上机作业 1——简单的客户-服务器程序

1711430 江玥

2019 年 **10** 月 **27** 日

摘 要

基于 **UNIX** 下的 **BSDSocket** 框架，基于 **UDP** 协议，实现客户机和服务器之间的简单交互。利用 **Xcode** 的可视化编程，使用 **Storyboard** 和 **Xib** 可视化工具以及 **objective-c** 语言来编写界面，实现客户机和服务器的交互界面。在此次作业中，了解了 **iOS** 下的可视化编程，深入研究了 **UDP** 协议机制，比较了 **TCP** 和 **UDP** 之间的异同，实现了 **UDP** 交互的整个过程。

目录

1 介绍

1.1 UDP

1.1.1 UDP 简介

UDP 是 **User Datagram Protocol** 的简称，是 **TCP** / **IP** 体系结构中一种无连接的传输层协议，提供面向事物的简单不可靠信息传送服务。

1.1.2 UDP 特点

- **UDP** 缺乏可靠性。**UDP** 本身不提供确认序列号，超时重传等机制。**UDP** 数据报可能在网络中被复制，被重新排序。即 **UDP** 不保证数据报会到达其最终目的地，也不保证各个数据报的先后顺序，也不保证每个数据报只到达一次。
- **UDP** 数据报是有长度的。每个 **UDP** 数据报都有长度，如果一个数据报正确地到达目的地，那么该数据报的长度将随数据一起传递给接收方。而 **TCP** 是一个字节流协议，没有任何（协议上的）记录边界。
- **UDP** 是无连接的。**UDP** 客户和服务器之前不必存在长期的关系。**UDP** 发送数据报之前也不需要经过握手创建连接的过程。
- **UDP** 支持多播和广播。

1.1.3 UDP 传输流程

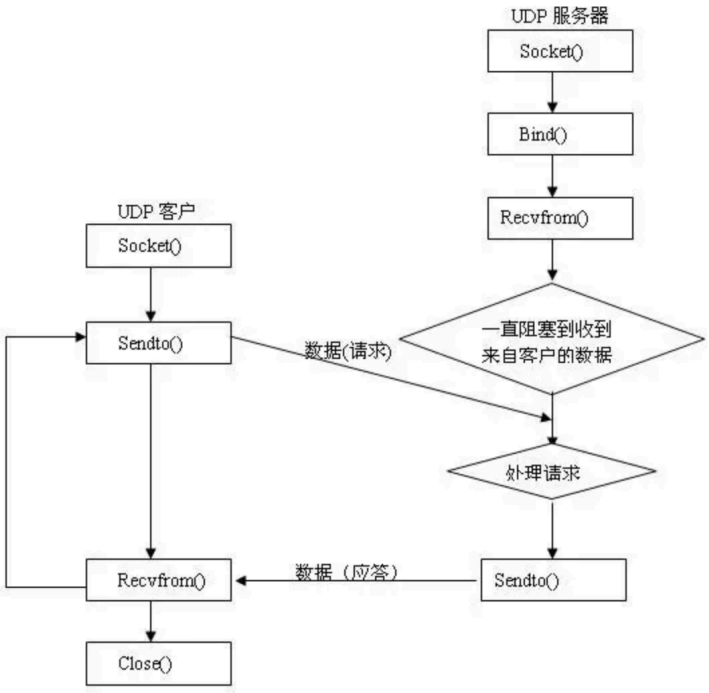

图 1: **UDP** 传输流程图

1.1.4 UDP 与 TCP 比较

	TCP	UDP
是否连接	面向连接	面向非连接
传输可靠性	可靠的	不可靠的
应用场合	传输大量的数据	传输少量的数据
传输速度	慢	快
报文	流模式	数据报模式
数据顺序	保证传输数据的顺序	不保证传输数据的顺序

