

Replikation

Grundlagen der Replikation

JGroups

Übungsaufgabe 4



Varianten

- Aktive Replikation („Hot Standby“)
 - Alle Replikate bearbeiten alle Anfragen
 - Vorteil: Schnelles Tolerieren von Ausfällen möglich
 - Nachteil: Vergleichsweise hoher Ressourcenverbrauch
- Passive Replikation
 - Ein Replikat bearbeitet alle Anfragen
 - Aktualisierung der anderen Replikate erfolgt über Sicherungspunkte
 - Unterscheidung: „Warm Standby“ vs. „Cold Standby“
 - Vorteil: Minimierung des Aufwands im fehlerfreien Fall
 - Nachteil: Im Fehlerfall schlechtere Reaktionszeit als bei aktiver Replikation

Replikationstransparenz

- Nutzer auf Client-Seite merkt nicht, dass der Dienst repliziert ist
- Replikatausfälle werden dem Nutzer verborgen



Referenzierung von Diensten

■ Nichtreplizierter Fall

- Remote-Referenz auf einzelnen Rechner bzw. einzelnes Objekt
- Beispiel aus den Übungsaufgaben

```
public class VSRemoteReference {  
    private String host;  
    private int port;  
    private int objectID;  
}
```

■ Replizierter Fall

- Gruppenreferenz auf Replikatgruppe [Vergleiche: IOGR in FT-CORBA]
- Beispiel aus Übungsaufgabe 4

```
public class VSRemoteGroupReference {  
    private VSRemoteReference[] references;  
}
```

- Ausfallsicherung auf Client-Seite
 - Verbindung zu einem der in der Gruppenreferenz enthaltenen Replikate
 - Im Fehlerfall: Wechsel zu einem anderen Replikat



Aktive Replikation von Diensten

■ Zustandslose Dienste

- Keine Koordination zwischen Replikaten notwendig
- Auswahl des ausführenden Replikats z. B. nach Last- oder Ortskriterien

■ Zustandsbehaftete Dienste

- Replikatzustände müssen konsistent gehalten werden
- Beispiel für inkonsistente Zustände zweier Replikate R_1 und R_2
 - `post()`-Anfragen A_1 („Hallo“) und A_2 („Welt“) von verschiedenen Nutzern
 - Annahme: A_1 erreicht R_1 früher als A_2 , bei R_2 ist es umgekehrt

	R_1	Schwarzes Brett	R_2	Schwarzes Brett
< <i>init</i> >		[]	< <i>init</i> >	[]
A_1		[„Hallo“]	A_2	[„Welt“]
A_2		[„Hallo“, „Welt“]	A_1	[„Welt“, „Hallo“]

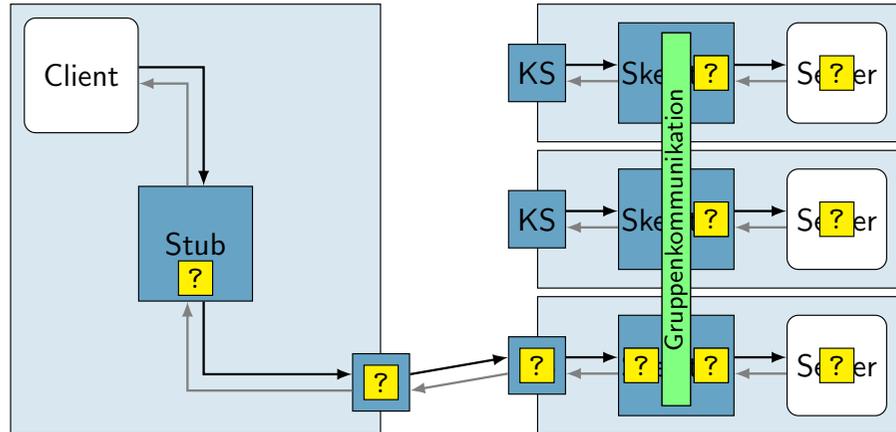
- Sicherstellung der Replikatkonsistenz
 - Alle Replikate müssen Anfragen in derselben Reihenfolge bearbeiten
 - Protokoll/Dienst zur Erstellung einer Anfragenreihenfolge nötig



