

F.1 Überblick

- Aufgabe 3
- RPC-System im Überblick
- Marshalling primitiver Datentypen
 - ◆ Byteorder
 - ◆ Fließkommawerte
- Stubs und Skeletons

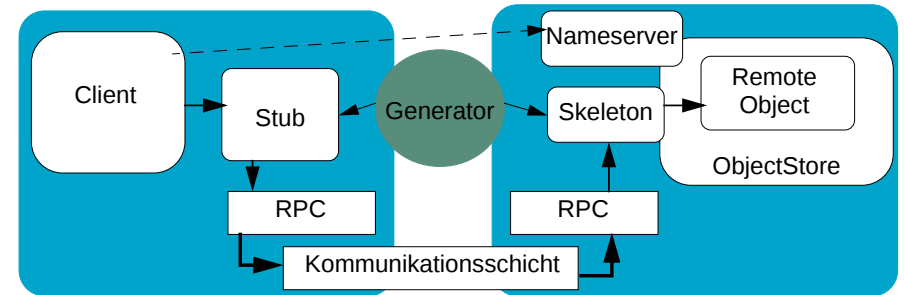
F.2 Aufgabe 3

F.2 Aufgabe 3

- Pufferklasse
 - ◆ mit Marshalling-Funktionen
 - ◆ ohne Puffer-Management
 - ◆ "stromorientiert"
- Stub und Skeleton
 - ◆ für einen Objekttyp
 - ◆ incl. Dispatcher zur Verwaltung von mehreren Objekten (dieses Typs)

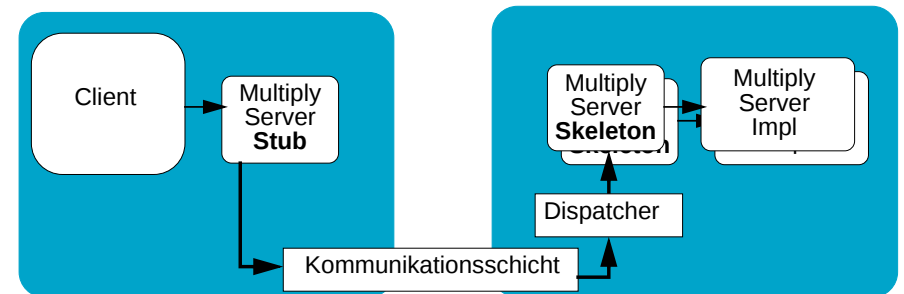
F.3 RPC-System im Überblick

- *Kommunikationsschicht*: tauscht Daten zwischen zwei Rechnern aus
- *RPC Schicht*: definiert die Aufrufsemantik und das Marshalling
- *Object Store*: verwaltet den Lebenszyklus der Objekte
- *Stub / Skeleton Generator*: erzeugt Code für die Stubs und die Skeletons
- *Nameserver*: findet Objekte anhand deren Namen



1 RPC-System in Aufgabe 3

F.3 RPC-System im Überblick



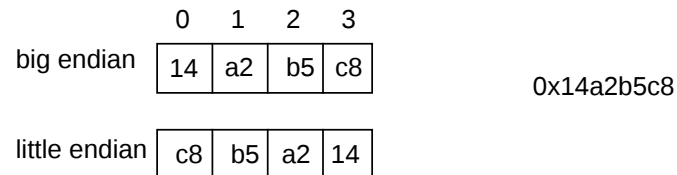
- Stub und Skeleton für einen Objekttyp (**MultiplyServer**)
- Server soll mehrere Objekte dieses Typs unterstützen (Dispatcher)
- Anfragen können (nacheinander) von verschiedenen Clients kommen
- kein Namensdienst

F.4 Marshalling

- Aufgabe: Verpacken und Entpacken von Daten zur Übertragung zwischen Rechnern
- Die wesentlichen Problemstellungen
 - ◆ Heterogenität der lokalen Repräsentation von Datentypen
 - Konvertierung in ein einheitliches Netzwerkformat notwendig
 - ◆ Unterschiedliche Arten von Datentypen und Datenübergabe
 - Primitive Datentypen
 - Benutzerdefinierte Datentypen
 - Objekte, Referenzen; "Call by value"?
- In dieser Übung zunächst nur: Marshalling von primitiven Datentypen

