Prüfung Verteilte Softwaresysteme

Datum	:	16.07.2019, 16:30 Uhr
Bearbeitungszeit	:	90 Minuten
Prüfer	:	Prof. Dr. Alf Zugenmaier, Prof. Dr. Oliver Braun
Hilfmittel	:	Keine
Erreichbare Punkte	:	96

Name: _____

Vorname: _____

Matrikelnummer: _____ Studiengruppe: _____

Hörsaal: _____ Platz Nr.: _____

Unterschrift: _____

Bitte kontrollieren Sie, ob Sie eine vollständige Angabe mit 4 Aufgaben auf 13 Seiten erhalten haben.

Aufgabe	1	2	3	4	Summe
max. Punkte	20	24	30	22	96

Anmerkungen:

- Schreiben Sie die Lösungen in die dafür vorgesehenen Kästchen. Sollte Ihnen der Platz dabei nicht reichen, benutzen Sie die Rückseite **und vermerken Sie das im dazugehörigen Kästchen!**
- Lösen Sie die Klammerung **nicht** auf und entfernen Sie keine Blätter.

Aufgabe 1 (20 Punkte)

(a) Erläutern Sie kurz unter welchen Voraussetzungen wir von einer *verteilten* Anwendung sprechen und beschreiben Sie eine mögliche Architektur. (4)

(b) Nennen Sie zwei (der vier) *Self*-Eigenschaften des *Autonomic Computing* und erklären Sie sie kurz. (4)

(c) Nennen sie zwei Transparenzeigenschaften, welche die Verteilung zur Leistungssteigerung und Fehlertoleranz ausnutzen und erklären Sie diese kurz. (4)

(d) Grenzen Sie die *parallele Verarbeitung* und *nebenläufige Verarbeitung* voneinander ab. (4)

(e) Nennen Sie zwei Probleme die durch die Offenheit eines verteilten Systems entstehen können und erläutern Sie diese kurz. (4)

Aufgabe 2 (24 Punkte)

(a) Gegeben sei folgender Go-Code unter Nutzung des Proto.Actor-Frameworks. (4)

```
type Actor0 struct{}

func (Actor0) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case string:
        fmt.Printf("Received: %v\n", msg)
        num, err := strconv.Atoi(msg)
        if err != nil {
            context.Respond(err)
        } else {
            context.Respond(num)
        }
    }
}
```

Erläutern Sie was der `Actor0` macht und wie er kommuniziert.

(b) Gegeben sei dazu folgende `main`-Funktion in der Datei `main.go`:

```
func main() {
    if len(os.Args) <= 1 {
        os.Exit(1)
    }
    rootContext := actor.EmptyRootContext
    // Stage 0
    props0 := actor.PropsFromProducer(
        func() actor.Actor { return &Actor0{} })
    actor0PID := rootContext.Spawn(props0)
    vals := make([]int, 0)
    for _, str := range os.Args[1:] {
        result, err := rootContext.RequestFuture(actor0PID,
            str, time.Second).Result()
        if err == nil {
            if val, ok := result.(int); ok {
                vals = append(vals, val)
            }
        }
    }
    fmt.Printf("Result Stage0: %v\n", vals)
    // Stage 1 kommt erst in der nächsten Teilaufgabe dazu
}
```

Geben Sie was bei folgenden Aufrufen ausgegeben wird.

i. `go run main.go 123 12` (2)

ii. `go run main.go zwölf` (2)

iii. `go run main.go 123 hallo 12` (2)

(c) Gegeben sei außerdem folgender Go-Code für einen weiteren Actor:

```
type Actor1 struct{}

func (Actor1) Receive(context actor.Context) {
    switch msg := context.Message().(type) {
    case []int:
        fmt.Printf("Received %v\n", msg)
        if len(msg) <= 1 {
            context.Respond(msg)
        } else {
            vals := []int(msg)
            future1, future2 := stage1Future(vals[:len(vals)/2]),
                stage1Future(vals[len(vals)/2:])
            result1, err1 := future1.Result()
            result2, err2 := future2.Result()
            if err1 == nil && err2 == nil {
                s1 := result1.([]int)
                s2 := result2.([]int)
                result := make([]int, 0)
                for {
                    if len(s1) == 0 {
                        result = append(result, s2...)
                        break
                    }
                    if len(s2) == 0 {
                        result = append(result, s1...)
                        break
                    }
                    if s1[0] <= s2[0] {
                        result = append(result, s1[0])
                        s1 = s1[1:]
                    } else {
                        result = append(result, s2[0])
                        s2 = s2[1:]
                    }
                }
                context.Respond(result)
            } else {
                context.Respond(vals)
            }
        }
    }
}
```

sowie die folgende Funktion:

```
func stage1Future(vals []int) *actor.Future {
    rootContext := actor.EmptyRootContext
    props := actor.PropsFromProducer(func() actor.Actor {
        return &Actor1{}
    })
    actorPID := rootContext.Spawn(props)
    return rootContext.RequestFuture(actorPID, vals,
        (time.Duration(len(vals))+1)*time.Second)
}
```

