“微信”实验报告

2015011308 计 53 唐适之

协议设计

总体来看，本项目的通信协议是一个实现在 TCP 之上的 ASCII 字符流协议。这主要是出于两点考虑：

1. “微信”程序涉及用户的身份验证，用户完成验证后应当保持在已登录的状态。选择基于连接的 TCP 比基于无连接的 UDP 更容易实现登录状态的保持；
2. 传输载荷选用 ASCII 字符流而不是二进制流的形式，能使调试更加便利。这样可以不经客户端，而直接使用 Telnet 连接服务端进行调试。

由于 TCP 是基于流而不是报文的协议，不同分组会被合并，所以协议中要能区分依次发送的各个消息的边界。因为本协议是 ASCII 字符流协议，若使用字符计数法，要以字符而非二进制形式记录消息长度，开销较大，故只使用了字符定界法。具体机制如下：

发送消息时，在每个消息结尾添加“$.”作为结束标记。在消息中如遇“$”字符，则转义为“$$”。接收消息时，如遇“$”字符，则根据下一个字符进行判断：若下一个字符仍为“$”，则表示原文中一个“$”字符；若下一个字符为“.”，则表示消息结束。

在确定消息边界后，每个消息采用 JSON 编码，每个消息可以解析为一个 JSON 对象。每个客户端发给服务端的消息可视作一个命令，其中必定包含“cmd”域表示所要执行的命令类型，对象中的其他域表示此命令的参数，具体约定如下：

“cmd”域	含义	其他参数
“register”	注册新用户	“name”：用户名 “password”：密码
“login”	登陆	“name”：用户名 “password”：密码
“contact”	更新联系人信息	“op”：“add”表示添加联系人，“del”表示删除联系人 “name”：联系人用户名（联系人是双向的，若 A 添加了 B，B 也会自动添加 A）
“chat”	发送聊天	“name”：目标用户 “message”：聊天内容
“log”	获取聊天记录	“since”：获取的聊天记录自此时间戳开始，最多 128 条
“profile”	更新用户信息	“data”：要更新的用户信息

协议中采用了端到端的原则：协议不对聊天内容、用户信息的格式做具体要求，而依客户端的实现决定。这使服务端不必关心具体的数据内容，减少了服务端与客户端的耦合性。

服务端可能回应客户端的请求，或主动给客户端发送消息，这两种情况不做区别对待。服务端向客户端发送的消息也采用 JSON 编码，每个消息可以解析为一个 JSON 对象。此对象包含若干个键值对，每个键值对的键表示消息类型，值表示消息内容。这样设计是为了同时发送多个不同类型的消息。具体约定如下：

键	含义	值
“register”	注册成功	必为“ok”
“login”	登陆成功	必为“ok”
“chat”	成功收到聊天消息	必为“ok”
“contact”	当前用户最新的联系人信息	数组，每一项表示一个联系人用户名
“income”	主动推送当前用户收到的新聊天消息，或是回复获取聊天记录的请求	数组，每一项有如下 4 个域： “timestamp”: 聊天发送时间 “from”: 发送人 “to”: 接收人 “body”: 聊天内容
“log”	请求的聊天记录是否发送完毕	“fin”表示已发送完毕，“more”表示还有更多
“info”	异常或错误信息	字符串。客户端会不做处理额外直接展示给用户

值得注意的是，由于聊天记录可能非常庞大，客户端请求聊天记录时，并非请求全部的聊天记录，而是请求某一个时间戳之后的记录；服务端回复时，也只回复最多 128 条记录，如果记录有超过 128 调，服务端会告知客户端再发送一个请求。这样将聊天记录分成多条消息发送的设计是出于公平性考虑，以避免服务端的资源长时间被某一个用户聊天记录请求占用，而无法处理其他用户的正常聊天。

服务端实现

服务端的 TCP 连接基于 Linux 原生的 socket 机制实现。其中一个 socket 负责监听所使用的 TCP 端口，一旦有客户端的请求到来，就与之建立一个 TCP 连接，并产生一个新的 socket 专用于与该客户端进行通信。