1 Heterogenität bei primitiven Datentypen

- "Byte Sex" (Big Endian vs. Little Endian)



- Kommunikation zwischen Rechnern verschiedener Architekturen (z.B. Intel Pentium (little endian) und Sun Sparc (big endian))
- Umwandlung:
 - ◆ von Host-spezifischer Ordnung in Netzwerk-Byteordnung (big endian): **htons**, **htonl** (für short bzw. long Werte)
 - ◆ Umgekehrt (Netzwerk-Byteordnung nach Host-spezifische Ordnung) **ntohs**, **ntohl**

1 Heterogenität bei primitiven Datentypen

- Der komplexere Fall: Fließkommazahlen
- Eine Fließkommazahl besteht aus drei Bestandteilen
 - Vorzeichen (s)
 - Mantisse (m)
 - Exponent (e)
- ◆ Wert der Zahl: $(-1)^s * m * 2^e$
- Große Variationsmöglichkeiten:
 - Wieviel bit für m? wieviel für e? (s ist immer ein bit...)
 - In welcher Reihenfolge werden m, e und s gespeichert
 - Wie wird e gespeichert (als unsigned mit offset; signed)
 - In welcher Byte-Ordnung?

1 Heterogenität bei primitiven Datentypen

- "Früher" machte hier jeder, was er will
- Seit einiger Zeit existiert IEEE-Standard (IEEE 754). Bei x86, PPC und Sparc wird dieser als lokale Repräsentation verwendet.
- IEEE single float: 32 bit
 - 1 bit Vorzeichen, 8 bit Exponent, 23 bit Mantisse.
 - Exponent mit Offset 127 (e==127 entspricht dem Wert 0)
 - Mantisse als Nachkommastellen einer impliziten "1"; außer bei Exponent -127 (e==0)
 - Spezielle Werte für 0, +/- unendlich, NaN
- IEEE double float: 64 bit
 - 1 bit Vorzeichen, 11 bit Exponent, 52 bit Mantisse
 - Exponent mit Offset 1023
 - ansonsten identisch

1 Heterogenität bei primitiven Datentypen

- einige einfache Beispiele (auf Sparc-Architektur)

```

◆ 1: s=0, m=1, e=0
    s=0 mant=0 exp=127    3f 80 00 00

◆ 256: s=0, m=1, e=8
    s=0 mant=0 exp=135 (127+8) 43 80 00 00

◆ 0,125: s=0, m=1, e=-3
    s=0 mant=0 exp=124 (127-3) 3e 00 00 00

◆ -1: s=1, m=1, e=0
    s=1 mant=0 exp=127    bf 80 00 00

◆ 0: s=0, m=0, e=0
    s=0 mant=0 exp=0 (127-127) 00 00 00 00
  
```

- ◆ weitere Beispiele:

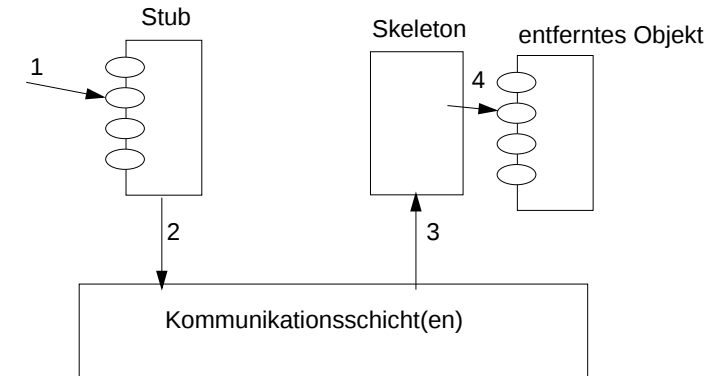
```

0.1: s=0 mant=4cccd exp=123 3d cc cc cd
9.94922e-44: s=0 mant=47 exp=0 00 00 00 47
  
```

- ◆ Auf IA32-Architektur: Umgekehrte Byte-Order, ansonsten identisch

F.5 Stubs und Skeleton

- Stub: Stellvertreter (Proxy) des entfernten Objekts.
- Skeleton: Ruft die Methoden am entfernten Objekt auf