总结：TCP 面向连接、传输可靠 (保证数据正确性，保证数据顺序)、用于传输大量数据 (流模式)、速度慢，建立连接需要开销较多 (时间，系统资源)。

1.2 Socket

1.2.1 Socket 简介

通过 ip 地址、协议、端口号唯一标识网络中的进程后，就可以利用 socket 进行通信了。

socket 即套接字，是在应用层和传输层之间的一个抽象层，它把 TCP/IP 层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。

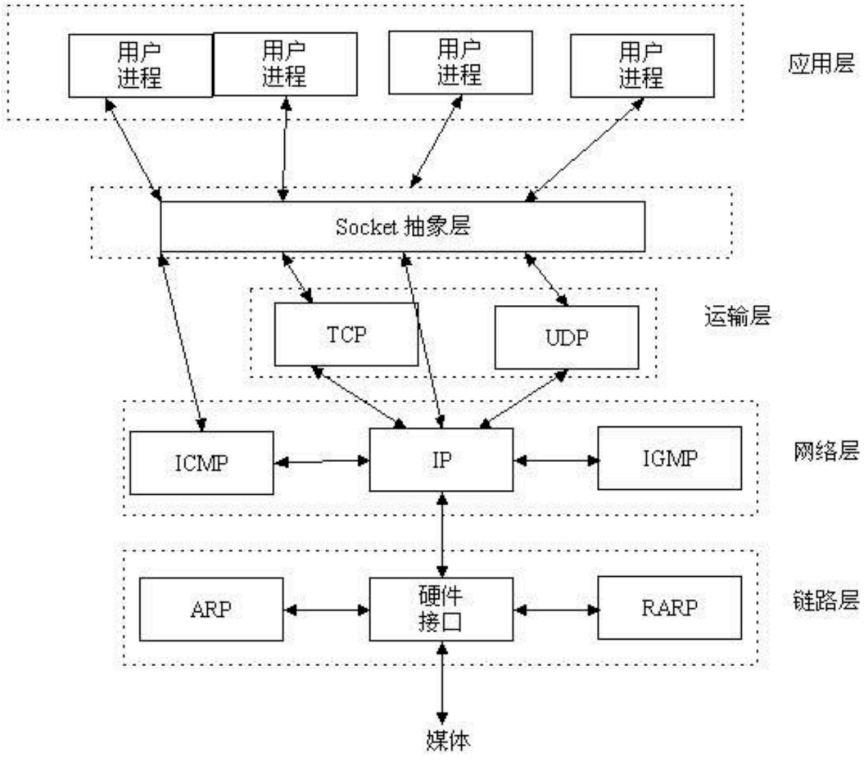

图 2: 网络传输架构

1.2.2 BSDSocket

BSDSocket，即 **BSD** 套接字，用于网络套接字与 **Unix** 域套接字，主要用于实现进程间通讯，在计算机网络通讯方面被广泛使用。

由于我的电脑的 **MAC OS** 系统，而 **BSD** 是 **Unix** 下的 **Socket** 框架，**iOS** 和 **MAC OS** 的内核就是 **Unix**，因此可以在 **Xcode** 上直接使用 **BSDSocket**。

BSDSocket 提供了一系列 **API** 函数，来支持网络之间的交互。在这次实现简单的 **UDP** 编程作业中，主要用到的 **API** 函数有：

- **socket()**
  创建一个新的确定类型的套接字，类型用一个整型数值标识（文件描述符），并为它分配系统资源。

- **bind()**
  一般用于服务器端，将一个套接字与一个套接字地址结构相关联，比如，一个指定的本地端口和 **IP** 地址。

- **sendto()**
  往远程发送套接字

- **recvfrom()**
  从远程接收套接字

这些函数的详细用法在之后分析程序时再进行说明。

1.3 可视化实现

为了实现在 **MAC OS** 下的可视化，在这里我运用了 **Xcode** 支持的 **storyboard** 工具，使用 **objective-c** 语言，实现了基于 **UDP** 协议的客户机-服务器的可视化交互。可视化项目的创建：
进入 **Xcode**，创建一个空项目，点击 **Single View App**，语言选择 **Objective-C**，用户界面（**User Interface**）选择 **storyboard**。至此，生成可一个可视化 **App** 的雏形框架。

