Middleware-Konzepte

Kommunikation

Dr. Gero Mühl

Kommunikations- und Betriebssysteme
Fakultät für Elektrotechnik und Informatik
Technische Universität Berlin

Übersicht

- Nachrichtenaustausch
  - Zuverlässigkeit
  - Synchrone vs. asynchrone Kommunikation
  - Nachrichtenreihenfolge
  - Transiente vs. persistente Kommunikation

- Externe Datenrepräsentation
  - ASN.1
  - Java Object Serialization
  - XML-basierte Serialisierung

Nachrichtenaustausch

Nachrichtenaustausch

- Nachricht

- Programmierung
  - *send*(nachricht) Senden einer Nachricht
  - *receive*(nachricht) Empfangen einer Nachricht
  - Callbacks sind eine Alternative zur *receive*-Anweisung
- Implementierung durch *Kommunikationssystem (KS)*
  - Abstrahiert von unterliegenden Schichten und bietet API
  - Oft für verschiedenste Programmiersprachen vorhanden
  - Urform: UNIX Sockets

Zuverlässigkeit

- Nachrichtenverluste (unbeabsichtigte)
  - (a) unerkannt, (b) gemeldet, (c) automatisch korrigiert
  - Implementierung von (b) und (c) durch das KS mittels Sequenznummern, Bestätigungen (negative, positive), Wiederholungen, Timeouts

- Nachrichtenverfälschungen (unbeabsichtigte)
  - (a) unerkannt, (b) gemeldet, (c) automatisch korrigiert
  - Implementierung von (b) durch das KS mittels Prüfsummen
  - Implementierung von (c) durch das KS mittels fehlerkorrigierender Codes (Redundanz erlaubt verfälschte Nachrichten zu rekonstruieren) oder durch Wiederholungen

Zuverlässigkeit

> Zur Erkennung *mutwilliger* Nachrichtenverluste und Nachrichtenverfälschungen sind verschlüsselte Sequenznummern bzw. verschlüsselte Prüfsummen (z.B. verschlüsselte SHA2-Prüfsumme) geeignet
> Das Warten auf Wiederholungen von Nachrichten führt aus Applikationssicht zu erhöhten Nachrichtenlaufzeiten

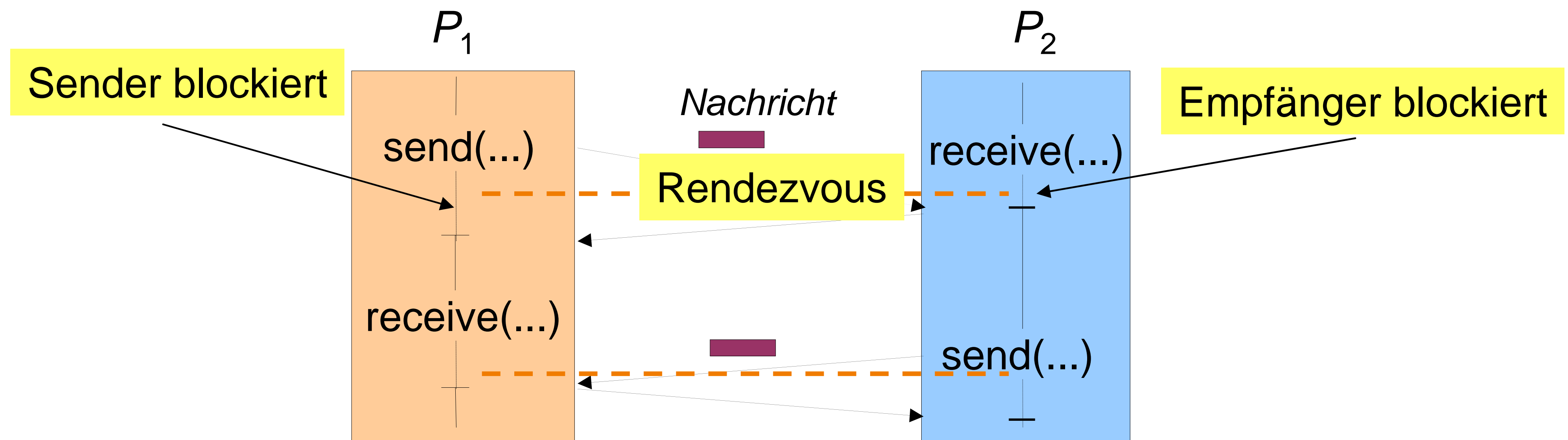
Blockierende Operationen?

- Blockieren nach Aufruf von *send* bis Nachricht …
  - übernommen vom lokalen KS?
  - übernommen vom entfernten KS?
  - übernommen vom Empfänger-Prozess?
  - bearbeitet vom Empfänger-Prozess?

- Blockieren nach Aufruf von *receive* …
  - nur Abtesten ob Nachricht verfügbar (ohne zu blockieren)
  - nur gewisse Zeit blockieren (Timeout)
  - **blockieren bis Nachricht verfügbar**
    (meist verwendet, vermeidet *Busy Waiting*)

Synchrone Kommunikation

- > Sender blockiert bis *receive* auf Empfängerseite erfolgt ist
- > Empfänger blockiert bis *send* auf Senderseite erfolgt ist
- ⇒ Zu einem Zeitpunkt gleichzeitige Blockierung! → Rendezvous
- > Transparente Implementierung durch KS (mittels Bestätigung)
- > Blockieren gleicht *Geschwindigkeitsunterschiede* zwischen Sender und Empfänger aus!

