

# 情報工学実験C

## ネットワークプログラミング レポート

里谷 佳紀  
09426568

提出日: 2017 年 1 月 26 日  
締切日: 2017 年 1 月 26 日

### 1 クライアント・サーバモデルの通信の仕組み

この章では、クライアント・サーバモデルの通信の一般的な仕組みについて説明する。クライアントとは、サービスを受けるプログラムやプロセスである。サーバに要求を送り、受け取った結果を元にさらに処理をする。サーバとは、サービスを提供するプログラムやプロセスである。クライアントから要求された処理を行い、その結果をクライアントに送る。クライアントがサーバと通信するには、プログラムで次の手順を行う必要がある。

1. サーバの IP アドレスを求める (`gethostbyname()` 関数でホスト名から検索することもできる)
2. `socket()` 関数で目的の通信形式に合わせてソケットを作成する
3. `connect()` 関数でソケットを元にサーバと通信を開始する
4. `send()` 関数でサーバにデータを送信し `recv()` 関数でサーバからの応答を待つことを繰り返す
5. 通信が終わったら `close()` 関数によりソケットを解放する

サーバがクライアントと通信するには、次の手順を行う必要がある。

1. `socket()` 関数で目的の通信形式に合わせてソケットを作成する
2. `bind()` 関数で作成したソケットにポート番号などを設定する
3. `listen()` 関数で設定したソケットに対するクライアントからの接続要求を待つ
4. `accept()` 関数でクライアントからの接続要求を受理して相手先のソケットを生成する
5. `recv()` 関数でクライアントからのデータを受け付け `send()` 関数で結果を送信することを繰り返す
6. 通信が終わったら `close()` 関数で相手側に接続されたソケットを解放する

クライアント・サーバモデルの通信の手順を図 1 にまとめる。

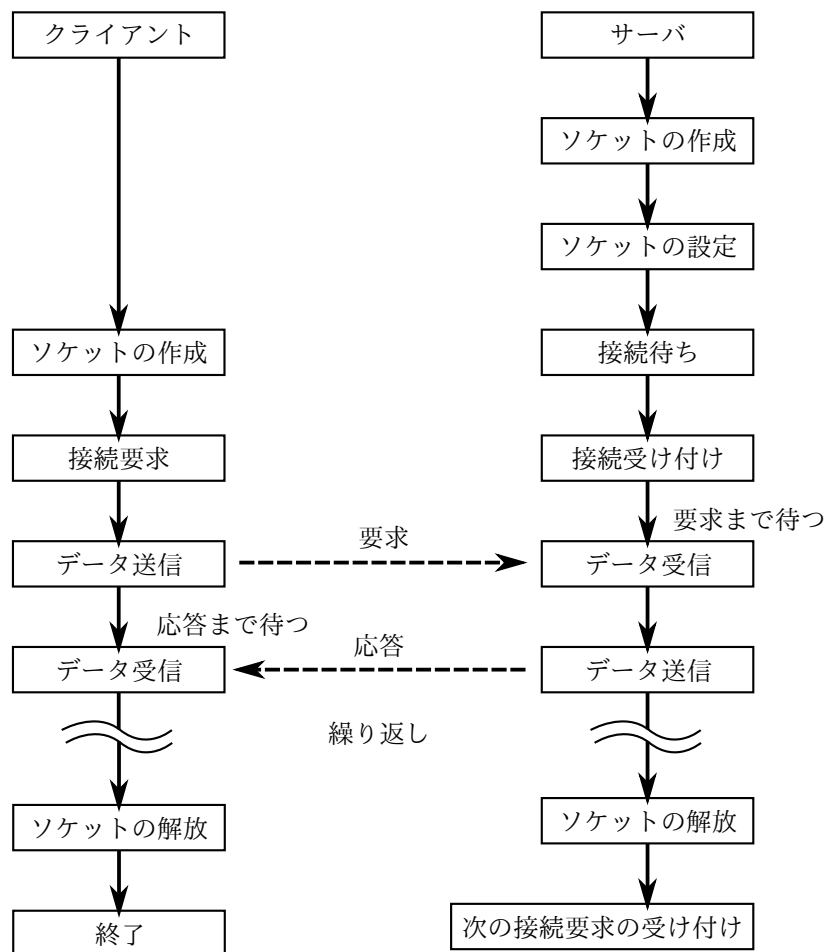


図 1: クライアント・サーバモデルの通信の流れ

表 1: 各コマンドの仕様

コマンド	名称	処理内容	
%C	データチェック	サーバ クライアント	保持しているデータの概要を送信 サーバからの結果を表示
%P <i>n</i>	データ表示	サーバ クライアント	名簿から指定された件数だけ抜きだし送信 サーバからの結果を表示
%R <i>filename</i>	読み込み (サーバ)	サーバ クライアント	指定されたファイルを読み込み サーバ側の読み込むファイルの名前を送信
%RL <i>filename</i>	読み込み (クライアント)	サーバ クライアント	送信されたデータを保持 クライアントでのファイルを読み込み内容を送信
%W <i>filename</i>	書き込み (サーバ)	サーバ クライアント	保持しているデータをファイルに書き込み サーバでのファイル名を送信
%WL <i>filename</i>	書き込み (クライアント)	サーバ クライアント	保持しているデータをクライアントに送信 受信したデータをクライアント側に保存
%F <i>word</i>	検索	サーバ クライアント	<i>word</i> に一致するエントリをすべて送信 受信したデータを表示
%S <i>col</i>	ソート	サーバ クライアント	列番号 <i>col</i> を基準にした並べ替えを実行 列番号 <i>col</i> をサーバに送信
%Q	終了	サーバ クライアント	クライアントとの接続を遮断 サーバに通信終了の旨を送信しプログラムを終了

## 2 プログラムの作成方針

本実験では、名簿管理プログラムをクライアント・サーバモデルに対応させる。この章では開発するプログラムの方針を示し、仕様を与える。プログラムの方針を以下に挙げる。

- サーバは名簿データを持つ。さらに、クライアントからの要求を受け、名簿データを変形させたり、名簿データを元にクライアントに結果を返す。
- クライアントは、ユーザからのコマンド入力を元に、サーバに処理の依頼を送る。さらにサーバからの結果を処理、表示を行う。
- プログラミング演習で実装されたコマンドに加え、新たにクライアントからの名簿データのアップロードとサーバからの名簿データのダウンロードのコマンドを実装する。
- 通信による遅延を少なくするため、可能な限り通信量を少なくする。

以上の方針を考慮して、実装するコマンドの仕様を表 1 に定める。

## 3 プログラムの実装

### 3.1 データ構造とデータの流れ

開発したプログラムの実装を説明する。まず、実装に使用したデータ構造を説明する。名簿のデータはサーバがファイルから読み込むか、クライアントからアップロードされることでサーバに保持される。そのため、サーバプログラムの `server.c` には、一件の名簿のデータ `profile` 構造体の配列の `current_profile_table` グローバル変数と、`profile` 構造体のポインタの配列 `current_profile_view` グローバル変数を持つ。`profile` 構造体の内容を表 2 に示す。サーバは、クライアントからサーバにあるファイルの読み込み要求や、クライアントからのデータのアップロードがあると、データの実体を `current_profile_table` に格納し、`current_profile_table` の一件ごとの参照を並べて `current_profile_view` とする。また、クライアント

表 2: profile 構造体の定義

メンバ名	大きさ (byte)	説明
id	4	識別番号
name	70	名前 (文字配列)
est	6	日付 (年, 月, 日それぞれ 2byte)
addr	70	所在地 (文字配列)
misc	任意	説明 (文字配列へのポインタ)

