Proxy Service

Introduction

Proxy servers have existed on the Internet for a very long time. Such proxy servers fulfill an immense variety of jobs, from link-layer operation such as allowing ARP to resolve names in several connected networks to application-layer operations that convert video formats between each other.

Essentially, “proxy” is a word that can be given to anyone or anything that does something on behalf of something else.

This assignment is inspired by IoT systems for home automation. These automation systems comprise a huge number of devices, where some can be tiny low-power nodes, while others have a lot of computing power. Even Cloud servers are part of many IoT systems.

To make development and debugging easy, communication between such IoT nodes is often implemented by sending messages in the XML format. However, some IoT nodes have such little computing power and such bad network connections that the huge overhead of XML cannot be supported. These IoT nodes are then included in the network by adding a proxy very close to that tiny IoT node, but with a bit more computing power and a better network connection. This proxy converts a very efficient but difficult-to-understand binary format to XML on behalf of the IoT node.

The Task

In this assignment, you are going to implement a proxy and a variety of helper functions of a proxy that converts student records back and forth between an XML format and a binary format.

The scenario is stripped down to the very basic functions. There is an example sender of student records in XML format called *xmlSender*, another one that sends records in binary format called *binSender*, and a receiver that can either receive XML or binary format called *anyReceiver*. The senders are reading from disk, the receiver is writing to disk.

All of these connect to the proxy, which acts as a TCP server. When a sender reads a record from disk, it sends it to the proxy. The proxy converts this record into an internal format and checks the destination ID in that record. If there is a node with this ID connected to the proxy, the proxy converts the records to the format (XML or binary) that is appropriate for this node and sends it.

The proxy must support all combinations: XML sender - XML receiver, XML sender - binary receiver, binary sender - XML receiver, binary sender - binary receiver.

Student records

The assignment considers 3 different representations of a student record. There is an internal representation using a data structure named struct Record that we recommend for the implementation of the proxy, and two representations stored on file and sent over the network, using XML and binary format.

XML format

An example record in XML format may look like this:

```
<record>
  <source="A" />
  <dest="D" />
  <username="griff" />
  <id="1003" />
  <group="200" />
  <semester="27" />
  <grade="PhD" />
  <courses>
    <course="IN1020" />
    <course="IN1060" />
  </courses>
</record>
```

This is an XML message that is being sent from node A to node D. It concerns user griff who has user ID 1003 and group ID 200. He's studying in the 27th semester and the highest degree he has received so far is PhD. From all the 1st year courses at IFI, he has taken only IN1020 and IN1060.

Each line in this format is optional. If a user has never taken any 1st year courses at IFI, the entire block from <courses> to </courses> will be absent.

Binary format

The hex representation of the binary version of the same record is

```
ff 42 44 00 00 00 05 67 72 69 66 66 00 00 03 eb 00 00 00 c8 1b 03 00 24
```

- `ff` - the first byte contains 8 flags that indicate which of the entries exist. In this case, all entries exist.
- `42` - the source is B, 42 is ASCII for B
- `44` - the destination is D, 44 is ASCII for D
- `00 00 00 05` - the string length of the username in network byte order. Since it is in network byte order, we can easily read that the length is 5.
- `67 72 69 66 66` - the ASCII representation for the username griff. Note that there is no terminating zero.
- `00 00 03 eb` - 0x3EB is hexadecimal for 1003
- `00 00 00 c8` - 0xC8 is hexadecimal for 200
- `1b` - 0x1b is hexadecimal for 27
- `03` - this is the enumeration value for "PhD" by our choice. This is documented in the precode file record.h

- 00 24 - the hex representation of the big endian int16_t value containing flags for courses taken. IN1020 has value (1<<2) = 4, IN1060 has value (1<<5) = 32. Hex for 36 is 0x24.

Internal format

The internal format that we propose uses the struct defined in record.h. The following shows how a struct Record could be filled manually (preferably you would use the helper functions). Note that r.username is assigned a copy of the string griff with its own heap memory.

```
struct Record r;
r.has_source = 1;       r.source = ‘B’;
r.has_dest = 1;         r.dest = ‘D’;
r.has_username = 1;     r.username = strdup( “griff” );
r.has_id = 1;           r.id = 1003;
r.has_group = 1;        r.group = 200;
r.has_semester = 1;     r.semester = 27;
r.has_grade = 1;        r.grade = Grade_PhD;
r.has_courses = 1;      r.courses = Course_IN1020 | Course_IN1060;
```

You don’t actually have to use `struct Record` but you can benefit from several helper functions that are already provided in the precode.