一个可视化项目下的文件组成：**(**项目名为 **project)**

```
project/
├── project
│   ├── AppDelegate.h
│   ├── AppDelegate.m
│   ├── SceneDelegate.h
│   ├── SceneDelegate.m
│   ├── ViewController.h
│   ├── ViewController.m
│   ├── Main.storyboard
│   ├── Assets.xcassets
│   ├── LaunchScreen.storyboard
│   ├── info.plist
│   └── main.m
├── projectTests
│   ├── projectTests.m
│   └── info.plist
├── projectUITests
│   ├── projectUITests.m
│   └── info.list
└── Products
    ├── project.app
    ├── projectTests.xctest
    └── projectUITests.xctest
```

其中，**ViewController.h** 含有主要实现类的变量成员；**ViewController.m** 含有主要实现类的所有方法 **(**函数**)**；**main.m** 是主函数，触发整个可视化页面的生成；**Main.storyboard** 主要负责可视化的设计，实现 **ViewController** 中的函数的可视化体现。其余文件在这次作业中没有做大的改动。

1.3.1 storyboard

storyboard 含有以下几个工作区域：

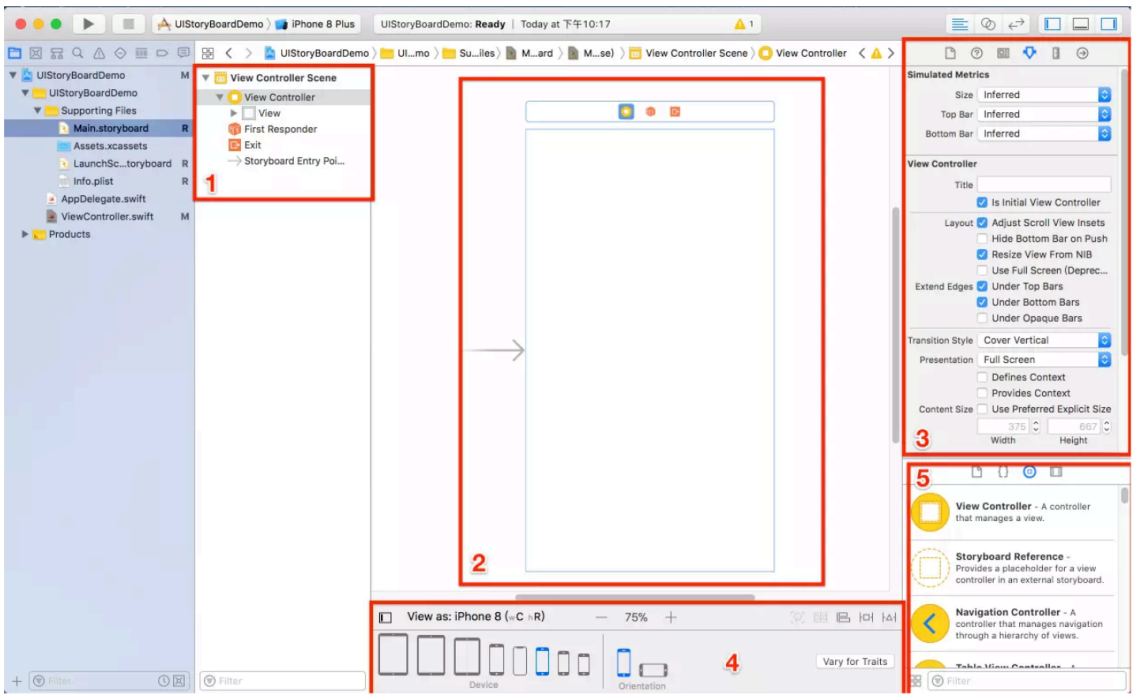

图 3： 工作区域展示

1. 菜单导航区域：添加的控制器、控制器之间的跳转 **Segue** 及控制器上面的控件和布局等等信息都在这里显示。

2. 工作展示区域：可以在这里给各个控制器添加控件和预览布局后的控件。

3. 配置区域：可以在这里将 **storyboard** 和代码文件关联和查看关联后的信息，也可以直接在这里配置控件的属性等等。

4. 布局区域：上面有很多机型选择，可以直接选择机型和方向，区域 **2** 会根据选择的机型和自动布局直接预览控件显示的效果和布局，中间的加减符号可以放大和缩小区域 **2** 中的内容，右上角的几个按钮可以进行自动布局操作。