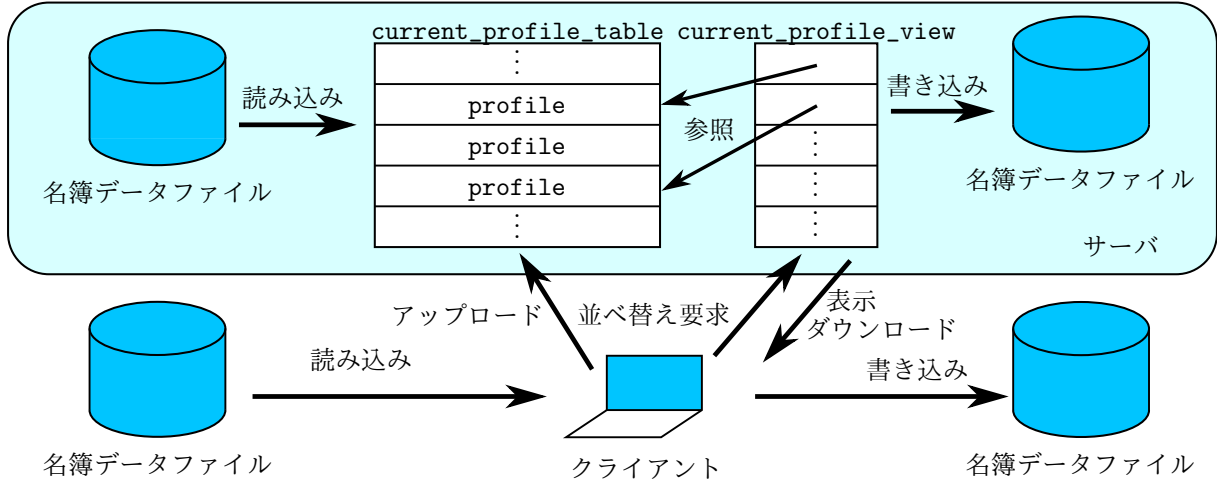


図 2: システムの概要図  
クライアントの、サーバのファイルに対する読み込み要求と書き込み要求を省略している。

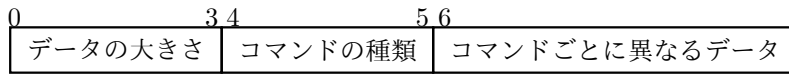
から整列の要求が来ると、名前などの基準に応じて `current_profile_view` の参照を入れ替える。このとき、`current_profile_table` に対する操作はない。さらに、クライアントから探索や表示の要求が来ると、`current_profile_view` から該当するデータを抜きだし、クライアントに返す。`current_profile_table` の内容は、次にサーバがファイルを読むか、クライアントから名簿データを受信すると上書きされる。サーバ上にある `current_profile_table` と `current_profile_view`、サーバ上にある名簿データを格納したファイルとクライアントの関係を図 2 にまとめる。

### 3.2 処理の流れと通信プロトコル

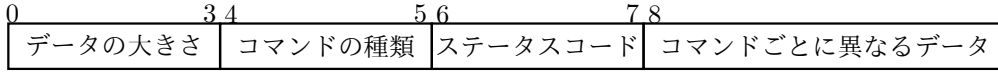
処理の流れと使用する通信プロトコルをクライアント側とサーバ側に分けて説明する。クライアントが送るデータとサーバが送るデータの様子を図 3 に示す。クライアントプログラムが送信するデータは次の要素を含む。

- 送信するデータの大きさ (4byte)
- サーバに実行を要求するコマンドの種類 (2byte)
- コマンドごとに異なるデータ (任意の長さ)

これらを、データの大きさとそれ以外の 2 回に分けて送信する。こうすることで、サーバプログラムが読み取るべきデータの大きさを認識できる。コマンドの種類は `common.h` の `command_index` 型として定義される。サーバプログラムが送信するデータは次の要素を含む。



(a) クライアントが送信するデータ



(b) サーバが送信するデータ

図 3: コマンドに共通するデータ (数字は byte 単位)

表 3: 各コマンドで送信されるデータ

コマンド	クライアントが送信するデータ	サーバが送信するデータ	ステータスコード
%C	なし	保持しているデータの件数 (4byte)	常に PS_SUCCESS
%P	表示するデータの件数 (4byte)	要求された件数を持つ <code>profile_table</code>	$n$ が 0 の場合 PS_INVARG それ以外は PS_SUCCESS
%R	ファイル名 (4byte+1byte× 長さ)	読み取った件数 (4byte)	常に PS_SUCCESS (エラーチェック未実装)
%RL	読み取った件数を持つ <code>profile_table</code>	読み取った件数 (4byte)	常に PS_SUCCESS (エラーはクライアント側で検知)
%W	ファイル名 (4byte+1byte× 長さ)	書き込んだ件数 (4byte)	常に PS_SUCCESS (エラーチェック未実装)
%WL	なし	保持している <code>profile_table</code>	常に PS_SUCCESS
%F	検索語 (4byte+1byte× 長さ)	検索結果の <code>profile_table</code>	常に PS_SUCCESS (エラーはクライアント側で検知)
%S	基準となる列 (2byte)	なし	有効な列番号は PS_SUCCESS それ以外は PS_INVCOL
%Q	なし	なし	常に PS_SUCCESS

- 送信するデータの大きさ (4byte)
- サーバが実行したコマンドの種類 (2byte)
- ステータスコード (2byte)
- コマンドごとに異なるデータ (任意の長さ)

これらを、クライアントと同様に 2 回に分けて送信する。実行したコマンドの種類を送る理由は、サーバが要求通りのコマンドを実行したかをクライアントでチェックできるようにするためである。ステータスコードとは、要求されたコマンドが正しく実行できたか、あるいは何らかのエラーが発生したかを示す。例えば、ソートコマンドで範囲外の列番号が指定された場合、サーバは `profile.h` の PS\_INVCOL をステータスコードとしてクライアントに送る。

コマンドごとに異なるデータの内容を表 3 に示す。表中にある `profile_table` 構造体は、`profile.h` で定義される。`profile_table` 構造体は、`profile` 構造体の配列と格納する `profile` 構造体の数を格納する。`profile_table` 構造体を送信するには、次の要領で整列化する (`profile.c` の `serialize_profile_table`, `serialize_profile_view` 参照)。

1. 格納する `profile` 構造体の数 `n_entries` を整列化する。
2. `entries` メンバの要素を順に `n_entries` 個だけ整列化する。表 2 を参考に整列化する。
  - (a) `id` メンバを整列化する。バイトオーダーを揃える `htonl` 関数を用いる。
  - (b) `name` メンバを整列化する。英数字を扱っているため、今回はバイトオーダーを考慮する必要はない。

- (c) `est` メンバを整列化する。 `date` 構造体の整列化であるが 2byte の整数型のメンバが 3 つ集まったものと解釈してよい。
- (d) `addr` メンバを整列化する。
- (e) `misc` メンバの長さを整列化する (4byte)。
- (f) `misc` メンバの実体を整列化する。

## 4 プログラムの使用方法

プログラムの使用法を例に沿って説明する。クライアントプログラム `client.out` とテスト用のデータ `sample.csv` が `~MyDirectory` に、サーバプログラム `server.out` が `~YourDirectory` にあると仮定する。まず、`server.out` をポート番号を与えて実行する。例えば、

```
./server.out 9000
```

とする。プログラムが起動すると、クライアントの接続を待つ。次にサーバ側のホスト名とポート番号を与えて `client.out` を起動する。例えば、

```
./client.out localhost 9000
```

とする。起動後、サーバと接続に成功するとコマンド入力促される。

以後、実行例としてファイルのアップロードと編集、サーバ側への書き込みを示す。まず、クライアント側で `%RL` コマンドを使って `sample.csv` の内容をサーバへ送信する。具体的には、

```
%RL sample.csv
```

とする。実行後、ファイルの内容がサーバにアップロードされる。次に、確認のため、`%C` コマンドで読み込んだ名簿の件数を確認し、`%P` コマンドで最初の 1 件を表示する。

```
%C
```

を実行すると、

```
2886 profiles in view
```

と表示される。また、

```
%P 1
```

を実行すると、

```
ID: 5100046
```

```
Name: The Bridge
```

```
Est.: 1845/11/2
```

```
Addr.: 14 Seafield Road Longman Inverness
```

```
Misc.: SEN Unit 2.0 Open
```

と表示される。さらに、`%S` コマンドでこのデータを名前順に並べ替える。

```
%S 2
```

を実行すると、並び替えが終了した旨のメッセージが返ってくる。ここで%P コマンドで最初の 1 件を表示すると、

ID: 8212627

Name: Abbey Primary School

Est.: 1918/8/23

Addr.: Claremont Crescent Kilwinning

Misc.: 01294 552251 01294 550525 Primary 295 10.5 Open

と表示される。

最後に、このデータをサーバ側のファイルに書き込む。%W コマンドを

%W sample2.csv

として実行すると、server.out がある~/YourDirectory に sample2.csv が保存される。%Q コマンドで終了して、sample2.csv の内容を確認すると、

8212627,Abbey Primary School,...

