

Verteilte Systeme – Übung

Aufgabe 6

Sommersemester 2024

Harald Böhm, Laura Lawniczak, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Informatik 4 (Systemsoftware)

<https://sys.cs.fau.de>



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Aufgabe 6

Aufgabe 6

Übungsaufgabe 6: Überblick

- Verteilte Synchronisation mittels Lamport-Lock-Protokoll (für alle)
 - Sperrobjekt: Blockieren/Deblockieren und Umwandlung von lokalen Sperranfragen in Ereignisse für Lamport-Lock-Protokollkomponente

```
public class VSLamportLock {  
    public void lock();  
    public void unlock();  
}
```

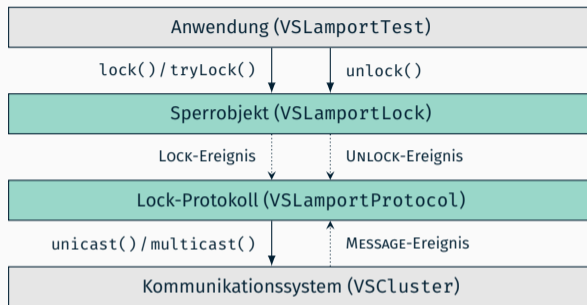
- Implementierung der Lamport-Lock-Protokollkomponente

```
public class VSLamportProtocol {  
    public void init();  
    public void event(VSLamportEvent event);  
}
```

- Zeitbeschränkte Sperrversuche (optional für 5,0 ECTS)
 - Spezifizierbare, maximale Wartedauer
 - Schnittstellenerweiterung

```
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
```

Zusammenspiel der Komponenten



- Bereitgestellte *Test-Anwendung* mit 4 Testfällen
- Zu implementierende Lock-Protokoll-Logik
 - **Benutzerschnittstelle** (Sperrobject)
 - **Protokollschicht** (Lock-Protokoll)
- Bereitgestelltes *Kommunikationssystem*: Klasse zum zuverlässigen Senden von Nachrichten an bestimmte (`unicast()`) oder alle Prozesse (`multicast()`) im Verbund (Cluster)

- Implementierung in zwei Teilen
 - Benutzerschnittstelle (`VSLamportLock`)
 - Protokollschicht (`VSLamportProtocol`)
- Bietet Anwendungen blockierenden `lock()`-Aufruf und `unlock()`-Aufruf zum Entsperren
- Implementierung des blockierenden Verhaltens durch lokalen Semaphore
- Interaktion mit Protokollschicht erfolgt mittels der übergebenen `VSLamportProtocol`-Referenz im Konstruktor von `VSLamportLock`

```
public class VSLamportLock {  
    public VSLamportLock(VSLamportProtocol protocol) { [...] }  
}
```

- Koordinierung von Ressourcenzugriffen
- Zentrale Methoden:

```
Semaphore(int permits);  
  
void acquireUninterruptibly();  
boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException;  
void release();
```

`Semaphore()` Initialisiert Semaphore mit Startwert

`acquireUninterruptibly()` Semaphore **un**unterbrechbar belegen

`tryAcquire()` Semaphore unterbrechbar belegen, schlägt nach Timeout fehl

`release()` Semaphore freigeben

- Beispiel:

```
Semaphore s = new Semaphore(1);  
s.acquireUninterruptibly();  
[...]  
s.release();
```

- Implementierung in Klasse `VSLampportProtocol` verarbeitet Ereignisse vom Typ `VSLampportEvent` sequentiell (aus Konsistenzgründen)
 - Ereignisse haben einen Typ (`type`) und ein zugeordnetes Objekt (`content`)

```
public class VSLampportEvent {  
    [...]  
    public VSLampportEventType getType() { return type; }  
    public Object getContent() { return content; }  
}
```

- Trennung Protokoll-interner Ereignisse von Ereignissen für die höhere Protokollschicht
- Vorgegebene Ereignistypen:

```
public enum VSLampportEventType { MESSAGE, LOCK, UNLOCK }
```