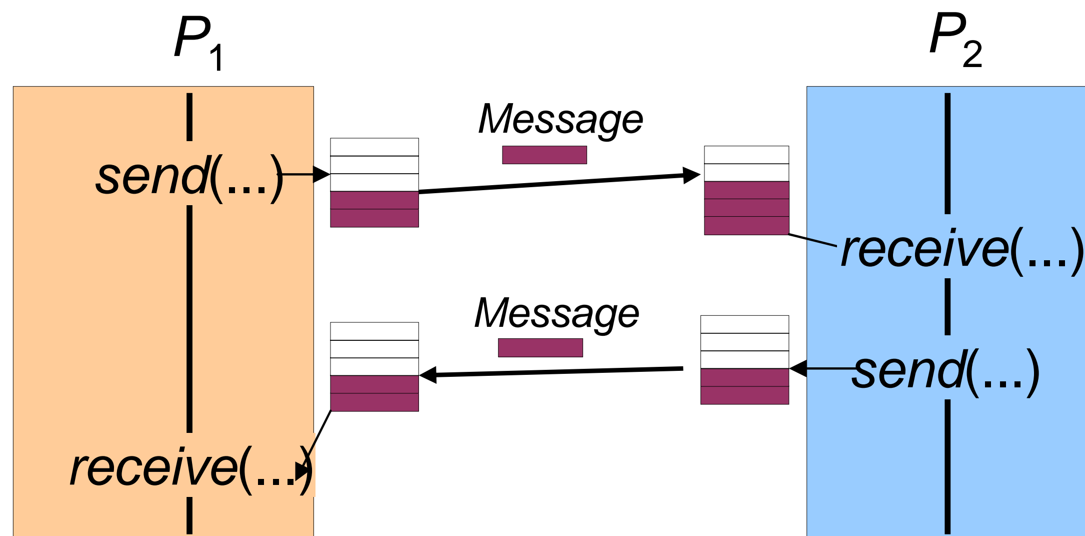


Synchrone Kommunikation

- Vorteile synchroner Kommunikation
  - Sender weiß, dass Empfänger Nachricht erhalten hat
  - Nur eine Nachricht muss gepuffert werden
  - Ausgleich unterschiedlicher Geschwindigkeiten von Sender und Empfänger durch Warten
- Nachteile
  - Enge Kopplung: Sender und Empfänger müssen gleichzeitig laufen
  - Sender wird blockiert
    - Geringere Parallelität *oder* komplexes Multi Threading
    - Deadlock-Gefahr
- **Request/Reply-Kommunikation** kombiniert das Senden eines Requests mit dem Empfangen des zugehörigen Replies in einer Anweisung

Asynchrone Kommunikation

- Keine Blockierung des Senders
- Jeweiliges KS puffert die ausgehenden Nachrichten beim Sender und die eingehenden Nachrichten beim Empfänger
  - *Sendepuffer* ermöglicht vorübergehend schneller zu Senden, als Nachrichten übertragen werden können
  - *Empfangspuffer* ermöglicht vorübergehend schneller zu Empfangen, als Nachrichten vom Empfänger abgenommen werden können
- Achtung: Puffer gleichen die *Geschwindigkeitsvarianz* von Sender und Empfänger aus, nicht aber *dauerhafte Geschwindigkeitsunterschiede*!

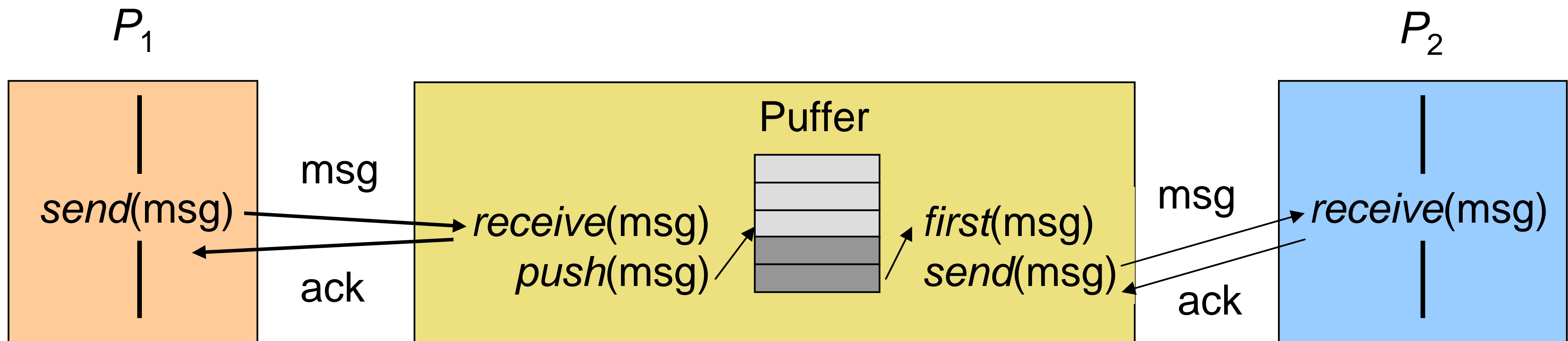


Asynchrone Kommunikation

- Vorteile
  - Sender und Empfänger *müssen nicht* gleichzeitig laufen
    → Losere Kopplung
  - Höherer Grad an Parallelität möglich
  - Gefahr von Deadlocks geringer
- Nachteile
  - Sender weiß nicht, ob oder wann eine Nachricht angekommen ist
  - Synchronisation schwieriger
  - Puffer können überlaufen
  - Schwierigere Programmierung

