

ESPRESSIF

ESP32 Protocol Stack Key



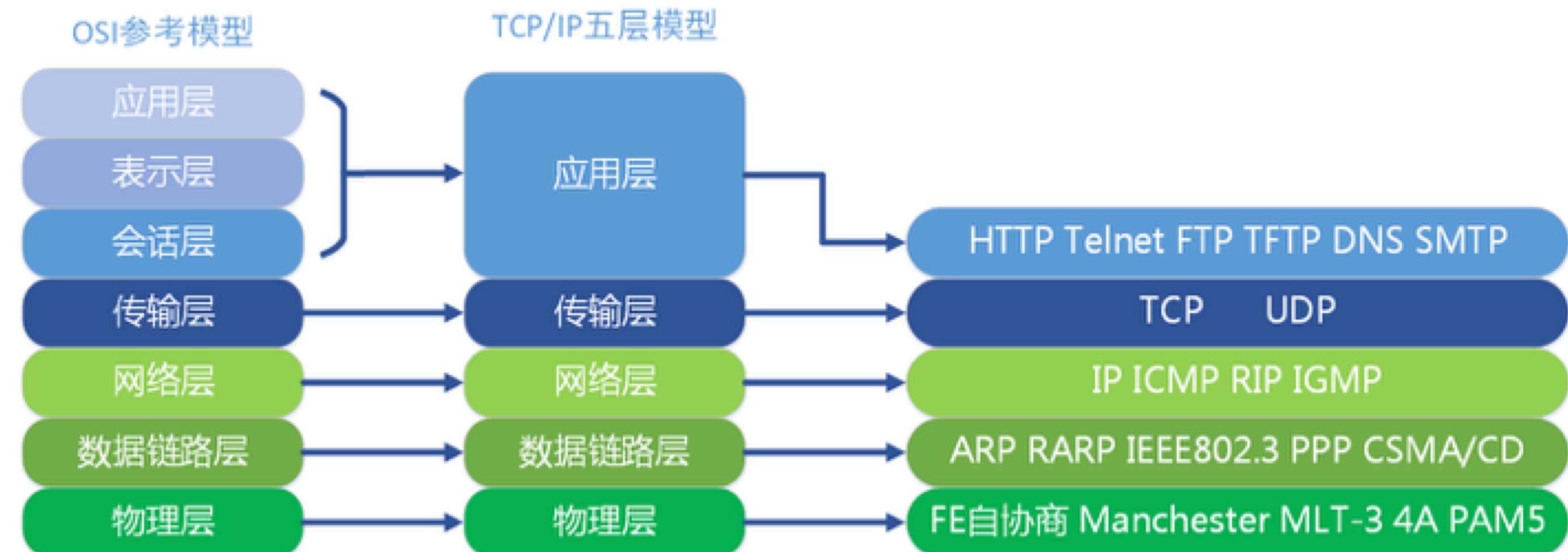
概述

- TCP / IP
- TLS



TCPPIP

TCP/IP参考模型分为五个层次：应用层、传输层、网络层、数据链路层和物理层，其中数据链路层和物理层在四层模型中统称为网络接口层

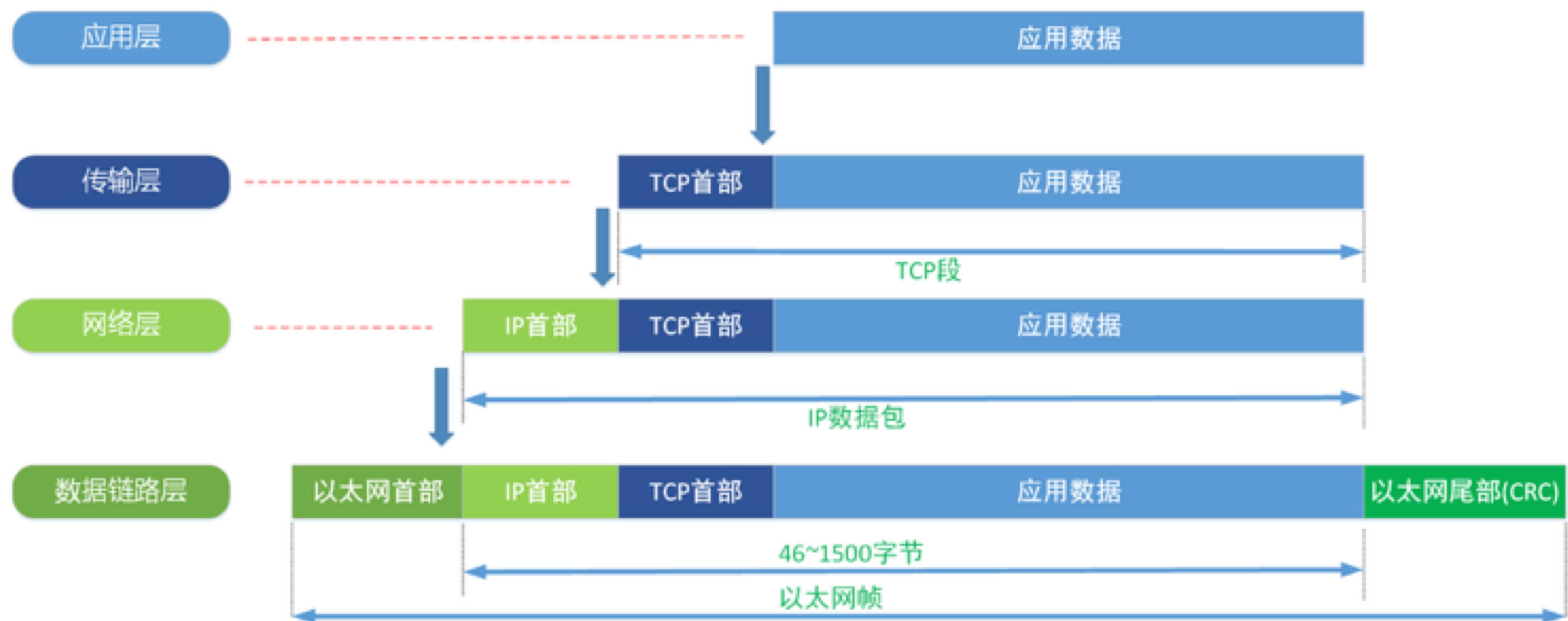


- 应用层：最靠近用户的一层，是为计算机用户提供应用接口，也为用户直接提供各种网络服务。我们常见的：HTTP，HTTPS，FTP，POP3、SMTP等属于该层
 - 传输层：建立主机端到端的链接层，其作用是为上层协议提供端到端的可靠和透明的数据传输服务，包括处理差错控制和流量控制等问题。常见的TCP，UDP等属于该层
 - 网络层：通过IP寻址建立两个节点之间的连接，为源端的运输层送来的数据，选择合适的交换节点，准确地传送给目的端的运输层。常见的IP，ICMP，IGMP等属于该层
 - 网络接口层：统一描述为链路层的部分，旨在对具体的网络硬件，软件进行统一的封装，为上层提供统一接口的服务，常见的环回接口，PPP，SLIP等属于该层



数据封装

当我们应用程序传输数据的时候，数据被送入协议栈中，然后逐个通过每一层，直到最后到物理层数据转换成比特流，送入网络。在这个过程中，每一层都会对要发送的数据加一些首部信息。以TCP传输为例，整个过程如下图：