Die `main`-Funktion wird an der Stelle `Stage 1` wie folgt erweitert:

```
    // Stage 1
    future := stage1Future(vals)
    result, err := future.Result()
    if err != nil {
        panic(err)
    }
    fmt.Printf("Result Stage1: %v\n", result)
```

i. Beschreiben Sie kurz was der `Actor1` in seiner `Receive`-Methode macht. (6)

ii. Wie heisst der Algorithmus der damit umgesetzt wird. (1)

iii. Was wird als letzte Zeile beim Aufruf (2)
`go run main.go 123 hallo 12 welt 56 1 ! 19`
ausgegeben

iv. Leiten Sie her und geben Sie an, was als letzte Zeile beim Aufruf (5)
`go run main.go 123 hallo 12 welt 56 1 ! 19`
ausgegeben wird, wenn der Actor, der das Slice `[123]` bearbeiten soll, nicht antwortet.

Aufgabe 3 (30 Punkte)

Gegeben sei folgender Go-Code:

```
type channels []chan int

type process struct {
    id   int
    time int
}

func main() {
    chs := make([]chan int, 0)
    for i := 0; i < 11; i++ {
        chs = append(chs, make(chan int))
    }
    go p1(chs)
    go p2(chs)
    go p3(chs)
    go p4(chs)
    fmt.Scanln() // wait for enter key
}
```

Außerdem 4 "Prozesse" `p1`, `p2`, `p3` und `p4` die mit Hilfe der Lamportzeit synchronisiert werden sollen. Die 4 Prozesse sind als Go-Funktionen implementiert und auf der nächsten Seite zu sehen.

Die 4 Prozesse können einen Bearbeitungsschritt durchführen (`doSomething`), eine Nachricht senden (`sendTo`) oder empfangen (`receiveFrom`).

(a) Erläutern sie kurz was in den Funktionen `doSomething`, `sendTo` und `receiveFrom` jeweils mit den Prozesszeiten passieren muss. Was wird im Channel übertragen? (4)

(b) Geben Sie auf den Strichen die Zeit an, die der jeweilige Prozess **nach** dem in der selben Zeile davor stehenden Schritt hat, wenn die `main`-Funktion ausgeführt wird. (Lassen Sie noch Platz für die Vektoruhren aus Teilaufgabe d.) (11)

```
func p1(chs channels) {
    p := &process{id: 1}
    p.sendTo(chs[0])       // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.receiveFrom(chs[5]) // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.receiveFrom(chs[7]) // p.time = _____________________________
}
func p2(chs channels) {
    p := &process{id: 2}
    p.receiveFrom(chs[2]) // p.time = _____________________________
    p.receiveFrom(chs[3]) // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.receiveFrom(chs[6]) // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.sendTo(chs[1])       // p.time = _____________________________
}
func p3(chs channels) {
    p := &process{id: 3}
    p.receiveFrom(chs[9])  // p.time = _____________________________
    p.sendTo(chs[2])       // p.time = _____________________________
    p.receiveFrom(chs[0])  // p.time = _____________________________
    p.sendTo(chs[3])       // p.time = _____________________________
    p.sendTo(chs[4])       // p.time = _____________________________
    p.sendTo(chs[5])       // p.time = _____________________________
    p.receiveFrom(chs[10]) // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.sendTo(chs[6])       // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.sendTo(chs[7])       // p.time = _____________________________
    p.sendTo(chs[8])       // p.time = _____________________________
}
func p4(chs channels) {
    p := &process{id: 4}
    p.sendTo(chs[9])       // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.receiveFrom(chs[4]) // p.time = _____________________________
    p.sendTo(chs[10])      // p.time = _____________________________
    p.doSomething()        // p.time = _____________________________
    p.receiveFrom(chs[1]) // p.time = _____________________________
    p.receiveFrom(chs[8]) // p.time = _____________________________
}
```