Asynchrone mittels synchroner Kommunikation

- Pufferprozess wird zwischengeschaltet
  - Speichert empfangene Nachrichten in einem Ringpuffer
  - Sendet gepufferte Nachrichten an Empfänger
  - Pufferprozess läuft am sinnvollsten auf dem Rechner des Senders
- Problem
  - Soll der Pufferprozess in einem *receive* auf weitere Nachrichten vom Sender warten oder in einem send auf die Abnahme einer Nachricht vom Empfänger?



Nachrichtenreihenfolge

- Kommunikationssystem kann Garantien geben, dass Nachrichten beim Empfänger in einer bestimmten Ordnung (d.h. Reihenfolge) eintreffen
  - FIFO-Sender Ordnung
  - Kausale Ordnung (impliziert FIFO-Sender)
  - Totale Ordnung (ist orthogonal zu den anderen beiden Ordnungen) → kann mit FIFO-Sender- oder auch mit kausaler Ordnung kombiniert werden

Transiente vs. Persistente Kommunikation

> transient = vergänglich, flüchtig
  persistent = anhaltend, dauernd

> *Transiente* Kommunikation
  > Kommunikationssystem speichert Nachrichten nicht über die Lebenszeit von Sender und Empfänger hinaus

> *Persistente* Kommunikation
  > Kommunikationssystem speichert Nachrichten über die Lebenszeit des Senders hinaus, bis sie dem Empfänger zugestellt werden können → Queues

Externe Datenrepräsentation

Motivation

- Sockets unterstützen per se nur den Transport von Byteströmen bzw. Bytearrays
- Bei der Nutzung von Sockets muss der Programmierer händisch
  - Die Struktur der Datenströme bzw. Nachrichtentypen festlegen
  - Die zu versendenden Applikationsdaten in Bytes umwandeln (aka. Serialisierung, Marshalling)
  - Die empfangenen Bytes wieder zurück in Applikationsdaten umwandeln (aka. Deserialisierung, Unmarshalling)
  - Bei verketteten, im Speicher nicht zusammenhängenden, Datenstrukturen ist dies nicht trivial („Deep Copy“ vs. „Shallow Copy“)
- Heterogenität macht alles noch deutlich komplexer!

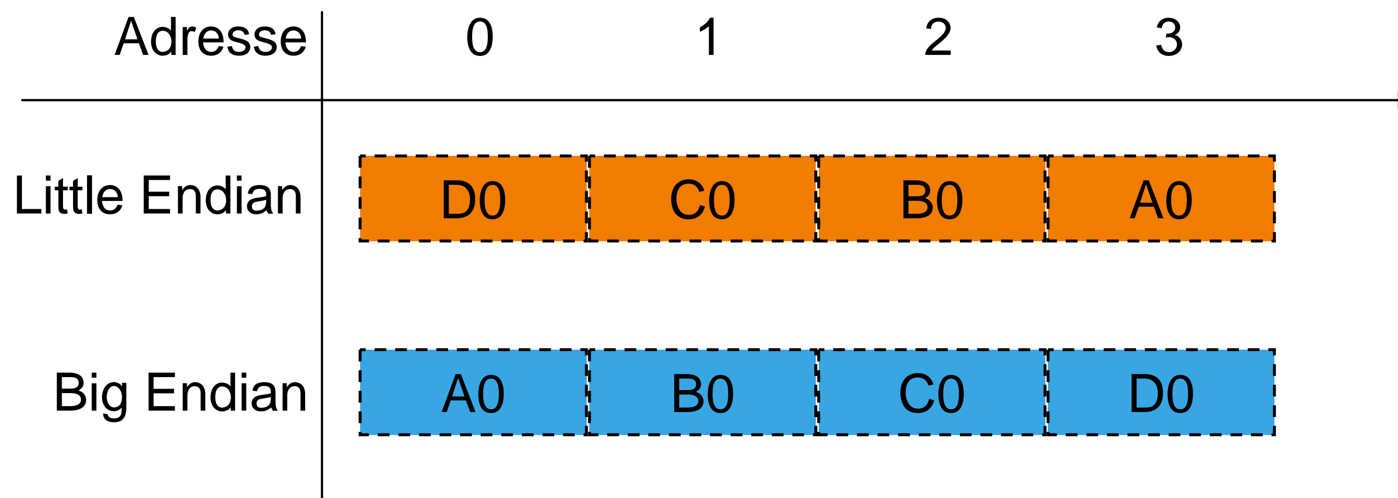
Motivation

- Sinnvoll wäre eine Bibliothek mit Routinen
  - zur Serialisierung und Deserialisierung von
    - primitiven Datentypen,
    - zusammengesetzte Datentypen (z.B. Arrays) und
    - für verkettete Datenstrukturen (z.B. Listen und Graphen)
  - zum Senden und Empfangen ganzer Nachrichten, inklusive der enthaltenen Datenstrukturen falls notwendig auch mit automatischer *Fragmentierung* beim Sender bzw. *Defragmentierung* beim Empfänger
- Die Bibliothek müsste für verschiedenste Hardware, Betriebssysteme und Programmiersprachen verfügbar sein, um auch in heterogenen Systemen anwendbar zu sein

