

电子科技大学

计算机专业类课程

实验报告

课程名称：计算机网络编程

学 院：计算科学与工程学院

学院专业：计算机科学与技术

学 号：2023080901012

学生姓名：古文斌

指导教师：向渝老师

日 期：2025年11月29日

电子科技大学

实验报告

学生姓名：古文斌 学号：2023080901012 指导教师：向渝老师

实验地点：电子科技大学清水河 校区主楼 A2-412

一、实验室名称：电子科技大学清水河校区主楼 A2-412

二、实验项目名称：无连接的 UDP TIME 服务器软件设计与实现

三、实验原理

(一) TIME 协议概述

TIME 协议（RFC 868）是一种用于获取网络时间的简单协议，由 Jon Postel 和 Ken Harrenstien 于 1983 年制定。该协议设计简洁，适用于需要时间同步的网络应用场景。TIME 协议使用 32 位无符号整数表示时间，该整数表示自 1900 年 1 月 1 日 00:00:00 UTC 以来经过的秒数。

TIME 协议可以基于 TCP 或 UDP 传输层协议实现。在本实验中，我们采用无连接的 UDP 协议实现 TIME 服务。UDP 协议具有以下特点：

- 无连接性：无需建立和维护连接状态，减少了系统开销
- 低延迟：没有 TCP 的三次握手过程，响应更快
- 简单性：协议实现简单，适合轻量级时间查询服务

标准的 TIME 服务使用端口 37（TCP 和 UDP 均可）。客户端向服务器发送任意数据（通常是一个空数据报或单字节）作为请求，服务器收到请求后返回一个 4 字节的大端序无符号整数，表示当前时间。

(二) 时间纪元转换

TIME 协议使用 1900 年纪元，而 Unix/Linux 系统使用 1970 年纪元（Unix 时间戳）。两者之间的转换关系如下：

$$\text{TIME 协议时间} = \text{Unix 时间戳} + 2208988800 \text{ 秒}$$

其中，2208988800 秒是从 1900 年 1 月 1 日到 1970 年 1 月 1 日之间经过的秒数（70 年）。在实现 TIME 协议时，需要进行这种纪元转换：

- 服务器端：将 Unix 时间戳加上 2208988800 得到 TIME 协议时间
- 客户端：将接收到的 TIME 协议时间减去 2208988800 得到 Unix 时间戳

(三) UDP 套接字编程模型

UDP 套接字编程遵循无连接的通信模型。与 TCP 不同，UDP 服务器不需要调用 `listen()` 和 `accept()` 函数，客户端也不需要调用 `connect()` 函数。UDP 通信使用 `sendto()` 和 `recvfrom()` 函数直接发送和接收数据报。

服务器端流程：

- 1) 创建 UDP 套接字：`socket(AF_INET, SOCK_DGRAM, 0)`
- 2) 绑定本地地址和端口：`bind()`
- 3) 循环接收客户端请求：`recvfrom()`
- 4) 处理请求并发送响应：`sendto()`

客户端流程：

- 1) 创建 UDP 套接字: `socket(AF_INET, SOCK_DGRAM, 0)`
- 2) 向服务器发送请求: `sendto()`
- 3) 接收服务器响应: `recvfrom()`
- 4) 处理接收到的时间数据

(四) 网络字节序

网络协议规定使用大端序 (Big-Endian) 传输多字节数据，即高位字节在前，低位字节在后。不同的计算机体系结构可能使用不同的字节序，因此在网络编程中需要进行字节序转换：

- `htonl()`: 将 32 位整数从主机字节序转换为网络字节序
- `ntohl()`: 将 32 位整数从网络字节序转换为主机字节序
- `htons()`: 将 16 位整数从主机字节序转换为网络字节序
- `ntohs()`: 将 16 位整数从网络字节序转换为主机字节序

四、 实验目的

- 1) 掌握服务端软件的工作原理
- 2) 掌握无连接的 TIME 服务器软件的编程实现方法
- 3) 掌握客户端软件的工作原理
- 4) 掌握针对 TIME 服务的 UDP 客户端软件的编程实现方法

五、 实验内容

本实验包括以下主要任务：

(一) 无连接的 TIME 服务器软件的实现

- 1) 使用指定编程工具编写无连接的 TIME 服务器软件的源代码
- 2) 使用指定编程工具对代码进行编译和调试，生成执行程序
- 3) 在 PC 机上执行程序，检验程序能提供的 TIME 服务的功能

(二) 针对 TIME 服务的 UDP 客户软件的实现

- 1) 使用指定编程工具编写针对 TIME 服务的 UDP 客户端软件的源代码
- 2) 使用指定编程工具对代码进行编译和调试，生成执行程序
- 3) 在 PC 机上执行程序，检验程序访问 TIME 服务的功能

六、 实验器材(设备、元器件)

硬件环境

- 计算机：个人 PC
- 网络环境：本地回环接口（127.0.0.1）

软件环境

- 操作系统：Ubuntu 22.04 LTS / Linux
- 编译器：GCC 11.4
- 开发工具：VS Code
- 网络库：POSIX Sockets API
- 版本控制：Git
- 构建工具：Make

