Einsendeaufgabe 2

Kommunikation, Namen und Koordination

Wintersemester 2025/2026
30.11.2025

EINGEREICHT BEI
FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Kooperative Systeme
Verteilte Systeme
Prof. Dr. Christian Icking

EINGEREICHT DURCH
Emily Lucia Antosch
emilyluciaantosch@web.de
Hamburg

Contents

1. Aufgabe 2.1

Ein Client führt RPCs auf einem Einprozessorsystem aus, angesprochene Server verfügen aber über genügend Prozessorkapazitäten. Setzen wir nun folgende Zeitanforderungen voraus, um Gesamtzeiten zu berechnen: Bei jedem RPC benötigt der Client zunächst 5 Millisekunden, um den Server zu lokalisieren, auf dem der Aufruf ausgeführt werden soll. Marshaling und Unmarshaling kosten jeweils 0,5 Millisekunden, sowohl auf dem Client als auch auf dem Server. Der Server braucht 10 Millisekunden für die Bearbeitung einer Anfrage. Die lokalen Betriebssysteme beim Client und beim Server benötigen jeweils 0,5 Millisekunden Rechenzeit, um eine der Operationen send und receive auszuführen. Die Übertragung einer Nachricht zwischen den Client und Server dauert 3 Millisekunden. Andere Zeiten werden zur Vereinfachung ignoriert.

1. Wie viel Zeit benötigt die komplette Bearbeitung eines einzelnen RPCs?
2. Wie viel Zeit vergeht, wenn zwei RPCs vorliegen und der Client insgesamt nur einen Thread zur Bearbeitung verwendet?
3. Wie viel Zeit vergeht, wenn zwei RPCs vorliegen und der Client mehrere Threads dafür einsetzt? (Tipp: erst nach Aufruf von `send()` blockiert der erste Thread, und der zweite Thread fängt an zu arbeiten.)

Lösung 2.1 Lösung 1

1. Es sind
   - 5 Millisekunden, um den Server zu finden,
   - $2 \cdot 0,5$ Millisekunden für Marshaling,
   - $2 \cdot 0,5$ Millisekunden für Unmarshaling,
   - $2 \cdot 0,5$ Millisekunden für `send()`,
   - $2 \cdot 0,5$ Millisekunden für `receive()`,
   - 10 Millisekunden für Bearbeitung,
   - $2 \cdot 3$ Millisekunden für Übertragung.

   Insgesamt also 25 Millisekunden.
2. Es sind 50 Millisekunden, da der eine Thread auf das Empfangen der Antwort wartet und erst danach mit der nächsten Anfrage beginnen kann. Wenn die Serverlokalisierung bei der zweiten Anfrage noch im Cache ist, könnte sich die Zeit auf 45 Millisekunden reduzieren.
3. Bei zwei Threads können die Anfragen nach 31 Millisekunden beendet werden, da wenn der zweite Thread nach dem `send()` anfängt zu arbeiten, dann ist die Antwort für den zweiten Thread $6 = 5 + 0,5 + 0,5$ Millisekunden verzögert zu der Antwort vom ersten Thread. Wenn die Serverlokalisierung vernachlässigt werden kann, dann sind beide Anfragen nach 26 Millisekunden beendet (nur Marschaling und `send()`).

2. Aufgabe 2.2

1. Worum geht es bei der persistenten und transienten Kommunikation? Worum geht es bei der synchronen und asynchronen Kommunikation?
2. Um persistente und asynchrone Kommunikation zu implementieren, wird es message-queuing-System benötigt. Welche Probleme muss ein message-queuing-System lösen?

Lösung 2.2.1 Lösung 2

Bei persistenten Kommunikation werden Nachrichten zwischen Parteien solange gespeichert, bis dem Empfänger die Nachricht des Senders zugestellt bekommen kann. Weder Sender noch Empfänger müssen bei der Übertragung aktiv sein. Beispielsweise werden E-Mails auch zugestellt, wenn der Empfänger offline ist, während der Sender die Nachricht verschickt. Bei transienter Kommunikation wird die Nachricht nur solange gespeichert, wie Sender und Empfänger aktiv sind. Wenn bei der Übertragung der Sender inaktiv ist oder ein anderer Fehler bei der Übertragung passiert, wird die Nachricht verworfen. Websockets sind ein Beispiel für transienter Kommunikation, da hier sowohl Empfänger als auch Sender aktiv sein müssen, um eine Nachricht erfolgreich zu übertragen.