⇒ Externe Datenrepräsentationen

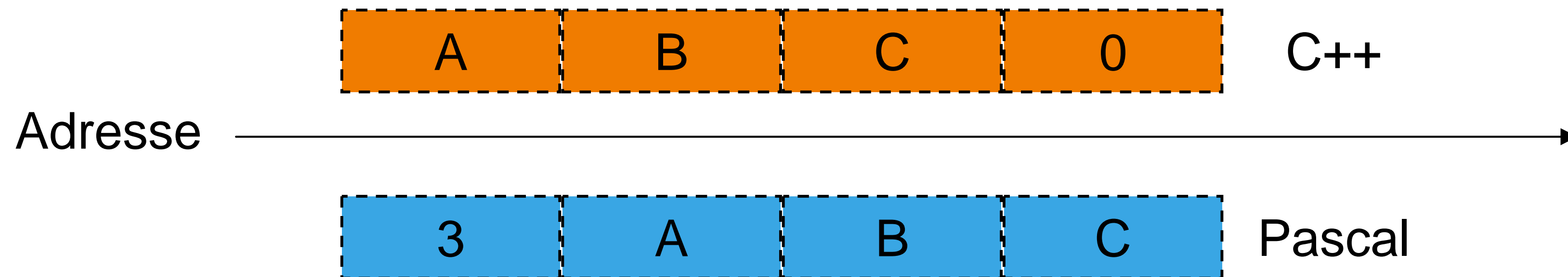
Heterogenität – Hardware

- Verschiedene Hardware-Architekturen speichern die Bytes eines Wortes (hier 32 Bit) in unterschiedlicher Reihenfolge
  - ***Little Endian*** (z.B. Intel x86) vs. ***Big Endian*** (z.B. Motorola G4) am Beispiel des Wertes 0xA0B0C0D0



Heterogenität – Programmiersprachen

- Verschiedene Programmiersprachen speichern Datentypen unterschiedlich
  - Z.B. String „ABC“



⇒ Beim Datenaustausch muss passend konvertiert werden

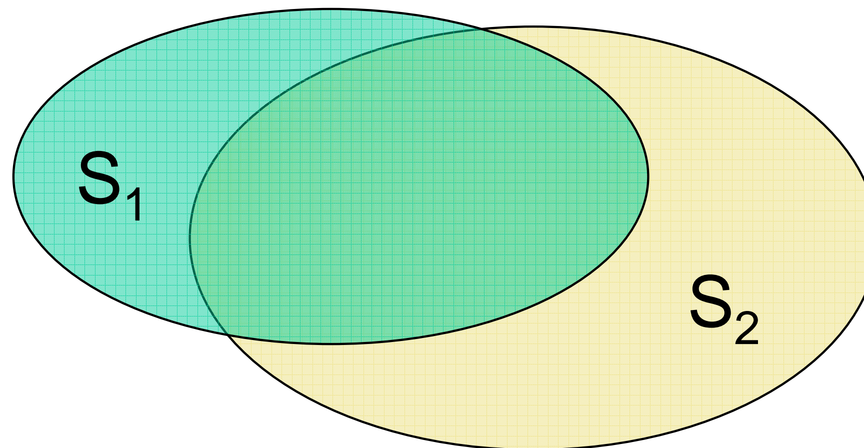
Transformation zwischen Darstellungen

> Heterogenität der lokalen Repräsentationen

⇒ Transformation zwischen verschiedenen Darstellungen notwendig

> Hierbei gehen evtl. Informationen verloren

> 2 mögliche Realisierungen (s. nächste 2 Folien)

