UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Excercise Sheet 2
Solution

Lecture Distributed Systems
Summer Term 2020

Exercise 1: Communication forms in distrubited systems

In the case of reliable message streams, a virtual connection is usually established between two communication partners, which is then used for data exchange. This ensures that no packets are lost (unless the connection breaks down, which is determined by both partners). Furthermore, the packets are received in exactly the same order as they were sent. The connection behaves essentially like a pipe.

Datagrams, on the other hand, are individual, independent packets. These can be lost in case of problems on underlying network layers without either side noticing. There are no guarantees about the packet order either.

The advantages of a reliable connection are obvious. Possible areas of application are the transmission of large amounts of data (e.g. ftp) or generally all types of connections that are session-oriented (e.g. telnet, ssh).

Datagrams are used if you want to save the additional overhead and can cope with lost and exchanged packets. Multicast connections are also possible with datagrams. One field of application is e.g. web radio. Protocols based on the datagram service ensure that changes in sequence and packet loss are detected. Individual lost packets can usually be compensated by interpolating the audio stream. In general, all real-time protocols where a low latency time is more important than the completeness of the data use datagrams. Another example are position transfers in multi-user games.

Exercise 2: Conversion of data formats

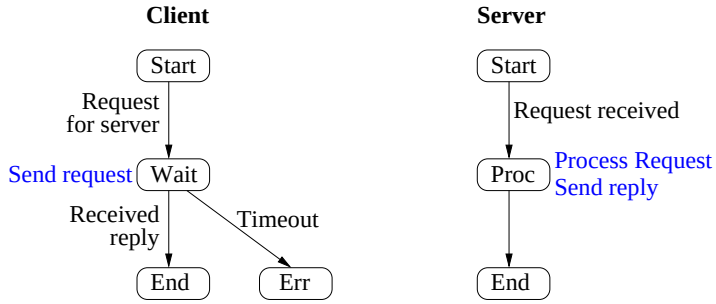
The data must be converted because a distributed system can generally be heterogeneous. Therefore, the internal representation of the data on the individual nodes of the system can be different. The most important type of heterogeneity here is the use of different processors and operating systems. To ensure that the different processes still understand each other, the data is usually converted into a common (canonical) network data format before it is sent and then converted back again at the receiver.

Important aspects for different data types:

- *Integer types:* The most common problem here is that of the *byte order*, i.e. the order in which the individual bytes of a larger integer type are stored in memory. A distinction is made between *big endian* and *little endian*. If no conversion were carried out, the (unsigned) big endian 16-bit number 3 on a little endian system would become the number 49152. Here it is usually sufficient to agree on one of the two formats and convert if necessary.
- *Floating point:* Here the situation is a little bit more complicated, because floating point numbers are not so simple. Essential problems here are different number of bits at mantissa or exponent. Further the numbers can be stored with different bases, the most common numbers are those to base 2 and 10. Usually, an exact 1-to-1 mapping of all floating point numbers of one system to those of another system is not possible. Today, however, most computers work with the standardized representation according to IEEE 754.
- *Strings:* Here a conversion between different character sets is necessary, because a byte or word in different character sets is interpreted as different characters. Some characters are completely missing in some character sets. There are also character sets with 8-bit (e.g. ASCII, ISO-8859-1), 16-bit (e.g. Java: Unicode) and 32-bit characters.

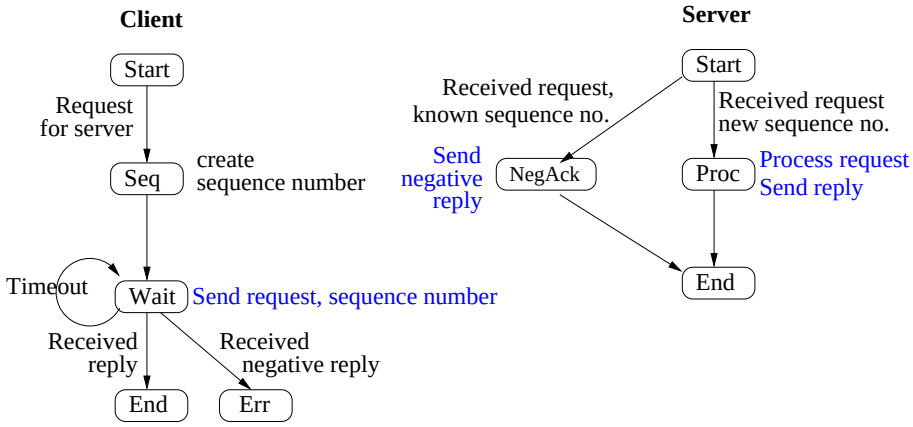
Exercise 3: Semantics of client/server communication

a) The following figure shows state diagrams for client and server with an implementation of the *at most once* semantics:



The server remains unchanged here. However, the client also terminates when a timeout event occurs. In this case, it is not certain whether the request was processed on the server or whether the request message to the server was actually lost.

