Praktikum Verteilte Systeme

Das Praktikum im Fach Verteilte Systeme findet in den Poolräumen SI0024/25 des Laborbereichs Technische Informatik statt. Auf den Web-Seiten des Laborbereichs finden Sie Informationen, wie Sie von außerhalb auf die Rechner im Pool zugreifen können. Für das Praktikum ist eine Vorbereitung erforderlich. Während des Praktikums ist ein Praktikumsprotokoll mit Namen, E-Mail-Adresse, Datum und allen wichtigen Ergebnissen zu erstellen und mit dem Betreuer am Bildschirm durchzusprechen. Das Protokoll wird zusammen mit dem gezippten Source-Code im OSCA-Dokumentenabgabeordner unter `praktx_Name1_Name2.doc` abgelegt (x=Nr. der Praktikumsaufgabe)

Bitte wählen Sie bei allen Aufgaben andere Dateinamen als die im Skript verwendeten!

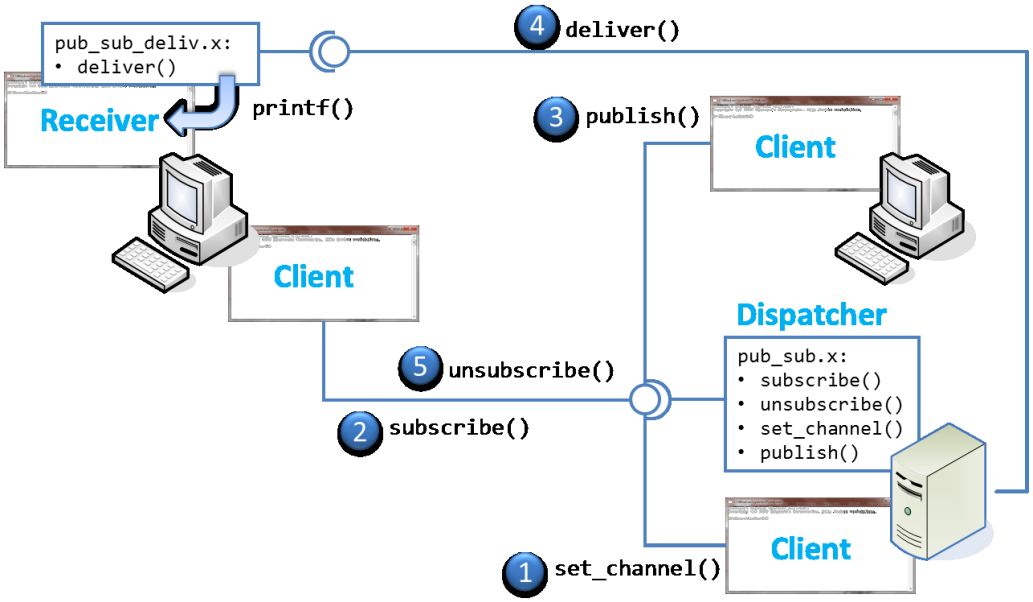
Praktikumsaufgabe 4 - Publish-Subscribe-System mit RPCs

Ziel dieses Praktikums ist es, ein System zum Austauschen von Nachrichten, die in Nachrichtenkanälen gruppiert (sog. *Channels*) sind, durch RPC-Schnittstellen umzusetzen . Das System arbeitet dabei nach dem Publish-Subscribe-Prinzip. Die Aufgabe muss bis

Montag: 08.04.2019
Dienstag: 09.04.2019
Donnerstag: 11.04.2019

zu 80% fertiggestellt werden. Als Vorlage dienen die Beispiele **`rls`** und **`math`** aus **`kap5beispiele.zip`**.

Systembeschreibung



- Ein zentraler Verteiler-Prozess (Dispatcher) implementiert das Interface **`pub_sub.x`**. Es enthält Funktionen zur Registration / Deregistration von Empfängern, die in einer Liste verwaltet werden, sowie zum Veröffentlichen von Nachrichten. Außerdem kann per **`set_channel()`** ein Topic gesetzt werden, welches beim Ausliefern der Nachrichten mitgeschickt wird. Das Setzen des Topics soll nur vom lokalen Rechner, auf dem der Dispatcher läuft, möglich sein.
- Nachrichten können per **`publish()`** verschickt werden.
- Der Empfang von Nachrichten erfolgt durch einen separaten Prozess (*Receiver*), der die Schnittstelle **`pub_sub_deliv.x`** implementiert. Die über die Funktion **`deliver()`** empfangenen Nachrichten werden der Einfachheit halber auf der Konsole ausgegeben. Der Aufruf des RPC soll *asynchron* (also aus Sicht des Aufrufers nicht blockierend) erfolgen. Die Steuerung des Nachrichtenempfangs erfolgt über einen *Client*, der auf der Seite des Receivers läuft. Dieser ruft das Interface des Dispatchers auf, indem er

1. nach dem Start des Receivers, den lokalen Rechnern per **`subscribe()`** für den Nachrichtenempfang registriert
2. und diesen nach Beendigung des Receivers per **`unsubscribe()`** deregistriert.

Hinweise:

- Die o.g. Schnittstellen-Dateien werden vorgegeben. Sie finden Sie im Lernraum zur Veranstaltung unter **`Input_prakt04.zip`**.
- Bei der Implementierung Verteilter Systeme sollte man generell auf den sparsamen Umgang mit Netzwerkressourcen achten. In der vorliegenden Aufgabe wird dem dadurch Rechnung getragen, dass die Rückgabewerte der Funktionen des Interfaces **`pub_sub.x`** in einem **`short`**-Wert kodiert werden. Die Semantik wird in der Header-Datei **`return_codes.h`**. Die dort definierten Werte können als Indizes auch zur Ausgabe der entsprechenden Statusmeldungen im Client verwendet werden.
- Sie können für das Einrichten der notwendigen Projekte auf die Beispiele aus der Vorlesung zurückgreifen. Achten Sie aber darauf, dass Sie die **Bezeichner sinnvoll abändern**!
- Die Bestimmung der IP-Adresse eines RPC-Aufrufers im Server kann über **`char* req_addr = inet_ntoa(request->rq_xprt->xp_raddr.sin_addr)`** erfolgen.
- Beachten Sie das in der Vorlesung besprochene Speichermanagement bei der Abwicklung eines RPC:
  Der Speicher für dynamische Datentypen muss bis zur Fertigstellung der Filter-Funktion vorgehalten werden. Die Filter-Funktion gibt danach den Speicher frei. Soll mit globalen Daten (z.B. Zeichenketten) nach dem RPC weitergearbeitet werden, so muss das Datum also dupliziert werden (z.B. per **`strdup`**), ansonsten kann es zu einem Segmentierungsfehler kommen.
- Receiver und Client sind voneinander getrennte Prozesse. Versuchen Sie sich klar zu machen, warum dies so ist! Als freiwillige Zusatzaufgabe können sich überlegen, wie man den Receiver als Kindprozess des Clients vor der Registration starten (**`fork`**) und vor
- Deregistration (**`kill`**) beenden kann.

Hinweis zur Fehlersuche:

Kompilieren Sie mit **`gcc`** und **`-g`** zur Erzeugung von Debug-Informationen. Speicherprobleme können mit **`valgrind`** gefunden werden**:** **`valgrind -v <ausführbare Datei>`**
Auf manchen Linux-Systemen muss der Aufruf mit **`/usr/bin/valgrind`** erfolgen**.**