Bei synchroner Kommunikation wird der Sender solange blockiert, bis der Empfänger die Nachricht empfangen hat. Der Empfänger kann hier zwar der tatsächliche Empfänger sein, aber auch eine Form der Middleware-Schicht, die die Nachricht bis zur finalen Übertragung speichert. Bei asynchroner Kommunikation wird der Sender nicht blockiert, sondern fährt mit der Ausführung seiner Tätigkeiten fort, ohne auf eine Antwort des Empfängers der Nachricht zu warten. Der Empfänger ist hier oft eine Middleware-Schicht, die mit der Nachricht dann etwas anfangen kann (speichern oder weiterleiten).

Lösung 2.2.2 Lösung 3

Ein Message-Queuing-System muss folgende Probleme lösen:

- Naming und Adressierung
  - Eine Ziel-Queue muss mittels (im Idealfall logischer (Location Transparency)) Namen/Adressen auffindbar sein. Jeder Name muss ein eindeutiges Ziel in der Form eines (Host,Port)-Pairs haben.
  - Das Mapping muss allen Queue Managern zur Verfügung gestellt werden.
  - Das System muss skalierbar sein, um mehrere Anwendungen zu unterstützen.
- Routing
  - Das System muss in der Lage sein, von einem Ort zu einem anderen, die Nachrichten weiterzuleiten. Da häufig nicht jeder Queue-Manager mit jedem anderen direkt verbunden sein kann, müssen Router-Queue-Manager zusammen mit einem Overlay Network diese Übertragung übernehmen.
- Heterogenität der Anwendungen
  - Das System muss in der Lage sein, Anwendungen miteinander sprechen zu lassen, die unterschiedliche Formate unterstützen. Message Broker können hier zwischen den Formaten hin und her konvertieren, um die Kommunikation der Anwendungen zu ermöglichen.
- Wartung
  - Das System, inbesondere die Mappings innerhalb des Systems, müssen auf Stand gehalten werden.
- Garantien
  - Das System muss garantieren können, dass die Nachricht irgendwann in die Queue des Empfängers eingefügt wird.
- Integration von heterogenen Subsystemen
  - Das Message-Queuing-System muss in der Lage sein, Subsysteme, wie beispielsweise verteilte Datenbanken, mit in das System zu integrieren.
  - Beispielsweise müssen Nachrichten an Datenbanken in Subqueries aufgeteilt und an den entsprechenden Queue-Manager verschickt werden können.

3. Aufgabe 2.3

Betrachten Sie das Python-Programmbeispiel aus dem Buch in Fig. 4.22 auf Seite 216 (Distributed Systems, 4th edition, Version 4.02X, 2024) zum Thema Publish/Subscribe. Bringen Sie das Programm auf Ihrem Computer zum Laufen, der Subscriber (Client) soll einfach immer wieder nach der Zeit fragen. Zum Testen starten Sie einen Publisher und mehrere Subscriber gleichzeitig.

Lösung 2.3 Lösung 4

Wenn man die Funktionen `client()` und `server()` in verschiedene Dateien teilt und das Package `pyzmq` installiert, werden auch bei mehreren clients (parallel ausgeführt mittels `tmux`), auch die entsprechenden fünf Nachrichten eines einzelnen Servers ausgeteilt.

Python

```python
import zmq
import time

def server():
    context = zmq.Context()
    socket = context.socket(zmq.PUB)
    socket.bind("tcp://*:12345")
    while True:
        time.sleep(5)
        t = "TIME " + time.asctime()
        socket.send(t.encode())

server()
```

Python

```python
import zmq
def client():
    context = zmq.Context()
    socket = context.socket(zmq.SUB)
    socket.connect("tcp://localhost:12345")
    socket.setsockopt(zmq.SUBSCRIBE, b"TIME")

    for i in range(5):
        time = socket.recv()
        print(time.decode())

client()
```

4. Aufgabe 2.4

Betrachten Sie das Chord-System aus 16 Knoten in der Abbildung, wobei die Knoten mit den Bezeichnern 0, 2, 6, 8, 10 und 12 aktiv sind und durch weiß gefüllte Kreise mit duchgezogenem Rand dargestellt werden.

1. Welche Nachfolger (successor) haben die Schlüssel 0, 2, 4, 6, 11 und 15?
2. Erstellen Sie die Finger-Tabellen, die für die Suche nach dem Nachfolger von Schlüssel k = 9 mit Startknoten 2 benötigt werden.
3. Welchen Pfad ausgehend von Knoten 2 nimmt die Auflösung von Schlüssel k = 9?