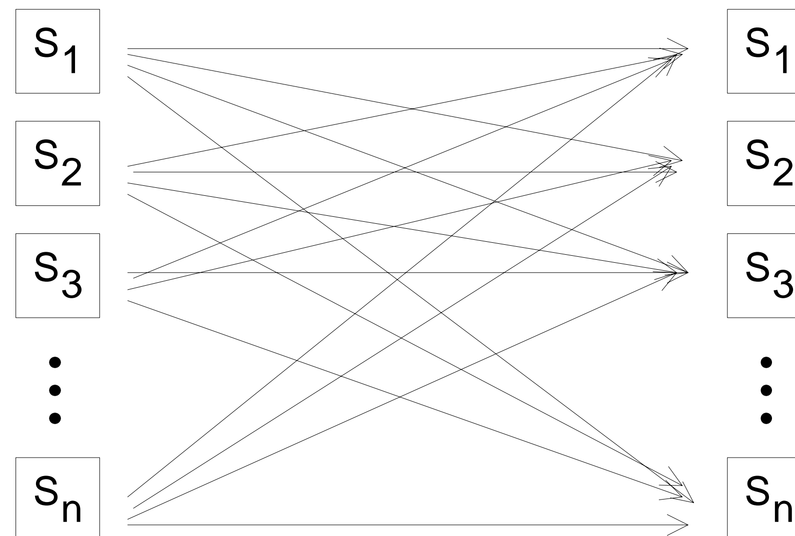


 © Gero Mühl

1. Möglichkeit: Paarweise Transformation

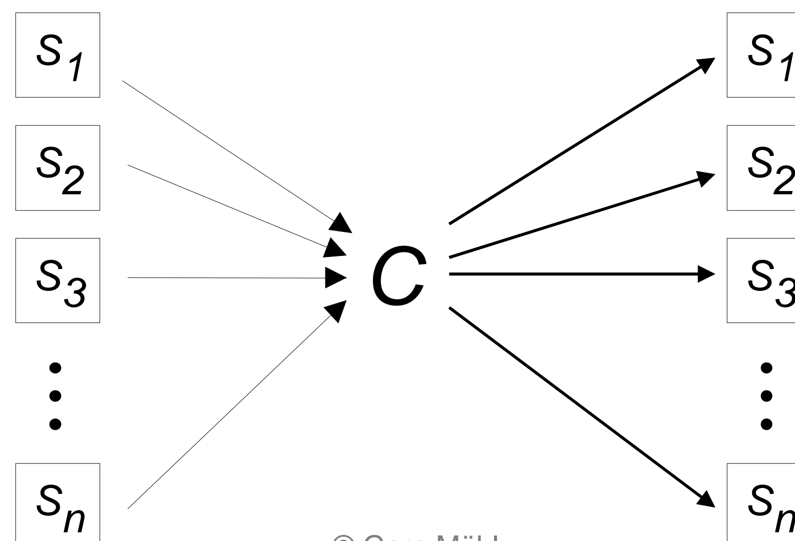
- *Paarweise* Transformation zwischen *n* verschiedenen lokalen Repräsentationen
- Entweder Sender oder Empfänger muss transformieren
  - „Sender makes it right“ vs. „Receiver makes it right“
  - ⇒ eine Transformation pro Kommunikation