七、实验步骤

(一) 问题描述

7.1.1. 服务器端程序

设计并实现 UDP TIME 服务器程序，该程序能够：

- 监听指定的 UDP 端口，等待客户端请求
- 接收任意客户端发送的数据报作为时间请求
- 获取当前系统时间，转换为 TIME 协议格式
- 将时间值以网络字节序发送给客户端
- 支持通过命令行参数指定监听端口
- 显示每次请求的客户端 IP 地址和端口号

7.1.2. 客户端程序

设计并实现 UDP TIME 客户端程序，该程序能够：

- 向指定的 TIME 服务器发送时间请求
- 接收服务器返回的 4 字节时间值
- 将 TIME 协议时间转换为本地时间格式显示
- 支持通过命令行参数指定服务器地址和端口
- 实现超时处理机制，避免长时间阻塞
- 显示时间差信息，便于时间同步分析

(二) 算法分析与概要设计

7.2.1. 服务器端算法设计

输入：命令行参数（可选端口号）

输出：向客户端返回当前时间，控制台打印请求日志

算法流程：

- 1) 解析命令行参数，获取端口号（默认 37）
- 2) 调用 `socket()` 创建 UDP 套接字
- 3) 设置套接字选项 `SO_REUSEADDR` 允许地址重用
- 4) 初始化 `sockaddr_in` 结构体，设置监听地址和端口
- 5) 调用 `bind()` 将套接字绑定到指定端口
- 6) 进入主循环：
 - a) 调用 `recvfrom()` 阻塞等待客户端请求
 - b) 调用 `time()` 获取当前 Unix 时间戳
 - c) 加上 2208988800 转换为 TIME 协议时间
 - d) 调用 `htonl()` 转换为网络字节序
 - e) 调用 `sendto()` 发送 4 字节时间值给客户端
 - f) 打印客户端信息和时间日志

7.2.2. 客户端算法设计

输入：命令行参数（服务器 IP 地址，可选端口号）

输出：控制台显示从服务器获取的时间

算法流程：

- 1) 解析命令行参数，获取服务器地址和端口
- 2) 调用 `socket()` 创建 UDP 套接字

- 3) 设置套接字接收超时选项 `SO_RCVTIMEO`
- 4) 初始化服务器地址结构体
- 5) 调用 `sendto()` 向服务器发送请求数据报
- 6) 调用 `recvfrom()` 接收服务器响应
- 7) 检查接收数据长度是否为 4 字节
- 8) 调用 `ntohl()` 将时间值转换为主机字节序
- 9) 减去 2208988800 转换为 Unix 时间戳
- 10) 调用 `localtime()` 和 `strftime()` 格式化时间
- 11) 显示时间信息和与本地时间的差值

(三) 核心算法的详细设计与实现

7.3.1. 公共头文件设计

为提高代码复用性和可维护性，将服务器和客户端共用的常量、宏定义和工具函数放在公共头文件中：

代码 1: 公共头文件 common.h

```

1  /* TIME 协议默认端口 */
2  #define TIME_PORT 37
3
4  /* 从1900年到1970年的秒数差 */
5  #define TIME_OFFSET 2208988800UL
6
7  /* 将 Unix 时间戳转换为 TIME 协议时间 */
8  static inline uint32_t unix_to_time_protocol(time_t unix_time) {
9      return (uint32_t)(unix_time + TIME_OFFSET);
10 }
11
12 /* 将 TIME 协议时间转换为 Unix 时间戳 */
13 static inline time_t time_protocol_to_unix(uint32_t time_protocol) {
14     return (time_t)(time_protocol - TIME_OFFSET);
15 }
```

7.3.2. 服务器端核心实现

创建 UDP 套接字并绑定端口：

代码 2: 服务器端套接字创建与绑定

```

1  /* 创建 UDP 套接字 */
2  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
3  if (sockfd < 0) {
4      error_exit("Failed to create socket");
5  }
6
7  /* 设置套接字选项，允许地址重用 */
8  int opt = 1;
9  setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
10
11 /* 初始化服务器地址结构 */
12 memset(&server_addr, 0, sizeof(server_addr));
13 server_addr.sin_family = AF_INET;
14 server_addr.sin_addr.s_addr = INADDR_ANY; /* 监听所有接口 */
```

```

15 server_addr.sin_port = htons(port);
16 /* 绑定套接字 */
17 bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr));

```

处理客户端请求并返回时间：

代码 3: 服务器端请求处理逻辑

```

1 while (1) {
2     client_len = sizeof(client_addr);
3
4     /* 接收客户端请求 */
5     recv_len = recvfrom(sockfd, buffer, BUFFER_SIZE, 0,
6                         (struct sockaddr *)&client_addr, &client_len);
7
8     /* 获取当前时间并转换为TIME协议格式 */
9     current_time = time(NULL);
10    time_value = unix_to_time_protocol(current_time);
11    network_time = htonl(time_value); /* 转换为网络字节序 */
12
13    /* 打印请求信息 */
14    printf("[Request] From %s:%d\n",
15           inet_ntoa(client_addr.sin_addr),
16           ntohs(client_addr.sin_port));
17
18    /* 发送时间值给客户端 */
19    sendto(sockfd, &network_time, sizeof(network_time), 0,
20           (struct sockaddr *)&client_addr, client_len);
21 }