5520924,Abbeyhill Primary School,...

5237521,Abbotswell School,...

となっており、名前順で並び替えしたのちに保存できたことが分かる。

## 5 プログラムの作成過程に関する考察

### 5.1 工夫した点

主に工夫した点は、整列化と非整列化である。これにより、文字列だけを送受信する場合に比べて扱えるデータの種類が増え、やりとりするデータ量が小さくなることが期待できる。データ量の改善については、6 で考察する。

また、ネットワークプログラムとは直接関係ないが、名簿のエントリの参照を並べるデータ構造 `profile_view` を採用することで、更なる機能を追加できる。例えば、クライアントがサーバに現在のデータから新しく `profile_view` を作成する要求を出せるようにする機能などが考えられる。

### 5.2 苦労した点

苦労した点もやはり整列化と非整列化である。非整列化のときに、プロトコルで定義したデータの大きさを超えるデータを受信することによる不具合があった。緊急の処置として、プロトコルで定義したデータの大きさだけ受信したら、次の要求までのデータを無効化することを行った。具体的には、`select()` 関数を使って `recv()` にタイムアウトを設けつつ、`recv()` で受信データを読み込み続けることを繰り返した。原因は今でも分かっていないので、今後の課題の一つである。

## 6 得られた結果に関する考察

ここで、開発したプログラムが2の方針を満足するか振り返る。2で示した方針を再び示す。

- サーバは名簿データを持つ。さらに、クライアントからの要求を受け、名簿データを変形させたり、名簿データを元にクライアントに結果を返す。
- クライアントは、ユーザからのコマンド入力を元に、サーバに処理の依頼を送る。さらにサーバからの結果を処理、表示を行う。
- プログラミング演習で実装されたコマンドに加え、新たにクライアントからの名簿データのアップロードとサーバからの名簿データのダウンロードのコマンドを実装する。
- 通信による遅延を少なくするため、可能な限り通信量を少なくする。

4つの方針のうち、上の三つは、4に実例を示したとおり、達成できた。最後の方針について考察する。`sample.csv`を対象に、ファイルをそのまま文字列で送信した場合と、整列化して送信した場合とを比較する。まず、ファイルをそのまま送信する場合のデータ量は、Linuxの`wc`コマンドで見ることができ、343021byteである。整列化して送信する場合のデータ量を計算する。表2より、1件の名簿の固定長の部分の大きさは、 $4 + 70 + 6 + 70 + 4 = 154\text{byte}$ である。残りの領域は、miscメンバが指す先の長さである。`sample.csv`は、Linuxで`awk -F "\"*,\"" '{print $5}' sample.csv | wc`で計算する。全体で143052byteであるが、2885個の改行が含まれ、整列化はヌル文字を含むので、実際の大きさは $143052 - 2885 + 2886 = 143053\text{byte}$ である。以上のことから、整列化して送信する場合のデータ量は、送信する名簿の件数4byteを足して、 $4 + 154 \times 2886 + 143053 = 587501\text{byte}$ である。これは、ファイルをそのまま送る方法に比べて大きく、方針は達成できなかった。



## A プログラムリスト

### A.1 client.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5
6  #include <netdb.h>
7  #include <sys/types.h>
8  #include <sys/socket.h>
9  #include <fcntl.h>
10 #include <unistd.h>
11
12 #include "common.h"
13 #include "mystring.h"
14 #include "profile.h"
15
16 int cmd_invalid(int argc, char* argv[], int sfd, struct buffer* buf) {
17     fprintf(stderr, "command \"%s\" not found\n", argv[0]);
18     return PS_SUCCESS;
19 }
20
21 int cmd_check(int argc, char* argv[], int sfd, struct buffer* buf) {
22     short cmd_index, status;
23     int n_entries = 0;
24     write_buffer(sfd, buf, 1024);
25     reset_buffer(buf);
26
27     read_buffer(sfd, buf, 1024);
28     cmd_index = deserialize_short(buf);
29     status = deserialize_short(buf);
30     assert(cmd_index == CMD_CHECK);
31     n_entries = deserialize_long(buf);
32
33     printf("%d profiles in view\n", n_entries);
34     return status;
35 }
36
37 int cmd_print(int argc, char* argv[], int sfd, struct buffer* buf) {
38     short cmd_index, status;
39     int n_entries;
40     struct profile_table table;
41     struct profile_view view;
42     if(argc < 2) {
43         fprintf(stderr, "profile number not provided\n");
44         return PS_INVARG;
45     }
46
47     n_entries = atoi(argv[1]);
48     new_profile_table(&table, MAX_PROFILES);
49     new_profile_view(&view, MAX_PROFILES);
50     serialize_long(buf, n_entries);
51     write_buffer(sfd, buf, 1024);
52     reset_buffer(buf);
53
54     read_buffer(sfd, buf, 1024);
55     cmd_index = deserialize_short(buf);
56     status = deserialize_short(buf);
57     assert(cmd_index == CMD_PRINT);
58     deserialize_profile_table(&table, buf);
59
60     create_view_from(&view, &table);
61     write_profiles_fancy(stdout, &view);
62
63     delete_profile_view(&view);
64     delete_profile_table(&table);
```

```

65     return status;
66 }
67
68 int cmd_readn(int argc, char* argv[], int sfd, struct buffer* buf) {
69     short cmd_index, status;
70     int n_entries;
71     char* filename;
72     if(argc < 2) {
73         fprintf(stderr, "file name is not provided\n");
74         return PS_INVARG;
75     }
76
77     filename = argv[1];
78     serialize_str(buf, filename);
79     write_buffer(sfd, buf, 1024);
80     reset_buffer(buf);
81
82     read_buffer(sfd, buf, 1024);
83     cmd_index = deserialize_short(buf);
84     status = deserialize_short(buf);
85     assert(cmd_index == CMD_READ);
86     n_entries = deserialize_long(buf);
87
88     printf("read %d profiles on server (status=%d)\n", n_entries, status);
89     return status;
90 }
91
92 int cmd_readl(int argc, char* argv[], int sfd, struct buffer* buf) {
93     short cmd_index, status;
94     FILE *fp;
95     struct profile_table table;
96     long n_entries;
97     if(argc < 2) {
98         fprintf(stderr, "file name not provided\n");
99         return PS_INVARG;
100     }
101
102     fp = fopen(argv[1], "r");
103     if(!fp) {
104         fprintf(stderr, "file not found\n");
105         return PS_INVARG;
106     }
107
108     new_profile_table(&table, MAX_PROFILES);
109     read_profiles_csv(&table, fp);
110     fclose(fp);
111
112     serialize_profile_table(buf, &table);
113     write_buffer(sfd, buf, 1024);
114     reset_buffer(buf);
115
116     read_buffer(sfd, buf, 1024);
117     cmd_index = deserialize_short(buf);
118     status = deserialize_short(buf);
119     assert(cmd_index == CMD_READ_L);
120     n_entries = deserialize_long(buf);
121
122     printf("read %li profiles on client (status=%d)\n", n_entries, status);
123     return status;
124 }
125
126 int cmd_writen(int argc, char* argv[], int sfd, struct buffer* buf) {
127     short cmd_index, status;
128     int n_entries;
129     char* filename;
130     if(argc < 2) {
131         fprintf(stderr, "file name is not provided\n");
132         return PS_INVARG;
133     }

```