The Precode

This assignment comes with a considerable amount of precode including a Makefile. By calling make, you are going to create 4 programs:

- binSender
- xmlSender
- anyReceiver
- proxy

These programs compile, but they do not work. All of them rely on at least one file that is incomplete and that you must write, connection.c. The program proxy relies on proxy.c, which is mostly empty apart from command line parsing and some hints. Here is an overview of files that are included in the precode but incomplete:

- connection.c: This file contains a sketch of the functions that implement TCP functionality. It’s header file connection.h includes their declarations. You must implement these functions. All 4 programs depend on them.
- proxy.c: this file contains the main() function of the proxy program. You must extend this file to process an event loop that can listen for newly connecting senders and receivers, disconnecting receivers, and newly arriving messages from any of the connected receivers. When a message arrives and an appropriate receiver is connected, you may have to convert the arriving XML or binary format into the format expected by the receiver. Writing the proxy is the biggest effort of the home exam.
- recordFromFormat.c: These are two helper functions that create internal data structures on the heap called Record from input buffers that contain records in XML or in binary format. You do not have to implement these functions, but we strongly recommend that you use them instead of doing everything in proxy.c. The header file recordFromFormat.h includes their declarations.

Several other files are included in the precode:

- Makefile: our makefile, extended as you like
- anyReceiver.c: contains main() for the program anyReceiver
- binSender.c: contains main() for the program binSender
- xmlSender.c: contains main() for the program xmlSender
- record.c / record.h: declaration of the internal struct Record and a variety of helper functions to create and delete Record objects on the heap and set values in them.
- recordToFormat.c / recordToFormat.h: definition and declaration for helper functions that convert an internal struct Record into buffers containing the record in XML or binary format.
- binfile.c / binfile.h: helper functions used by binSender to read binary format from file.
- xmlfile.c / xmlfile.h: helper functions used by xmlSender to read XML format from file.

Documentation

Write a design document that explains your design choices as well as the code.

The main topics of the design document should be the TCP code you write in connection.c, the design of your event loop in proxy.c and the conversion between XML and binary formats.

Explain what works, and if parts of the task have not been solved successfully, clarify in which way they fail. Your documentation must be a PDF file and it should be between 1 and 2 pages long (10pt font or bigger).

Advice

Development steps

One possible way to solve this task is in the following steps:

1. Start by writing the TCP-related functions in connection.h that make it possible to connect and send data over TCP.
    - Hint: use the program `nc` (netcat) to receive data from your `xmlSender` and print it to stdout.
2. Extend `connection.c` and implement the `proxy.c` until it can receive connection from one or more `xmlSenders` at the same time, receive records in XML format from them and write it to the screen.
3. Extend `proxy.c` until it can also receive connections from `anyReceiver` in XML mode.
    - Send dummy data to the receiver first.
    - After that, send all records that you receive from an `xmlSender` to this `anyReceiver`.
    - After that, send records only to that `anyReceiver` if the ID is matching. Connect several receivers with different IDs and make sure that records are sent only to the correct one using the `dest` field in the record.
4. After that, get `binSenders` to talk with `anyReceiver` in Binary mode.
5. After that, get `binSenders` to talk to `anyReceivers` in XML mode.
6. Finally, get `xmlSender` to talk to `anyReceivers` in Binary mode.

These recommendations are built around the idea that XML is readable and debugging is therefore easier. You don't have to follow the steps in this order.

Using valgrind

To check the program for memory leaks, we recommend that you run Valgrind with the following flag:

```
valgrind \
    --leak-check=full \
    DITT_PROGRAM
```

Submission

You must submit all your code in a single TAR, TGZ or ZIP archive.

Include your Makefile and include all of the precode.

If your file is called `<candidatenumber>.tar` or `<candidatenumber>.tgz`, we will use the command tar on `login.ifi.uio.no` to extract it. If your file is called `<candidatenumber>.zip`, we will use the command unzip on login.ifi.uio.no to extract it. Make sure that this works before uploading the file. It is also prudent to download and test the code after delivering it.

Your archive must contain the Makefile, which will have at least these options:

- `make` - compiles your code into the following executable binaries `proxy`, `binSender`, `xmlSender` and `anyReceiver`
- `make realclean` - deletes the executables and any temporary files (e.g. `*.o`)

About the Evaluation

The home exam will be evaluated on the computers of the login.ifi.uio.no pool. The programs must compile and run on these computers. If there are ambiguities in the assignment text you should point them out in comments in the code and in your documentation. Write about your choices and assumptions in your documentation that is deliver it alongside the code.