⇒ $n^2 - n$ verschiedene Transformationen

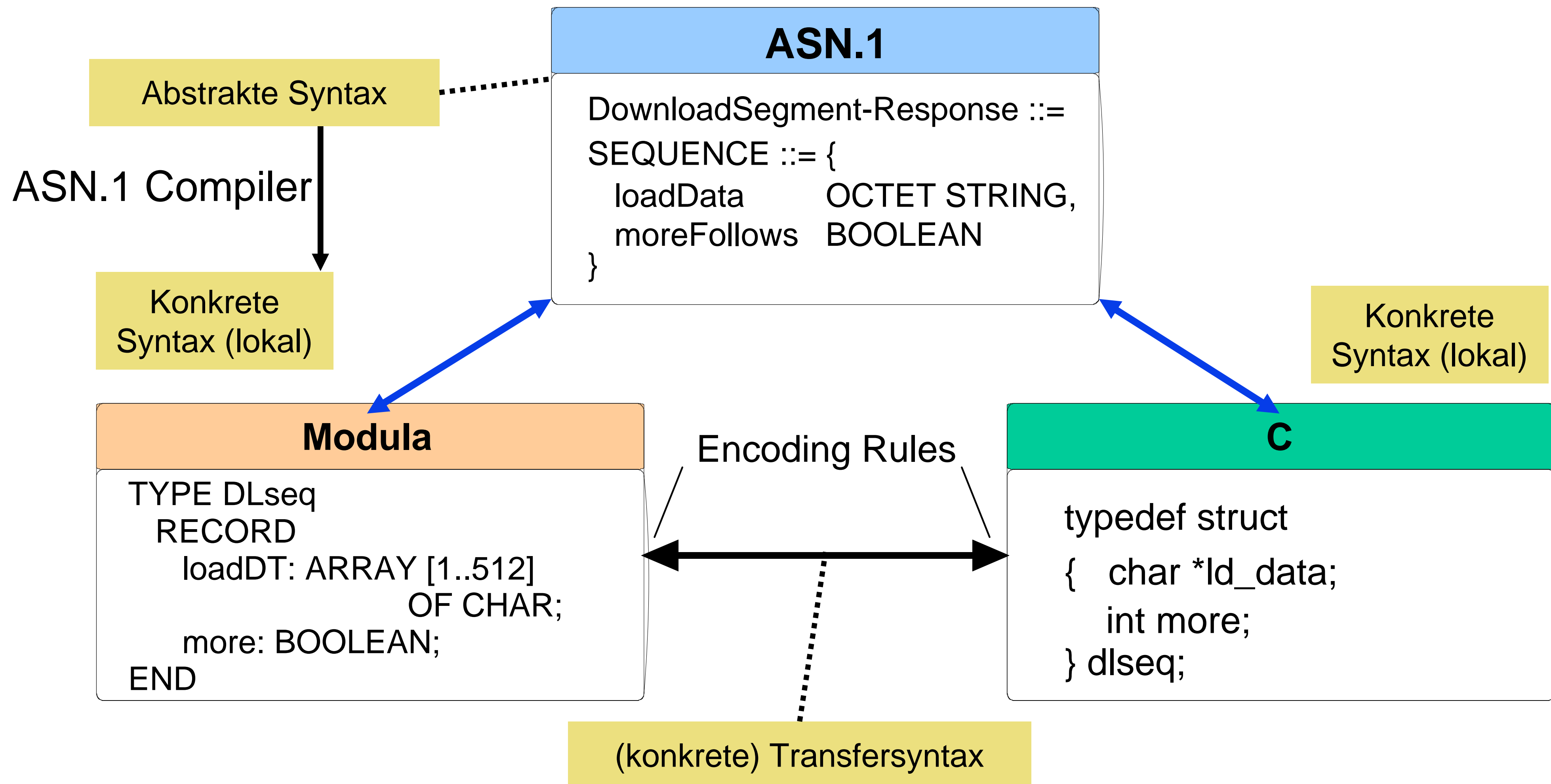


2. Möglichkeit: Kanonische Repräsentation

- Verwendung einer *kanonischen* Datenrepräsentation C
- Keine Informationen über die Repräsentation des Gegenübers notwendig
- Nur n verschiedene Transformationen für jede der beiden Richtungen (nach C, von C) notwendig
- 2 Transformationen pro Kommunikation notwendig (jeweils von S_i nach C und von C nach S_j) → mehr Aufwand



Bsp.: ISO Standard Abstract Syntax Notation.1 (ASN.1)



ASN.1

- Abstract Syntax
  - Beschreibt die allgemeine Struktur von Daten (*unabhängig* von Programmiersprachen etc.)
- Concrete Syntax
  - Beschreibt die lokale Darstellung der Daten (*abhängig* von Programmiersprache etc.)
- Transfer Syntax
  - Beschreibt die Darstellung der Daten bei der Übertragung (z.B. als Tripel <Datentyp, Byteanzahl, Bytes>)
- Encoding Rules (z.B. Basic Encoding Rules (BER))
  - Beschreiben die Umwandlung von der konkreten in die Transfer Syntax

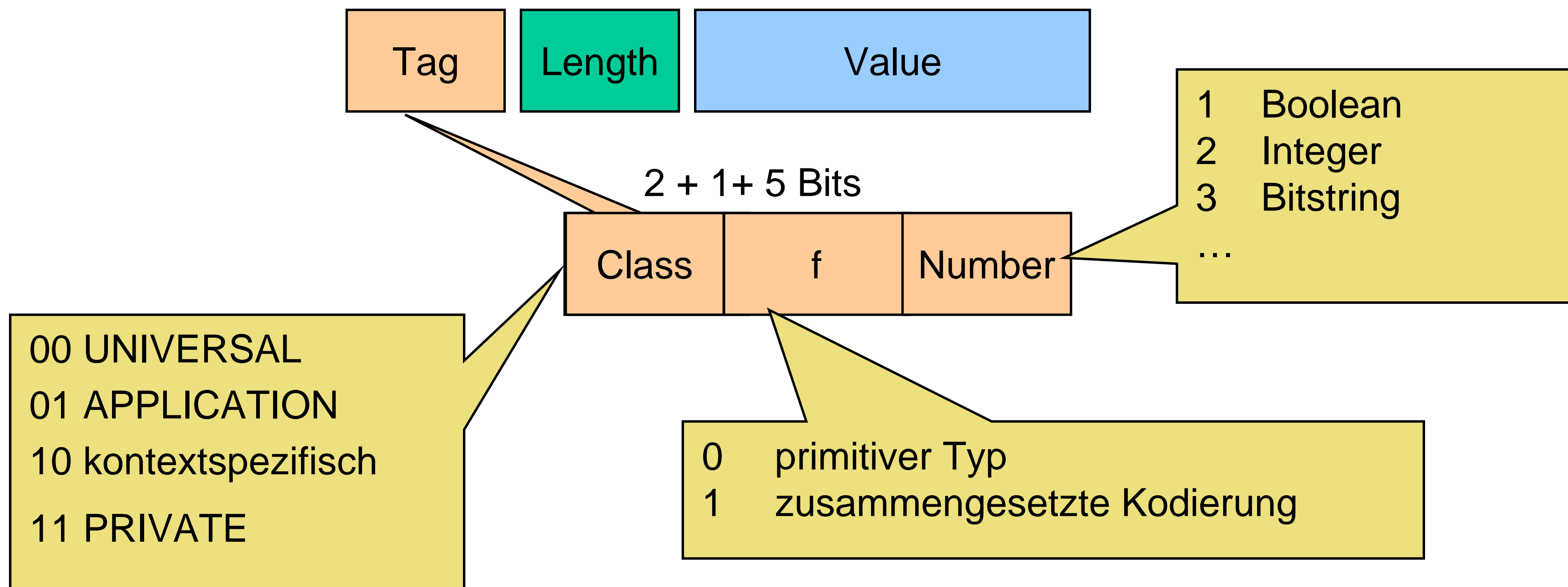
ASN.1 Compiler

> Überprüft die Syntax einer Abstract Syntax Specification
> Erzeugt aus der Abstract Syntax die Concrete Syntax (z.B. für C)
> Erzeugt die Routinen für Codierung/Decodierung (z.B. in C)