5. 控件区域：可以直接在这里选择控件，然后拖入区域 **2** 中。

1.3.2 Objective-C

Objective-C 是一种通用、高级、面向对象的编程语言。**Objective-C** 是 **C** 语言的严格超集- - 任何 **C** 语言程序不经修改就可以直接通过 **Objective-C** 编译器，在 **Objective-C** 中使用 **C** 语言代码也是完全合法的。两个语言的逻辑非常相似的，只是在有些语法上和用法上有出入的地方，并且使用的范围不同。

Objective-C 最大的特点是继承了 **Smalltalk** 的消息传递模型，执行函数时，与其说对象互相调用方法，不如说是对象之间互相传递消息。

在实际使用的过程中，因为不了解 **Objective-C** 语言的语法结构和特性，在实现功能的时候费了些周折。

2 外部函数 & 类型

2.1 引入的头文件

- <sys/socket.h>

  核心 **BSD** 套接字核心函数和数据结构。
  AF_INET、**AF_INET6** 地址集和它们相应的协议集 **PF_INET**、**PF_INET6**.
  广泛用于 **Internet**，这些包括了 **IP** 地址和 **TCP**、**UDP** 端口号。

- <netinet/in.h>

  AF_INET 和 **AF_INET6** 地址家族和他们对应的协议家族 **PF_INET** 和 **PF_INET6**。在互联网编程中广泛使用，包括 **IP** 地址以及 **TCP** 和 **UDP** 端口号。

- <arpa/inet.h>

  处理数值型 **IP** 地址的函数。

2.2 API 函数

2.2.1 socket()

```
1 int socket(int domain, int type, int protocol)
```

socket()：创建 **Socket**
socket() 有三个参数：

- **domain** 为创建的套接字指定协议集（或称做地址族 **address family**）。
  例如：
  - **AF_INET IPv4** 网络协议
  - **AF_INET6 IPv6** 网络协议

  在这次作业中，创建 **socket** 时使用的都是 **IPv4** 协议。

- **type socket** 类型：
  - **SOCK_STREAM** 流式套接字：**TCP**
  - **SOCK_DGRAM** 数据报式套接字：**UDP**

- **protocol** 指定实际使用的传输协议：
  - **IPPROTO_TCP TCP** 传输协议
  - **IPPROTO_UDP UDP** 传输协议
  - **protocol = 0** 时，根据**type**参数自动选择类型

返回值：

- 调用成功就返回新创建的套接字的描述符（大于 0）
- 失败返回-1

2.2.2 bind()

```
int bind(int sockfd, const struct sockaddr * addr, socklen_t addrlen)
```

bind() 为一个套接字分配地址，绑定 **ip** 地址 **+** 端口。
bind() 有三个参数：

- **sockfd** 套接字描述符，就是上面 **socket()** 的返回值
- **addr** 是一个 **sockaddr(**用于表示锁分配地址**)** 结构指针，该结构中包含了要结合的地址和端口号。
- **addrlen** 用 **socklen_t** 字段指定了 **sockaddr** 结构的长度

返回值：如果函数执行成功，返回值为 **0**，否则为 **-1**

2.2.3 sendto

```
ssize_t sendto(int sockfd, const void * buf, size_t size, int flags,
    const struct sockaddr * dest_addr, socklen_t addrlen)
```

sendto()： 发送消息
sendto() 一共有 **6** 个参数：

- **sockfd**：套接字描述符
- **buf**：待发送数据的缓冲区
- **size**：缓冲区长度，是字节的个数，需使用 **strlen()** 计算所有字节的长度
- **flags**：调用方式标志位，一般为 **0**，改变 **flags**，将会改变 **sendto** 发送的形式
- **dest_addr**：可选指针，指向目的套接字的地址
- **addrlen**：**dest_addr** 的长度