数据封装

如图可以看出，每一层数据是由上一层数据+本层首部信息组成的，其中每一层的数据称为本层的协议数据单元，即PDU。

应用层数据在传输层添加TCP报头后得到的PDU被称为 Segment(数据段)，如图示 TCP段
传输层的数据 (TCP段) 传给网络层, 网络层添加IP报头得到的PDU被称为Packet(数据包); 如图示IP数据包
网络层数据报 (IP数据包) 被传递到数据链路层, 封装数据链路层报头得到的PDU被称为Frame(数据帧)，图示以太网帧。

最后, 帧被转换为比特, 通过网络介质传输。这种协议栈逐层向下传递数据, 并添加报头和报尾的过程称为封装。

这里所说的以太网帧是IEEE II 格式，如下



前导码：Ethernet II是由8个8'b10101010构成

目的地址：目的设备的MAC物理地址。

源 地址：发送设备的MAC物理地址。

类型(Ethernet II)：以太网首部后面所跟数据包的类型，例如Type为0x8000时为IP协议包，Type为8060时，后面为ARP协议包。

FCS: 就是CRC校验值

以太网帧首部，如下





ARP 数据格式

硬件类型	协议类型
硬件地址长度	协议长度
发送方的硬件地址 (0-3字节)	操作类型
源物理地址 (4-5字节)	源IP地址 (0-1字节)
源IP地址 (2-3字节)	目标硬件地址 (0-1字节)
目标硬件地址 (2-5字节)	
目标IP地址 (0-3字节)	