Basic Encoding Rules (BER) for ASN.1

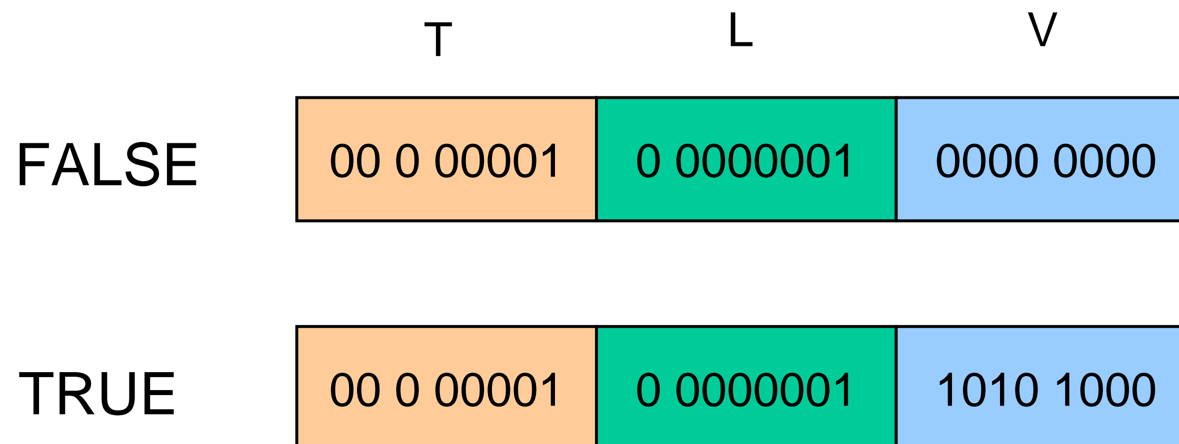
> Daten werden als Folge von Oktetts (= Bytes) kodiert

> TLV-Codierung



Basic Encoding Rules

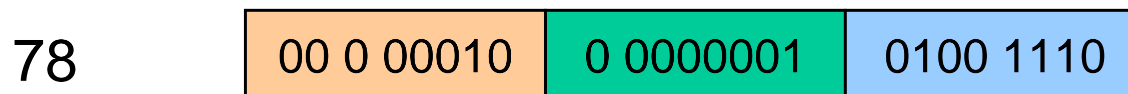
- Beispiel:
  - Zwei Werte für BOOLEAN
    - FALSE = 0000 0000　　　TRUE ≠ 0000 0000
  - Kodierung mit TLV



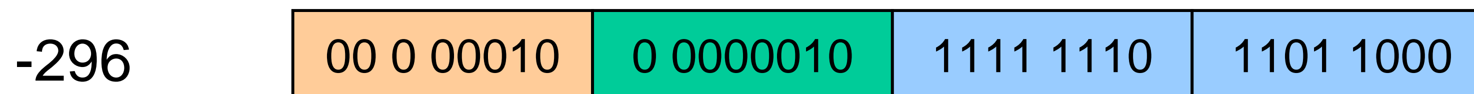
Basic Encoding Rules

> Beispiel INTEGER (im 2er-Komplement, Big Endian)

