LS Lab Assignment:
Simple Network File Server[*]

A. Bakker Vincent Breider Roy Vermeulen
{Arno.Bakker@os3.nl}

Feedback deadline:
November 22, 2021, 10:00 CET

Abstract

In this exercise you will develop a simple Network File Server (NFS) that allows you to access files on a remote server from your local machine. More advanced implementations of this concept are Sun's Network File System and Microsoft's Common Internet File System (CIFS), better know as the Samba file system. You will build this NFS system in a step-wise fashion, starting with a echo client and server to exchange messages over a network.

The goal of this exercise is to improve understanding of the Distributed Systems concepts of communication layers, virtualisation, and remote-procedure call (RPC).

1 Echo Client and Server

An echo server receives a message from the network and sends it back unmodified to the echo client that sent it.

→ Exercise:: Write a simple client and server program in Python 3 that can echo messages over the TCP protocol. Publish the code on your Wiki.

In this case you are allowed to use solutions from the Internet (provided you reference the source). Test the code by running the client and server on your desktop machine. Once that works, try running the echo server on your server. You can use GIT via `gitlab.os3.nl` for getting your source code on a different machine.

In the upcoming exercises you can benefit from a small helper class called `DatagramConnection` that we wrote to send blocks of data unmodified over a TCP connection. This is needed because a TCP connection is defined as delivering a stream of bytes, not a stream of blocks. This means that if you send 1024 bytes, it may be received as two blocks 512 bytes at the receiver, or vice versa. Standard procedure is to prefix the data with a 4 byte length field in network byte-order. You will find the `DatagramConnection` class on the Wiki.

[*]Based on an earlier document by Arno Bakker, Jeroen van der Ham. Version December 5, 2024

→ Exercise:: Reimplement your client and server to use the DatagramConnection class to wrap the (non-listening) UNIX sockets.

2 Reading and Writing Local Files

Next, consider this simple file-server interface:

```
class FileIO:

    def read(self,filename,offset,length):
        """ Read data from a file.
        @param filename The file to read from.
        @param offset   The offset to start reading from.
        @param length   The number of bytes to read.
        @return The data read or '' in case of error.
        """
        pass

    def write(self,filename,offset,block):
        """ Write data to a file.
        @param filename The file to write to.
        @param offset   The offset to write at.
        @param block    The block of data to write.
        @return The number of bytes written or -1 in case of error.
        """
        pass
```

→ Exercise: Implement the two methods such that they read and write to the local file system, by creating a class `LocalImpl` that inherits from this interface class. Publish the code on your Wiki.

Test this class by writing a few lines of text to a file, and reading them back to see if your writes were executed correctly, as follows.

```
import sys

from local import LocalImpl as FileIOImpl


if __name__ == "__main__":
    fs = FileIOImpl()
    lines = ["Hallo\r\n", "Hoe gaat het?\r\n", "CU\r\n"]
    offset = 0
    for line in lines:
        ret = fs.write("file.txt",offset,line)
        print ("write returns",ret,"is same as line length?",len(line)
== ret,file=sys.stderr)
        offset += len(line)

    offset = 0
    for line in lines:
        ret = fs.read("file.txt",offset,len(line))
        print (sys.stderr,"read returns original line?",line ==
ret,file=sys.stderr)
        offset += len(line)
```

You can find this test code on the Wiki.

→ Show a succesful execution of your local implementation

3 Network File Server

You are now going to implement a remote server and client that will allow you to read/write files stored on that remote server. To do so, you should create an implementation of the `FileIO` interface that encodes the calls made to it and transmits them to a remote server. In other words, you should implement a Remote Procedure Call (RPC) system, that takes a call e.g. `read("file.txt",0,100)` on the client side, encodes this in a network message, transports it to the server and decodes and executes the call on the local implementation you just wrote. The process of encoding these calls in such a form that they can be transmitted over the network is called marshalling, or serialisation.

→ Exercise: Implement the FileIO interface to use a remote server.

- This involves creating a client-side implementation of FileIO that marshalls the arguments to the method calls and ships them to your server process over TCP. This is called the client stub.
- Your server process should unmarshall the parameters, and then perform the read or write operation using the local implementation you just wrote and tested. The code that does this is called the server stub or skeleton.
- Finally, the server should return the result back to the client over TCP.

  - Hint: use a Python dictionary encoded with JSON for marshalling the operation, or use a binary representation.

Publish the code on your Wiki.

→ Show a succesful execution of your remote implementation. Client and server-side.

What you have just created is a simple network file server (note the security holes!). There are a number of important concepts illustrated in this exercise. First, notice how the interface for reading and writing files remained the same, whether you were writing to local files, or to files on the remote server. This example illustrates virtualisation. If you would extend the `FileIO` interface to include other operating-system interfaces you could even build a type of virtual machine.

Second, you have also built a remote-procedure call (RPC) system. The client calls the methods in the FileIO interface, and this call is then transmitted to the remote server to be executed. Again, this provides transparency to the programmer, in this case distribution transparency. If you take the test script from the previous section, and just change the class name `LocalImpl` to the name of your client stub class (e.g. `ClientImpl`) the program will work as before, just with remote execution. In theory, the programmer does not have to concern her/himself with how the method is implemented. In practise, things are a little different, as you will see later in the course when we discuss making RPC/NFS tolerant of packet loss and crashes.

4 Real-World Network File Systems

The traditional UNIX file interface is different from the FileIO interface you have been using:

```
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
```

→ Question: what is the major difference between these two interfaces, and how does that affect the server implementation?

True complexity comes when errors and crashes are taken into account, as you will see in Chapter 8 of the DS book. Hence, in practice, to handle these (when possible!), and keep as much of the traditional UNIX interface as possible, the Sun engineers had to write a completely custom implementation and network protocol on both client and server side. See Section 2.4 in the DS book. This again illustrates that RPC is an appealing idea, but its benefits are hard to obtain in practice.