

Aufgabenblatt 2

Musterlösung

Vorlesung Verteilte Systeme

Wintersemester 2025/26

Aufgabe 1: Kommunikationsformen in verteilten Systemen

Bei gesicherten Nachrichtenströmen wird in der Regel zwischen zwei Kommunikationspartnern eine virtuelle Verbindung aufgebaut, über die dann der Datenaustausch erfolgt. Dabei ist sichergestellt, dass keine Pakete verloren gehen (es sei denn, die Verbindung bricht zusammen, was aber dann von beiden Partnern festgestellt wird). Weiterhin werden die Pakete in genau der Reihenfolge empfangen, wie sie auch gesendet wurden. Die Verbindung verhält sich im Wesentlichen wie eine Pipe.

Bei Datagrammen hingegen handelt es sich um einzelne, voneinander unabhängige Pakete. Diese können bei Problemen auf darunterliegenden Netzwerkschichten verlorengehen, ohne dass eine von beiden Seiten dies merkt. Auch über die Paketreihenfolge gibt es keine Garantien.

Die Vorteile der gesicherten Verbindung sind offensichtlich. Mögliche Einsatzgebiete sind die Übertragung größerer Datenmengen (z.B. ftp) oder generell alle Arten von Verbindungen, die sitzungsorientiert ablaufen (z.B. telnet, ssh).

Auf Datagramme greift man zurück, wenn man den zusätzlichen Overhead einsparen will und mit verlorengegangenen und in der Reihenfolge vertauschten Paketen zurechtkommen kann. Weiterhin sind mit Datagrammen Multicast-Verbindungen möglich. Ein Einsatzgebiet ist z.B. Webradio. Dabei sorgen auf dem Datagrammdienst aufsetzende Protokolle dafür, dass eine Reihenfolgevertauschung und ein Paketverlust erkannt wird. Einzelne verlorengegangene Pakete können hier in der Regel kompensiert werden, indem der Audiostrom interpoliert wird. I.A. verwenden alle Echtzeit-Protokolle, bei denen eine geringe Latenzzeit wichtiger ist als die Vollständigkeit der Daten, Datagramme. Ein weiteres Beispiel sind hier Positionsübertragungen bei Multiuser-Spielen.

Aufgabe 2: Datenformat-Konvertierung

Die Daten müssen deshalb konvertiert werden, weil ein verteiltes System im allgemeinen heterogen aufgebaut sein kann. Daher kann die interne Darstellung der Daten auf den einzelnen Knoten des Systems unterschiedlich sein. Die wichtigste Art der Heterogenität ist hier der Einsatz unterschiedlicher Prozessoren und Betriebssysteme. Damit die verschiedenen Prozesse sich trotzdem verstehen, werden die Daten vor dem Verschicken in der Regel in ein gemeinsames (kanonisches) Netzwerk-Datenformat konvertiert und beim Empfänger entsprechend wieder zurückgewandelt.

Wichtige Aspekte für verschiedene Datentypen:

- *Ganzzahltypen*: Das häufigste Problem hier ist das der *Byte Order*, also der Reihenfolge, in der die einzelnen Bytes eines größeren Ganzzahltyps im Speicher abgelegt sind. Man unterscheidet hier *big endian* und *little endian*. Würde keine Konvertierung durchgeführt, so würde die (vorzeichenlose) *big endian* 16-Bit-Zahl 3 auf einem *little endian*-System zur Zahl 49152. Hier reicht also in der Regel, sich auf eines der beiden Formate zu einigen und ggf. zu konvertieren.
- *Floating-Point*: Hier ist die Lage etwas komplizierter, da Gleitkommazahlen nicht so einfach aufgebaut sind. Wesentliche Probleme sind hier unterschiedliche Bit-Anzahlen bei Mantisse oder Exponent. Weiterhin können die Zahlen zu unterschiedlichen Basen abgespeichert sein, am häufigsten sind Zahlen zur Basis 2 und 10. I.A. ist keine exakte 1-zu-1-Abbildung aller Fließkommazahlen eines Systems zu denen eines anderen Systems durchführbar. Heute arbeiten allerdings die meisten Rechner mit der standardisierten Darstellung nach IEEE 754.

- *Strings*: Hier ist eine Konvertierung zwischen unterschiedlichen Zeichensätzen notwendig, weil ein Byte oder Wort in verschiedenen Zeichensätzen als unterschiedliche Zeichen interpretiert wird. Manche Zeichen fehlen in einigen Zeichensätzen sogar komplett. Zudem gibt es Zeichensätze mit 8-Bit- (z.B. ASCII, ISO-8859-1), 16-Bit (z.B. Java: Unicode) und 32-Bit-Zeichen.

Aufgabe 3: Semantik der Client/Server-Kommunikation (Pflichtaufgabe, Abgabe über moodle!)

