Übungsaufgabe #4: Basisroutinen für das Marshalling

24.05.2004

In Aufgabe 3 wurde eine grundlegende Abstraktionsschicht für eine nachrichtenbasierte Kommunikation zwischen Rechnern entworfen. Für ein Fernaufruf-System sind darauf aufbauend Marshallingroutinen notwendig, die es erlauben, verschiedene Datentypen in eine Nachricht zu verpacken. Diese kann dann über die Kommunikationsschicht verschickt und beim Empfänger wieder entpackt werden. In dieser Aufgabe sollen nun solche Marshallingroutinen für primitive Datentypen und ausserdem ein Skeleton und Stub für einen einfachen Fernaufruf an einem Testobjekt erstellt werden.

a) Marshalling

Im ersten Teil dieser Aufgabe sind Marshallingroutinen für primitive Datentypen zu implementieren. Konzeptionell ist hierzu eine Klasse `Message` bereitszustellen, die es mittels `write`-Methoden erlaubt, Datentypen in einen internen Puffer zu schreiben (Marshalling) bzw. mittels `read`-Methoden diese aus diesem Puffer zu lesen. Ein Positionszeiger im internen Puffer wird bei jeder Operation weitergesetzt.
Die Pufferverwaltung selbst soll ausserhalb der Klasse `Message` geschehen. Das heisst, ein `Message`-Objekt bekommt im Konstruktor einen Puffer übergeben. Zum Marshalling ist dies ein leerer Puffer, zum Demarshalling ein Puffer mit den Daten, die über das Netzwerk empfangen wurden.
Es kann das Problem auftreten, dass die benötigte Puffergrösse nicht a priori bekannt ist, und der vorgesehene Puffer nicht ausreicht. Dies ist sowohl bei `write` als auch bei `read` zu überprüfen; alle diese Methoden liefern einen Status, den Erfolg (`TRUE`: Puffer reicht aus), zurück.
Der Anwender kann aber auch wünschen, den Puffer bei Bedarf dynamisch vergrössern zu können. Hierzu kann ein resize-Handler an der Klasse `Message` installiert werden, der ggf. den Puffer vergrössern kann. Als Standardvorgabe ist der Handler `null` und wird dann nicht ausgeführt.
Zu beachten ist die hardwareunabhängigkeit des erzeugten Netzwerk-Formats. Auf jeden Fall muss die eigene Implementierung zwischen Sparc-Solaris (Big Endian) und x86-Linux (Little Endian) Interoperabilität bereitstellen.

Das Interface könnte in etwa wie folgt aussehen:

```
class ResizeHandler {
     public: virtual Buffer *resize(Buffer *old_buffer) = 0;
};
class Message {
public:
     Message(Buffer *buffer);
     bool write(int8_t c);
     bool write(int16_t s);
     bool write(int32_t d);
     bool write(float shr);
     bool write(double dbl);
     bool read(int8_t &c);
     bool read(int16_t &s);
     bool read(int32_t &d);
     bool read(float &shr);
     bool read(double &dbl);
     void reset(); // set buffer position pointer to 0
     Buffer *getBuffer();
     void register_resize_handler(ResizeHandler *hndl);
};
```

b) Beispielanwendung mit Stub und Skeleton

Mit Hilfe der Marshalling-Funktionen aus `Message` und den Nachrichtenmechanismen aus Aufgabe 3 soll nun eine einfache Klient-Server-Beispielanwendung erstellt werden. Hierzu sind zu implementieren:

- Ein Test-Klient, der eine Initialisierung vornimmt, einen lokalen Stub erzeugt und mit dessen Hilfe Fernaufrufe am Server vornimmt.
- Ein Stub, der die Aufrufe des Klienten über die Marshalling-Routinen verpackt, zum Server schickt, auf eine Antwort wartet, das Ergebnis wieder entpackt und zum Klienten zurückliefert.
- Ein Server-Skeleton, der Aufrufe entgegennimmt, entpackt, an die eigentliche Implementierung weiterreicht, und das Ergebnis wieder verpackt und zurückschickt.
- Eine Implementierung des Server-Objekts.
- Einen Server, der die Initialisierung übernimmt, dabei eine Instanz eines Server-Objekts und des passenden Skeletons erzeugt, und dann auf Aufrufe wartet.

Das Server-Objekt soll diese Schnittstelle bereitstellen:

```
class MultiplyServer {
    int16_t multiply(int16_t val1, int16_t val2);
    int32_t multiply(int32_t val1, int32_t val2);
    float multiply(float val1, float val2); d
    ouble multiply(double val1, double val2);
}
```

Beachtet werden soll weiterhin folgendes:

- Falls bei der Kommunikation unerwartete Fehler auftreten, soll der Stub dies dem Klienten durch eine Exception signalisieren.
- Es gibt 2 Nachrichtentypen (Request und Reply)
- In alle Nachrichten sollen eine Typ-ID (Request oder Reply) sowie eine Objekt-ID codiert werden. Diese wird bei Erzeugung des Skeltons auf Server-Seite festgelegt (und z.B. am Bildschirm ausgegeben). Es soll also prinzipiell auch möglich sein, auf Serverseite mehrere Objekte (Instanzen einer oder mehrere Klassen) zu verwalten. Bei eingehenden Aufrufen ist entsprechend der Objekt-ID zu unterscheiden, an welches Objekt der Aufruf gehen soll.
- Der Klient bekommt Rechnername, Port und Objekt-ID des Servers über die Kommandozeile.

Weitere Hinweise (incl. Lösungsskize) werden in der Übungen am Mittwoch gegeben.

<u>Abgabe</u>: bis 08.06.2004/12:00 Uhr
Abgabe mittels /proj/i4vs/pub/abgabe