Aktive Replikation von Diensten

■ Weg der Anfrage

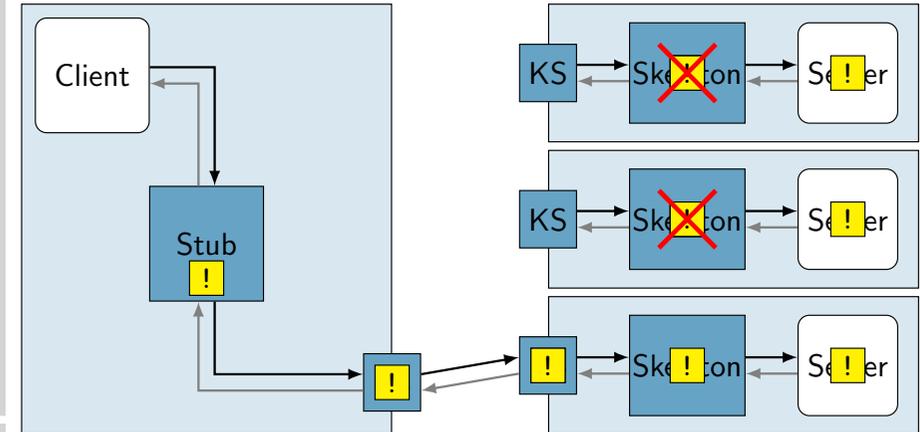
- Senden der Anfrage an ein Replikat
- Verteilen der Anfrage (z. B. durch ein Gruppenkommunikationssystem)
- Bearbeitung der Anfrage auf allen Replikaten



Aktive Replikation von Diensten

■ Weg der Antwort

- Mit dem Client verbundenes Replikat: Rückgabe der Antwort
- Alle anderen Replikate: Speichern/Verwerfen der Antwort, abhängig von der Semantik bzw. den Fehlertoleranz-Eigenschaften



Konsistenz von Replikaten

■ Voraussetzung für aktive Replikation: Anwendungsreplikate müssen dieselbe *deterministische Zustandsmaschine* realisieren

- Identische Ausgangszustände
- Identische Eingaben [→ Anfragen in derselben Reihenfolge.]

⇒ Identische Zustandsänderungen

⇒ Identische Ausgaben

■ Herausforderungen

- Mehrfädige Programme
 - Zutrittsreihenfolge von kritischen Abschnitten kann unterschiedlich sein
 - Ansatz: Zusätzliche Koordinierung zwischen Replikaten notwendig
- Nichtdeterministische Systemaufrufe
 - Beispiele: `System.currentTimeMillis()`, `Math.random()`
 - Ansatz: Einigung auf gemeinsamen Wert
- Seiteneffekte bzw. externalisierte Ereignisse
 - Beispiel: Anwendung greift auf externe Dienste zu [z. B. Rückruf am `VSBoardListener`]
 - Ansatz: Ein Replikat macht Aufruf, Ergebnisweitergabe an die anderen
- ...

Überblick

Replikation

Grundlagen der Replikation

JGroups

Übungsaufgabe 4

- Bereitstellung von *Virtual Synchrony*
 - Zusammenschluss von Knoten zu Gruppen
 - Senden von Nachrichten an die Gruppe (anstatt an jeden Knoten einzeln)
 - Alle Knoten erhalten jede
 - an die Gruppe versendete Nachricht
 - gruppeninterne Statusmeldung
 in derselben Reihenfolge
 → Alle Knoten einer Gruppe machen (vermeintlich) synchron Fortschritt
- Grundlegende Dienste
 - Membership-Service
 - Total-Ordering-Multicast
 - Zustandstransfer-Mechanismus
- Beispiele
 - **JGroups** [<http://www.jgroups.org/index.html>]
 - Spread
 - ...