```

7.3.3. 客户端核心实现

发送请求并接收响应：

代码 4: 客户端请求与响应处理

```

1 /* 设置接收超时 */
2 timeout.tv_sec = TIMEOUT_SEC;
3 timeout.tv_usec = 0;
4 setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));
5
6 /* 发送时间请求 */
7 sendto(sockfd, request, strlen(request), 0,
8        (struct sockaddr *)&server_addr, sizeof(server_addr));
9
10 /* 接收服务器响应 */
11 recv_len = recvfrom(sockfd, &network_time, sizeof(network_time), 0,
12                      (struct sockaddr *)&server_addr, &server_len);
13
14 /* 转换字节序并计算Unix时间 */
15 time_value = ntohl(network_time);
16 unix_time = time_protocol_to_unix(time_value);
17
18 /* 格式化并显示时间 */

```

```

19 strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S",
20         localtime(&unix_time));
21 printf("Local time: %s\n", time_str);

```

注：以上仅展示核心代码片段，完整源代码请参见[附录A](#)。

八、实验数据及结果分析

(一) 无连接的 TIME 服务器功能验证

本节验证 TIME 服务器软件的各项功能，检验程序能提供的 TIME 服务功能是否正常。

8.1.1. 服务器端口绑定与监听

服务器需要正确创建 UDP 套接字并绑定到指定端口，进入监听状态等待客户端请求。图1展示了服务器成功启动后的运行状态。

```

=====
< airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr4$ ./time_server 8037
=====
    无连接TIME服务器 (UDP)
=====
TIME Server started on port 8037
Waiting for client requests...
Press Ctrl+C to stop the server
=====

[Request] From 127.0.0.1:36464
TIME value: 3973237484
Local time: 2025-11-27 21:04:44
=====
```

图 1: TIME 服务器端口监听状态

结果分析：从图1可以看出，服务器成功完成以下功能：

- UDP 套接字创建成功，绑定到端口 8037
- 服务器正确显示协议类型（UDP）和监听端口号
- 当收到客户端请求时，能够正确解析并显示客户端的 IP 地址和端口号
- 服务器返回的 TIME 协议时间值被正确记录在日志中

这验证了服务器端 `socket()`、`bind()`、`recvfrom()` 和 `sendto()` 等核心 API 的正确使用，程序能够正常提供 TIME 服务。

8.1.2. 服务器时间获取与格式转换

服务器需要获取当前系统时间，将 Unix 时间戳转换为 TIME 协议格式（1900 纪元），并以网络字节序发送给客户端。

从图1中服务器日志可以验证：

- 服务器正确调用 `time()` 获取当前 Unix 时间戳
- 成功将 Unix 时间加上 2208988800 秒转换为 TIME 协议时间
- 使用 `htonl()` 将时间值转换为网络字节序（大端序）
- 以 4 字节格式发送给客户端，符合 RFC 868 规范

8.1.3. 简化版服务器功能验证

为进一步验证程序能提供的 TIME 服务功能，本实验还提供了简化版服务器实现，直接使用 POSIX API 实现，不依赖额外的头文件。图2展示了简化版服务器的运行状态。

结果分析：从图2可以看出：

- 简化版服务器同样能够正确监听 UDP 端口并接收客户端请求

```

gcc -std=c11 -Wall -Wextra -O2 -f time_client time_client.c
airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr3$ ./time_server 10037
TIME 服务器已启动，监听端口 10037 (UDP)
按 Ctrl+C 停止服务器

收到来自 127.0.0.1:34150 的请求，返回时间

```

图 2: 简化版 TIME 服务器功能验证

- 时间获取和转换功能正常，能够提供 TIME 服务功能
- 代码更加精简，验证了 TIME 协议核心逻辑的正确性

综上所述，通过完整版和简化版两种服务器实现的测试，验证了程序能提供的 TIME 服务功能正常，服务器能够正确绑定端口、监听请求、获取系统时间、转换为 TIME 协议格式并返回给客户端。

(二) 针对 TIME 服务的 UDP 客户端功能验证

本节验证 TIME 客户端软件的各项功能，检验程序访问 TIME 服务的功能是否正常。

8.2.1. 客户端请求发送与响应接收

客户端需要向服务器发送时间请求，并正确接收服务器返回的 4 字节时间数据。图3展示了客户端的完整运行结果。

```

airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr4$ ./time_client 127.0.0.1 8037
=====
无连接TIME客户端 (UDP)
=====
Connecting to TIME server 127.0.0.1:8037
=====

Sending time request...

=====
Response Received
=====
TIME protocol value : 3973237484
Unix timestamp      : 1764248684
Local time          : 2025-11-27 21:04:44
=====

Time difference from local: 0 seconds

```

图 3: TIME 客户端请求响应结果

结果分析：从图3可以看出，客户端成功完成以下功能：

- 成功创建 UDP 套接字并向服务器发送请求数据报
- 正确接收服务器返回的 4 字节时间数据
- **TIME protocol value** 字段显示原始的 32 位 TIME 协议时间值，验证了数据接收的完整性
- 客户端能够正常访问 TIME 服务并获取时间信息

8.2.2. 客户端时间解析与显示

客户端需要将接收到的 TIME 协议时间转换为 Unix 时间戳，并以人类可读的格式显示。

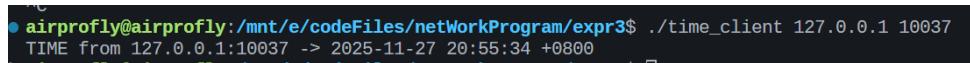
从图3的输出可以验证：

- 使用 `ntohl()` 将网络字节序转换为主机字节序
- 将 TIME 协议时间减去 2208988800 秒还原为 Unix 时间戳
- **Unix timestamp** 字段显示转换后的时间戳，验证了时间纪元转换的正确性
- **Local time** 字段显示格式化的本地时间字符串

