Send (faui45s, Message)
    - Wait (received, timeout)

Distributor auf faui45s

- Receive (status, Message)
- OKtoAccept (Message)

Der Prozeß des Distributors wird verdrängt (Prozeßwechsel)

Worker arbeitet wieder
- Message.state = return
- EndCall
    - Send (faui47e:4711, user-list) mit ID=4008
    - aufräumen
    - Wait (w->work)

Distributor auf faui47e

- Receive (status, Message)
- DoReturn (Message)
    - Nachricht verwerfen wegen alter ID

Distributor auf faui45s

- StartCall (Message)
    - ID=4057
    - Nachricht ablegen
    - Signal (w->work)

Worker (deblockiert an w->work)
- user-list = rusers();

- Message.state = return
- EndCall
    - Send (faui47e:4711, user-list) mit ID=4057
    - aufräumen
    - Wait (w->work)

Distributor auf faui47e

- Receive (status, Message)
- DoReturn (Message)
    - Signal (c->received)

Client auf faui47e:

- Übernahme der Ergebnisse des Fernaufrufs
- Aufräumarbeiten

d) Wie kann der Algorithmus von Lampson verbessert werden, um die Mehrfachausführungen von Aufrufen zu verringern?

Das Problem liegt darin, daß die Ergebnisse nach dem erfolgreichen Senden des Ergebnisses sofort verworfen werden. Wenn sich die Antwort-Nachricht mit einer wiederholten Anfrage auf dem Netz kreuzen, würde der Client zwar die Antwort-Nachricht verwerfen (relevant für die last of many-Semantik!), der Server müsste jedoch die Anfrage vollständig neu berechnen.

- Eine Verbesserung kann dadurch erreicht werden, daß der Server das Ergebnis der Aktion noch eine gewisse Zeit aufbewahrt. Erst nach Ablauf eines Timeouts werden die Daten gelöscht. Mehrfachausführungen von Aufträgen könen dadurch verringert werden. Da jedoch Wiederholungen auch nach dem Timeout noch beim Server ankommen können, ist eine Mehrfachausführung nicht ausgeschlossen.
- Bei theoretisch unbegrenztem Speicherplatz können die Ergebnisse beim Server so lange aufgehoben werden, bis der selbe Client-Prozeß einen neuen, anderen Fernaufruf startet. Dann kann der Server sicher sein, daß seine Daten angekommen sind und er kann sie verwerfen. Dazu muß jedoch der Algorithmus zur Identifikation des Client-Prozesses erweitert werden; außerdem entsteht die Frage, was mit den Ergebnissen des letzten Fernaufrufes eines Prozesses geschieht.