```

134
135     filename = argv[1];
136     serialize_str(buf, filename);
137     write_buffer(sfd, buf, 1024);
138     reset_buffer(buf);
139
140     read_buffer(sfd, buf, 1024);
141     cmd_index = deserialize_short(buf);
142     status = deserialize_short(buf);
143     assert(cmd_index == CMD_WRITE);
144     n_entries = deserialize_long(buf);
145
146     printf("write %d profiles on server\n", n_entries);
147     return status;
148 }
149
150 int cmd_writel(int argc, char* argv[], int sfd, struct buffer* buf) {
151     short cmd_index, status;
152     char* filename;
153     FILE* fp;
154     struct profile_table table;
155     struct profile_view view;
156
157     if(argc < 2) {
158         fprintf(stderr, "file name is not provided\n");
159         return PS_INVARG;
160     }
161     filename = argv[1];
162     fp = fopen(filename, "w");
163     if(!fp) {
164         fprintf(stderr, "cannot open file\n");
165         return PS_INVARG;
166     }
167     new_profile_table(&table, MAX_PROFILES);
168     new_profile_view(&view, MAX_PROFILES);
169
170     write_buffer(sfd, buf, 1024);
171     reset_buffer(buf);
172
173     read_buffer(sfd, buf, 1024);
174     cmd_index = deserialize_short(buf);
175     status = deserialize_short(buf);
176     assert(cmd_index == CMD_WRITE_L);
177     deserialize_profile_table(&table, buf);
178     create_view_from(&view, &table);
179
180     write_profiles_csv(fp, &view);
181     fclose(fp);
182     printf("write %d profiles on local\n", table.n_entries);
183
184     delete_profile_view(&view);
185     delete_profile_table(&table);
186     return status;
187 }
188
189 int cmd_sort(int argc, char* argv[], int sfd, struct buffer* buf) {
190     short cmd_index, status;
191     int col;
192
193     if(argc < 2) {
194         fprintf(stderr, "column number is not provided");
195         return PS_INVARG;
196     }
197     col = atoi(argv[1]);
198
199     serialize_short(buf, col);
200     write_buffer(sfd, buf, 1024);
201     reset_buffer(buf);
202

```

```

203     read_buffer(sfd, buf, 1024);
204     cmd_index = deserialize_short(buf);
205     status = deserialize_short(buf);
206     assert(cmd_index == CMD_SORT);
207
208     if(status == PS_SUCCESS)
209         printf("sort profile table by column %d\n", col);
210     else
211         fprintf(stderr, "invalid profile number %d\n", col);
212     return status;
213 }
214
215 int cmd_find(int argc, char* argv[], int sfd, struct buffer* buf) {
216     short cmd_index, status;
217     char word[128] = "\0";
218     int i;
219     struct profile_table table;
220     struct profile_view view;
221     if(argc < 2) {
222         fprintf(stderr, "search word not provided\n");
223         return PS_INVARG;
224     }
225     for(i = 1; i < argc; i++) {
226         strcat(word, argv[i]);
227         if(i < argc-1)
228             strcat(word, " ");
229     }
230     new_profile_table(&table, MAX_PROFILES);
231     new_profile_view(&view, MAX_PROFILES);
232
233     serialize_str(buf, word);
234     write_buffer(sfd, buf, 1024);
235     reset_buffer(buf);
236
237     read_buffer(sfd, buf, 1024);
238     cmd_index = deserialize_short(buf);
239     status = deserialize_short(buf);
240     assert(cmd_index == CMD_FIND);
241     deserialize_profile_table(&table, buf);
242
243     if(status == PS_SUCCESS) {
244         create_view_from(&view, &table);
245         write_profiles_fancy(stdout, &view);
246     }
247
248     delete_profile_view(&view);
249     delete_profile_table(&table);
250     return status;
251 }
252
253 int cmd_quit(int argc, char* argv[], int sfd, struct buffer* buf) {
254     short cmd_index, status;
255     write_buffer(sfd, buf, 1024);
256     reset_buffer(buf);
257
258     read_buffer(sfd, buf, 1024);
259     cmd_index = deserialize_short(buf);
260     status = deserialize_short(buf);
261     assert(cmd_index == CMD_QUIT);
262
263     printf("quit\n");
264     return status;
265 }
266
267 int main(int argc, char* argv[]) {
268     struct hostent* host;
269     struct sockaddr_in hostaddr;
270     int sfd;
271

```

```

272     char* host_name;
273     int port_no;
274
275     if(argc <= 2) {
276         return -1;
277     }
278     host_name = argv[1];
279     port_no = atoi(argv[2]);
280
281     host = gethostbyname(host_name);
282     if(host == NULL) {
283         fprintf(stderr, "Error: host not found\n");
284         return -1;
285     }
286
287     sfd = socket(AF_INET, SOCK_STREAM, 0);
288     if(sfd < 0) {
289         fprintf(stderr, "Error: could not open socket\n");
290         return -1;
291     }
292
293     hostaddr.sin_family = host->h_addrtype;
294     bzero((char*)&hostaddr.sin_addr, sizeof(hostaddr.sin_addr));
295     memcpy((char*)&hostaddr.sin_addr, (char*)host->h_addr,
296           host->h_length);
297     hostaddr.sin_port = htons(port_no);
298     if(connect(sfd, (struct sockaddr*)&hostaddr, sizeof(hostaddr)) < 0) {
299         fprintf(stderr, "Error: connection failed\n");
300         return -1;
301     }
302
303     struct buffer buf;
304     new_buffer(&buf, 1024);
305
306     char *request = NULL;
307     size_t req_len = 0;
308     ssize_t read;
309     command_index index = CMD_INVALID;
310     while(index != CMD_QUIT) {
311         // get request
312         read = getline(&request, &req_len, stdin);
313         if(read == -1) break;
314
315         int i;
316         for(i = 0; i < req_len; i++) {
317             if(request[i] == '\r' || request[i] == '\n')
318                 request[i] = '\0';
319         }
320
321         // send request
322         //exec_command(request);
323         char* argv[16];
324         int argc = split(request, argv, ' ', 16);
325         //int status;
326
327         index = command_index_of(argv[0]);
328         reset_buffer(&buf);
329         serialize_short(&buf, index); // header as command index
330         //status = command_table[index](argc, argv, sfd, &buf);
331         command_table[index](argc, argv, sfd, &buf);
332     }
333
334     if(request) free(request);
335     delete_buffer(&buf);
336     close(sfd);
337     return 0;
338 }

```

## A.2 server.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5
6  #include <netdb.h>
7  #include <sys/types.h>
8  #include <sys/socket.h>
9  #include <fcntl.h>
10 #include <unistd.h>
11
12 #include "common.h"
13 #include "mystring.h"
14 #include "profile.h"
15
16 struct profile_table current_profile_table;
17 struct profile_view current_profile_view;
18
19 int cmd_invalid(int argc, char* argv[], int sfd, struct buffer* buf) {
20     printf("server recieved invalid command\n");
21
22     reset_buffer(buf);
23     serialize_short(buf, CMD_INVALID);
24     serialize_short(buf, PS_SUCCESS);
25
26     return PS_SUCCESS;
27 }
28
29 int cmd_check(int argc, char* argv[], int sfd, struct buffer* buf) {
30     printf("check command: %d profiles in current table\n",
31           current_profile_view.n_entries);
32
33     reset_buffer(buf);
34     serialize_short(buf, CMD_CHECK);
35     serialize_short(buf, PS_SUCCESS);
36     serialize_long(buf, current_profile_view.n_entries);
37
38     return PS_SUCCESS;
39 }
40
41 int cmd_print(int argc, char* argv[], int sfd, struct buffer* buf) {
42     int n_entries, status;
43     struct profile_view shown;
44     new_profile_view(&shown, current_profile_view.n_entries);
45
46     n_entries = deserialize_long(buf);
47     status = take(&shown, &current_profile_view, n_entries);
48     printf("print command: argument=%d status=%d\n", n_entries, status);
49
50     reset_buffer(buf);
51     serialize_short(buf, CMD_PRINT);
52     serialize_short(buf, status);
53     serialize_profile_view(buf, &shown);
54
55     delete_profile_view(&shown);
56     return status;
57 }
58
59 int cmd_readn(int argc, char* argv[], int sfd, struct buffer* buf) {
60     int n_entries, status = PS_SUCCESS;
61     char* filename;
62     FILE *fp;
63
64     // TODO error handling
65     deserialize_str(&filename, buf);
66     fp = fopen(filename, "r");
67     delete_profile_table(&current_profile_table);
```

```

68     new_profile_table(&current_profile_table, MAX_PROFILES);
69     n_entries = read_profiles_csv(&current_profile_table, fp);
70     create_view_from(&current_profile_view, &current_profile_table);
71     printf("read net command: read %d profile(s), status=%d\n", n_entries, status);
72
73     reset_buffer(buf);
74     serialize_short(buf, CMD_READ);
75     serialize_short(buf, status);
76     serialize_long(buf, n_entries);
77
78     free(filename);
79     return status;
80 }
81
82 int cmd_readl(int argc, char* argv[], int sfd, struct buffer* buf) {
83     int status = PS_SUCCESS;
84
85     delete_profile_view(&current_profile_view);
86     delete_profile_table(&current_profile_table);
87     new_profile_table(&current_profile_table, MAX_PROFILES);
88     new_profile_view(&current_profile_view, MAX_PROFILES);
89
90     deserialize_profile_table(&current_profile_table, buf);
91     create_view_from(&current_profile_view, &current_profile_table);
92     printf("read local command: read %d profile(s)\n",
93           current_profile_table.n_entries);
94
95     reset_buffer(buf);
96     serialize_short(buf, CMD_READ_L);
97     serialize_short(buf, status);
98     serialize_long(buf, current_profile_table.n_entries);
99
100    return status;
101 }
102
103 int cmd_writen(int argc, char* argv[], int sfd, struct buffer* buf) {
104     int n_entries, status = PS_SUCCESS;
105     char* filename;
106     FILE *fp;
107
108     // TODO error handling
109     deserialize_str(&filename, buf);
110     fp = fopen(filename, "w");
111     n_entries = write_profiles_csv(fp, &current_profile_view);
112     fclose(fp);
113     //printf("write net command: write %d profile(s), status=%d\n", n_entries, status);
114
115     reset_buffer(buf);
116     serialize_short(buf, CMD_WRITE);
117     serialize_short(buf, status);
118     serialize_long(buf, n_entries);
119
120     free(filename);
121     return status;
122 }
123
124 int cmd_writel(int argc, char* argv[], int sfd, struct buffer* buf) {
125     reset_buffer(buf);
126     serialize_short(buf, CMD_WRITE_L);
127     serialize_short(buf, PS_SUCCESS);
128     serialize_profile_view(buf, &current_profile_view);
129
130     printf("write local command: write %d profile(s)\n",
131           current_profile_view.n_entries);
132
133     return PS_SUCCESS;
134 }
135
136 int cmd_sort(int argc, char* argv[], int sfd, struct buffer* buf) {