- (1) 硬件类型字段指明了发送方想知道的硬件接口类型，以太网的值为 1；
 - (2) 协议类型字段指明了发送方提供的高层协议类型，IP 为 0800 (16 进制)；
 - (3) 硬件地址长度和协议长度指明了硬件地址和高层协议地址的长度，这样 ARP 报文就可以在任意硬件和任意协议的网络中使用；
 - (4) 操作字段用来表示这个报文的类型，ARP 请求为 1，ARP 响应为 2，RARP 请求为 3，RARP 响应为 4；
 - (5) 发送方的硬件地址 (0 - 3 字节)：源主机硬件地址的前 3 个字节；
 - (6) 发送方的硬件地址 (4 - 5 字节)：源主机硬件地址的后 3 个字节；
 - (7) 发送方 IP (0 - 1 字节)：源主机硬件地址的前 2 个字节；
 - (8) 发送方 IP (2 - 3 字节)：源主机硬件地址的后 2 个字节；
 - (9) 目的硬件地址 (0 - 1 字节)：目的主机硬件地址的前 2 个字节；
 - (10) 目的硬件地址 (2 - 5 字节)：目的主机硬件地址的后 4 个字节；
 - (11) 目的 IP (0 - 3 字节)：目的主机的 IP 地址。



IP 数据格式



(1) 版本：占第一个字节的高四位，对于 IPv4，该值为 4，对于 IPv6，该值为 6。头长度：占第一个字节的低四位，在不包含任何选项字段的 IP 首部，长度为 $5(5 \times 4 = 20)$ 字节；由于该值最大为 15，最大 IP 首部长度为 60 字节。

(2) 服务类型：前 3 位为优先字段权，现在已经被忽略。接着 4 位用来表示最小延时、最大吞吐量、最高可靠性和最小费用。

(3) 封包总长度：整个 IP 报的长度，单位为字节。

(4) 16位标志字段，3位标志，13位片偏移字段常用于IP数据分片时使用。

(5) 存活时间：就是封包的生存时间。即最多能被转发的次数，通常用通过的路由器的个数来衡量，比如初始值设置为 32，则每通过一个路由器处理就会被减一，当这个值为 0 的时候就会丢掉这个包，并用 ICMP 消息通知源主机。

(6) 协议：用于描述该 IP 数据报中的数据来自哪个上层协议。例如：TCP(0x06)、UDP(0x11)、ICMP(0x01) 和 IGMP(0x02)

(7) 检验和：只针对 IP 首部做校验，内部数据传输的准确性由上层协议负责。

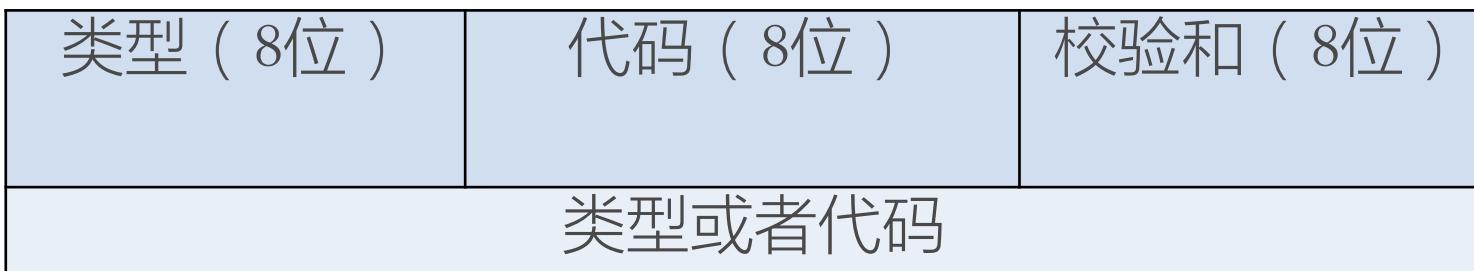
(8) 来源 IP 地址：将 IP 地址看作是 32 位数值则需要将网络字节顺序转化为主机字节顺序。

(9) 目的 IP 地址 : 转换方法和来源 IP 地址一样。

NB：IP 是面向非连接的，所谓的非连接就是传递数据的时候，不检测网络是否连通。所以是不可靠的数据报协议，IP 协议主要负责在主机之间寻址和选择数据包路由。



ICMP 数据格式



- (1) 类型：一个8位类型字段，表示ICMP数据包类型。
 - (2) 代码：一个8位代码域，表示指定类型中的一个功能。如果一个类型中只有一种功能，代码域置为0。
 - (3) 检验和：数据包中ICMP部分上的一个16位检验和。