返回值：如果成功，则返回发送的字节数，失败则返回 **-1**

2.2.4 recvfrom

```
ssize_t recvfrom(int sockfd, void * buf, size_t size, int flags,
    struct sockaddr * src_addr, socklen_t * addrlen);
```

recvfrom()：接收消息 **recvfrom()** 一共有 **6** 个参数：

- **sockfd** ：套接字描述符
- **buf** ：接收数据缓冲区
- **size** ：缓冲区长度
- **flags** ：调用操作方式
- **src_addr**：可选指针，指向装有源地址的缓冲区
- **addrlen** ：可选指针，指向 **address** 缓冲区长度值

返回值：如果成功，返回实际所读的字节数，如果返回的值是 **0** 表示已经读到文件的结束了，小于 **0** 表示出现了错误。

2.3 一些类结构

```
struct sockaddr_in {
        __uint8_t sin_len;
        // 地址长度
        sa_family_t sin_family;
        // IP 地址协议族，必须设为 AF_INET
        in_port_t sin_port;
        // 通信端口
        struct in_addr sin_addr;
        // 以网络字节排序的 4 字节 IPv4 地址
        char sin_zero[8];
        // 填充项，是为了让 sockaddr 与 sockaddr_in
        两个数据结构保持大小相同而保留的空字节
    };

    struct sockaddr {
        __uint8_t sa_len;
        // 地址长度
        sa_family_t sa_family;
        // IP 地址协议族，必须设为 AF_INET
        char sa_data[14];
        // 地址值
    };

```

```
    struct in_addr {
        uint32_t s_addr;
        // 按照网络字节顺序存储 IP 地址
    };
```

sockaddr_in 和 **sockaddr** 是并列的结构，指向 **sockaddr_in** 的结构体的指针也可以指向 **sockaddr** 的结构体，并代替它。

3 服务器

创建服务器页面：这里使用的是 **Mac** 的界面开发。

3.1 创建 ViewController 类

```
#import "ViewController.h"
#import <sys/socket.h>
#import <netinet/in.h>
#import <arpa/inet.h>

@interface ViewController ()
{

    NSString *_loc_ipAdr,*_loc_port; // 本地 ip 地址、端口号
    int _udp_serverSockfd; // 服务端套接字描述符

}
@end

@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad]; // 调用父类方法
    _loc_ipAdr = @"127.0.0.1"; // 本地 ip
    _loc_port = @"10000"; // 本地端口

    [NSThread detachNewThreadSelector:@selector(creatUDPSocket)
        toTarget:self withObject:nil];
    // 创建一个生成 UDP 套接字的线程
}
```

3.2 创建 Socket

```
- (void)creatUDPSocket{
    _udp_serverSockfd  = socket(AF_INET, SOCK_DGRAM, 0);
    // ipv4, 数据报式
    if (_udp_serverSockfd > 0 ) {
        NSLog(@"UDP socket 创建成功");
        [self UDPBind];
        // 如果创建成功, 向控制台输出创建成功消息, 并对新创建的 socket
        // 进行 ip 地址和端口号的绑定 (进入到 Bind() 函数中)
    }else {
        NSLog(@"UDP socket 创建失败");
        // 打印失败信息, 不进行下一步的操作
    }
}
```

