E 6. Übung

E.1 Überblick

- Fragen zu Aufgabe 2
  - Thread-Klasse von Teilaufgabe a
  - Templates für Teilaufgabe b
- Aufgabe 3
  - Überblick: Architektur eines RPC Systems
  - Stubs und Skeletons
- Marshalling primitiver Datentypen
  - Byteorder
  - Fließkommawerte

E.2 Fragen zu Aufgabe 2

1 Thread Klasse: möglich Lösung

- Schnittstelle

```
// Interface

class Runnable{
 public: virtual void run() = 0;
};

class Thread {
    private:
        THREAD_DESC t;
        static void *run_fn (void *);
    public:
        Thread(Runnable *run);
        void join();
};
```


1 Klasse Thread: möglich Lösung

- Implementierung:

```
class Thread {
    private:
        THREAD_DESC t;
        static void *run_fn (void *);
    public:
        Thread(Runnable *run);
        void join();
};
```

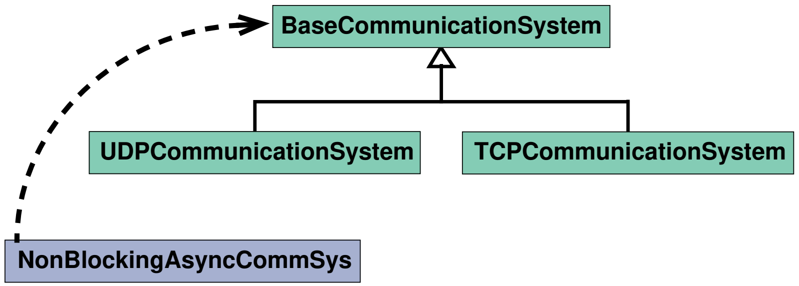
```
void* Thread::run_fn(void *run){
  Runnable *r =(Runnable *)run;
  r->run();
  return NULL;
}

Thread::Thread(Runnable *run){
    if (pthread_create(&t, NULL, run_fn, run) != 0){
        perror("Thread::start: pthread_create failed");
        t = 0;
    }
}

void Thread::join(){
    if (pthread_join(t, NULL) != 0)
        perror("thread::join: pthread_join failed");
}
```


2 Templates für Teilaufgabe b

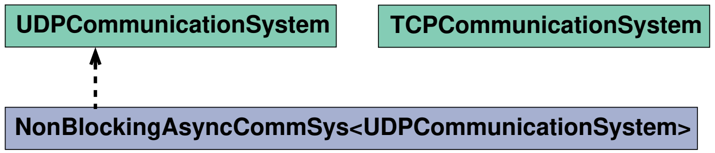
- Aufgabe:
  - Implementierung verschiedener Kommunikationsvarianten
  - basierend auf einem Basiskommunikationssystem
  - welches eine einheitlicher Schnittstelle anbietet
- Die Wahl des Basissystems soll/kann zum Erstellungszeitpunkt getroffen werden.
- Lösungsmöglichkeiten:
  - durch Ableitungshierarchie des Basissystems
  - mit Hilfe von Templates

2 Lösungsmöglichkeiten für Teilaufgabe b)

- Lösungsweg 1: Ableitungshierarchie



- Lösungsweg 2: mit Hilfe von Templates



2 Ableitungshierarchie

- **UDPCommunicationSystem** von **BaseCommunicationSystem** abgeleitet

```
class UDPCommunicationSystem
                          : public BaseCommunicationSystem {
 public:
    void send(Address dest, Buffer *message);
    void register_send_handler(send_signal_t fn);
    void register_receive_handler(...),
...
}
```

- **NonBlockingAsyncCommSys** verwendet **BaseCommunicationSystem**

```
class NonBlockingAsyncCommSys {
    private:
        BaseCommunicationSystem *com;
    public:
        NonBlockingAsyncCommSys(
                BaseCommunicationSystem *com) : com(com){};

        void send(Address dest, Buffer *message)
            { com->send(dest, message);}
...
}
```


2 Templates

- **UDPCommunicationSystem** hat keine Oberklasse

```
class UDPCommunicationSystem
                        : public BaseCommunicationSystem {
public:
    void send(Address dest, Buffer *message);
    void register_send_handler(send_signal_t fn);
    void register_receive_handler(...),
...
}
```

- **NonBlockingAsyncCommSys** ist durch Template parametrisiert

```
template <class BaseCommunicationSystem>
class NonBlockingAsyncCommSys {
    private:
        BaseCommunicationSystem *com;
    public:
        NonBlockingAsyncCommSys(
                BaseCommunicationSystem *com) : com(com){};

        void send(Address dest, Buffer *message)
            { com->send(dest, message);}
...
}
```


2 Unterschiede

- Vergleich der Grösse (an diesem Beispiel!):
  - Ableitungshierarchie

```
-rw-r----- 1 felser i4staff 9180 May 12 16:53 base.o
-rw-r----- 1 felser i4staff 4292 May 12 16:53 main.o
```

  - Templates

```
-rw-rw-r-- 1 felser i4staff 6724 May 12 16:53 base.o
-rw-rw-r-- 1 felser i4staff 5136 May 12 16:53 main.o
```

- Vergleich der Ausführungsgeschwindigkeit ($1*10^9$ Aufrufe von **register_receive_handler**)
  - Ableitungshierarchie: 22.580 s
  - Templates: 21.880 s