- **Time difference** 显示与本地时间的差值接近 0 秒，证明时间获取准确

8.2.3. 简化版客户端功能验证

为进一步验证客户端访问 TIME 服务的功能，本实验还提供了简化版客户端实现，使用 `getaddrinfo()` 进行地址解析。图4展示了简化版客户端的运行结果。



```
airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr3$ ./time_client 127.0.0.1 10037
TIME from 127.0.0.1:10037 -> 2025-11-27 20:55:34 +0800
```

图 4: 简化版 TIME 客户端功能验证

结果分析：从图4可以看出：

- 客户端成功访问 TIME 服务并获取服务器时间
- 输出格式简洁，直接显示格式化后的时间字符串
- 使用 `getaddrinfo()` 实现地址解析，支持域名和 IP 地址两种输入方式
- 时间显示结果正确，验证了客户端访问 TIME 服务功能正常

综上所述，通过完整版和简化版两种客户端实现的测试，验证了程序访问 TIME 服务的功能正常，客户端能够成功发送请求、接收响应、解析时间数据并以可读格式显示。

九、实验结论

通过本次实验，达成了全部实验目标：

- 1) **掌握服务端软件的工作原理：**通过设计并实现 UDP TIME 服务器程序，深入理解了服务端软件的工作模式——创建套接字、绑定端口、循环等待客户端请求、处理请求并返回响应。服务器能够监听指定端口，接收任意客户端的时间请求，并返回符合 TIME 协议格式的时间数据。
- 2) **掌握无连接的 TIME 服务器软件的编程实现方法：**成功使用 C 语言和 POSIX Sockets API 编写了无连接的 TIME 服务器软件，实现了 UDP 套接字创建、端口绑定、数据报收发等核心功能。程序经过编译调试后运行正常，能够持续提供 TIME 服务功能，验证了程序能提供的 TIME 服务的功能。
- 3) **掌握客户端软件的工作原理：**通过设计并实现 UDP TIME 客户端程序，理解了客户端软件如何主动发起请求、等待服务器响应、解析返回数据。客户端程序能够正确处理超时等异常情况，保证了程序的健壮性。
- 4) **掌握针对 TIME 服务的 UDP 客户端软件的编程实现方法：**成功使用 C 语言编写了针对 TIME 服务的 UDP 客户端软件，实现了向服务器发送时间请求、接收 4 字节时间数据、将 TIME 协议时间转换为本地可读时间格式等功能。程序经过编译调试后运行正常，验证了程序访问 TIME 服务的功能。

实验结果表明，服务器能够正确提供 TIME 服务功能，客户端能够成功访问 TIME 服务并获取准确的时间信息。本实验加深了对服务端软件和客户端软件工作原理的理解，为后续网络编程学习奠定了基础。

十、总结及心得体会

通过本次实验，我对服务端软件和客户端软件的工作原理有了深入的理解，同时在网络编程实践、协议理解和工程能力等方面都有了显著提升。以下是我的主要收获和心得体会：

- 1) **服务端软件设计理解：**通过编写 TIME 服务器，理解了服务端软件的基本工作模式——创建套接字、绑定端口、循环等待请求、处理请求并返回响应。这种“请求-响应”模型是网络服务的核心范式。

- 2) **客户端软件设计理解:** 通过编写 TIME 客户端，理解了客户端软件如何主动发起请求、等待服务器响应、解析返回数据。客户端需要处理超时、错误等异常情况，保证程序的健壮性。
- 3) **UDP 编程实践:** 与 TCP 编程相比，UDP 编程更加简洁——无需建立连接，直接使用 `sendto()` 和 `recvfrom()` 收发数据。这种无连接模式适合 TIME 这类简单的请求-响应场景。
- 4) **网络协议理解:** 通过实现 TIME 协议，理解了协议设计的基本要素：数据格式（4 字节无符号整数）、字节序（网络字节序）、时间基准（1900 年纪元）。简单的 TIME 协议包含了协议设计的核心思想。
- 5) **调试能力提升:** 在实验过程中遇到了字节序转换错误、端口绑定失败等问题，通过查阅资料和调试分析解决了这些问题，提升了网络程序调试能力。
- 6) **工程实践能力:** 通过使用 `Makefile` 管理编译、编写公共头文件复用代码，学习了如何组织和管理多文件 C 语言项目，培养了良好的工程实践习惯。

总之，本次实验不仅让我掌握了 UDP 网络编程的核心技术，更重要的是培养了独立分析问题、解决问题的能力。这些经验和技能将为今后学习更复杂的网络协议和开发实际的网络应用奠定坚实的基础。

十一、对本实验过程及方法、手段的改进建议及展望

虽然本次实验成功实现了基本的 TIME 服务器和客户端功能，但在功能完善性、代码健壮性和工程规范性等方面仍有提升空间。针对当前实现的不足之处，提出以下改进建议和未来展望：