3.3 绑定信息

```
- (void)UDPBind {
    struct sockaddr_in loc_addr;
    // 获取本地地址
    memset(&loc_addr, 0, sizeof(loc_addr));
    // 清空指向的内存中的存储内容, 因为分配的内存是随机的
    loc_addr.sin_family = AF_INET;
    // 设置协议族为 IPv4
    loc_addr.sin_port = htons(_loc_port.intValue);
    // 设置端口
    loc_addr.sin_addr.s_addr = inet_addr(_loc_ipAdr.UTF8String);
    // 设置 IP 地址
    int udpCode = bind(_udp_serverSockfd, (const struct
        sockaddr *)&loc_addr,sizeof(loc_addr) );
    // 绑定, 调用 bind 函数
    if (udpCode == 0) {
        NSLog(@"UDP socket 绑定成功");
        [self UDPRecv];
        // 若绑定成功, 则进入 UDPRecv 函数, 进行对消息的不断监听状态
    }else{
        NSLog(@"UDP socekt 绑定失败");
        // 绑定失败则不进行下面的操作
    }
}
```

3.4 接收客户端消息并进行响应

```
- (void)UDPRecv{
    struct sockaddr_in des_addr; // 目标地址
    bzero(&des_addr, sizeof(des_addr));
    // 清空指向的内存中的存储内容，否则会影响到下次的使用
    char buf[1024]; // 存储接收到的消息
    while(1) { // 接收数据
        bzero(buf, sizeof(buf));
        //清空指向的内存中的存储内容
        socklen_t des_addr_len = sizeof(des_addr);
        // 目标地址类型的字节长度
        ssize_t recvLen = recvfrom(_udp_serverSockfd, buf,
            sizeof(buf), 0, (struct sockaddr *) &des_addr, &des_addr_len);
        // 通过函数 recvfrom 将 message 放入到 buf 中
        NSString *recvStr, *sendMsg;
        NSString * time = [self getTime:0]; // 现阶段的时间
        NSString * date = [self getTime:1]; // 现阶段的日期
        if (recvLen > 0) {
            if (strcasecmp(buf, "date") == 0) {
                sendMsg = date;
            } else if (strcasecmp(buf, "time") == 0) {
                sendMsg = time;
            } else {
                sendMsg = @" 错误信息";
            }
            // strcasecmp 为忽略大小写的字符串比较函数
            recvStr = [NSString stringWithFormat:
             @"%@   %@   [UDP 消息][来自客户端%@ : %hu]:
             请求【%@】, 响应【%@】\n",
             date,
             time,
             [NSString stringWithUTF8String:inet_ntoa(des_addr.sin_addr)],
             des_addr.sin_port, 、
             [NSString stringWithUTF8String:buf],
             sendMsg];
             // 定义显示在服务器日志上的格式
            sendto(_udp_serverSockfd, sendMsg.UTF8String,
            strlen(sendMsg.UTF8String), 0, (struct sockaddr *)&des_addr,
                sizeof(des_addr));
                    // 给客户端响应，发送返回的消息
            dispatch_async(dispatch_get_main_queue(), ^{
```

```
                    self->_recTextView.string =
                    [_recTextView.string stringByAppendingString:recvStr];
                // 将新输入的消息 append 进去，显示在服务器的日志上。

            });
        }
    }
}
```

这个函数里的类型转换比较复杂，为了将消息显示在服务器的日志上，需要对消息里代入的每个变量进行适当的类型转换，转换成可以输出在服务器上的 **NSString** 类型。