为了同时处理多个用户的请求，采用了 Linux 原生的 epoll 机制（类似早期 Linux 的 select 机制）。epoll 本身也是一个 socket，负责以轮询的方式同时监听其他多个 socket，一旦某个被监听的 socket 的状态变为可读，epoll 自身的状态就会变成可读，并使程序可以得知是哪一个被监听的 socket 变成可读了。如果此 socket 是负责监听 TCP 端口的 socket，这意味着有新的连接请求；如果此 socket 是负责具体通信的 socket，这意味着有新消息到来。

与 epoll 相对，另一种同时监听多个 TCP 连接的办法是采用多线程。但由于处理 TCP 连接是 I/O 密集型，而不是 CPU 密集型的任务，多个线程并不能同时占用网卡与外界交互，所以多线程并不能提高性能。反而，线程间的上下文切换会消耗额外的时间。因此，本项目没有采取此方案。

本项目中 Conn 类负责使用 epoll 管理各连接，当每个连接请求到来时，产生一个 Context 对象。每个 Context 对象负责维护某一个 TCP 连接的上下文，包括 TCP 连接相关信息和当前登陆用户的信息，并提供收发数据的接口。Context 在收发数据时还负责处理以上协议中约定的编解码。

主程序维护用户、消息、联系人等全局信息，并在 Context 对象中设置回调函数，指明有新消息从客户端发来时要如何处理，处理完毕后再调用 Context 对象的接口将回复发回客户端。主程序也可以在 Context 类中主动查询某个用户当前有没有连接，如果有的话连接是哪一个，并主动向其发送消息。

客户端实现

由于协议中采用了端到端原则，服务端不对聊天内容等做具体约定，这就要求客户端对其进行处理。

聊天内容可能是文字或图片，客户端需要将这两种不同的内容编码进同一个“聊天内容”字段中以 ASCII 字符流的形式发送。这里依然采用了 JSON 编码，每条聊天消息编码为一个对象，其中“type”字段表示类型（“text”或“file”），“data”字段表示 Base64 编码的有效载荷。采用 Base64 编码是为了使二进制文件可以在 ASCII 字符流协议上发送。由于文件可能较大，为了不长时间占用系统资源使 UI 失去响应，一个文件会被切分成多个块发送，所以此对象中还包含了文件总大小、此块的偏移量等信息（类似 IP 协议的设计）。

类似的，用户信息域在客户端被实现为 Base64 编码的用户头像。

由于服务端除了被动响应请求以外，只会主动推送新聊天消息以及其他少量信息，登陆时的具体流程需要在客户端处理。用户登录时，按如下流程执行：

1. 与服务端建立 TCP 连接，若失败则放弃（客户端会展示“info”类型消息包含的错误信息）；
2. 如果要注册新用户，发送注册请求，成功后转 3；
3. 发送登陆请求，若失败则主动断开连接，这样可以按 1 失败来处理；
4. 服务端会在登录后主动推送最新的联系人信息，客户端不需主动同步；
5. 服务端会在登录后主动推送最新的用户信息，客户端不需主动同步；
6. 从时间戳 0 开始请求同步消息记录，每收到一组回复就增加时间戳再次发送请求，直到同步完成。

登陆完成后，客户端只需及时响应用户操作和服务器来信即可，再无复杂流程。

客户端的 UI 基于 Electron 实现。Electron 提供了一种机制使客户端程序的 UI 可以用类似网页的方式实现，这样可以方便地利用 HTML/CSS 提供的灵活特性。此 UI 中提供了新消息提示、文件上传/下载进度条、Shift + Enter 快捷键发送消息等人性化功能。

项目地址

https://github.com/roastduck/im

使用的第三方库

1. *JSON for Modern C++*：一个 C++的 JSON 解析库；
2. *Electron*：一个 Node.js 的前端 UI 框架，用于在客户端程序上使用 HTML/CSS 编写 UI；
3. *AngularJS*：一个 Javascript 前端框架，用于动态设置 HTML 中元素的内容；
4. *Bootstrap*：一个 HTML/CSS 样式库，用于美化 UI。

思考题

1. *Linux 系统里，Socket 与文件的关系。*

Socket 提供了与文件一致的系统调用与 API 接口，使软件可以统一地处理网络收发和文件读写，降低了系统地复杂性。

2. *即时通信时，服务器程序的角色*

服务器程序验证和管理用户身份，存储和转发用户通信，并存储用户配置等额外信息。

3. *服务器端口与客户端连接个数的关系。*

服务器端口与客户端连接个数没有关系。一个服务器端口可以处理多个客户端连接。