种类:

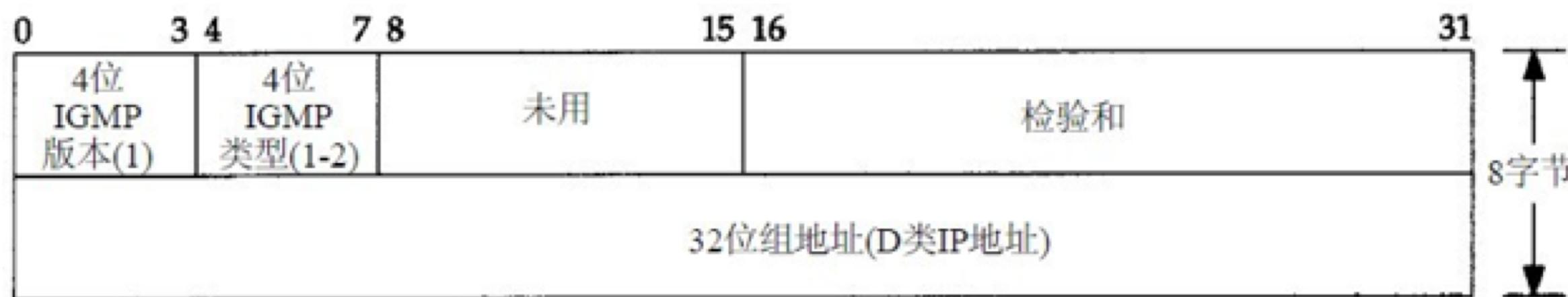
- ICMP 差错报告报文:
 - 终点不可达 3
 - 源点抑制(Source quench) 4
 - 时间超过 11
 - 参数问题 12
 - 改变路由 (重定向) (Redirect) 5
 - ICMP 询问报文:
 - 回送请求和回答报文 8 或 0
 - 时间戳请求和回答报文 13 或 14
 - 路由器询问和通告 10 或 9
 - 信息请求请求或回答报文 15 或 16
 - 地址掩码请求请求或回答报文 17 或 18

注意

- 允许主机和路由器报告差错情况和提供有关异常情况的报告
 - ICMP不是高层协议,而是IP层的协议
 - ICMP报文作为IP层数据报的数据,加上数据报的首部,组成 IP 数据报发送出去
 - ICMP报文的前4个字节是统一的格式,共有三个字段 : 即类型,代码和检验和.接着的4个字节的内容与ICMP的类型有关