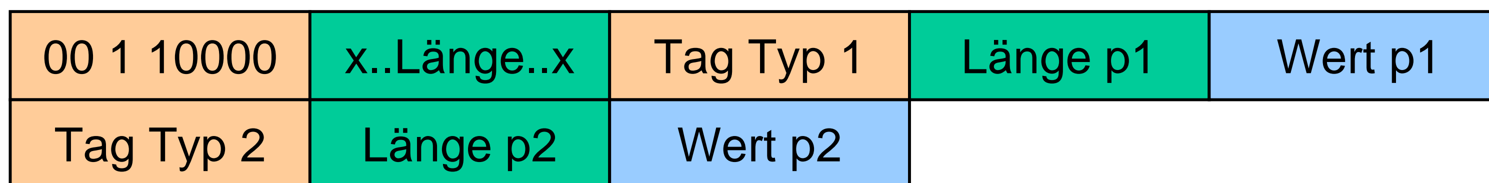
78



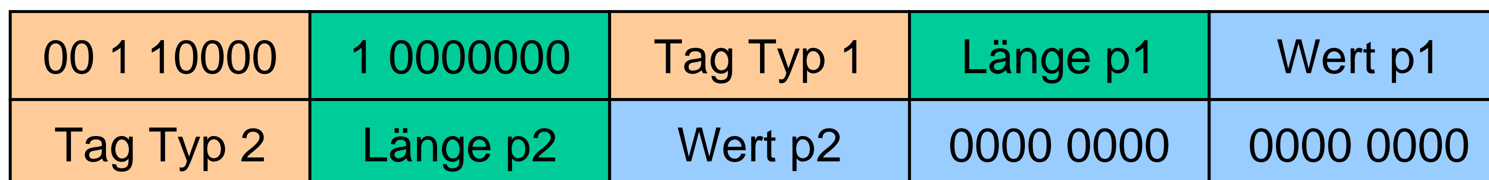
-296



> Beispiel SEQUENCE S ::= SEQUENCE { p1 Typ1, p2 Typ2 }



oder



Java Object Serialization

- Erlaubt die streambasierte Übertragung von serialisierten Objekten
  - Z.B. über TCP- oder UDP-Sockets oder als Parameter bei Java RMI
- Zu serialisierende Klassen müssen das Markerinterface `java.io.Serializable` implementieren (auch alle Klassenfelder)
- Auf Java beschränkt!
- Empfänger muss (Zugriff auf) Implementierungen der Klassen haben
- Serialisierung benötigt vom Entwickler einer Klasse keinen Klassen-spezifischen Code
  - Benutzt die Reflektions-Mechanismen von Java
- Binäres Format
- Beinhaltet Meta-Informationen
- Kommt auch mit zyklischen Datenstrukturen zurecht
  - Jedes Objekt wird einmal geschrieben und ihm wird eine Referenz zugewiesen
  - Rückwärtsverweise auf Referenzen für bereits geschriebene Objekte

Java Object Serialization

- Spezielle Darstellungen für `null` Objekte, neue Objekte, Klassen, Arrays, Strings und Rückwertsverweise
- Es ist wichtig zu wissen, von welcher Klasse ein Objekt ist!
  - Stream enthält auch Informationen über alle geschriebenen Klassen, deren Version und deren Superklassen (falls serialisierbar)
    $\Rightarrow$ Instanzen der Klasse `ObjectStreamClass`
  - Ermöglicht dem Empfänger auch, die benötigten Klassen zu laden
- Ein Reset des Streams löscht die Informationen über bereits geschriebene Objekte (und Klassen)
  - Ermöglicht die neuerliche Übertragung von (z.B. geänderten) Objekten
  - Ineffizient, da Klasseninformationen auch gelöscht werden
- Objekt-Implementierungen können Serialisierung durch Implementierung spezieller Methoden beeinflussen

Klonen mittels Serialisierung

```
public class ObjectCloner {

   public static byte[] toByteArray(Object obj)
      throws IOException {
      ByteArrayOutputStream baos =
            new ByteArrayOutputStream();
      ObjectOutputStream oos =
            new ObjectOutputStream(baos);
      oos.writeObject(obj); oos.flush(); baos.flush();
      return baos.toByteArray();
   }

   public static Object fromByteArray(byte[] ba)
         throws IOException, ClassNotFoundException {
      ByteArrayInputStream bais =
            new ByteArrayInputStream(ba);
      return new ObjectInputStream(bais).readObject();
   }
```

Klonen mittels Serialisierung

```
public static Object clone(Object obj) {
   Object clone = null;
   try {
      clone = fromByteArray(toByteArray(obj));
   } catch (Exception exception) {
      exception.printStackTrace();
   }
   return clone;
}

public static void main(String[] args) {
   Date d = new Date();
   Date d2 = (Date)ObjectCloner.clone(d);
   System.out.println(d==d2); // false
}
}
```

Gleichheitstests mittels Serialisierung

```
public class ObjectComparer {
   public static boolean equals(Object obj1, Object obj2) {
      boolean b = false;
      try {
         byte[] ba1 = ObjectCloner.toByteArray(obj1);
         byte[] ba2 = ObjectCloner.toByteArray(obj2);
         return Arrays.equals(ba1,ba2);
      } catch (Exception exeption) { // ignored }
      return b;
   }

   public static void main(String[] args) {
      Date d = new Date();
      Date d2 = (Date)ObjectCloner.clone(d);
      System.out.println(ObjectComparer.equals(d,d2)); // true
   }
}
```

XML-basierte Serialisierung

- eXtensible Markup Language (XML)
- XML-Dokumente
  - Sind hierarchisch strukturiert
  - Enthalten Informationen über die eigene Struktur
  - Basieren auf Klartext
    - „Menschen-lesbar“
    - Wiederherstellung archivierter Dokumente leichter
    - Größere Länge als binäre Formate
- XML-Parser und andere XML-Tools können genutzt werden

XML-basierte Serialisierung

```
<?xml version='1.0' encoding='UTF-8'?>
    <!DOCTYPE koml SYSTEM "http://koala.ilog.fr/XML/koml12.dtd">
    <koml version='1.2'>
        <classes>
            <class name='java.net.URL'>
                <field name='port' type='int'/>
                <field name='file' type='java.lang.String'/>
                <field name='host' type='java.lang.String'/>
                <field name='protocol' type='java.lang.String'/>
            </class>
        </classes>
        <object class='java.net.URL' id='i2'>
            <value type='int' name='port'>80</value>
            <value type='java.lang.String‘ name='file'>/koala/XML/</value>
            <value type='java.lang.String‘ name='host'>www.inria.fr</value>
            <value type='java.lang.String' name='protocol'>http</value>
        </object>
    </koml>
```

Literatur

1. Sun Microsystems, Inc. *Java Object Serialization Specification 1.5.0*, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serial-title.html.
2. O. Dubuisson. *ASN.1 Communication between Heterogeneous Systems*. Morgan Kaufmann Publishers, 2000. translated from French by Philippe Fouquart. http://asn1.elibel.tm.fr/en/book/ http://www.oss.com/asn1/booksintro.html
3. M. Campione, K. Walrath, and A. Huml. *The Java Tutorial. The Java Series*. Addison-Wesley, 3rd edition, 2001. http://java.sun.com/docs/books/tutorial/