Lösung 2.4 Lösung 5

1. Die Schlüssel haben folgende Finger-Tabellen, die die Nachfolger definieren ($m = 4$, $16 = 2^4$):

Level (0)	Nachfolger
1	2
2	2
3	6
4	8

Level (2)	Nachfolger
1	6
2	6
3	6
4	10

Level (4 -> 2)	Nachfolger
1	6
2	6
3	6
4	10

Level (6)	Nachfolger
1	8
2	8
3	10
4	0

Level (11 -> 10)	Nachfolger
1	12
2	12
3	0
4	2

Level (15 -> 12)	Nachfolger
1	0
2	0
3	0
4	6

2. Die Finger-Tabelle für Knoten 8 wird noch benötigt für Schüssel $k = 9$:

Level (8)	Nachfolger
1	10
2	10
3	12
4	0

3. Denn der Weg von Knoten 2 zu Schlüssel $k = 9$ ist:

Wir suchen immer den Knoten, der in der Finger-Tabelle von dem derzeitigen Knoten zu finden ist, der geringer ist als der Schlüssel (in diesem Fall Knoten 6). Von Knoten 6 ausgehend finden wir Knoten 8 nach dem selben Prinzip. Da Knoten 8 sieht, das der direkte Nachfolger Knoten 10 ist, weiß Knoten 8, dass er für $(7, 10]$ verantwortlich ist. Also ist der Zielknoten Knoten 8. $2 \rightarrow 6 \rightarrow 8$.

5. Aufgabe 2.5

Betrachten wir das System von drei Prozessen P_0, P_1, und P_2 in der Abbildung unten, die jeweils über eine eigene Lamport-Uhr verfügen. Beim Prozess P_i haben sich die Events e_{ij} für $0 \leq i \leq 2$ und $1 \leq j \leq 12$ ereignet, wobei ein Pfeil von e_{ij} nach e_{kl} das Absenden einer Nachricht von Prozess P_i und das Empfangen beim Prozess P_k bedeutet. Nach jedem Empfangsereignis e_{kl} gibt es ein Auslieferungsereignis $e_{k(l+1)}$. Wir nehmen an, dass alle Uhren mit dem Wert 0 initialisiert sind.

1. Geben Sie jedem Ereignis einen Zeitstempel nach dem Algorithmus von Raynal und Singhal.
2. Geben Sie für jedes Ereignis einen Vektorzeitstempel an.

Lösung 2.5.1 Lösung 6

Ereignis	Zeitstempel
e_{01}	(1,0)
e_{02}	(2,0)
e_{03}	(3,0)
e_{04}	(5,0)
e_{05}	(6,0)
e_{06}	(7,0)
e_{07}	(8,0)
e_{08}	(15,0)
e_{09}	(16,0)
e_{11}	(2,1)
e_{12}	(3,1)
e_{13}	(4,1)
e_{14}	(5,1)
e_{15}	(6,1)
e_{16}	(7,1)
e_{17}	(8,1)
e_{18}	(9,1)
e_{19}	(10,1)
e_{110}	(11,1)
e_{111}	(12,1)
e_{112}	(13,1)
e_{21}	(1,2)
e_{22}	(2,0)
e_{23}	(3,0)
e_{24}	(4,0)
e_{25}	(12,0)
e_{26}	(13,0)
e_{27}	(14,0)

Ereignis	**Zeitstempel**
Initial	(0, 0, 0), (0, 0, 0), (0, 0, 0)
e_{01}	(1, 0, 0)
e_{02}	(2, 0, 0)
e_{03}	(3, 0, 0)
e_{04}	(4, 3, 1)
e_{05}	(5, 3, 1)
e_{06}	(6, 3, 1)
e_{07}	(7, 3, 1)
e_{08}	(8, 10, 7)
e_{09}	(9, 10, 7)
e_{11}	(0, 1, 1)
e_{12}	(0, 2, 1)
e_{13}	(0, 3, 1)
e_{14}	(3, 4, 1)
e_{15}	(3, 5, 1)
e_{16}	(3, 6, 3)
e_{17}	(3, 7, 3)
e_{18}	(6, 8, 3)
e_{19}	(6, 9, 3)
e_{110}	(6, 10, 3)
e_{111}	(7, 11, 3)
e_{112}	(7, 12, 3)
e_{21}	(0, 0, 1)
e_{22}	(0, 0, 2)
e_{23}	(0, 0, 3)
e_{24}	(0, 0, 4)
e_{25}	(6, 10, 5)
e_{26}	(6, 10, 6)
e_{27}	(6, 10, 7)