(c) **Beschreiben** Sie was Sie ändern müssten (insbesondere in `doSomething`, `sendTo` und `receiveFrom`), wenn Sie statt der Lamportzeit Vektoruhren verwenden wollen? (4)

(d) Tragen Sie im hinter den Lamportzeiten bei Teilaufgabe b auch die Vektoruhren ein, die, mit entsprechend geändertem Code, **nach** den jeweiligen Schritten dann im Prozess gespeichert sind. (11)

Aufgabe 4 (22 Punkte)

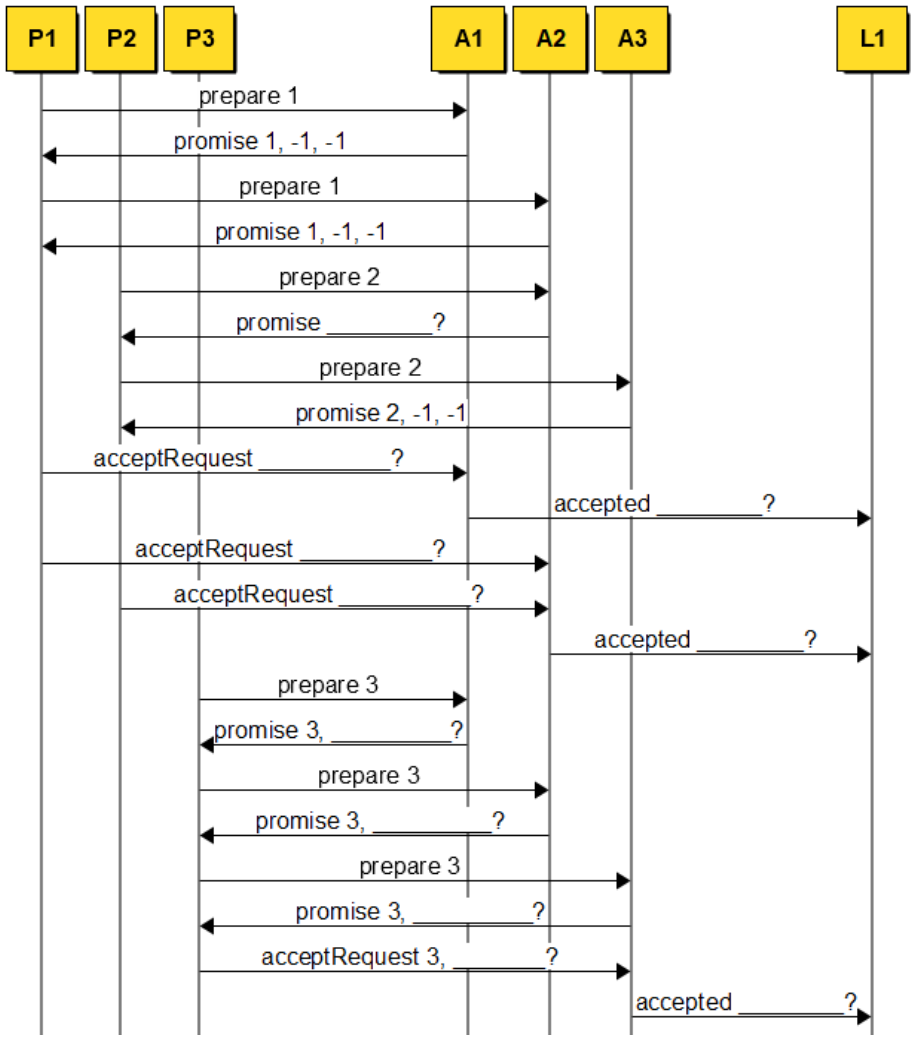
(a) Welche Problemstellung wird mit Paxos gelöst? (2)

(b) Angenommmen, es gibt (11)

- 3 Proposer: P1, P2, P3,
- 3 Akzeptoren: A1, A2, und A3, und
- 1 Learner: L1.

Zu Beginn des Protokolls besteht folgende Situation: P1 möchte den Wert a vorschlagen, P2 den Wert b und P3 den Wert c.

Tragen Sie im MSC die Werte auf den mit ? gekennzeichneten Linien ein.



(c) Welchen Wert lernt L1? (1)

(d) Welcher Fehler könnte auftreten, wenn die Proposer nicht auf eine Mehrheit an Promises warten? (4)

(e) Welcher Fehler könnte auftreten, wenn der Learner nicht auf eine Mehrheit an accepted Nachrichten wartet? (4)