IGMP 数据格式



作用：

它是TCP/IP 协议族中负责IP 组播成员管理的协议,用来在IP 主机和与其直接相邻的组播路由器之间建立、维护组播组成员关系

功能：

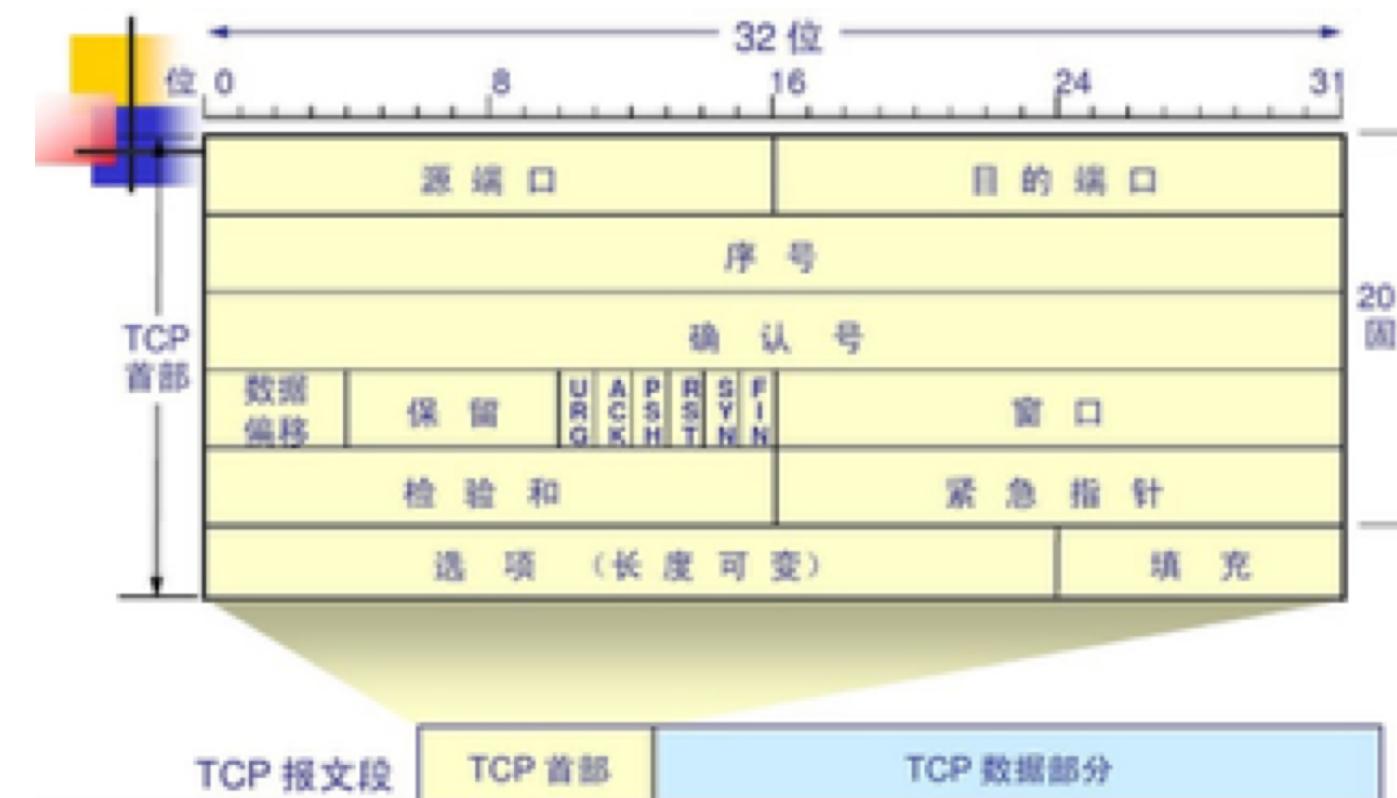
当一台主机加入到一个新的组时,它发送一个IGMP消息到组地址以宣告它的成员身份,多播路由器和交换机就可以从中学习到组的成员.利用从IGMP中获取到的信息,路由器和交换机在每个接口上维护一个多播组成员的列表

两个阶段：

- **加入**：当主机加入新的多播组时,向多播组的多播地址发送IGMP 报文,声明自己要成为该组的成员.本地的多播路由器收到 IGMP 报文后,将组成员关系转发给因特网上的其他多播路由器
 - **询问**：周期性地探询本地局域网上的主机,以便知道这些主机是否还继续是组的成员



