COMP3331 Assignment - Ryan McClue (z5346008)

Program Design

Utilise a unity build system to reduce linkage time and remove need for an external build tool. Build by running script `chmod +x build && ./build`, which will produce `client` and `server` binaries.

File Structure

Entrypoints to client and server binaries are `client.c` and `server.c` respectively.

`common.h` contains various common includes and function definitions that are used by both the client and server.

`io.h/io.c` contain functions related to file reading/writing and string parsing

`commands.c` contains functions implementing client side commands

`messages.h` contains definitions for application layer message format

Files produced by `EDG` are placed in the current working directory of the `client` binary and are named `[device-name]-[fileid].txt`

Files uploaded by `UED` are placed in the current working directory of the `server` binary and are named `server-[device-name]-[fileid].txt`

Files uploaded by `UVF` are placed in a folder named `[recieving-device-name]/[sending-file-name]`

Key Functions/Definitions

`readx()/writex()`: read and write packets and exit if error encountered

`read_entire_file()/write_entire_file()`: read and write entire file contents into buffer and from buffer respectively

Client

`Tokens`: struct containing array of tokens

`split_into_tokens()`: parse command line string into individual tokens

`process_edg_command()/process_ued_command()/process_scs_command()/process_dte_command()/process_aed_command()/process_out_command()/process_uvf_command()`:
command handlers

Server

`SharedState`: struct containing state information to be shared amongst server threads managing client connections

`BlockedDevice`: struct containing information relating to a blocked device

`DevInfo`: struct containing information related to a connected device

`parse_credentials()/verify_credentials()`: load and verify credentials

`clear_file()`: create file if does not exist and clear its contents

`append_to_file()`: append data to a file

`write_active_devices_to_log_file()`: overwrite log file with current connected devices

`populate_timestamp()`: generate a string timestamp formatted as per assignment specification

Application Layer Message Format

Defined in `messages.h` is a discriminated union `Message`. The particular content of the message is determined by `MESSAGE_TYPE` enum field. For each message, there is a request and response enum field, e.g. for UED message there is `UED_REQUEST` and `UED_RESPONSE`
Therefore, inside of the union, there are two anonymous structs for each message. One for the request type and the other for the response type, e.g:

```
typedef struct
{
  MESSAGE_TYPE type;
  union
  {
    // DTE_REQUEST
    struct
    {
      u32 dte_file_id;
    };
    // DTE_RESPONSE
    struct
    {
      s32 dte_response_code;
    };

    // ...
  };
} Message;
```

Program Flow

Client

Will first prompt for device name and password. Once authenticated, will start a thread listening for UDP connections. In the main thread, a loop is entered that will prompt for a command. The command is parsed and the appropriate command handler is called.

Server

Will first parse credentials file, clear related log files and allocate shared state. The welcoming TCP socket is set to non-blocking. So, after checking if a new connection is present, will update the timing information of any blocked devices. If a new connection is recieved, will create a new thread to handle this client. The client TCP socket is set to blocking. Inside the client thread, a while loop is entered where a message will be read each time. Inside this loop, a switch statement is checked on the `MESSAGE_TYPE` field of the read message. Each case block contains the appropriate server response to the particular message type.

Design Tradeoffs

- **Security**:
  - **Encryption**: Message format can be easily reverse engineered by sniffing traffic. Fix by encrypting messages
  - **Buffer Overflow**: For file transfer, the size of the data transfer is specified in the packet allowing clients to manipulate this value. Fix by inferring size from message type
- **Bandwidth**: As using discriminated union, each message will be the size of the largest struct. Fix by writing bitpacker/bitreader for message writing/reading
- **Endianness**: Assume all client and server are running on little-endian. Fix by writing endian agnostic bitpacker/bitreader for message writing/reading
- **Scalability**: Device information stored on stack for simplicity, not allowing for very large number of concurrent clients. Fix by storing dynamically on heap
- **Concurrency**: Possible race conditions in relation to server threads reading/writing to shared state. Fix by using locks
- **Reliability**:
  - **Packet Loss**: As running on localhost, assume no UDP loss. Fix by writing reliability system on top of UDP
  - **MTU**: Assuming large MTU of localhost interface. Fix by writing smaller sized messages closer to typically WiFi or ethernet interfaces.