INF
Informatik

Thero. Block -3:

- **REMOTE PROCEDURE CALL (RPC)**
- **MULTICAST COMMUNICATION**

Katja Haas

Agenda

I. Remote Procedure Call (RPC)
   I. Was ist Remote Procedure Call (RPC)?
   II. Client und Server Stubs
   III. Schritte eines RPC Aufrufs
   IV. Parameterübergabe
   V. Synchrones vs. Asynchrones RPC
   VI. Verzögerter synchroner RPC

II. Multicast Communication
   I. Grundlage –Unterschied Uni- , Multi- und Broadcast
   II. IP- Multicast
   III. Systemmodell für Multicast
   IV. Basic-Multicast (B-Multicast)
   V. Reliable Multicast
      I. Reliable multicast over IP multicast
   VI. Ordered Multicast

REMOTE PROCEDURE CALL (RPC)

Was ist Remote Procedure Call (RPC)?

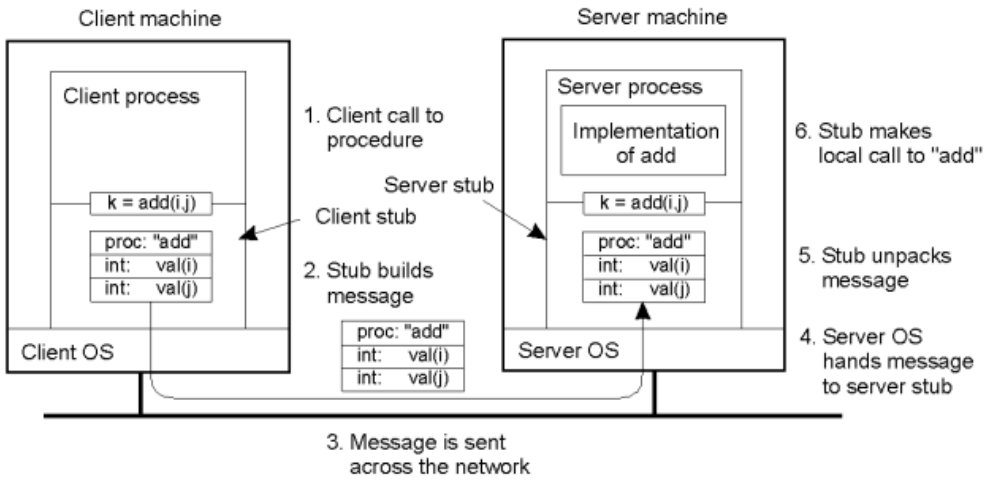
- „**Aufruf einer fernen Prozedur**“ welches vor allem für Client/Server-Anwendungen entwickelt wurde
- Konzept (entwickelt von Birrel und Nelson 1984), welches die **Interprozesskommunikation**, also den Informationsaustausch zwischen den Systemprozessen **regelt**
  - Bsp.: Prozess auf Rechner A will eine Prozedur auf Rechner B ausführen
- Basiert auf den Mechanismen eines **synchron entfernten Dienstaufruf**, der die Kontrollfluss- und Datenübergabe als Prozeduraufrufen zwischen unterschiedlichen Adressräumen über ein schmalbandiges Netz transferiert
- Zum Kommunikationsprozess via RPC gehören die **Übergabe von Parametern** und die **Rückgabe eines Funktionswertes**
- RPC **sieht wie ein lokaler Prozeduraufruf aus**, da die notwendige Netzwerkkommunikation und Ein- & Ausgabe Mechanismen verborgen bleiben

Client und Server Stubs:

- Wenn ein RPC gestartet wird, ist die entfernte Prozedur auf dem gleichen Rechner nicht vorhanden (anderer Adressraum).
  Dazu gibt es **Stubprozeduren**:
- Stubs sind lokale **Stellvertreter-Prozedure**n
- Stubs-Objekte kümmern sich um das
  - Verpacken((**Marshalling**),
  - Senden,
  - Empfangen und
  - Entpacken der Funktionsparameter (**Unmarshalling**)
- Client-Stub ist ein Stellvertreter der entfernten Server-Prozedur auf der Client Seite
- Server-Stub ist ein Stellvertreter des aufrufenden Client-Codes auf der Server Seite

Schritte eines RPC Aufrufs:

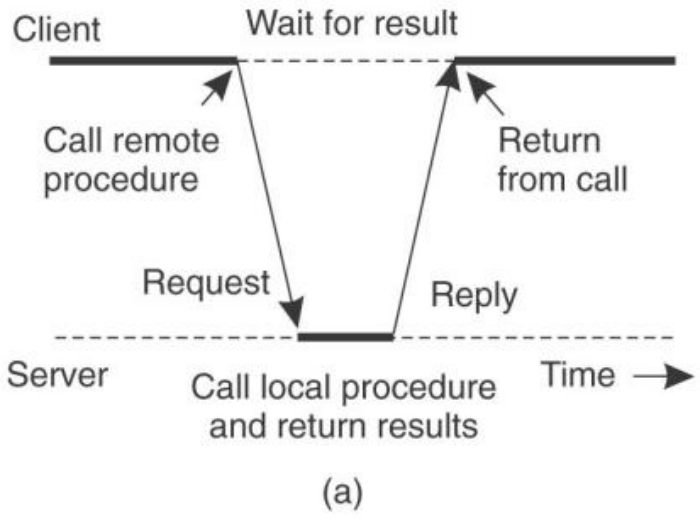
1. Client ruft Client-Stub auf
2. Client-Stub verpackt die übergebenen Parameter des Prozeduraufrufs in eine Nachricht (Marshalling) und ruft das lokale Betriebssystem (OS) auf
3. Das lokale OS sendet Nachricht an das entfernte OS (unter Nutzung eines Transportschichtprotokolls wie UDP oder TCP)
4. Entferntes OS übergibt Nachricht an Server-Stub
5. Server-Stub entpackt die in der Nachricht enthaltenen Parameter (Unmarshalling) und ruft den Server auf
6. Server verarbeitet den Aufruf und übergibt Ergebnis an Server-Stub
7. Server-Stub verpackt es in eine Nachricht und ruft das lokale OS auf
8. Lokales OS sendet Nachricht an Client OS
9. Client OS übergibt Nachricht an Client-Stub
10. Client-Stub entpackt Nachricht und gibt das Ergebnis an den Client weiter

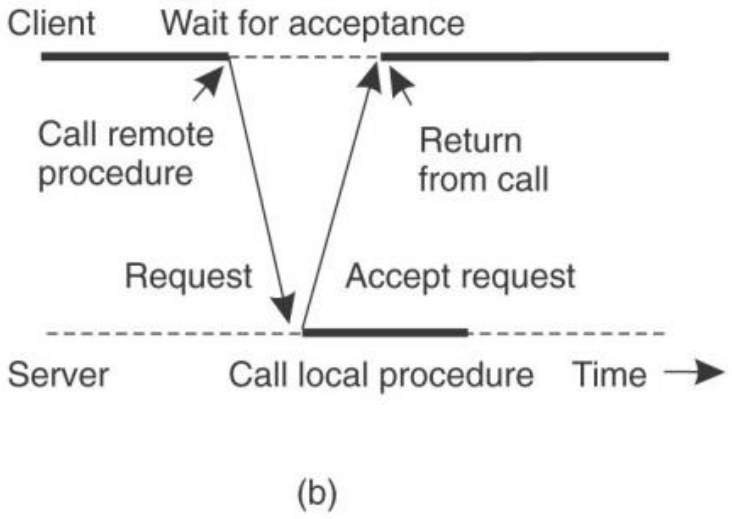


Parameterübergabe

- Die Übertragung der Parameter über das Netzwerk erfordert die Behandlung von Problemen der unterschiedlichen Datenformate und Adressierungen. Unterstützung erhält man durch das Verpacke (Marschalling) und Entpacken (Unmarshalling) der Parameter.
- **Referenz-Parameter:**
  - Bei Referenzparameter (Call by Reference) werden im lokalen Fall tatsächliche Adressen übergeben, welches bei RPC problematisch ist da eine übergeben Adresse im Server-Adressraum eine andere Bedeutung als im Client-Adressraum haben
  - Statt der Adresse muss der unter der Adresse abgelegte Wert(Wertfolge) übertragen und dann in eine entsprechende Variable im Server-Stub kopiert werden. Die Parameter müssen vom Server-Stub wieder in die Antwort-Nachricht zurückkopiert und zurück zum Client übertragen werden. In diesem Fall wird Call by Reference durch Call by Copy/Restore ersetzt
- **Wert-Parameter:**
  - Der übergeben Parameter wird im lokalen Fall zu einer lokalen Variable der Prozedur, die mit dem Parameterwert initialisiert ist (Call by Value) übergeben
  - Bei RPC genügt es daher nur den Wert des jeweiligen Parameters in die Nachricht aufzunehmen und zu übertragen (Achtung: Bei Arrays nicht nur Startzeiger sondern gesamte Datenstruktur übergeben)