2 Pufferklasse

- Schnittstellen-Beispiel

```

class Message {
public:
    Message(Buffer *b);
    Message(MsgType t, Buffer *b);

    MsgType getType() const;
    Buffer *getBuffer() const;
    void reset();
    void register_resize_handler(ResizeHandler *hdl);

    // marshalling primitiver Typen
    bool write(int16_t s);
    bool write(int32_t d);
    ...
    bool read(int16_t &s);
    bool read(int32_t &d);
    ...
    // alternatives Interface
    Message &operator<< (const int16_t value);
    Message &operator<< (const int32_t value);
    ...
    Message &operator>> (int16_t &value);
    Message &operator>> (int32_t &value);
    ...
};
  
```

```

class ResizeHandler{
public:
    virtual Buffer *resize
        (Buffer *old_buffer) = 0;
};
  
```

1 Stub: Aufgaben, Funktion

- implementiert den gleichen Typen wie das entfernte Objekt (gleiches Interface)
- Durchführung des Fernaufrufs:
 1. Methodenaufbau in eine Anfrage verpacken:
 - ◆ Objekt ID, Methoden ID, Parameter, ...
 2. Kommunikationsschicht verwenden um eine Anforderung zu versenden
 3. Warten auf das Ergebnis
 4. Rückgabeobjekt in den entsprechenden Typ transformieren

1 Stub Beispiel

- Beispiel: Stub-Methode (ohne Ausnahme- und Fehlerbehandlung)

```
int16_t ExampleStub::testMethode(const int16_t value){
    // Anfrage erstellen
    Buffer buf(Request::HDR_SZ + sizeof(value));
    Request req(oid, testMethode_MID, &buf);
    req << value; // req.write(value);

    // los geht's
    c->send(req.getBuffer());

    // warten auf Antwort
    c->receive(buf);

    // Antwort auspacken
    Message m(&buf);
    Result res(m);
    int16_t result;
    res >> result; // res.read(result);

    //fertig
    return result;
}
```

2 Skeleton: Aufgaben, Funktion

- ruft Methoden am "echten" Objekt auf
- notwendige Informationen:
 - ◆ Objektreferenz (z.B. aus Konstruktor)
 - ◆ Methoden ID
 - ◆ Parameter
- Vorgehen:
 1. MID aus Anfrage ermitteln (--> entsprechende Methode auswählen)
 2. Parameter aus Anfrage auspacken
 3. Methodenaufruf durchführen
 4. Rückgabeobjekt in Antwort verpacken
- Kommunikation ist in den Broker ausgelagert

2 Skeleton Beispiel

- Beispiel: Skeleton

```
Result ExampleSkel::invoke(Request &m) const{
    switch (m.getMID()){
        ...
        case testMethode_MID:{

            // Parameter auspacken
            int16_t param1;
            m >> param1;

            // Methode aufrufen
            int16_t result = obj->testMethode(param1);

            // Antwort verpacken
            Result r(new Buffer());
            r << result;;
            return r;

        } ...
        default:
            cerr << "unknown mid" << endl;
            ...
    }
}
```

3 Broker

- empfängt ankommenden Anfragen
- sucht (im ObjectStore) den Skeleton an den die Anfrage gerichtet ist
- aktiviert den Skeleton (mit den Parametern aus der Anfrage)
- sendet das Ergebnis zurück

3 Initialisierung Stub und Skeleton in Aufgabe 3

■ Beispiel - Client:

```
MultiplyServerStub stub(oid, comSys);
result = stub.multiply(6, 7);
```

■ Beispiel - Server:

```
MultiplyServerImpl obj;
MultiplyServerSkel skel(&obj);

Broker dispatcher;
int oid = dispatcher.registerSkel(&skel);
cout << "register Skel as OID: " << oid << endl;
dispatcher.run();
cerr << "never reached!" << endl;
```

■ Beispiel: einfacher Dispatcher

- ◆ empfangt Paket (oid, mid, parameter)
- ◆ sucht Skeleton und ruft dort "invoke" mit entsprechenden Parametern auf
- ◆ Skeleton sendet das Ergebnis selbst zurück

F.6 Zusammenfassung

■ RPC-System

- ◆ Marshalling
 - Byteorder
 - Fließkommawerte durch IEEE-Standard meist unproblematisch
- ◆ Stubs und Skeletons
 - einpacken von Parametern und Rückgabewerten
 - Marshalling
- ◆ Broker / Dispatcher
 - OID -> Skeleton