```

```

137     uint16_t which, status;
138     struct profile_view sorted;
139     which = deserialize_short(buf);
140     new_profile_view(&sorted, current_profile_view.n_entries);
141
142     status = sorted_by(&sorted, &current_profile_view, which);
143     copy_view(&current_profile_view, &sorted);
144
145     reset_buffer(buf);
146     serialize_short(buf, CMD_SORT);
147     serialize_short(buf, status);
148
149     delete_profile_view(&sorted);
150     return status;
151 }
152
153 int cmd_find(int argc, char* argv[], int sfd, struct buffer* buf) {
154     int status;
155     char* word;
156     struct profile_view found;
157     deserialize_str(&word, buf);
158     new_profile_view(&found, current_profile_view.n_entries);
159
160     status = search_word_in_all(&found, &current_profile_view, word);
161
162     reset_buffer(buf);
163     serialize_short(buf, CMD_FIND);
164     serialize_short(buf, status);
165     serialize_profile_view(buf, &found);
166
167     delete_profile_view(&found);
168     return status;
169 }
170
171 int cmd_quit(int argc, char* argv[], int sfd, struct buffer* buf) {
172     reset_buffer(buf);
173     serialize_short(buf, CMD_QUIT);
174     serialize_short(buf, PS_SUCCESS);
175
176     return PS_SUCCESS;
177 }
178
179 int main(int argc, char* argv[]) {
180     int host_sfd, client_sfd;
181     struct sockaddr_in my_addr, peer_addr;
182     socklen_t peer_addr_size;
183     int port_no;
184
185     if(argc < 2) {
186         fprintf(stderr, "Error: no port number provided\n");
187         return -1;
188     }
189     port_no = atoi(argv[1]);
190
191     host_sfd = socket(AF_INET, SOCK_STREAM, 0);
192     if(host_sfd < 0) {
193         fprintf(stderr, "Error: cannot open socket\n");
194         return -1;
195     }
196
197     memset(&my_addr, 0, sizeof(struct sockaddr_in));
198     my_addr.sin_family = AF_INET;
199     my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
200     my_addr.sin_port = htons(port_no);
201     if(bind(host_sfd, (struct sockaddr*)&my_addr, sizeof(my_addr)) < 0) {
202         fprintf(stderr, "Error: failed to bind socket\n");
203         return -1;
204     }
205

```



```

206
207 if(listen(host_sfd, 5) < 0) {
208     fprintf(stderr, "Error: cannot listen\n");
209     return -1;
210 }
211
212 while(1) {
213     peer_addr_size = sizeof(struct sockaddr_in);
214     client_sfd = accept(host_sfd,
215                         (struct sockaddr*)&peer_addr, &peer_addr_size);
216     if(client_sfd < 0) {
217         fprintf(stderr, "Error: cannot accept\n");
218         continue;
219     }
220
221     struct buffer buf;
222     new_profile_table(&current_profile_table, MAX_PROFILES);
223     new_profile_view(&current_profile_view, MAX_PROFILES);
224     new_buffer(&buf, 1024);
225
226     while(1) {
227         // recieve request
228         read_buffer(client_sfd, &buf, 1024);
229
230         // exec. requested command
231         uint16_t index = CMD_INVALID;
232         //int status;
233         index = deserialize_short(&buf);
234         printf("command no. %d\n", index);
235         if(index < 0 || index >= CMD_END) index = CMD_INVALID;
236         command_table[index](0, NULL, client_sfd, &buf);
237         //status = command_table[index](0, NULL, client_sfd, &buf);
238         //printf("status: %d\n", status);
239
240         // send response
241         write_buffer(client_sfd, &buf, 1024);
242         reset_buffer(&buf);
243
244         if(index == CMD_QUIT) break;
245     }
246
247     delete_profile_table(&current_profile_table);
248     delete_profile_view(&current_profile_view);
249     delete_buffer(&buf);
250     close(client_sfd);
251 }
252
253 close(host_sfd);
254 return 0;
255 }

```

### A.3 common.h

```

1  #pragma once
2
3  #include "profile.h"
4
5  #define MAX_PROFILES 10000
6
7  typedef int(*command)(int, char**, int, struct buffer*);
8
9  typedef enum {
10     CMD_INVALID,
11     CMD_CHECK, CMD_PRINT,
12     CMD_READ, CMD_READ_L,
13     CMD_WRITE, CMD_WRITE_L,
14     CMD_SORT, CMD_FIND,

```

```

15     CMD_QUIT, CMD_END
16 } command_index;
17
18 char* command_name_table[] = {
19     "%",
20     "%C", "%P",
21     "%R", "%RL",
22     "%W", "%WL",
23     "%S", "%F",
24     "%Q"
25 };
26
27 command_index command_index_of(char* name) {
28     int i;
29     command_index index = CMD_INVALID;
30     for(i = 0; i < CMD_END; i++)
31         if(strcmp(name, command_name_table[i]) == 0) {
32             index = i;
33             break;
34         }
35     return index;
36 }
37
38 int cmd_invalid(int argc, char* argv[], int sfd, struct buffer* buf);
39 int cmd_check(int argc, char* argv[], int sfd, struct buffer* buf);
40 int cmd_print(int argc, char* argv[], int sfd, struct buffer* buf);
41 int cmd_readn(int argc, char* argv[], int sfd, struct buffer* buf);
42 int cmd_readl(int argc, char* argv[], int sfd, struct buffer* buf);
43 int cmd_writen(int argc, char* argv[], int sfd, struct buffer* buf);
44 int cmd_writel(int argc, char* argv[], int sfd, struct buffer* buf);
45 int cmd_sort(int argc, char* argv[], int sfd, struct buffer* buf);
46 int cmd_find(int argc, char* argv[], int sfd, struct buffer* buf);
47 int cmd_quit(int argc, char* argv[], int sfd, struct buffer* buf);
48
49 command command_table[] = {
50     cmd_invalid,
51     cmd_check, cmd_print,
52     cmd_readn, cmd_readl,
53     cmd_writen, cmd_writel,
54     cmd_sort, cmd_find,
55     cmd_quit
56 };