Synchrones vs. Asynchrones RPC





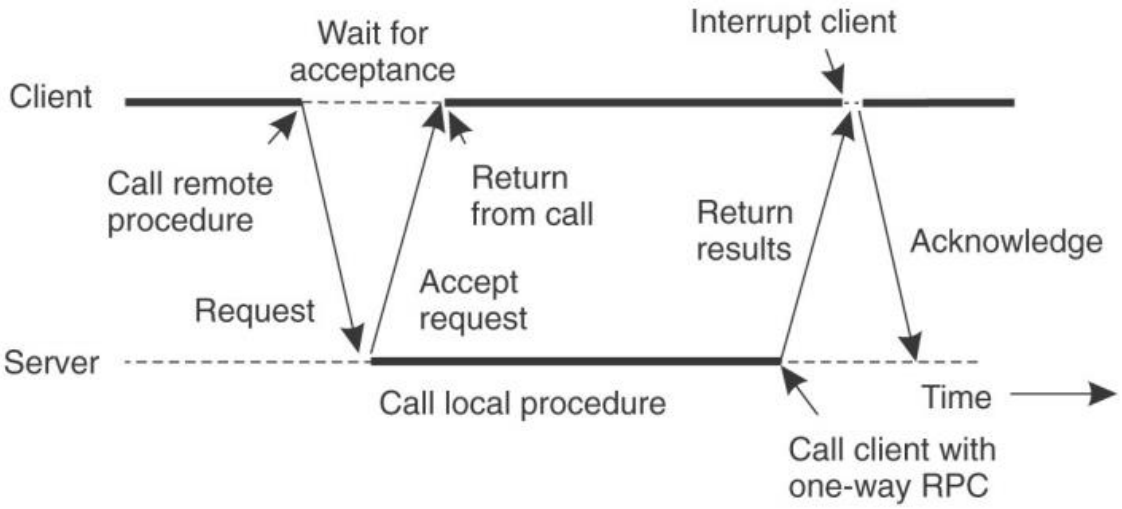
(a): synchroner RPC:

Blockierend, da der Client auf ein Ergebnis des RPCs wartet bzw. bis der Server geantwortet hat.

(b): asynchroner RPC:

Nicht Blockierend, da der Client auf eine Empfangsbestätigung des RPCs wartet, er kann andere Aufgaben bearbeitet bis der Server antwortet.

Verzögerter synchroner RPC



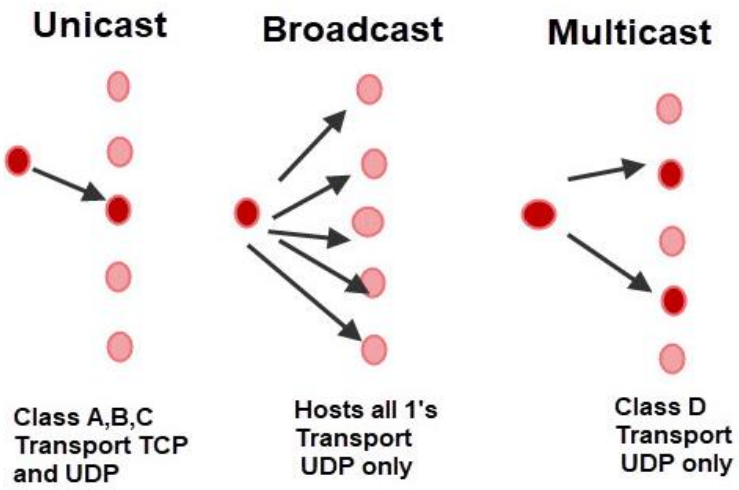
Verzögerter synchroner RPCS besteht aus zwei asynchronen PRCs:

- Beim ersten asynchronen RPC wird eine Anfrage vom Client an den Server gesendet
- Beim zweiten asynchronen PRC wird das Ergebnis vom Server an den Client gesendet

Beide Asynchrone RPCs ergeben einen verzögerten RPC da das Ergebnis verzögert gesendet wird

MULTICAST COMMUNICATION

Grundlage –Unterschied Uni- , Multi- und Broadcast



- **Unicast**: Punkt zu Punkt- Übertragung. Die Daten werden von einem Endpunkt über eventuelle Knoten zu einem anderen Endpunkt übertragen.