The following figure shows the state diagrams for client and server with an implementation of the *exactly once* semantics:



Here each transmitted request is assigned a unique sequence number, which remains the same even if it is repeated. The client behaves like in the implementation of the *at least once* semantics: it repeats the request to the server until it receives a response. The server, on the other hand, checks when a request arrives whether it has not yet processed the request with this sequence number. If this is the case, the request is processed and the result sent to the client. If, however, the sequence number is already known, then in the simplest case a negative response is sent directly to the client without the request being executed again. In general, however, the originally calculated response should be cached in order to send it to the client again. To do this, the server must buffer the last response for each client (for an indefinite time). Only when the next request from the same client arrives can the server assume that this client has received the last reply.

b)

at least once semantics	at most once semantics
pressing an elevator button	write/append data to a file
translating a program	ordering a pizza
to get a bank account statement	to make an electronic transfer
	cast one vote in an electronic voting service

Exercise 4: Request/reply protocol

The three operations *doOperation*, *getRequest* and *sendReply* must be implemented on the basis of *send* and *recv*. Since no fragmentation of messages is necessary, only one packet needs to be sent per message. To keep the protocol free of overhead as much as possible, the basic idea is as follows:

For a complete request/reply cycle exactly 2 packets are sent, namely the request of the client itself and then the response of the server to it. The reply also serves as implicit confirmation of the receipt of the message. If there is no reply within

a certain time, it is assumed that the packet has been lost and must be resent. In order to associate requests and responses, unique sequence numbers are generated and appended to the messages. The following pseudo code excerpt shows a typical sequence in the client. The client gets the address of the server (e.g. from a name service) and generates a request message. It then sends the request to the server and processes the response:

```
severaddr = getServerAddr();
request = createRequest();
doOperation(serveraddr, request, repl);
handleReply(repl);
```

The server typically runs in an infinite loop in which it receives arbitrary requests, processes them and finally sends back a corresponding response.

```
for ( ; ; )
{
  getRequest(clientaddr, request);
  repl = handleRequest(request);
  sendReply(clientaddr, repl);
}
```

The following are examples of implementations of the three required operations:

a) The *doOperation* operation is implemented by sending the packet using *send* and then waiting for a response. If this response is not received within a certain time (here 5s), the packet is sent again. A serious error handling should give up after a maximum number of repetitions.

```
doOperation(in s_address, in request, out reply)
{
  ok = false;
  while (! ok)
  {
    send(s_address, request);
    ok = recv(s_address, reply, 5s);
  }
}
```

b) *getRequest* is the most complex function. It waits for the arrival of any message. Usually this message is received by *recv* and the message ID is stored in an associative array (map). The function then returns. However, if a request is received repeatedly (i.e. it is already entered in the map), it is checked whether an answer has already been generated. If this is the case, it is sent again. If not, the message is ignored.

```
getRequest(out address, out request)
{
  newrequest = false;
  while (! newrequest)
  {
    recv(address, request, -1);          // no timeout
    if (map.lookup(request.id, reply))
    {
      if (reply != 0)
        sendReply(address, reply);
    }
    else
    {
       newrequest = true;
       map.store(request.id, 0);
    }
  }
}
```

c) *sendReply* is implemented by simply sending the reply packet. The response is then stored in the existing map entry so that it can be sent if the packet is lost in the network.

```
sendReply(in address, in reply)
{
  send(address, reply);
  map.store(reply.id, reply) // store reply in a map
}
```

Exercise 5: Middleware

In principle, a reliable multicast service could simply be part of the transport layer or even the network layer. For example, the unreliable IP multicasting service is implemented in the network layer. However, because these services are not yet available, they are generally implemented using services of the transport layer, which automatically places them in the middleware. However, when it comes to scalability, reliability can only be guaranteed if application requirements are taken into account. This is a strong argument for implementing such services on higher, less generic layers.

Exercise 6: Transparency of RPC

The system call would be executed on another node. This is problematic, for example, if you want to read from an open file (it would not be accessible on the other node, or at least not open). Screen output or setting an alarm timer would also be problematic.

As a remedy you can intercept system calls and send them (again via RPC) back to the client computer.