```

## A.4 profile.h

```

1  #pragma once
2
3  #include <stdio.h>
4  #include <stdint.h>
5  #include "buffer.h"
6  #include "date.h"
7
8  /* a profile */
9
10 typedef enum {
11     PC_ID = 1, PC_NAME, PC_EST, PC_ADDR, PC_MISC, PC_END
12 } profile_column;
13
14 #define NAME_SIZE 70
15 #define ADDR_SIZE 70
16
17 struct profile {
18     uint32_t id;
19     char name[NAME_SIZE];
20     struct date est;
21     char addr[ADDR_SIZE];
22     char* misc;

```

```

23 };
24
25 struct profile_table {
26     int n_entries;
27     struct profile* entries;
28 };
29
30 /* profile_table */
31 struct profile_view {
32     int n_entries;
33     struct profile** p_entries;
34 };
35
36 typedef enum {
37     PS_SUCCESS, PS_INVCMD, PS_INVARG,
38     PS_INVID, PS_INVDATE, PS_INVCOL
39 } profile_status;
40
41 /* parse and dump a profile from/to string */
42 int parse_profile_csv(struct profile* ret, char* str);
43 int dump_profile_csv(char* ret, struct profile* p);
44 int dump_profile_fancy(char* ret, struct profile* p);
45
46 /* allocate and free(deallocate) profile_table/profile_view */
47 int delete_profile(struct profile* p);
48 int new_profile_table(struct profile_table* t, int n_max_profile_table);
49 int clear_profile_table(struct profile_table* t);
50 int delete_profile_table(struct profile_table* t);
51 int new_profile_view(struct profile_view* v, int n_profile_table);
52 int clear_profile_view(struct profile_view* v);
53 int delete_profile_view(struct profile_view* v);
54
55 /* make profile_view from profile_table */
56 int create_view_from(struct profile_view* v, struct profile_table* t);
57 /* transform profile_view */
58 int copy_view(struct profile_view* t, struct profile_view* s);
59 int take(struct profile_view* to, struct profile_view* from, int n);
60 int search_word_in(struct profile_view* to, struct profile_view* from,
61                  char* word, profile_column which);
62 int search_word_in_all(struct profile_view* to, struct profile_view* from,
63                       char* word);
64 int sorted_by(struct profile_view* w, struct profile_view* v,
65              profile_column which);
66
67 /* read and write profile_table */
68 int read_profiles_csv(struct profile_table* t, FILE* fp);
69 int write_profiles_csv(FILE* fp, struct profile_view* v);
70 int write_profiles_fancy(FILE* fp, struct profile_view* v);
71
72 /* serialize / deserialize profile, profile_table, profile_view */
73 void serialize_profile(struct buffer* buf, struct profile* p);
74 void serialize_profile_table(struct buffer* buf, struct profile_table* t);
75 void serialize_profile_view(struct buffer* buf, struct profile_view* v);
76 void deserialize_profile(struct profile* p, struct buffer* buf);
77 void deserialize_profile_table(struct profile_table* t, struct buffer* buf);

```

## A.5 profile.c

```

1  #include "profile.h"
2
3  #include <stdio.h>
4  #include <stdint.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <arpa/inet.h>
8
9  #include <fcntl.h>

```

```

10  #include <unistd.h>
11
12  #include "generic_search.h"
13  #include "generic_sort.h"
14  #include "mystring.h"
15
16  /*****
17   * handling single profile
18   *****/
19  int parse_profile_csv(struct profile* ret, char* str) {
20      char *splited[5], *e;
21      char buf[strlen(str) + 1];
22      int n;
23      memcpy(buf, str, sizeof(buf));
24
25      n = split(buf, splited, ',', 5);
26      if(n != 5) return 0;
27      ret->id = strtol(splited[0], &e, 10);
28      if(*e != '\0') return 0;
29      strcpy(ret->name, splited[1]);
30      if(!parse_date(&(ret->est), splited[2]))
31          return 0;
32      strcpy(ret->addr, splited[3]);
33      ret->misc = (char*) malloc(strlen(splited[4]) + 1);
34      strcpy(ret->misc, splited[4]);
35      return 1;
36  }
37
38  int dump_profile_csv(char* ret, struct profile* p) {
39      char buf[20];
40      dump_date_csv(buf, &p->est);
41      return sprintf(ret, "%d,%s,%s,%s,%s",
42                    p->id, p->name, buf, p->addr, p->misc);
43  }
44
45  int dump_profile_fancy(char* ret, struct profile* p) {
46      char buf[20];
47      dump_date_fancy(buf, &p->est);
48      return sprintf(ret, "ID:\t%d\nName:\t%s\nEst.:\t%s\nAddr.:\t%s\nMisc.:\t%s",
49                    p->id, p->name, buf, p->addr, p->misc);
50  }
51  }
52
53  /*****
54   * handling profile_table
55   *****/
56  int delete_profile(struct profile* p) {
57      free(p->misc);
58      p->misc = 0;
59      return 1;
60  }
61
62  int new_profile_table(struct profile_table* t, int max_n_profile_table) {
63      t->n_entries = 0;
64      t->entries = (struct profile*)
65          malloc(sizeof(struct profile)*max_n_profile_table);
66      return max_n_profile_table;
67  }
68
69  int clear_profile_table(struct profile_table* t) {
70      t->n_entries = 0;
71      return 1;
72  }
73
74  int delete_profile_table(struct profile_table* t) {
75      int i, deleted_entries = t->n_entries;
76      for(i = 0; i < t->n_entries; i++)
77          delete_profile(&t->entries[i]);
78      free(t->entries);

```

```

79     t->n_entries = 0;
80     t->entries = NULL;
81     return deleted_entries;
82 }
83
84 int new_profile_view(struct profile_view* v, int n_profile_table) {
85     v->n_entries = 0;
86     v->p_entries = (struct profile**)
87         malloc(sizeof(struct profile*)*n_profile_table);
88     return n_profile_table;
89 }
90
91 int clear_profile_view(struct profile_view* v) {
92     v->n_entries = 0;
93     return 1;
94 }
95
96 int delete_profile_view(struct profile_view* v) {
97     int deleted_entries = v->n_entries;
98     free(v->p_entries);
99     v->n_entries = 0;
100    v->p_entries = NULL;
101    return deleted_entries;
102 }
103
104 /*
105  * convert profile_table to profile_view
106  */
107 int create_view_from(struct profile_view* v, struct profile_table* t) {
108     int i;
109     v->n_entries = 0;
110     for(i = 0; i < t->n_entries; i++) {
111         v->p_entries[i] = &t->entries[i];
112         v->n_entries++;
113     }
114     return v->n_entries;
115 }
116
117 /*
118  * profile getter and comparator
119  */
120 int get_profile_id(struct profile* p) {
121     return p->id;
122 }
123
124 char* get_profile_name(struct profile* p) {
125     return p->name;
126 }
127
128 struct date* get_profile_est(struct profile* p) {
129     return &p->est;
130 }
131
132 char* get_profile_addr(struct profile* p) {
133     return p->addr;
134 }
135
136 char* get_profile_misc(struct profile* p) {
137     return p->misc;
138 }
139
140 int comp_int(int a, int b) {
141     return b - a;
142 }
143
144 int comp_str(char* a, char* b) {
145     return strcmp(b, a);
146 }
147