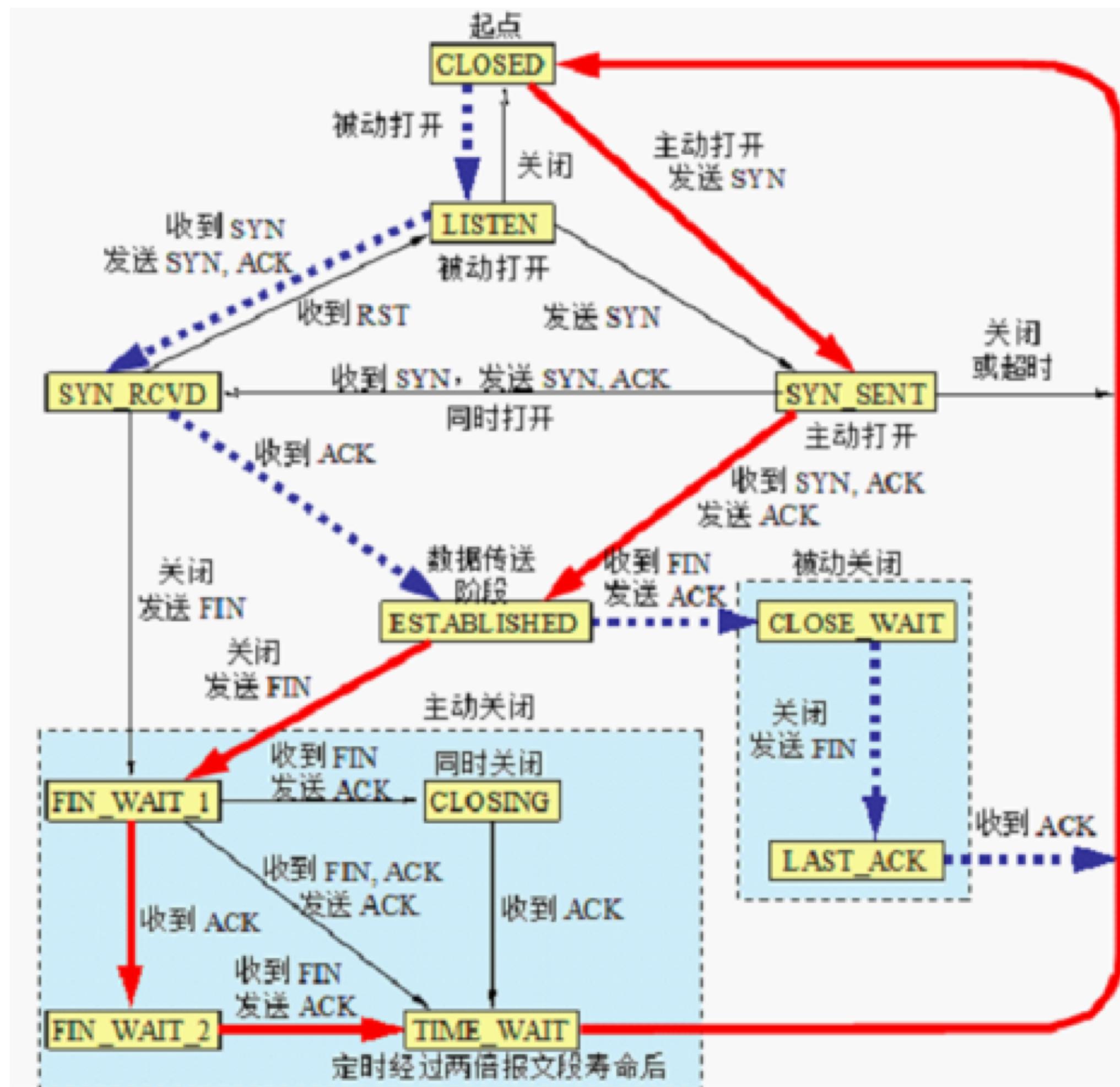
TCP 数据格式



- (1) TCP源端口 (Source Port) : 16 位的源端口包含初始化通信的端口号。源端口和 IP 地址的作用是标识报文的返回地址。
 - (2) TCP目的端口 (Destination Port) : 16 位的目的端口域定义传输的目的。这个端口指明报文接收计算机上的应用程序地址接口。
 - (3) 序列号 (Sequence Number) : TCP 连线发送方向接收方的封包顺序号。
 - (4) 确认序号 (Acknowledge Number) : 接收方回发的应答顺序号。
 - (5) 头长度 (Header Length) : 表示 TCP 头的双四字节数，如果转化为字节个数需要乘以 4。
 - (6) URG : 是否使用紧急指针，0 为不使用，1 为使用。
 - (7) ACK : 请求/应答状态。0 为请求，1 为应答。
 - (8) PSH : 以最快的速度传输数据。
 - (9) RST : 连接复位，首先断开连接，然后重建。
 - (10) SYN : 同步连接序号，用来建立连接。
 - (11) FIN : 结束连接。
 - (12) 窗口大小 (Window) : 目的机使用 16 位的域告诉源主机，它想收到的每个 TCP 数据段大小。
 - (13) 校验和 (Check Sum) : 这个校验和和 IP 的校验和有所不同，不仅对头数据进行校验还对封包内容校验。
 - (14) 紧急指针 (Urgent Pointer) : 当 URG 为 1 的时候才有效。TCP 的紧急方式是发送紧急数据的一种方式。



