# Practical Work #2 – RPC File Transfer

Nguyen Tien Dung – 23BI14116

December 4, 2025

## 1 Goal of the practical work

The goal of this practical work is to upgrade the TCP file transfer system from Practical Work #1 into a Remote Procedure Call (RPC) based file transfer system.

Instead of manually working with sockets and byte streams, the client calls a remote procedure `GET_FILE` that returns the requested file as a typed result. The lab focuses on:

- Designing a simple RPC interface for one–to–one file transfer.
- Generating client/server stubs using `rpcgen`.
- Implementing the server and client logic in C.
- Building and running the system on Kali Linux using `libtirpc` and `rpcbind`.

## 2 Environment setup

All experiments were done on a Kali Linux virtual machine running inside VMware Workstation.

### 2.1 ONC RPC libraries

Kali uses `libtirpc` instead of the legacy `rpc/rpc.h` headers. I installed the necessary packages as follows:

Listing 1: Installing RPC dependencies

```
sudo apt update
sudo apt install rpcbind libtirpc-dev
```

`rpcbind` is required at runtime to register RPC services, while `libtirpc-dev` provides the header files and library needed for compilation.

### 2.2 Network issues and fix

At first the VM had no network connectivity (ping to `8.8.8.8` failed). I fixed this by:

- Setting the VMware network adapter to *Bridged* mode.
- Restarting NetworkManager and renewing DHCP:

```
sudo systemctl restart NetworkManager
sudo dhclient -r eth0
sudo dhclient eth0
```

After this, `apt` could download packages and the rest of the lab could continue.

# 3 RPC interface design

## 3.1 file_transfer.x

I designed a minimal RPC interface with one remote procedure `GET_FILE`. The RPC definition file `file_transfer.x` is:

Listing 2: RPC specification in file$_transfer.x$

```
const MAXFILESIZE = 1048576; /* 1 MB */

typedef string filename_t<>; /* requested file name */
typedef opaque filedata_t<>; /* raw file data bytes */

struct file_result {
    int status; /* 0 on success, non-zero = errno */
    filedata_t data; /* file content if status == 0 */
};

program FILE_TRANSFER_PROG {
    version FILE_TRANSFER_VERS {
        file_result GET_FILE(filename_t) = 1;
    } = 1;
} = 0x31234567;
```

- `filename_t`: the name of the file requested by the client.
- `filedata_t`: a variable-length opaque byte array containing the file data.
- `file_result.status`: error code (0 on success, non–zero on error).
- `file_result.data`: valid only when `status == 0`.

## 3.2 Generating stubs with rpcgen

From `file_transfer.x`, I generated the RPC header and stub files:

Listing 3: Generating RPC code

```
rpcgen -C file_transfer.x
```

This command produced the following files:

- `file_transfer.h`: header with data types and function prototypes.
- `file_transfer_xdr.c`: XDR (de)serialization functions.
- `file_transfer_clnt.c`: client stub.
- `file_transfer_svc.c`: server stub and RPC event loop.

I did not edit these generated files except for a small change in the `main` function of `file_transfer_svc.c` to relax error handling for UDP registration in my environment.

# 4 System organization

The final RPC file transfer system consists of:

- `file_transfer.x`: RPC specification.
- `file_transfer.h`: generated header.
- `file_transfer_xdr.c`: generated XDR routines.
- `file_transfer_clnt.c`: generated client stub.
- `file_transfer_svc.c`: generated server stub and RPC loop.
- `server_impl.c`: server-side implementation of `GET_FILE`.
- `client.c`: client application.

# 5 Server implementation

## 5.1 get_file_1_svc

The function `get_file_1_svc` is declared in `file_transfer.h` and implemented in `server_impl.c`. It is called by the server stub when the client invokes `GET_FILE`.

Listing 4: Server-side implementation in server$_i$*mpl.c*

```
#include "file_transfer.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

file_result *get_file_1_svc(filename_t *argp, struct svc_req *rqstp)
{
    static file_result result;
    FILE *fp;
    long filesize;
    char *buf;

    /* Free previous buffer if any */
    if (result.data.filedata_t_val != NULL) {
        free(result.data.filedata_t_val);
        result.data.filedata_t_val = NULL;
        result.data.filedata_t_len = 0;
    }

    result.status = 0;

    fp = fopen(*argp, "rb");
    if (!fp) {
        perror("fopen");
        result.status = errno;
        return &result;
    }

    if (fseek(fp, 0, SEEK_END) != 0) {
        perror("fseek");
        result.status = EIO;
        fclose(fp);
        return &result;
    }

    filesize = ftell(fp);
    if (filesize < 0 || filesize > MAXFILESIZE) {
        fprintf(stderr, "file␣too␣large␣or␣ftell␣error\n");
        result.status = EFBIG;
        fclose(fp);
        return &result;
    }
    rewind(fp);

    buf = malloc(filesize);
    if (!buf) {
        perror("malloc");
        result.status = ENOMEM;
        fclose(fp);
```

```
        return &result;
    }

    if (fread(buf, 1, filesize, fp) != (size_t)filesize) {
        perror("fread");
        result.status = EIO;
        free(buf);
        fclose(fp);
        return &result;
    }
    fclose(fp);

    result.data.filedata_t_val = buf;
    result.data.filedata_t_len = (u_int)filesize;
    return &result;
}
```

**Behaviour.** The function:

1. Opens the requested file in binary mode.
2. Checks its size and ensures it does not exceed `MAXFILESIZE`.
3. Allocates a buffer and reads the entire file into memory.
4. On success, fills the `file_result` structure and returns it.
5. On error, sets `status` to a non-zero error code and returns no data.