View

- Aktuelle Sicht auf die Gruppe
 - Liste aller aktiven Gruppenmitglieder
- Problem
 - Keine gemeinsame Zeitbasis
 - Was bedeutet also „aktuell“?
- Lösung
 - Änderung der Gruppenzusammensetzung: Erzeugung einer neuen View
 - Aktuelle Teilnehmer einigen sich auf die neue View
 ⇒ Abfolge von Views fungiert als gemeinsame Zeitbasis
- JGroups: Klasse `org.jgroups.View`
 - Ausgabe der Gruppenmitglieder


```
Vector<Address> getMembers();
```
 - Ausgabe der Gruppengröße


```
int size();
```



- Problemstellungen
 - Zusammensetzung einer Gruppe kann dynamisch variieren
 - Knoten kommen neu hinzu
 - Knoten verlassen die Gruppe
 - Fehlersituationen
 - Verbindungsabbruch zu einzelnen Knoten
 - Gruppenpartitionierung
- Aufgabe des Membership-Service
 - Benachrichtigung aller Gruppenmitglieder über die gegenwärtige Zusammensetzung der Gruppe
- JGroups: Schnittstelle `org.jgroups.MembershipListener`
 - Benachrichtigung über Gruppenänderungen


```
void viewAccepted(View new_view);
```
 - Mitteilung eines Ausfallverdachts


```
void suspect(Address suspected_mbr);
```



Total-Ordering-Multicast

- Problemstellung
 - Clients sollen ihre Anfragen an einen beliebigen Server senden können
 - Alle Server müssen alle Anfragen in derselben Reihenfolge bearbeiten
 - Bewahrung konsistenter Server-Zustände
 - Bereitstellung konsistenter Antworten (z. B. für Fehlertoleranz)
- Total-Ordering-Multicast: Alle aktiven Knoten einer Gruppe bekommen alle Nachrichten in derselben Reihenfolge zugestellt
 - Interne Algorithmen, die
 - jeder Nachricht eine eindeutige Sequenznummer zuweisen → totale Ordnung
 - sicherstellen, dass jeder aktive erreichbare Knoten jede Nachricht erhält
 - dafür sorgen, dass jeder Knoten die Nachrichten in der richtigen Reihenfolge an die Anwendung weiter gibt
 - Hinweis
 - Jede Nachricht wird an **alle** Gruppenmitglieder zugestellt; also auch an den Knoten, der die Nachricht ursprünglich gesendet hat



Nachricht

■ Klasse `org.jgroups.Message`

- Kapselung der eigentlichen Nutzdaten
- Container für Protokoll-Header
- Konstruktoren

```
Message(Address dst);  
Message(Address dst, Address src, Serializable obj);  
[...]
```

- `dst` Zieladresse; falls `null` → alle
- `src` Ursprungsadresse; falls `null` → durch JGroups ausgefüllt
- `obj` Nutzdaten als Payload

■ Wichtigste Methoden

```
Object getObject();  
Message copy();
```

- `getObject()` Getter-Methode für Payload
- `copy()` Erzeugung einer Kopie der Nachricht



Kommunikationskanal

■ Klasse `org.jgroups.JChannel`

■ Konstruktoren

```
JChannel() // Standardkonfiguration  
JChannel(File properties) // XML-Datei  
JChannel(String properties) // Konfig. als Zeichenkette
```

■ Wichtigste Methoden

- Verbindungsaufbau zur Gruppe `cluster_name`

```
void connect(String cluster_name);
```

- Diverse Getter-Methoden

```
Address getLocalAddress() // Eigene Adresse  
String getClusterName() // Gruppenname  
View getView() // Aktuelle View
```

- Nachrichtenversand

```
void send(Message msg)  
void send(Address dst, Address src, Serializable obj)
```

Hinweis: Senden einer Nachricht an alle → `dst = null` setzen



Nachrichteneingang

■ Synchron: am `JChannel`

■ Blockierende Methode (deprecated)

```
Object receive(long timeout);
```

■ Beispiel

```
Object object = channel.receive(0); // blockierend  
if(object instanceof Message) {  
    Message msg = (Message) object;  
    [...] // Daten mittels msg.getObject() extrahieren  
} else if(object instanceof View) {  
    [...] // View behandeln  
} else {  
    [...] // Ereignis behandeln  
}
```

■ Asynchron: per `org.jgroups.MessageListener`

```
public interface MessageListener {  
    void receive(Message msg);  
    [...]  
}
```



Kombinierte Listener und Adapter

■ Kombinierte Schnittstelle: `org.jgroups.Receiver`

```
public interface Receiver extends MembershipListener,  
    MessageListener {}
```