TCP 状态机

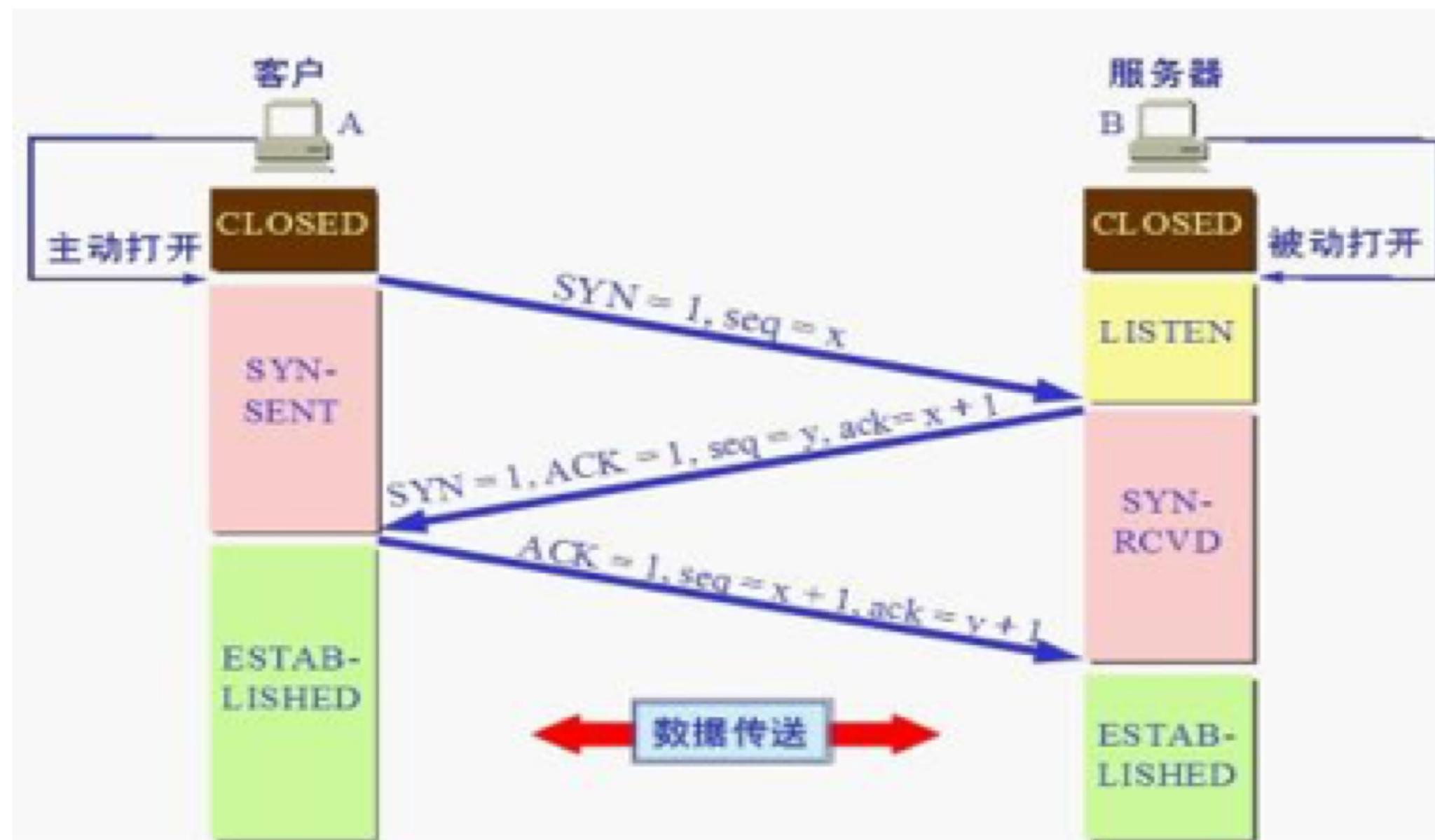


说明

- TCP 有限状态机的图中每一个方框都是 TCP 可能具有的状态
 - 每个方框中的大写英文字符串是 TCP 标准所使用的 TCP 连接状态名.状态之间的箭头表示可能发生的状态变迁
 - 箭头旁边的字,表明引起这种变迁的原因,或表明发生状态变迁后又出现什么动作
 - 图中有三种不同的箭头
 - 粗实线箭头表示对客户进程的正常变迁
 - 粗虚线箭头表示对服务器进程的正常变迁
 - 另一种细线箭头表示异常变迁