```

```

148 int comp_date(struct date* a, struct date* b) {
149     if(b->y != a->y) return b->y - a->y;
150     if(b->m != a->m) return b->m - a->m;
151     return b->d - a->d;
152 }
153
154 /*
155  * define search_profile and sort_profile
156  */
157 generic_search(profile_str, struct profile*, char*);
158 generic_sort(profile_str, struct profile*, char*);
159 generic_search(profile_int, struct profile*, int);
160 generic_sort(profile_int, struct profile*, int);
161 generic_search(profile_date, struct profile*, struct date*);
162 generic_sort(profile_date, struct profile*, struct date*);
163
164 /*
165  * transform profile_view
166  */
167 int copy_view(struct profile_view* t, struct profile_view* s) {
168     t->n_entries = s->n_entries;
169     memcpy(t->p_entries, s->p_entries, sizeof(struct profile*)*s->n_entries);
170     return t->n_entries;
171 }
172
173 int take(struct profile_view* to, struct profile_view* from, int n) {
174     int i, j = 0, minidx = 0, maxidx = from->n_entries;
175     if(n >= 0 && n <= from->n_entries) maxidx = n;
176     else if(n < 0 && n >= -from->n_entries) minidx = from->n_entries + n;
177     else return PS_INVARG;
178
179     to->n_entries = maxidx - minidx;
180     for(i = minidx; i < maxidx; i++) {
181         to->p_entries[j++] = from->p_entries[i];
182     }
183     return PS_SUCCESS;
184 }
185
186 int search_word_in(struct profile_view* to, struct profile_view* from,
187     char* word, profile_column which) {
188     int found_idx[from->n_entries];
189     int i, n_found = 0;
190     int id;
191     char* e;
192     struct date est;
193
194     switch(which) {
195     case(PC_ID):
196         id = strtoul(word, &e, 10);
197         if(*e == '\0') {
198             n_found = search_profile_int(found_idx, from->p_entries, id,
199                                     from->n_entries,
200                                     get_profile_id, comp_int);
201         } else {
202             // invalid id
203             return PS_INVID;
204         }
205         break;
206     case(PC_NAME):
207         n_found = search_profile_str(found_idx, from->p_entries, word,
208                                     from->n_entries,
209                                     get_profile_name, comp_str);
210         break;
211     case(PC_EST):
212         if(parse_date(&est, word)) {
213             n_found = search_profile_date(found_idx, from->p_entries, &est,
214                                     from->n_entries,
215                                     get_profile_est, comp_date);
216         } else {

```

```

217         // invalid date
218         return PS_INVDATE;
219     }
220     break;
221 case(PC_ADDR):
222     n_found = search_profile_str(found_idx, from->p_entries, word,
223                                 from->n_entries,
224                                 get_profile_addr, comp_str);
225     break;
226 case(PC_MISC):
227     n_found = search_profile_str(found_idx, from->p_entries, word,
228                                 from->n_entries,
229                                 get_profile_misc, comp_str);
230     break;
231 default:
232     // invalid column number
233     return PS_INVCOL;
234     break;
235 }
236
237 to->n_entries = n_found;
238 for(i = 0; i < n_found; i++)
239     to->p_entries[i] = from->p_entries[found_idx[i]];
240 return PS_SUCCESS;
241 }
242
243 int search_word_in_all(struct profile_view* to, struct profile_view* from,
244                      char* word) {
245     int idx[from->n_entries];
246     struct profile_view result_per_col;
247     new_profile_view(&result_per_col, from->n_entries);
248
249     int i, j, col, n_found = 0;
250     for(i = 0; i < from->n_entries; i++)
251         idx[i] = 0;
252     for(col = PC_ID; col < PC_END; col++) {
253         if(search_word_in(&result_per_col, from, word, col) != PS_SUCCESS)
254             continue;
255         // TODO poor performance (algorithm)
256         for(i = 0; i < from->n_entries; i++)
257             for(j = 0; j < result_per_col.n_entries; j++)
258                 if(idx[i] == 0 &&
259                    from->p_entries[i] == result_per_col.p_entries[j]) {
260                     idx[i] = 1;
261                     n_found++;
262                 }
263     }
264     delete_profile_view(&result_per_col);
265
266     i = 0;
267     to->n_entries = n_found;
268     for(j = 0; j < n_found; j++) {
269         while(idx[i] == 0) i++;
270         to->p_entries[j] = from->p_entries[i++];
271     }
272     return PS_SUCCESS;
273 }
274
275 int sorted_by(struct profile_view* w, struct profile_view* v,
276              profile_column which) {
277     copy_view(w, v);
278     switch(which) {
279     case(PC_ID):
280         sort_profile_int(w->p_entries, w->n_entries,
281                         get_profile_id, comp_int);
282         break;
283     case(PC_NAME):
284         sort_profile_str(w->p_entries, w->n_entries,
285                         get_profile_name, comp_str);

```

```

286     break;
287 case(PC_EST):
288     sort_profile_date(w->p_entries, w->n_entries,
289                     get_profile_est, comp_date);
290     break;
291 case(PC_ADDR):
292     sort_profile_str(w->p_entries, w->n_entries,
293                    get_profile_addr, comp_str);
294     break;
295 case(PC_MISC):
296     sort_profile_str(w->p_entries, w->n_entries,
297                    get_profile_misc, comp_str);
298     break;
299 default:
300     // invalid column number
301     return PS_INVCOL;
302 }
303
304 return PS_SUCCESS;
305 }
306
307 /*
308  * read / write function
309  */
310 int read_profiles_csv(struct profile_table* t, FILE* fp) {
311     int i, n_added = 0;
312     char buf[1024];
313
314     while(fgets(buf, 1024, fp)) {
315         // skip newline
316         for(i = 0; i < 1024 && buf[i] != '\0'; i++)
317             if(buf[i] == '\r' || buf[i] == '\n') buf[i] = '\0';
318         if(parse_profile_csv(&t->entries[t->n_entries], buf)) {
319             t->n_entries++;
320             n_added++;
321         }
322     }
323     return n_added;
324 }
325
326 int write_profiles_csv(FILE* fp, struct profile_view* v) {
327     int i;
328     char buf[1024];
329     for(i = 0; i < v->n_entries; i++) {
330         dump_profile_csv(buf, v->p_entries[i]);
331         fprintf(fp, "%s", buf);
332         //if(i < v->n_entries-1)
333         fprintf(fp, "\n");
334     }
335     return i;
336 }
337
338 int write_profiles_fancy(FILE* fp, struct profile_view* v) {
339     int i;
340     char buf[1024];
341     for(i = 0; i < v->n_entries; i++) {
342         dump_profile_fancy(buf, v->p_entries[i]);
343         fprintf(fp, "%s\n\n", buf);
344     }
345     return i;
346 }
347
348 void serialize_profile(struct buffer* buf, struct profile* p) {
349     serialize_long(buf, p->id);
350     serialize_data(buf, p->name, NAME_SIZE);
351     serialize_date(buf, &p->est);
352     serialize_data(buf, p->addr, ADDR_SIZE);
353     serialize_str(buf, p->misc);
354 }

```



```

355
356 void serialize_profile_table(struct buffer* buf, struct profile_table* t) {
357     int i;
358     serialize_long(buf, t->n_entries);
359     for(i = 0; i < t->n_entries; i++)
360         serialize_profile(buf, &t->entries[i]);
361 }
362
363 void serialize_profile_view(struct buffer* buf, struct profile_view* v) {
364     int i;
365     serialize_long(buf, v->n_entries);
366     for(i = 0; i < v->n_entries; i++)
367         serialize_profile(buf, v->p_entries[i]);
368 }
369
370 void deserialize_profile(struct profile* p, struct buffer* buf) {
371     p->id = deserialize_long(buf);
372     deserialize_data(p->name, buf, NAME_SIZE);
373     deserialize_date(&p->est, buf);
374     deserialize_data(p->addr, buf, ADDR_SIZE);
375     deserialize_str(&p->misc, buf);
376 }
377
378 void deserialize_profile_table(struct profile_table* t, struct buffer* buf) {
379     int i;
380     t->n_entries = deserialize_long(buf);
381     for(i = 0; i < t->n_entries; i++)
382         deserialize_profile(&t->entries[i], buf);
383 }