■ Erweiterte Adapterklasse: `org.jgroups.ExtendedReceiverAdapter`

■ Implementiert (unter anderem) `Receiver`

■ Eigene `Receiver`-Klasse als Unterklasse von `ExtendedReceiverAdapter`

■ Beispiel

```
public class VSReceiver extends ExtendedReceiverAdapter {  
    public void receive(Message msg) {  
        System.out.println("received message " + msg);  
    }  
  
    public void viewAccepted(View newView) {  
        System.out.println("received view " + newView);  
    }  
}
```

■ Registrierung am `JChannel`

```
void setReceiver(Receiver r);
```



Zustandstransfer

- Problem
 - Knoten bearbeiten alle Anfragen, um ihre Zustände konsistent zu halten
 - Was ist mit Knoten, die
 - später hinzu kommen, also nicht alle Anfragen kennen
 - mit der Bearbeitung der Anfragen nicht hinterher kommen oder
 - aufgrund eines Fehlers über kaputte Zustandsteile verfügen?
- Lösung: Unterstützung von Zustandstransfers
 - Die Gruppenkommunikation sorgt dafür, dass ein Knoten (z. B. beim Gruppenbeitritt) eine Kopie des aktuellen Zustands erhält
 - Der aktuelle Zustand stammt von einem Knoten aus der Gruppe
- JGroups: zusätzliche Methoden des `MessageListener`
 - Bereitstellung des eigenen Zustands:

```
byte[] getState();
```
 - Setzen des lokalen Zustands auf `state`:

```
void setState(byte[] state);
```
- JGroups: Holen des Zustands eines anderen Replikats (`JChannel`)

```
boolean getState(Address target, long timeout);
```

Überblick

Replikation

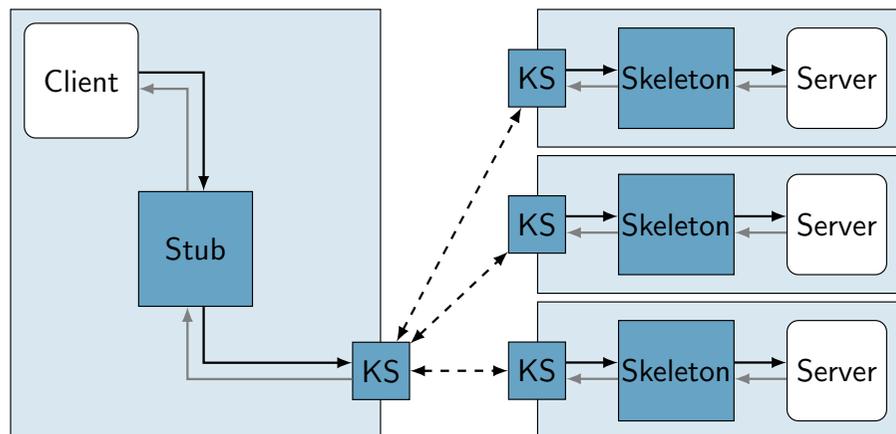
Grundlagen der Replikation

JGroups

Übungsaufgabe 4

Übungsaufgabe 4

- Replikation des Diensts
- Ausfallsicherung der Client-Seite



Replikation des Diensts

- Aktive Replikation des eigenen Fernaufrufsystems
 - Drei Replikate auf verschiedenen Rechnern
 - Alle Replikate bearbeiten alle Anfragen in derselben Reihenfolge
 - Nur eines der Replikate sendet eine Antwort zum Client
- Replikation mittels JGroups
 - JGroups-Bibliothek im Pub-Verzeichnis (`/proj/i4vs/pub/aufgabe4`)
 - Konfiguration für Total-Ordering-Multicast: Einsatz eines *Sequencer*
 - Zu verteilende Nachrichten werden (intern) an den Sequencer geschickt
 - Sequencer legt Reihenfolge der Nachrichten fest
 - Sequencer verteilt die Nachrichten an alle Replikate

```
// Kanal erstellen
JChannel channel = new JChannel();

// Erweiterung des Standard-Protokoll-Stacks um Sequencer
ProtocolStack protocolStack = channel.getProtocolStack();
protocolStack.addProtocol(new SEQUENCER());

// Receiver registrieren und Verbindung öffnen
[...]
```

Logging in JGroups

■ Konfiguration

- Granularitätsstufen: OFF, SEVERE, WARNING, INFO, FINE, FINER, ALL,...
- Konfiguration in Datei
- Programmstart

```
java -Djava.util.logging.config.file=<Datei> <Programm>
```