- 1) **支持 IPv6:** 当前实现仅支持 IPv4，可以扩展支持 IPv6 协议，使程序适应更广泛的网络环境。使用 `AF_INET6` 和 `sockaddr_in6` 结构体即可实现。
- 2) **多线程服务器:** 虽然 UDP 服务器天然支持并发（每个请求独立处理），但可以考虑使用多线程来处理耗时操作，提高服务器的响应能力。
- 3) **日志系统:** 可以添加更完善的日志记录功能，将请求日志写入文件，便于后续分析和问题排查。可以使用 `syslog` 或自定义日志模块。
- 4) **配置文件支持:** 将端口号、超时时间等参数放入配置文件中，避免硬编码，使程序更加灵活。
- 5) **NTP 协议实现:** TIME 协议较为简单，精度有限。可以进一步学习和实现 NTP（Network Time Protocol）协议，提供更高精度的时间同步服务。
- 6) **图形界面客户端:** 可以使用 GTK 或 Qt 开发图形界面客户端，提供更友好的用户交互体验。
- 7) **跨平台支持:** 当前代码主要针对 Linux 系统，可以通过条件编译支持 Windows 平台，使用 Winsock API。
- 8) **单元测试:** 添加单元测试用例，对时间转换函数等核心逻辑进行自动化测试，确保代码质量。

以上改进方向涵盖了协议兼容性、系统架构、运维管理和开发流程等多个层面。在后续学习中，我将逐步尝试实现这些改进，不断提升自己的网络编程能力和软件工程素养。

报告评分：
指导教师签字：

附录一 详细代码

GitHub 代码仓库地址: <https://github.com/airprofly/networkProgram.git>

(一) 代码文件说明

表 1: 代码文件说明表

文件名	所属部分	说明
expr3/time_server.c	简化版	简化版 TIME 服务器
expr3/time_client.c	简化版	简化版 TIME 客户端
expr4/common.h	完整版	公共头文件
expr4/time_server.c	完整版	完整版 TIME 服务器
expr4/time_client.c	完整版	完整版 TIME 客户端

(二) 简化版 TIME 服务器代码

代码 5: expr3/time_server.c - 简化版 TIME 服务器

```
1  /*
2   * time_server.c - UDP TIME 服务器 (RFC 868)
3   *
4   * 监听 UDP 端口 37, 接收任意请求后返回当前时间。
5   * 时间格式: 自 1900-01-01 00:00:00 以来的秒数, 32 位大端序无符号整数。
6   */
7
8 #define __GNU_SOURCE
9 #include <arpa/inet.h>
10 #include <stdint.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include <sys/socket.h>
15 #include <time.h>
16 #include <unistd.h>
17
18 #define TIMEPORT 37
19 #define TIME_DIFF_1900_TO_1970 2208988800U
20
21 int main(int argc, char *argv[]) {
22     int port = TIMEPORT;
23     if (argc >= 2) {
24         port = atoi(argv[1]);
25         if (port <= 0 || port > 65535) {
26             fprintf(stderr, "无效端口号: %s\n", argv[1]);
27             return 1;
28         }
29     }
30
31     int sock = socket(AF_INET, SOCK_DGRAM, 0);
32     if (sock < 0) {
33         perror("socket");
34         return 1;
```

```

35 }
36
37 /* 允许端口复用 */
38 int opt = 1;
39 setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
40
41 struct sockaddr_in addr;
42 memset(&addr, 0, sizeof(addr));
43 addr.sin_family = AF_INET;
44 addr.sin_addr.s_addr = INADDR_ANY;
45 addr.sin_port = htons((uint16_t)port);
46
47 if (bind(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
48     perror("bind");
49     close(sock);
50     return 1;
51 }
52
53 printf("TIME 服务器已启动, 监听端口 %d (UDP)\n", port);
54 printf("按 Ctrl+C 停止服务器\n\n");
55
56 while (1) {
57     unsigned char buf[64];
58     struct sockaddr_in client;
59     socklen_t clen = sizeof(client);
60
61     ssize_t n =
62         recvfrom(sock, buf, sizeof(buf), 0, (struct sockaddr *)&client,
63                  &clen);
64     if (n < 0) {
65         perror("recvfrom");
66         continue;
67     }
68
69     /* 获取当前时间并转换为 1900 纪元 */
70     time_t now = time(NULL);
71     uint32_t time1900 = (uint32_t)(now + TIME_DIFF_1900_TO_1970);
72     uint32_t net_time = htonl(time1900);
73
74     char client_ip[INET_ADDRSTRLEN];
75     inet_ntop(AF_INET, &client.sin_addr, client_ip, sizeof(client_ip));
76     printf("收到来自 %s:%d 的请求, 返回时间\n", client_ip,
77           ntohs(client.sin_port));
78
79     /* 发送 4 字节时间 */
80     sendto(sock, &net_time, sizeof(net_time), 0, (struct sockaddr *)&
81            client,
82            clen);
83 }
84
85 close(sock);
86 return 0;
}

