Verteilte Systeme – Übung

Java RMI: Marshalling und Unmarshalling

Sommersemester 2022

Laura Lawniczak, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.cs.fau.de

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

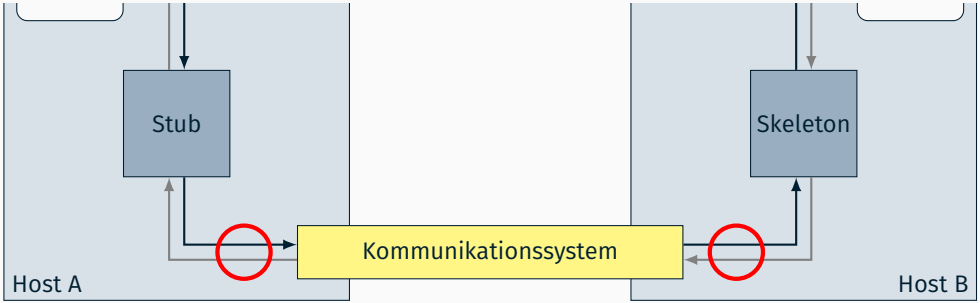
Überblick

Marshalling und Unmarshalling

Marshalling und Unmarshalling

Marshalling und Unmarshalling

- Definition
  - *Marshalling*: Verpacken von Informationen in einer Nachricht
  - *Unmarshalling*: Auspacken von Informationen aus einer Nachricht
- Problemstellungen
  - Unterschiedliche Datentypen
  - Heterogenität bei der lokalen Repräsentation von Datentypen



Unterschiedliche Datentypen

- Primitive Datentypen
  - z. B. `char`, `boolean`, `int`, ...
- Benutzerdefinierte Datentypen
  - z. B. `classes`
- Felder
  - z. B. `int[47]`, `Strings`
- Referenzen
  - z. B. `Object ref = new Object(); Object refDup = ref;`
- Ressourcen
  - z. B. `Threads`, Dateien, `Sockets`, ...
- ...

$\Rightarrow$ **Kein allgemeines Vorgehen möglich**

Heterogenität

- Byte-Reihenfolgeproblem
  - Big Endian (Network Byte Order)
    - Most-significant byte first
    - z. B. SPARC, Motorola
  - Little Endian
    - Least-significant byte first
    - z. B. Intel x86
- Repräsentation von Fließkommazahlen
  - Allgemein
    - Vorzeichen (s)
    - Mantisse (m)
    - Exponent (e)
    - Zahlenwert: $(-1)^s * m * 2^e$
  - Variationsmöglichkeiten
    - Anzahl der Bits für m und e
    - Speicherreihenfolge von m, e und s
    - Byte-Order
- Beispiel
  - `12345 =` `0x 30 39`
  - Big Endian: `00 00 30 39`
  - Little Endian: `39 30 00 00`

- Kanonische Repräsentation
  - Nutzung einer allgemeingültigen Form als Zwischenrepräsentation
  - z. B. IEEE-Standard

  ⇒ Eventuell unnötige Konvertierungen
  [z. B. wenn Sender und Empfänger identische Repräsentation nutzen]

- „Sender makes it right“
  - Sender kennt Datenrepräsentation des Empfängers
  - Sender konvertiert Daten

  ⇒ Multicast an heterogene Gruppe nicht möglich

- „Receiver makes it right“
  - Kennzeichnung des Datenformats
  - Empfänger konvertiert Daten

  ⇒ Bereitstellung sämtlicher Konvertierungsroutinen notwendig
  [Unproblematisch für Byte-Order-Konvertierung]

- Hilfsklasse `java.nio.ByteBuffer`

```
public abstract class ByteBuffer [...] {
    public static ByteBuffer allocate(int capacity);
    public static ByteBuffer wrap(byte[] array);
    public byte[] array();
    public ByteBuffer put<Datentyp>(<Datentyp> value);
    public <Datentyp> get<Datentyp>();
    [...]
}
```

  - `allocate()` Anlegen eines neuen (leeren) Byte-Array
  - `wrap()` Verwendung eines bestehenden Byte-Array
  - `array()` Rückgabe des vom Puffer verwendeten Byte-Array
  - `put*(), get*()` Einfügen bzw. Lesen von Daten aus dem Puffer

- Beispiel: {S,Des}erialisierung eines `double`-Werts

```
double d = 0.47;
ByteBuffer buffer1 = ByteBuffer.allocate(Double.BYTES);
buffer1.putDouble(d);
byte[] byteArray = buffer1.array();
```

```
ByteBuffer buffer2 = ByteBuffer.wrap(byteArray);
double d2 = buffer2.getDouble();
```

- Objekt ⇔ Stream: `java.io.Object{Out,In}putStream`

```
public class ObjectOutputStream [...] {
    public ObjectOutputStream(OutputStream out);
    public void writeObject(Object obj); // Objekt serialisieren
    [...]
}
```

```
public class ObjectInputStream [...] {
    public ObjectInputStream(InputStream in);
    public Object readObject(); // Objekt deserialisieren
    [...]
}
```

- Stream ⇔ Byte-Array: `java.io.ByteArray{Out,In}putStream`

```
public class ByteArrayOutputStream extends OutputStream {
    public byte[] toByteArray(); // Rueckgabe des Byte-Array
    [...]
}
```

```
public class ByteArrayInputStream extends InputStream {
    public ByteArrayInputStream(byte buf[]);
    [...]
}
```

- Automatisierte {S,Des}erialisierung: `java.io.Serializable`
  - Muss von jedem Objekt implementiert werden, das von einem `Object{Out,In}putStream` serialisiert bzw. deserialisiert werden soll
  - Marker-Schnittstelle → keine zu implementierenden Methoden

⇒ {S,Des}erialisierung wird vom `Object{Out,In}putStream` übernommen

- Manuelle {S,Des}erialisierung: `java.io.Externalizable`
  - Klassenspezifische {S,Des}erialisierung

```
public interface Externalizable extends Serializable {
    void writeExternal(ObjectOutput out);
    void readExternal(ObjectInput in);
}
```

  - `writeExternal()` Objekt serialisieren
  - `readExternal()` Objekt deserialisieren
  - Objekt muss öffentlichen Konstruktor ohne Argumente bereitstellen

⇒ {S,Des}erialisierung wird vom Objekt selbst übernommen

- Einige Attribute einer Klasse sollen nicht serialisiert werden
  - Sicherheitsaspekte
  - Effizienzüberlegungen
- Einige Objekte können nicht serialisiert & deserialisiert werden, da sich ihr Zustand nicht so ohne weiteres wiederherstellen lässt
  - `FileInputStream`
  - `Socket`, `ServerSocket`
  - `Thread`

⇒ Schlüsselwort `transient`

- Mit `transient` gekennzeichnete Attribute werden bei der automatischen {S,Des}erialisierung vom `Object{Out,In}putStream` ignoriert
- Beispiel

```
public class TransientExample implements Serializable {
    private transient Thread t = new Thread();
}
```