- Protokollinterner Ereignistyp: MESSAGE
 - Typen für Ereignisse aus Benutzerschnittstelle heraus: LOCK und UNLOCK
- Vorsicht beim Umgang mit Lock-Anfragen in der Warteschlange
 - Korrekte Zuordnung zwischen `lock()`-Aufrufen und den erzeugten REQUEST-Nachrichten notwendig
 - Schnell aufeinanderfolgende Lock-Anforderungen können sonst zu Problemen führen

- Vorgegebene Klasse `VSClusterImpl` implementiert die Schnittstelle des Kommunikationssystems (`VSCluster`)
 - Ausgelieferte Nachrichten/Ereignisse sind immer vom Typ `MESSAGE`
 - Jeder einzelne Lamport-Protokoll-Prozess im Verbund hat ein eigenes, lokales `VSCluster`-Objekt
 - Kommunikationssystem läuft stets in einem eigenen Thread, d. h., Ereigniszustellung erfolgt immer aus demselben Thread heraus
- Methoden der Kommunikationsschnittstelle `VSCluster`
 - ID des lokalen Lamport-Protokoll-Prozesses und #Prozesse im Verbund

```
public int getProcessID();  
public int getSize();
```

- Nachricht an einen bestimmten Prozess im Verbund senden

```
public void unicast(Serializable msg, int processID) throws IOException;
```

- Nachricht an alle Prozesse im Verbund senden

```
public void multicast(Serializable msg) throws IOException;
```

- Über `unicast()` bzw. `multicast()` gesendete Nachrichten werden sequentiell in FIFO-Reihenfolge durch die `event()`-Methode am jeweiligen Protocol-Objekt ausgeliefert

- Einfaches Testen der Implementierung durch Test-Anwendung
- Konfiguration: Zu verwendende Rechner in Datei `my_hosts` ablegen
- Ausführung: Start im CIP-Pool mit `distribute.sh`
 - 1. Parameter gibt Art des Testfalls an (siehe unten)
 - Skripte können im Basisverzeichnis der eigenen Paket-Hierarchie abgelegt werden; alternativ:
 - Explizites Spezifizieren des Basisverzeichnisses (2. Parameter, optional)
 - und ggf. (3. Parameter, optional) des Verzeichnisses von `my_hosts`
- Überprüfung: Skript `checklogs.sh` ausführen
- Verschiedene Testfälle (Mindestlaufzeit: 1 Minute)
 - Einfacher Fall (Aufruf mit Parameter `simple`)
 - Beantragen (`lock()`) und Freigeben (`unlock()`) in Schleife
 - Darf nicht stehen bleiben
 - Komplexer Fall (Aufruf mit Parameter `fancy`)
 - Gegenseitiges Umbuchen von Beträgen zwischen Konten
 - „Sum is“-Zeile darf sich nicht ändern (max. Betrag pro Rechner: 1000)
 - Darf nicht stehen bleiben
 - Debugging-Hilfe (Aufruf mit Parameter `debug`): Testet auf häufige Probleme
 - Testfälle für erweiterte Variante: siehe nächste Folie

- Basisvariante (`lock()`) würde Anwendung so lange blockieren, bis der kritische Abschnitt tatsächlich für sie freigegeben ist
- Erweiterung der Sperrobjektimplementierung um folgende Methode

```
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException;
```

- Spezifizieren einer maximalen Blockierzeit über `timeout` und `unit` (z. B. `TimeUnit.MILLISECONDS`)
 - Methode reagiert auf Unterbrechung des die Methode aufrufenden Threads mittels einer `InterruptedException`
- Zieht in der Regel Änderungen von `VSLamportLock` **sowie** `VSLamportProtocol` nach sich
 - Zwei weitere Testfälle
 - Funktionalität von einfachem (Parameter `simple-try`) und komplexem Fall (Parameter `fancy-try`) grundlegend analog zu `simple-` bzw. `fancy-`Testfall
 - `tryLock()`- statt `lock()`-Aufrufe (jeweils so lange, bis Lock vergeben wurde)
 - Dynamische Anpassung des Timeouts in Abhängigkeit von erfolgreichen und nicht erfolgreichen Aufrufen von `tryLock()`