```

(三) 简化版 TIME 客户端代码

代码 6: expr3/time_client.c - 简化版 TIME 客户端

```
1 #define _GNU_SOURCE
2 #include <arpa/inet.h>
3 #include <netdb.h>
4 #include <stdint.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <sys/socket.h>
9 #include <time.h>
10 #include <unistd.h>
11
12 #define TIMEPORT 37
13 #define TIME_DIFF_1900_TO_1970 2208988800U
14
15 int main(int argc, char *argv[]) {
16     const char *host = (argc >= 2) ? argv[1] : "127.0.0.1";
17     int port = (argc >= 3) ? atoi(argv[2]) : TIMEPORT;
18     struct addrinfo hints, *res = NULL;
19     int sock = -1;
20     uint32_t net_time = 0;
21
22     if (port <= 0 || port > 65535) {
23         fprintf(stderr, "无效端口号: %s\n", argv[2]);
24         return 1;
25     }
26
27     memset(&hints, 0, sizeof(hints));
28     hints.ai_family = AF_INET; /* IPv4 only (TIME is simple) */
29     hints.ai_socktype = SOCK_DGRAM;
30
31     char port_str[6];
32     snprintf(port_str, sizeof(port_str), "%d", port);
33
34     if (getaddrinfo(host, port_str, &hints, &res) != 0) {
35         perror("getaddrinfo");
36         return 1;
37     }
38
39     sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
40     if (sock < 0) {
41         perror("socket");
42         freeaddrinfo(res);
43         return 1;
44     }
45
46     /* Send a single zero byte to request the time (many servers accept
47      * empty or
48      * any data) */
49     unsigned char buf[1] = {0};
50     ssize_t sent =
51
52     sendto(sock, buf, 1, 0, res->ai_addr, res->ai_addrlen);
53
54     if (sent != 1) {
55         perror("sendto");
56         close(sock);
57         freeaddrinfo(res);
58         return 1;
59     }
60
61     /* Read the response */
62     if (recvfrom(sock, buf, 1, 0, res->ai_addr, &res->ai_addrlen) != 1) {
63         perror("recvfrom");
64         close(sock);
65         freeaddrinfo(res);
66         return 1;
67     }
68
69     /* Convert the received time to a uint32_t value */
70     net_time = ((uint32_t)buf[0] << 24) | ((uint32_t)buf[1] << 16) |
71             ((uint32_t)buf[2] << 8) | ((uint32_t)buf[3]);
72
73     /* Print the received time */
74     printf("Received time: %u\n", net_time);
75
76     /* Close the socket */
77     close(sock);
78
79     /* Free the address information */
80     freeaddrinfo(res);
81
82     /* Return success */
83     return 0;
84 }
```

```

50     sendto(sock, buf, sizeof(buf), 0, res->ai_addr, res->ai_addrlen);
51 if (sent < 0) {
52     perror("sendto");
53     close(sock);
54     freeaddrinfo(res);
55     return 1;
56 }
57
58 /* Receive 4 bytes: 32-bit unsigned big-endian seconds since
59  * 1900-01-01 */
60 struct sockaddr_storage from;
61 socklen_t fromlen = sizeof(from);
62 ssize_t rec = recvfrom(sock, &net_time, sizeof(net_time), 0,
63                         (struct sockaddr *)&from, &fromlen);
64 if (rec < 0) {
65     perror("recvfrom");
66     close(sock);
67     freeaddrinfo(res);
68     return 1;
69 }
70 if (rec < 4) {
71     fprintf(stderr, "received %zd bytes, expected 4\n", rec);
72     close(sock);
73     freeaddrinfo(res);
74     return 1;
75 }
76 uint32_t seconds1900 = ntohl(net_time);
77 time_t unix_seconds = (time_t)(seconds1900 - TIME_DIFF_1900_TO_1970);
78
79 struct tm tmres;
80 if (localtime_r(&unix_seconds, &tmres) == NULL) {
81     perror("localtime_r");
82     close(sock);
83     freeaddrinfo(res);
84     return 1;
85 }
86
87 char timestr[256];
88 if (strftime(timestr, sizeof(timestr), "%Y-%m-%d %H:%M:%S %z", &tmres
89 ) == 0)
90     strncpy(timestr, "<time-format-failed>", sizeof(timestr));
91
92 printf("TIME from %s:%d -> %s\n", host, port, timestr);
93
94 close(sock);
95 freeaddrinfo(res);
96 return 0;
97 }
```

(四) 公共头文件

代码 7: expr4/common.h - 公共定义与工具函数

```

1  /**
2   * common.h - 公共头文件
3   *
4   * 定义TIME服务器和客户端共用的常量和结构
5   */
6
7 #ifndef COMMON_H
8 #define COMMON_H
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <unistd.h>
14 #include <time.h>
15 #include <sys/types.h>
16 #include <sys/socket.h>
17 #include <netinet/in.h>
18 #include <arpa/inet.h>
19 #include <errno.h>
20
21 /* TIME协议默认端口 */
22 #define TIME_PORT 37
23
24 /* 从1900年1月1日到1970年1月1日的秒数差 */
25 /* TIME协议使用1900年为基准，而Unix时间戳使用1970年为基准 */
26 #define TIME_OFFSET 2208988800UL
27
28 /* 缓冲区大小 */
29 #define BUFFER_SIZE 1024
30
31 /* 超时时间（秒） */
32 #define TIMEOUT_SEC 5
33
34 /**
35  * 错误处理函数
36  * @param msg 错误信息
37  */
38 static inline void error_exit(const char *msg)
39 {
40     perror(msg);
41     exit(EXIT_FAILURE);
42 }
43
44 /**
45  * 将Unix时间戳转换为TIME协议时间
46  * @param unix_time Unix时间戳
47  * @return TIME协议时间（从1900年开始的秒数）
48  */
49 static inline uint32_t unix_to_time_protocol(time_t unix_time)
50 {
51     return (uint32_t)(unix_time + TIME_OFFSET);
52 }
53