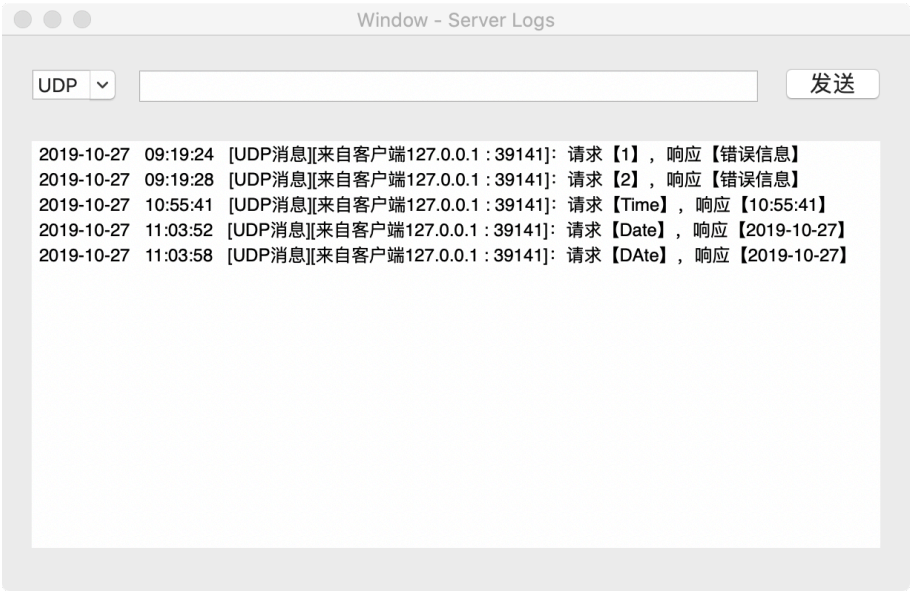

图 **4:** 服务器显示效果

4 客户端

4.1 创建 ViewController 类

```
@interface ViewController (){
    int _udp_clientSockfd;
    // 客户端套接字描述符
    NSString *_loc_ipAdr,*_loc_port ,*_dest_ipAdr,*_dest_port;
    // 客户端 ip、端口号；服务器 ip、端口号
    int success;
    // 相当于一个 flag(标记)
}
@property (weak, nonatomic) IBOutlet UITextField *des_ipAdress;
```

```
@property (weak, nonatomic) IBOutlet UITextField *des_port;
@property (weak, nonatomic) IBOutlet UITextField *sendTF;
@property (weak, nonatomic) IBOutlet UITextView *recvTextView;
@end
// 设置控制部件、输入部件
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    _loc_ipAdr = @"127.0.0.1";
    _dest_ipAdr = _dest_port = @"";
    // 对服务器的 ip 值和端口号赋初值为空字符串
    [NSThread detachNewThreadSelector:@selector(creatUDPSocket)
       toTarget:self withObject:nil];
     // 初始化一个 UDPSocket 进程
}
```

4.2 创建 Socket

```
- (void)creatUDPSocket {
    _udp_clientSockfd = socket(AF_INET, SOCK_DGRAM, 0);
    // protocol 根据 type 的值自动设定
    if (_udp_clientSockfd > 0) {
        NSLog(@"UDP socket 创建成功");
        success = 1;
        // 创建成功，标记为 1

    } else {
        NSLog(@"UDP socket 创建失败");
        success = 0;
        // 创建失败，标记为 0
    }
}
```

4.3 发送消息到服务器

该函数是由部件触发的函数；在可视化页面上，点击发送按钮，触发 **send** 函数，客户端向服务器发送请求。

```
- (IBAction)send:(id)sender{

    if (!self.sendTF.text.length) {
        return;
    }
    if (!success) {
        NSString *str = [NSString stringWithFormat:
             @"socket 创建失败，无法传输消息！"];
        // 如果上述操作有失败，即是触发按钮也无法发送消息!
        dispatch_async(dispatch_get_main_queue(), ^{
            self->_recvTextView.text = str;
        });
    }

    ssize_t sendLen = 0;

    struct sockaddr_in des_addr;
    bzero(&des_addr, sizeof(des_addr));
    // 发送数据
    ssize_t recvLen = 0;
    char buf[256];
    do {
        _dest_ipAdr = _des_ipAdress.text;
        _dest_port = _des_port.text;
        // 对输入到相应文本框内的内容进行摘取赋值，传给 id 和 port
        des_addr.sin_family      = AF_INET;
        des_addr.sin_port        = htons(_dest_port.intValue);
        des_addr.sin_addr.s_addr = inet_addr(_dest_ipAdr.UTF8String);
        // 进行必要的类型转换
        sendLen = sendto(_udp_clientSockfd, _sendTF.text.UTF8String,
        strlen(_sendTF.text.UTF8String), 0,
                (struct sockaddr *)&des_addr, sizeof(des_addr));
        if (sendLen < 0) {
            NSLog(@" 传输失败");
            continue;
        }
        // 如果发送消息失败，执行重传操作
        socklen_t des_addr_len = sizeof(des_addr);
        recvLen = recvfrom(_udp_clientSockfd, buf,
            sizeof(buf), 0, (struct sockaddr*)&des_addr, &des_addr_len);
        if (recvLen > 0) {
```

```
42              NSString *recvStr = [NSString stringWithFormat:
43              @"[UDP 消息][来自服务端%@:%@]: %@\n",
44              _dest_ipAdr, _dest_port, [NSString stringWithUTF8String:buf]];
45              dispatch_async(dispatch_get_main_queue(), ^{
46                  self->_recvTextView.text = recvStr;
47              });
48          }
49          // 如果接收到了服务器的响应，将响应消息打印在响应文本框中
50          else {
51              NSString *recvStr = [NSString stringWithFormat
52              :@" 消息接收失败，正在尝试重传。"];
53              dispatch_async(dispatch_get_main_queue(), ^{
54                  self->_recvTextView.text = recvStr;
55              });
56          } // 打印错误信息到客户端页面上
57      }while (recvLen <= 0); // 如果没有 (recvLen<=0)，则重传。
58  }
```