■ Beispiele für Konfigurationsdateien

- Ausgabe der Log-Meldungen auf der Konsole (Stufe: FINE)

```
handlers=java.util.logging.ConsoleHandler  
.level=FINE
```

- Ausgabe der Log-Meldungen in einer Datei vs.log (Stufe: INFO)

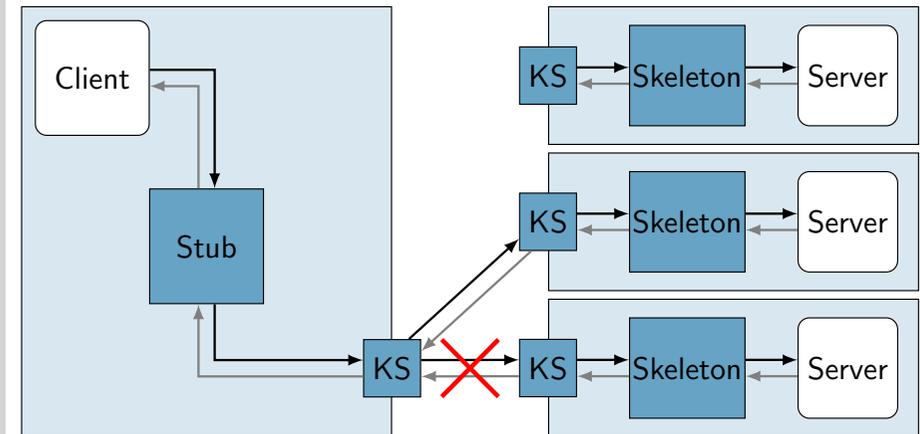
```
handlers=java.util.logging.FileHandler  
.level=INFO  
java.util.logging.FileHandler.pattern=vs.log
```



Ausfallsicherung der Client-Seite

■ Gruppenreferenz auf replizierten Dienst

- Zusammenfassung gewöhnlicher Remote-Referenzen
- Wechsel des Replikats bei Verbindungsabbrüchen



[Im Optimalfall funktionieren die RPC-Semantiken aus Übungsaufgabe 3 weiterhin. Sollte dies nicht gelingen, darf statt der VSbuggyConnection wieder auf die fehlerfreie VSObjectConnection zurückgegriffen werden.]



Neustart nach Replikatausfall

■ Problem

- VSBoard-Implementierung verwaltet ihren Zustand im Hauptspeicher
- Datenverlust bei Ausfall eines Replikats

→ Kein Neustart des Replikats möglich

■ Lösung

- VSRemoteObjectStateHandler: Schnittstelle zum Auslesen/Serialisieren und Setzen/Deserialisieren des Zustands eines Remote-Objekts

```
public interface VSRemoteObjectStateHandler {  
    public byte[] getState();  
    public void setState(byte[] state);  
}
```

- Verwendung von JGroups für Zustandstransfer

- Annahme: Zu jeder Zeit ist mindestens ein Replikat verfügbar
- Neustart: Holen der Remote-Objekt-Zustände von einem anderen Replikat
- Konsistentes Einspielen der Zustände wird von JGroups gewährleistet



Vereinfachungen in Übungsaufgabe 4

■ Transparente Rückrufe

- Problem

- Einzelner post()-Aufruf führt zu mehreren Rückrufen (einer pro Replikat) am VSBoardListener → Verletzung der Replikationstransparenz
- Rückruf ist externalisiertes Ereignis

- Möglicher Lösungsansatz

- Zusätzliche Einigung über Gruppenkommunikation, wer den Aufruf ausführt
- Offene Fragestellung: Was ist, wenn das ausgewählte Replikat ausfällt?

- In Übungsaufgabe 4 zu realisierender Ansatz: Problem ignorieren

■ Aktualisierung von Replikatreferenzen

- Problem

- Adressen neu hinzu kommender Replikate sind eventuell nicht vorab bekannt
- Veraltete Informationen in Gruppenreferenz

- Möglicher Lösungsansatz

- Bekanntgabe der eigenen Adresse per Gruppenkommunikation (z. B. in View)
- Dynamische Anpassung der Gruppenreferenz

- Annahme in Aufgabe 4: Adressen sind auf Server-Seite statisch bekannt



Übungsaufgabe 4: Klassenübersicht