```

```

54 /**
55 * 将TIME协议时间转换为Unix时间戳
56 * @param time_protocol TIME协议时间
57 * @return Unix时间戳
58 */
59 static inline time_t time_protocol_to_unix(uint32_t time_protocol)
60 {
61     return (time_t)(time_protocol - TIME_OFFSET);
62 }
63
64 #endif /* COMMON_H */

```

(五) 完整版 TIME 服务器代码

代码 8: expr4/time_server.c - 完整版 TIME 服务器

```

1 /**
2 * time_server.c - 无连接TIME服务器
3 *
4 * 使用UDP协议实现RFC 868 TIME协议服务器
5 *
6 * 功能：
7 * 1. 监听指定UDP端口
8 * 2. 接收客户端请求
9 * 3. 返回当前时间（从1900年1月1日开始的秒数）
10 *
11 * 用法：./time_server [port]
12 */
13
14 #include "common.h"
15
16 int main(int argc, char *argv[])
17 {
18     int sockfd;                      /* UDP套接字描述符 */
19     struct sockaddr_in server_addr; /* 服务器地址结构 */
20     struct sockaddr_in client_addr; /* 客户端地址结构 */
21     socklen_t client_len;           /* 客户端地址长度 */
22     int port;                       /* 服务端口号 */
23     char buffer[BUFFER_SIZE];      /* 接收缓冲区 */
24     ssize_t recv_len;              /* 接收数据长度 */
25     time_t current_time;           /* 当前Unix时间 */
26     uint32_t time_value;            /* TIME协议时间值 */
27     uint32_t network_time;          /* 网络字节序时间值 */
28     char time_str[64];              /* 时间字符串 */
29
30     /* 解析端口参数 */
31     if (argc > 1)
32     {
33         port = atoi(argv[1]);
34         if (port <= 0 || port > 65535)
35         {
36             fprintf(stderr, "Invalid port number: %s\n", argv[1]);
37             fprintf(stderr, "Usage: %s [port]\n", argv[0]);

```

```

38         exit(EXIT_FAILURE);
39     }
40 }
41 else
42 {
43     port = TIME_PORT;
44 }
45
46 /* 创建 UDP 套接字 */
47 sockfd = socket(AF_INET, SOCK_DGRAM, 0);
48 if (sockfd < 0)
49 {
50     error_exit("Failed to create socket");
51 }
52
53 /* 设置套接字选项，允许地址重用 */
54 int opt = 1;
55 if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
56     < 0)
57 {
58     error_exit("Failed to set socket option");
59 }
60
61 /* 初始化服务器地址结构 */
62 memset(&server_addr, 0, sizeof(server_addr));
63 server_addr.sin_family = AF_INET;           /* IPv4 */
64 server_addr.sin_addr.s_addr = INADDR_ANY;   /* 监听所有网络接口 */
65 server_addr.sin_port = htons(port);         /* 端口号（网络字节序）
66 */
67
68 /* 绑定套接字到指定端口 */
69 if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(
70     server_addr)) < 0)
71 {
72     close(sockfd);
73     error_exit("Failed to bind socket");
74 }
75
76 printf("=====\\n");
77 printf("      无连接TIME服务器 (UDP)\\n");
78 printf("=====\\n");
79 printf("TIME Server started on port %d\\n", port);
80 printf("Waiting for client requests...\\n");
81 printf("Press Ctrl+C to stop the server\\n");
82 printf("=====\\n\\n");
83
84 /* 主循环：等待并处理客户端请求 */
85 while (1)
86 {
87     client_len = sizeof(client_addr);
88
89     /* 接收客户端请求（阻塞等待） */
90     /* 对于TIME协议，客户端只需发送任意数据即可请求时间 */

```

```

88     recv_len = recvfrom(sockfd, buffer, BUFFER_SIZE, 0,
89                         (struct sockaddr *)&client_addr, &
90                         client_len);
91
91     if (recv_len < 0)
92     {
93         perror("Failed to receive data");
94         continue;
95     }
96
97     /* 获取当前时间 */
98     current_time = time(NULL);
99
100    /* 转换为TIME协议时间（从1900年开始的秒数） */
101    time_value = unix_to_time_protocol(current_time);
102
103    /* 转换为网络字节序（大端序） */
104    network_time = htonl(time_value);
105
106    /* 获取可读的时间字符串 */
107    strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S",
108             localtime(&current_time));
109
110    /* 打印客户端信息 */
111    printf("[Request] From %s:%d\n",
112           inet_ntoa(client_addr.sin_addr),
113           ntohs(client_addr.sin_port));
114    printf(" TIME value: %u\n", time_value);
115    printf(" Local time: %s\n\n", time_str);
116
117    /* 发送时间值给客户端 */
118    if (sendto(sockfd, &network_time, sizeof(network_time), 0,
119               (struct sockaddr *)&client_addr, client_len) < 0)
120    {
121        perror("Failed to send data");
122        continue;
123    }
124
125
126    /* 关闭套接字（实际上不会执行到这里） */
127    close(sockfd);
128
129 }

```

(六) 完整版 TIME 客户端代码

代码 9: expr4/time_client.c - 完整版 TIME 客户端

```

1 /**
2  * time_client.c - 无连接TIME客户端
3  *
4  * 使用UDP协议实现RFC 868 TIME协议客户端
5  *