这个函数完成了对服务器发送请求，处理服务器发来的相应的整个过程。并且还增加了重传机制，加强了数据传输的可靠性。

图 5: 客户端

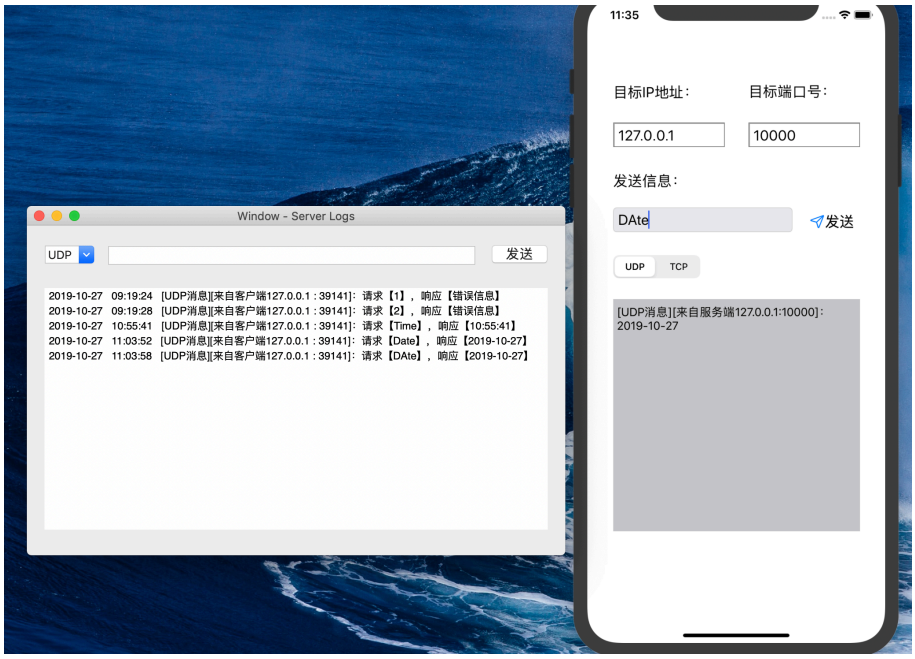

图 6: 客户机 & 服务器交互

5 疑问 & 思考

1. 在 **recvfrom** 消息以后，对目标地址的 **port** 进行输出时，一直出现指针访问错误的提示，导致不能正常输出。查阅了相关资料后发现，**sockaddr** 类型里的 **port** 的类型是 **in_port_t**，占 2 个字节 16 个 **bit**，强制转换为 **NSString** 类型无法访问它的位置。后来，我发现了在 **NSString** 中能替代半字类型的“斜杠 **n**”字符，用它对 **port** 进行替代，可以正确输出。

2. **UDP** 的一些功能还没有完全实现，之后可以对其进行一一实现。