  Broadcast: Rundsendung: Die Daten werden von einem Endpunkt an alle Endpunkte in einer Broadcast-Domain verteilt. Der Empfänger muss dann entscheiden ob er die erhaltenen Daten verarbeiten möchte oder nicht.

  Multicast: Punkt zu Mehrpunkt-Übertragung. Die Daten werden von einem Endpunkt gesendet und von den Knotenpunkten an diejenigen Empfänger verteilt, welche die Daten angefordert haben. Die Knotenpunkte sind für die Verteilung der Daten zuständig.

IP- Multicast

- Implementierung einer **Gruppenkommunikation**
- ermöglicht dem Sender IP Pakete (sind an Computer adressiert) **an viele Empfänger, die eine Multicast-Gruppe bildet** zur gleichen Zeit zu senden ohne die Identität der einzelnen Empfänger und die Größe der Gruppe zu kennen
- Eine Multicast-Gruppe wird durch eine **Internetadresse der Klasse D** spezifiziert, d.h. eine Adresse, deren erste 4 Bits in 1110 in IPv4 (Adressbereich:224.0.0.0 – 239.255.255.255) sind
- Die Mitgliedschaft in Multicast-Gruppen ist **dynamisch**, jeder Computer kann jederzeit ein- oder austreten und einer beliebigen Anzahl an Gruppen beitreten & es ist möglich Datagramme über eine Multicast-gruppe zu senden ohne Mitglied zu sein
- wird durchgeführt indem es **UDP-Datagramme** mit Multicast Adressen und Port-Nummern sendet
- durch das erstellen eines Sockets können Nachrichten der Gruppe empfangen werden
- **Failure model**:
  - IP-Multicast arbeitet UDP-basiert und somit ist die Nachrichtenzustellung nicht garantiert, da UDP bestätigte, verbindungslose Kommunikation verwendet d.h.
    - **Ommision failures**: Einige aber nicht alle Mitglieder können eine Nachricht erhalten
    - **Unrelibale multicast**: IP-Pakete kommen nicht in Absenderreihenfolge an & Gruppenmitglieder können Nachrichten in unterschiedlicher Reihenfolge empfangen

Systemmodell für Multicast

- Das System besteht aus einer Sammlung von Prozessen, die zuverlässig über 1-1 Kanäle kommunizieren können
- Prozesse können in einer Gruppe enthalten sein, die Multicast-Nachrichten empfangen können
- Im Allgemeinen können Prozesse zu mehr als einer Gruppe gehören (Prozesse können dadurch Informationen aus mehreren Quellen empfangen)
- **Operationen:**
  - *multicast(g,m)* sendet Nachricht m an alle Mitglieder der Prozessgruppe g
  - *deliver (m)* ,der eine per multicast gesendete Nachricht an den aufrufenden Prozess übermittelt
- Multicast-Nachricht m enthält die ID des Absender Prozess *sender(m*) und die ID der Empfängergruppe *group(m).*
- Wir gehen davon aus, dass es keine Verfälschung des Ursprungs und des Ziels von Nachrichten gibt

Basic-Multicast (B-Multicast)

- sichert zu, im Gegensatz zu IP Multicast, dass ein korrekter Prozess die Nachricht schließlich ausliefert, solange der Multicaster nicht abstürzt
- Eine verlässliche Unicast-Send-Operation (one-to-one send operation) wird für die Implementierung verwendet:
  - To *B-multicast(g,m):* for each process *p ϵ g, send(p,m);*
  - On *receive(m)* at p: *B-deliver(m)* at *p*.
- Nachteil:
  - Bei großen Anzahl von Prozessen kann das Protokoll unter einer „*ack-implosion*“ leiden.
  - Die Acknowledgements, die gesendet werden können von vielen Prozessen gleichzeitig ankommen, sodass sich der Puffer des Multicasting-Prozesses schnell füllt.
    - Gefahr, dass diese verloren gehen
    - Auslastung der Netzwerkbandbreite

    →Eine praktische Implementierung von Basic Multicast kann mithilfe von über IP-Multicast umgesetzt werden

Reliable Multicast

- Kriterien für ein zuverlässigen Multicast sind:

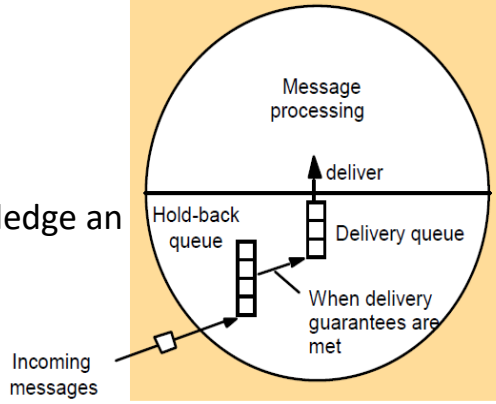
  - ***Integrity****:* Ein korrekter Prozess p liefert eine Nachricht m höchstens einmal aus (at-most-once-delivery). Darüber hinaus ist
    p ϵ group(m) und m wurde von *sender(m)* durch den Multicast-Vorgang verschickt.

  - ***Validity****:* Wenn ein korrekter Prozess die Nachricht *m* per multicast versendet oder empfängt, dann wird *m* garantier ausgeliefert

  - ***Agreement****:* Wenn ein korrekter Prozess Nachricht *m* ausliefert, dann wird *m* auch allen anderen korrekten Prozesse in *group(m)* zugestellt

Reliable multicast over IP multicast

- **Grundidee:**
  - Protokoll geht davon aus, dass die Gruppen geschlossen sind.
    Es verwendet:
    - Piggybacked acknowledgements (acknowledgments die an anderen Nachrichten angehängt sind)
    - Negative acknowledgements (wenn Nachrichten versäumt werden)
  - Jeder Prozess *p* verwaltet
    - Sequenznummer *S(p,g)* für jede *group(g)* zu der er gehört
    - Sequenznummer *R(q,g)*, der letzten Nachricht, die er vom Prozess *q* geliefert bekommen hat und die an Gruppe g gesendet wurde
  - Wenn p eine Nachricht *m R-multicastet* dann:
    - piggyback S(p,g) und gebe acknowledgements für empfangene Nachrichten in folgender Form <q,R(q,g)>
    - IP multicast die Nachricht an g, und erhöhe die Sequenznummer S(p,g) um 1
  - Vorgehen beim Empfang einer Nachricht von q mit Sequenznummer S gilt:
    - Wenn S=R(p,g) +1: R-deliver die Nachricht und erhöht R(p,g) um 1
    - Wenn S<= R(p,g): verwerfe die Nachricht, da diese schon empfangen wurde
    - Wenn S> R(p,g)+1 oder wenn R>R(q,g), for enclosed acknowledgement<q,R>:
      - dann wurde Nachricht verpasst und fordere sie mit einer negative acknowledge an
      - stelle neue Nachrichten in die Warteschlange für eine spätere Zustellung



Ordered Multicast

- **FIFO-ordering:**
  - Wenn ein korrekter Prozess zunächst *multicast(g, m)* und dann *multicast(g, m‘)* ausführt, dann liefert jeder korrekte Prozess aus g, der *m‘* ausliefert, *m* vor *m‘* aus
    - lokale Senderreihenfolge wird garantiert

- **Causal-ordering:** Wenn *multicast(g, m)* → *multicast(g, m‘)* mit → als „happens-before“-Operator als logische Reihenfolge gilt, dann liefert jeder korrekte Prozess aus g, der *m‘* ausliefert, *m* vor *m‘* aus
  - garantier die Zustellung nach der Reihenfolge, die durch die Relation „→“ festgelegt wird

- **Total-ordering:** Wenn ein korrekter Prozess eine Nachricht *m* vor einer Nachricht *m‘* ausliefert, dann liefert jeder korrekte Prozess aus g, der *m‘* ausliefert, *m* vor *m‘* aus
  - garantiert die Empfangsreihenfolge über alle Prozesse der Gruppe
    Bsp.: wenn 2 Prozesse an 2 andere Prozesse etwas senden, dann wird die Nachricht bei beiden in gleicher Reihenfolge zugestellt

Quellen:

- **Buch**: Distributed Systems – Concept and Design
  - George Colouris, Jean Dollimore, Tim Kindberg, Gordon Blair
  - ISBN 10: 0-273-76059-9
  - ISBN 13: 978-0-273-76059-7

- **Buch**: Distirbuted Systems – Principles and Paradigms
  - Andrew S. Tanenbaum, Martin Van Stehen
  - ISBN 10: 1-292-02552-2
  - ISBN 13: 978-1-292-02552-0

Vielen Dank