```

```

6  * 功能：
7  * 1. 向TIME服务器发送请求
8  * 2. 接收服务器返回的时间值
9  * 3. 将时间值转换为可读格式并显示
10 *
11 * 用法: ./time_client <server_ip> [port]
12 */
13
14 #include "common.h"
15
16 int main(int argc, char *argv[])
17 {
18     int sockfd;                                /* UDP套接字描述符 */
19     struct sockaddr_in server_addr;           /* 服务器地址结构 */
20     socklen_t server_len;                     /* 服务器地址长度 */
21     int port;                                 /* 服务端口号 */
22     const char *server_ip;                    /* 服务器IP地址 */
23     char request[] = "TIME";                  /* 请求消息(内容不重要) */
24     uint32_t network_time;                   /* 网络字节序时间值 */
25     uint32_t time_value;                     /* TIME协议时间值 */
26     time_t unix_time;                       /* Unix时间戳 */
27     char time_str[64];                      /* 时间字符串 */
28     struct timeval timeout;                 /* 超时设置 */
29     ssize_t recv_len;                        /* 接收数据长度 */
30
31     /* 检查参数 */
32     if (argc < 2)
33     {
34         fprintf(stderr, "Usage: %s <server_ip> [port]\n", argv[0]);
35         fprintf(stderr, "Example: %s 127.0.0.1 8037\n", argv[0]);
36         exit(EXIT_FAILURE);
37     }
38
39     server_ip = argv[1];
40
41     /* 解析端口参数 */
42     if (argc > 2)
43     {
44         port = atoi(argv[2]);
45         if (port <= 0 || port > 65535)
46         {
47             fprintf(stderr, "Invalid port number: %s\n", argv[2]);
48             exit(EXIT_FAILURE);
49         }
50     }
51     else
52     {
53         port = TIME_PORT;
54     }
55
56     printf("=====\\n");
57     printf("      无连接TIME客户端 (UDP)\\n");
58     printf("=====\\n");

```

```

59     printf("Connecting to TIME server %s:%d\n", server_ip, port);
60     printf("=====\\n\\n");
61
62     /* 创建UDP套接字 */
63     sockfd = socket(AF_INET, SOCK_DGRAM, 0);
64     if (sockfd < 0)
65     {
66         error_exit("Failed to create socket");
67     }
68
69     /* 设置接收超时 */
70     timeout.tv_sec = TIMEOUT_SEC;
71     timeout.tv_usec = 0;
72     if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(
73         timeout)) < 0)
74     {
75         perror("Warning: Failed to set timeout");
76     }
77
78     /* 初始化服务器地址结构 */
79     memset(&server_addr, 0, sizeof(server_addr));
80     server_addr.sin_family = AF_INET; /* IPv4 */
81     server_addr.sin_port = htons(port); /* 端口号（网络字节序） */
82
83     /* 转换IP地址 */
84     if (inet_pton(AF_INET, server_ip, &server_addr.sin_addr) <= 0)
85     {
86         close(sockfd);
87         fprintf(stderr, "Invalid server IP address: %s\\n", server_ip);
88         exit(EXIT_FAILURE);
89     }
90
91     /* 发送时间请求 */
92     printf("Sending time request...\\n");
93     if (sendto(sockfd, request, strlen(request), 0,
94                 (struct sockaddr *)&server_addr, sizeof(server_addr)) <
95                     0)
96     {
97         close(sockfd);
98         error_exit("Failed to send request");
99     }
100
101    /* 接收服务器响应 */
102    server_len = sizeof(server_addr);
103    recv_len = recvfrom(sockfd, &network_time, sizeof(network_time), 0,
104                        (struct sockaddr *)&server_addr, &server_len);
105
106    if (recv_len < 0)
107    {
108        close(sockfd);
109        if (errno == EAGAIN || errno == EWOULDBLOCK)
110        {
111            fprintf(stderr, "Error: Request timed out (no response from

```

```

                server)\n");
110     }
111     else
112     {
113         perror("Failed to receive response");
114     }
115     exit(EXIT_FAILURE);
116 }
117
118 if (recv_len != sizeof(network_time))
119 {
120     close(sockfd);
121     fprintf(stderr, "Error: Invalid response size (expected %zu,
122             got %zd)\n",
123             sizeof(network_time), recv_len);
124     exit(EXIT_FAILURE);
125 }
126
127 /* 转换字节序 */
128 time_value = ntohl(network_time);
129
130 /* 转换为 Unix 时间戳 */
131 unix_time = time_protocol_to_unix(time_value);
132
133 /* 获取可读的时间字符串 */
134 strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%M:%S",
135           localtime(&unix_time));
136
137 /* 显示结果 */
138 printf("\n===== Response Received\n");
139 =====\n";
140 printf("TIME protocol value : %u\n", time_value);
141 printf("Unix timestamp       : %ld\n", (long)unix_time);
142 printf("Local time           : %s\n", time_str);
143 =====\n");
144
145 /* 与本地时间比较 */
146 time_t local_time = time(NULL);
147 long diff = (long)(unix_time - local_time);
148 printf("\nTime difference from local: %ld seconds\n", diff);
149
150 /* 关闭套接字 */
151 close(sockfd);
152
153 return 0;
154 }
```