# 6 Client implementation

The client application in `client.c` creates an RPC client, calls `GET_FILE` and writes the result to a local file.

Listing 5: Client code in client.c

```
#include "file_transfer.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    if (argc != 4) {
        fprintf(stderr,
            "Usage: %s <server_host> <remote_file> <local_file>\n",
            argv[0]);
        exit(EXIT_FAILURE);
    }

    char *server_host = argv[1];
    char *remote_file = argv[2];
    char *local_file = argv[3];

    CLIENT *clnt = clnt_create(server_host,
                               FILE_TRANSFER_PROG,
                               FILE_TRANSFER_VERS,
                               "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror(server_host);
        exit(EXIT_FAILURE);
```

```
    }

    file_result *res = get_file_1(&remote_file, clnt);
    if (res == NULL) {
        clnt_perror(clnt, "RPC call failed");
        clnt_destroy(clnt);
        exit(EXIT_FAILURE);
    }

    if (res->status != 0) {
        fprintf(stderr, "Server error, status = %d\n", res->status);
        clnt_destroy(clnt);
        exit(EXIT_FAILURE);
    }

    FILE *out = fopen(local_file, "wb");
    if (!out) {
        perror("fopen");
        clnt_destroy(clnt);
        exit(EXIT_FAILURE);
    }

    fwrite(res->data.filedata_t_val, 1,
           res->data.filedata_t_len, out);
    fclose(out);
    clnt_destroy(clnt);

    printf("Downloaded %u bytes to %s\n",
           res->data.filedata_t_len, local_file);
    return 0;
}
```

# 7 Build commands

Because Kali uses `libtirpc`, the compilation commands explicitly add the include path and link against this library:

Listing 6: Build commands for server and client

```
# Generate RPC stubs
rpcgen -C file_transfer.x

# Build the server
gcc -Wall -g -o rpc_server \
    file_transfer_svc.c file_transfer_xdr.c server_impl.c \
    -I/usr/include/tirpc -ltirpc

# Build the client
gcc -Wall -g -o rpc_client \
    file_transfer_clnt.c file_transfer_xdr.c client.c \
    -I/usr/include/tirpc -ltirpc
```

After successful compilation, the directory contains the binaries `rpc_server` and `rpc_client`.

# 8 Execution and test results

## 8.1 Starting rpcbind and the server

Before running the server, I start the `rpcbind` service and check that it is active:

```
sudo systemctl start rpcbind
sudo systemctl status rpcbind
rpcinfo -p
```

Then I start the RPC server (in the lab I run it with `sudo`):

```
sudo ./rpc_server
```

The server runs in the foreground and prints messages for each client request.

## 8.2 Client request and file download

In another terminal, I create a test file and call the client:

```
echo "Hello␣from␣RPC␣server" > test.txt
./rpc_client 127.0.0.1 test.txt test_copy.txt
```

A typical output is:

```
Downloaded 22 bytes to test_copy.txt
```

On the server side, the log shows:

```
Client requested file: test.txt
```

Finally, I verify that the original and the copied files are identical:

```
cat test.txt
cat test_copy.txt
```

Both files contain the same text “Hello from RPC server”, which confirms that the RPC file transfer works correctly.

# 9 Discussion

Compared to the TCP socket implementation from Practical Work #1, the RPC-based solution has several advantages:

- The communication interface is a typed function call (`GET_FILE`) instead of manual send/receive operations.
- The `rpcgen` tool generates XDR code and stubs automatically, reducing boilerplate and potential bugs.
- Extending the service (for example, adding file upload or directory listing) is as simple as adding more procedures to `file_transfer.x`.

However, the current implementation still reads the entire file into memory at once and is limited by `MAXFILESIZE`. A more scalable design would send the file in chunks to support larger files.

## 10 Personal work

For this practical work I:

- Configured the Kali VM networking and installed `rpcbind` and `libtirpc-dev`.
- Designed the RPC interface in `file_transfer.x`.
- Generated the C stubs with `rpcgen`.
- Implemented the server logic in `server_impl.c`.
- Implemented and tested the client in `client.c`.
- Wrote this report documenting the design, implementation steps, build commands and test results.

## 11 Conclusion

In this practical work I implemented a simple RPC-based file transfer service on top of ONC RPC. The system successfully transfers files from the server to the client through the `GET_FILE` remote procedure. The lab demonstrates how RPC can abstract away low-level socket details and provide a convenient programming model for distributed systems.