Aufgabe 4: Request/Reply-Protokoll

Es müssen die drei Operationen *doOperation*, *getRequest* und *sendReply* auf Basis von *send* und *recv* implementiert werden. Da keine Fragmentierung von Nachrichten notwendig ist, muss pro Nachricht nur genau ein Paket versendet werden. Um das Protokoll wie gefordert möglichst overheadfrei zu halten, ist die prinzipielle Idee folgende:

Für einen kompletten request-reply-Zyklus werden genau 2 Pakete geschickt, nämlich die Anfrage des Clients selbst und dann die Antwort des Servers darauf. Die Antwort dient gleichzeitig als implizite Bestätigung des Erhalts der Nachricht. Wenn innerhalb einer bestimmten Zeitspanne eine Antwort ausbleibt, wird davon ausgegangen, dass das Paket verlorengegangen ist, und es muss neu verschickt werden. Damit Anfragen und Antworten zugeordnet werden können, werden eindeutige Sequenznummern erzeugt und an die Nachrichten angehängt. Folgender Pseudo-Code-Auszug zeigt einen typischen Ablauf im Client. Dieser besorgt sich die Adresse des Servers (z.B. von einem Namensdienst) und erzeugt eine Request-Nachricht. Dann stellt er die Anfrage an den Server und verarbeitet die Antwort:

```
serveraddr = getServerAddr();
request = createRequest();
doOperation(serveraddr, request, repl);
handleReply(repl);
```

Der Server läuft typischerweise in einer Endlosschleife, in der er beliebige Anfragen entgegennimmt, diese bearbeitet und schließlich eine entsprechende Antwort zurückschickt.

```
for ( ; ; )
{
    getRequest(clientaddr, request);
    repl = handleRequest(request);
    sendReply(clientaddr, repl);
}
```

Es folgen beispielhafte Implementierungen der drei geforderten Operationen:

- a) Die *doOperation* Operation wird implementiert, indem das Paket mittels *send* verschickt wird und anschließend auf eine Antwort gewartet wird. Bleibt diese innerhalb einer bestimmten Zeit aus (hier 5s), so wird das Paket erneut gesendet. Eine ernsthafte Fehlerbehandlung sollte nach einer maximalen Anzahl von Wiederholungen aufgeben.

```
doOperation(in s_address, in request, out reply)
{
    ok = false;
    while (! ok)
    {
        send(s_address, request);
        ok = recv(s_address, reply, 5s);
    }
}
```

- b) *getRequest* ist die aufwendigste Funktion. Sie wartet auf das Eintreffen einer beliebigen Nachricht. Im Regelfall wird diese Nachricht mittels *recv* entgegengenommen und die Nachricht-Id in einem assoziativen Array (Map) gespeichert. Die Funktion kehrt dann zurück. Wird jedoch eine Anfrage wiederholt empfangen (d.h. sie ist in der Map bereits eingetragen), dann wird überprüft, ob schon eine Antwort generiert wurde. Ist dies der Fall, wird diese erneut gesendet. Wenn nicht, wird die Nachricht ignoriert.

```

getRequest(out address, out request)
{
    newrequest = false;
    while (! newrequest)
    {
        recv(address, request, -1);           // no timeout
        if (map.lookup(request.id, reply))
        {
            if (reply != 0)
                sendReply(address, reply);
        }
        else
        {
            newrequest = true;
            map.store(request.id, 0);
        }
    }
}

```

- c) *sendReply* wird umgesetzt, indem ganz einfach das Antwortpaket gesendet wird. Anschließend wird die Antwort in dem schon vorhandenen Eintrag der Map gespeichert, damit diese bei Verlust des Paketes im Netz erneut gesendet werden kann.

```

sendReply(in address, in reply)
{
    send(address, reply);
    map.store(reply.id, reply) // store reply in a map
}

```

Aufgabe 5: Middleware

Im Prinzip könnte ein zuverlässiger Multicast-Dienst einfach Teil der Transportschicht oder sogar der Netzwerkschicht sein. Bspw. wird der unzuverlässige IP-Multicasting-Dienst in der Netzwerkschicht implementiert. Weil diese Dienste jedoch momentan noch nicht zur Verfügung stehen, werden sie im Allg. unter Verwendung von Diensten auf der Transportschicht implementiert, wodurch sie automatisch in der Middleware platziert werden. Geht es jedoch um die Skalierbarkeit, kann die Zuverlässigkeit nur dann garantiert werden, wenn die Applikationsanforderungen berücksichtigt werden. Dies ist ein starkes Argument für die Implementierung solcher Dienste auf höheren, weniger allgemeinen Schichten.

Aufgabe 6: Transparenz des RPC (**Pflichtaufgabe, Abgabe über moodle!**)