```

## A.6 date.c

```

1  #include "date.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "mystring.h"
6
7  /*****
8   * handling date
9   *****/
10 int parse_date(struct date* d, char* str) {
11     char buf[strlen(str) + 1];
12     char *ret[3], *e;
13     int n;
14     memcpy(buf, str, sizeof(buf));
15
16     n = split(buf, ret, '-', 3);
17     if(n != 3) return 0;
18     d->y = strtol(ret[0], &e, 10);
19     if(*e != '\0') return 0;
20     d->m = strtol(ret[1], &e, 10);
21     if(*e != '\0') return 0;
22     d->d = strtol(ret[2], &e, 10);
23     if(*e != '\0') return 0;
24     return 1;
25 }
26
27 int dump_date_csv(char* ret, struct date* d) {
28     return sprintf(ret, "%d-%d-%d", d->y, d->m, d->d);
29 }
30
31 int dump_date_fancy(char* ret, struct date* d) {
32     return sprintf(ret, "%d/%d/%d", d->y, d->m, d->d);
33 }
34
35 void serialize_date(struct buffer* buf, struct date* d) {

```

```

36     serialize_short(buf, d->y);
37     serialize_short(buf, d->m);
38     serialize_short(buf, d->d);
39 }
40
41 void deserialize_date(struct date* d, struct buffer* buf) {
42     d->y = deserialize_short(buf);
43     d->m = deserialize_short(buf);
44     d->d = deserialize_short(buf);
45 }

```

## A.7 buffer.c

```

1  #include "buffer.h"
2  #include <stdlib.h>
3  #include <string.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6  #include <arpa/inet.h>
7
8  /*****
9   * buffer function
10  *****/
11 void new_buffer(struct buffer* buf, uint32_t init_size) {
12     buf->data = malloc(init_size);
13     buf->size = init_size;
14     buf->next = 0;
15 }
16
17 void new_buffer_from_data(struct buffer* buf, uint32_t buf_size, void* data) {
18     buf->data = data;
19     buf->size = buf_size;
20     buf->next = 0;
21 }
22
23 void reserve_space_for_buffer(struct buffer* buf, uint32_t additional) {
24     while(buf->next + additional > buf->size) {
25         buf->data = realloc(buf->data, buf->size * 2);
26         buf->size *= 2;
27     }
28 }
29
30 void reset_buffer(struct buffer* buf) {
31     buf->next = 0;
32 }
33
34 void delete_buffer(struct buffer* buf) {
35     free(buf->data);
36     buf->data = NULL;
37     buf->size = 0;
38     buf->next = 0;
39 }
40
41 int read_buffer(int sfd, struct buffer* buf, int bufsize) {
42     char rbuf[bufsize];
43     int rsize, rsize_sum = 0;
44     uint32_t rsize_all;
45
46     rsize = read(sfd, &rsize_all, sizeof(uint32_t));
47     rsize_all = ntohl(rsize_all);
48
49     while(rsize_sum < rsize_all) {
50         if(rsize_sum + bufsize > rsize_all)
51             rsize = rsize_all - rsize_sum;
52         else
53             rsize = bufsize;
54         rsize = read(sfd, rbuf, rsize);

```

```

55     serialize_data(buf, rbuf, rsize);
56     rsize_sum += rsize;
57 }
58 buf->next = 0;
59
60 // clear buffer
61 fd_set rfd;
62 struct timeval tv;
63 int retval;
64 FD_ZERO(&rfd);
65 FD_SET(sfd, &rfd);
66 while(1) {
67     tv.tv_sec = 0;
68     tv.tv_usec = 1000;
69     retval = select(sfd+1, &rfd, NULL, NULL, &tv);
70     if(retval > 0 && read(sfd, rbuf, bufsize) <= 0) {
71         if(read(sfd, rbuf, bufsize) <= 0) break;
72     }
73     else break;
74 }
75
76 return rsize_all;
77 }
78
79 int write_buffer(int sfd, struct buffer* buf, int bufsize) {
80     int wsize, wsize_sum = 0;
81     uint32_t wsize_all = buf->next;
82     uint32_t wsize_all_n = htonl(wsize_all);
83     wsize = write(sfd, &wsize_all_n, sizeof(uint32_t));
84     while(wsize_sum < wsize_all) {
85         if(wsize_sum + bufsize > wsize_all)
86             wsize = buf->next - wsize_sum;
87         else
88             wsize = bufsize;
89         wsize = write(sfd, (char*)buf->data + wsize_sum, wsize);
90         wsize_sum += wsize;
91     }
92
93     // wait for buffer
94     fd_set wfd;
95     struct timeval tv;
96     int retval;
97     FD_ZERO(&wfd);
98     FD_SET(sfd, &wfd);
99     while(1) {
100         tv.tv_sec = 0;
101         tv.tv_usec = 1000;
102         retval = select(sfd+1, NULL, &wfd, NULL, &tv);
103         if(retval > 0) break;
104         else continue;
105     }
106
107     return wsize_all;
108 }
109
110 /*****
111  * serialize / deserialize primitives
112  *****/
113 void serialize_short(struct buffer* buf, short x) {
114     //printf("serialize short: %d\n", x);
115     x = htons(x);
116     reserve_space_for_buffer(buf, sizeof(uint16_t));
117     memcpy(buf->data + buf->next, &x, sizeof(uint16_t));
118     buf->next += sizeof(uint16_t);
119 }
120
121 short deserialize_short(struct buffer* buf) {
122     uint16_t x;
123     memcpy(&x, buf->data + buf->next, sizeof(uint16_t));

```

```

124     buf->next += sizeof(uint16_t);
125     x = ntohs(x);
126     //printf("deserialize short: %d\n", *x);
127     return x;
128 }
129
130 void serialize_long(struct buffer* buf, long x) {
131     //printf("serialize long: %d\n", x);
132     x = htonl(x);
133     reserve_space_for_buffer(buf, sizeof(uint32_t));
134     memcpy(buf->data + buf->next, &x, sizeof(uint32_t));
135     buf->next += sizeof(uint32_t);
136 }
137
138 long deserialize_long(struct buffer* buf) {
139     uint32_t x;
140     memcpy(&x, buf->data + buf->next, sizeof(uint32_t));
141     buf->next += sizeof(uint32_t);
142     x = ntohl(x);
143     //printf("deserialize long %d\n", *x);
144     return x;
145 }
146
147 void serialize_data(struct buffer* buf, void* data, uint32_t size) {
148     reserve_space_for_buffer(buf, size);
149     memcpy((char*)buf->data + buf->next, data, size);
150     buf->next += size;
151 }
152
153 void deserialize_data(void* data, struct buffer* buf, uint32_t size) {
154     memcpy(data, (char*)buf->data + buf->next, size);
155     buf->next += size;
156 }

```

## A.8 mystring.c

```

1  #include "mystring.h"
2  #include <stdlib.h>
3  #include <string.h>
4
5  /*****
6   * string function
7   *****/
8  int split(char* str, char* ret[], char delim, int max) {
9      int count = 0;
10     ret[count++] = str;
11     while(*str && count < max) {
12         if(*str == delim) {
13             *str = '\0';
14             ret[count++] = str + 1;
15         }
16         str++;
17     }
18     return count;
19 }
20
21 void serialize_str(struct buffer* buf, char* str) {
22     uint32_t size = strlen(str) + 1;
23     serialize_long(buf, size);
24     serialize_data(buf, str, size);
25 }
26
27 void deserialize_str(char** str, struct buffer* buf) {
28     uint32_t size;
29     size = deserialize_long(buf);
30     *str = (char*) malloc(size);
31     deserialize_data(*str, buf, size);

```

```
32 }  
33  
34 void deserialize_str_nocopy(char** str, struct buffer* buf) {  
35     uint32_t size;  
36     size = deserialize_long(buf);  
37     *str = buf->data + buf->next;  
38     buf->next += size;  
39 }
```