E.3 Aufgabe 3

- Pufferklasse
  - mit Marshalling-Funktionen
  - ohne Puffer-Management
  - "stromorientiert"
- Stub und Skeleton
  - für ein Objekt
  - incl. Dispatcher zur Verwaltung von mehreren Objekten

1 Pufferklasse

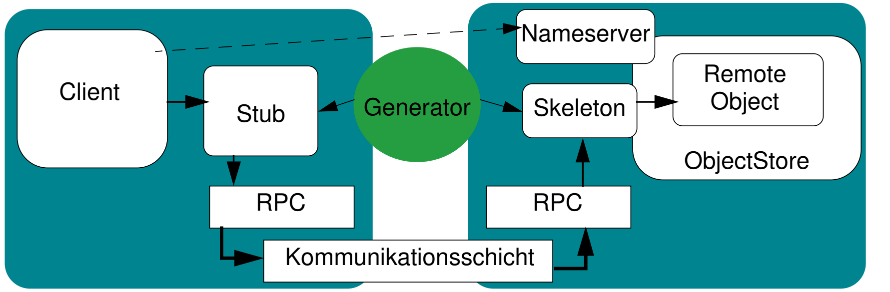
- Schnittstellen-Beispiel

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

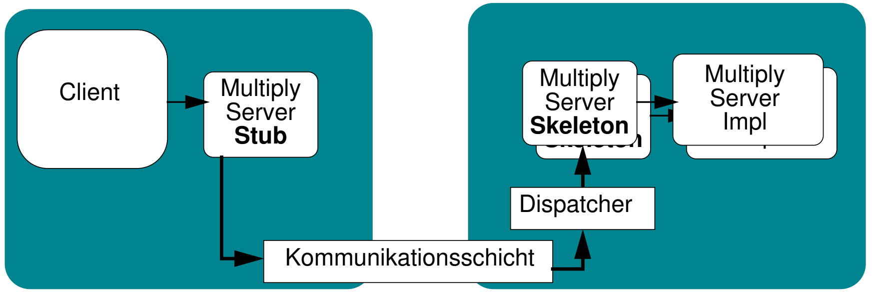
```
class ResizeHandler{
 public:
  virtual char *resize
    (char *old_buffer) = 0;
};
```


2 Überblick: Stub und Skeleton im RPC-System

- *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



2 RPC-System in Aufgabe 3



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

2 Stub und Skeleton in Aufgabe 3

- Beispiel - Client:

```
MultiplyStub stub(oid, comSys);

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

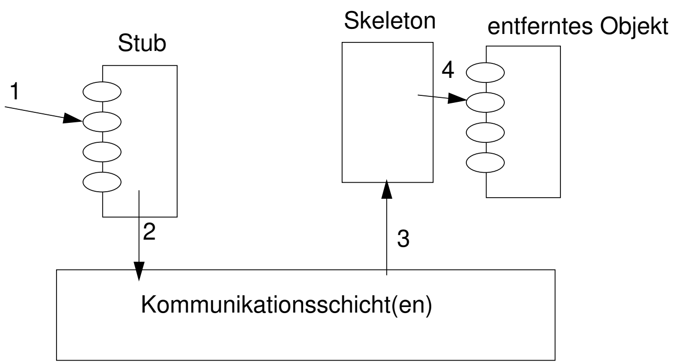
- Beispiel - Server:

```
ExampleObjImpl obj;
ExampleSkel skel(&obj);

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

- Beispiel Dispatcher
  - ◆ empfange Paket
  - ◆ suche Skeleton und rufe dort "invoke" mit entsprechenden Parametern auf
  - ◆ sende Ergebnis

E.4 Stubs und Skeletons

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



1 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

1 Stub Beispiel

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

```
short ExampleStub::shortTest(const short value){

    // Anfrage erstellen
    Buffer *buf = new Buffer(Request::HDR_SZ+sizeof(value));
    Request m(oid, shortTest_mid, buf);
    m << value;
    // los geht's
    c->send(m.getBuffer());

    // warten auf Antwort
    c->receive(buf);
    // Antwort auspacken
    Result r(buf);
    short result;
    r >> result;
    delete buf;
    //fertig
    return result;
}
```

2 Skeleton

- ruft Methoden am "echten" Objekt auf
- notwendige Informationen:
  - Objektreferenz
  - Methoden ID
  - Parameter

2 Skeleton Beispiel

- Beispiel: Skeleton

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

        // Parameter auspacken
        short param1;
        m >> param1;

        // Methode aufrufen
        short result = obj->shortTest(param1);

        // Antwort verpacken
        Result r(m.getBuffer());
        r << result;;
        return r;

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


E.5 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

E.5 Heterogenität bei primitiven Datentypen

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

	0	1	2	3
big endian	14	a2	b5	c8
little endian	c8	b5	a2	14

0x14a2b5c8

- 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**

E.5 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?

E.5 Heterogenität bei primitiven Datentypen

- "Früher" machte hier jeder, was hier 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

E.5 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

E.5 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;
```

E.6 Zusammenfassung

- Einsatzbeispiel von Templates
- Anteil von Stub und Skeleton in einem RPC-System
  - Aufgabe 3
- Stubs und Skeletons
  - einpacken von Parametern und Rückgabewerten
  - Marshalling
- Marshalling
  - Byteorder
  - Fließkommawerte durch IEEE-Standard meist unproblematisch