B 6. Übung

B.1 Überblick

- Aufgabe 4
- Marshalling primitiver Datentypen
  - Byteorder
  - Fließkommawerte
- RPC-System
  - Überblick: Architektur eines RPC Systems
  - Stubs und Skeletons

B.2 Aufgabe 4

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

1 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 *hndl);

        // 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);
        ...
};
```

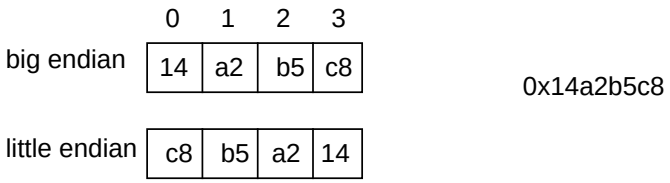
```
class ResizeHandler{
 public:
  virtual Buffer *resize
    (Buffer *old_buffer) = 0;
};
```


B.3 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

B.3 Heterogenität bei primitiven Datentypen

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



- Kommunikation zwischen Rechnern verschiedener Architekturen (z.B. Intel Pentium (litte 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`**

B.3 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

B.3 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?

B.3 Heterogenität bei primitiven Datentypen

- Beispiel

```
main()
{
    float f[]={1.0, 256, 0.125, 0.1, -1.0, 0, 1e-43};
    for(int i=0; i<7; i++) {
        char *ptr = (char *)&f[i];
        for(j=0; j<sizeof(f[0]); j++) printf("%02x ",ptr[j]);
        printf("\n");
    }
}
```

  - Auf Sparc-Architektur:

```
          1  = 3f 80 00 00      s=0 mant=0 exp=127
        256  = 43 80 00 00      s=0 mant=0 exp=135
      0.125  = 3e 00 00 00      s=0 mant=0 exp=124
        0.1  = 3d cc cc cd      s=0 mant=4ccccd exp=123
         -1  = bf 80 00 00      s=1 mant=0 exp=127
          0  = 00 00 00 00      s=0 mant=0 exp=0
 9.94922e-44 = 00 00 00 47      s=0 mant=47 exp=0
```

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

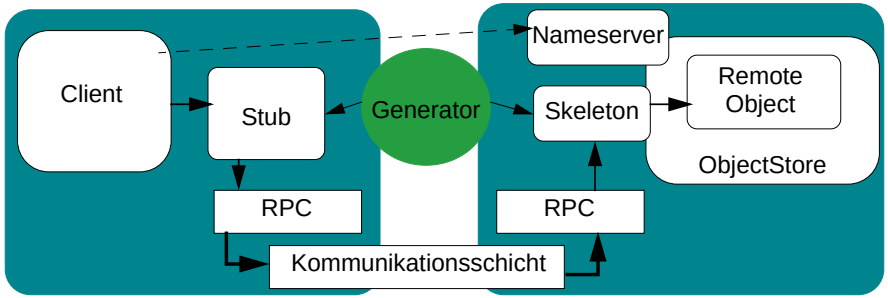
B.3 Heterogenität bei primitiven Datentypen

- Unions und Bit-Felder in C

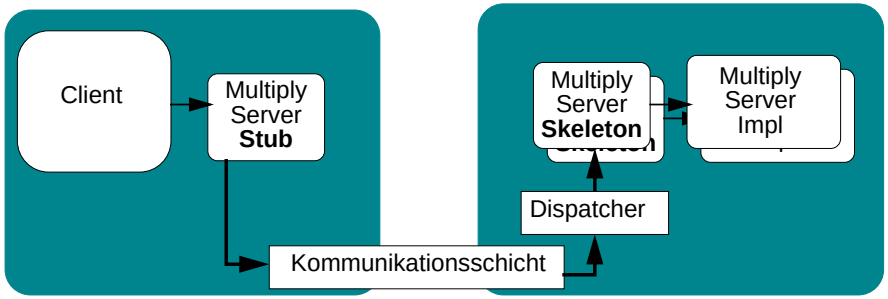
```
#ifdef BIGENDIAN
struct s_float {
    unsigned sign : 1;
    unsigned exponent : 8;
    unsigned mantisse : 23;
};
#else
struct s_float {
    unsigned mantisse : 23;
    unsigned exponent : 8;
    unsigned sign : 1;
};
#endif

union {
    float f;
    struct s_float sf;
} ieee;
```


B.4 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



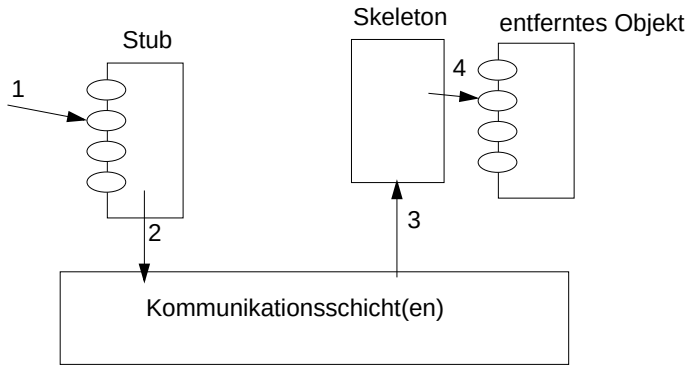
1 RPC-System in Aufgabe 4



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

2 Stubs und Skeleton

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



3 Stub

- implementiert den gleichen Typen wie das entfernte Objekt (gleiches Interface)
- verpackt einen Methodenaufruf in eine Anfrage:
  - Objekt ID, Methoden ID, Parameter, ...
- verwendet die Kommunikationsschicht um eine Anforderung zu versenden
- transformiert das Rückgabeobjekt in den entsprechenden Typ

4 Skeleton

- ruft Methoden am "echten" Objekt auf
- notwendige Informationen:
  - Objektreferenz (z.B. aus Konstruktor)
  - Methoden ID
  - Parameter

3 Stub Beispiel

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

```
short ExampleStub::testMethode(const int16_t value){

    // Anfrage erstellen
    Buffer buf(Request::HDR_SZ + sizeof(value));
    Request req(oid, testMethode_mid, &buf);
    req << 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;

    //fertig
    return result;
}
```


4 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(m.getBuffer());
        r << result;;
        return r;

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

5 Brooker

- 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

5 Initialisierung Stub und Skeleton in Aufgabe 4

- Beispiel - Client:

```
MultiplyServerStub stub(oid, comSys);

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

- Beispiel - Server:

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

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

- Beispiel Dispatcher
  - ◆ empfange Paket (oid, mid, parameter)
  - ◆ suche Skeleton und rufe dort "invoke" mit entsprechenden Parametern auf
  - ◆ Skeleton sendet das Ergebnis selbst zurück

B.5 Zusammenfassung

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