TCP 三个阶段



步骤：

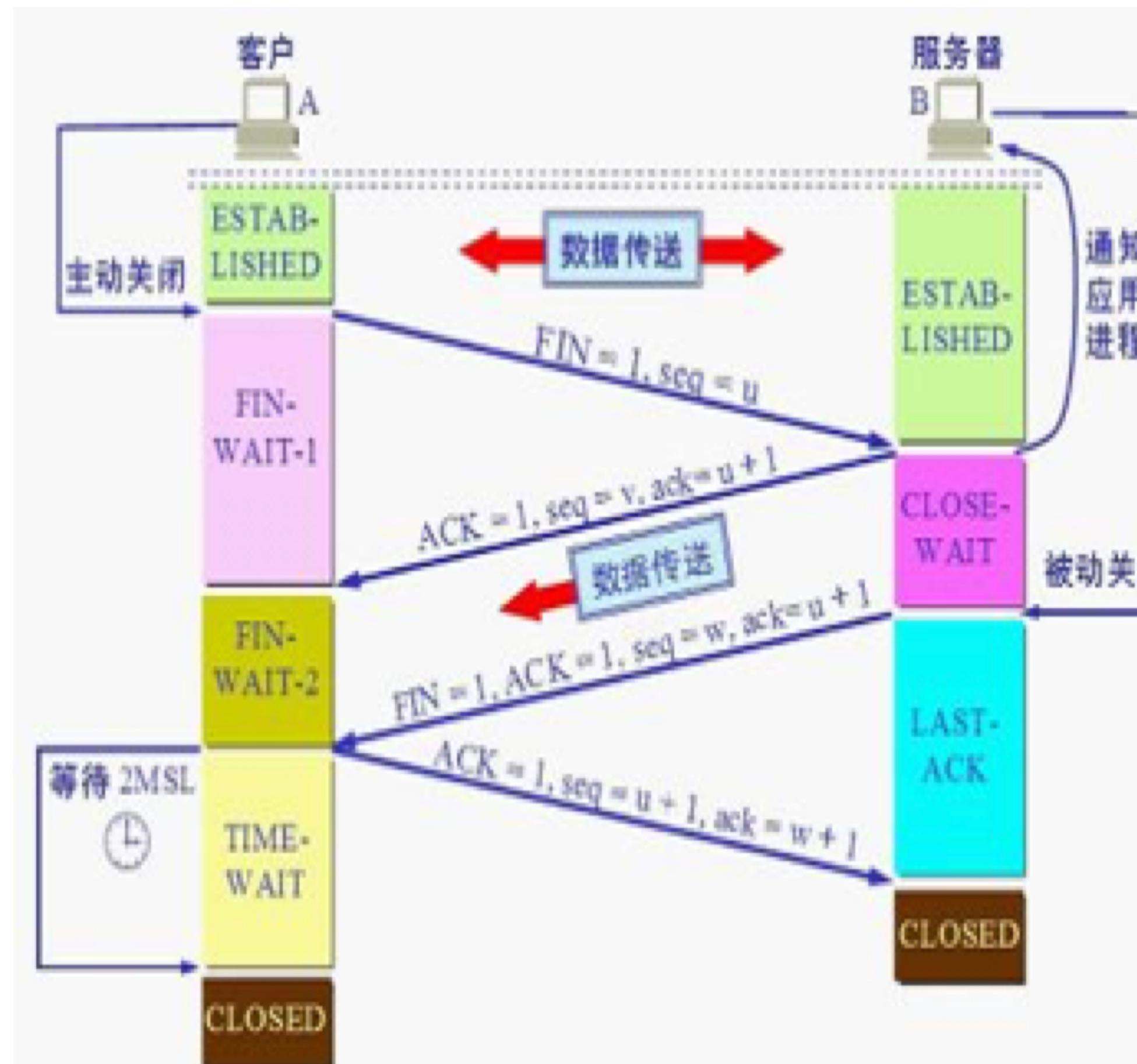
A 的 TCP 向 B 发出连接请求报文段，其首部中的同步位 $SYN = 1$ ，并选择序号 $seq = x$ ，表明传送数据时的第一个数据字节的序号是 x

B 的 TCP 收到连接请求报文段后，如同意，则发回确认（B 在确认报文段中应使 $SYN = 1$ ，使 $ACK = 1$ ，其确认号 $ack = x + 1$ ，自己选择的序号 $seq = y$ ）

A 收到此报文段后向 B 给出确认，其 $ACK = 1$ ，确认号 $ack = y + 1$ （A 的 TCP 通知上层应用进程，连接已经建立，B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立）



TCP 四个阶段



步骤

数据传输结束后，通信的双方都可释放连接。现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接（A 把连接释放报文段首部的 $FIN = 1$ ，其序号 $seq = u$ ，等待 B 的确认）

B 发出确认，确认号 $ack = u + 1$ ，而这个报文段自己的序号 $seq = v$ （TCP 服务器进程通知高层应用进程。从 A 到 B 这个方向的连接就释放了，TCP 连接处于半关闭状态。B 若发送数据，A 仍要接收）

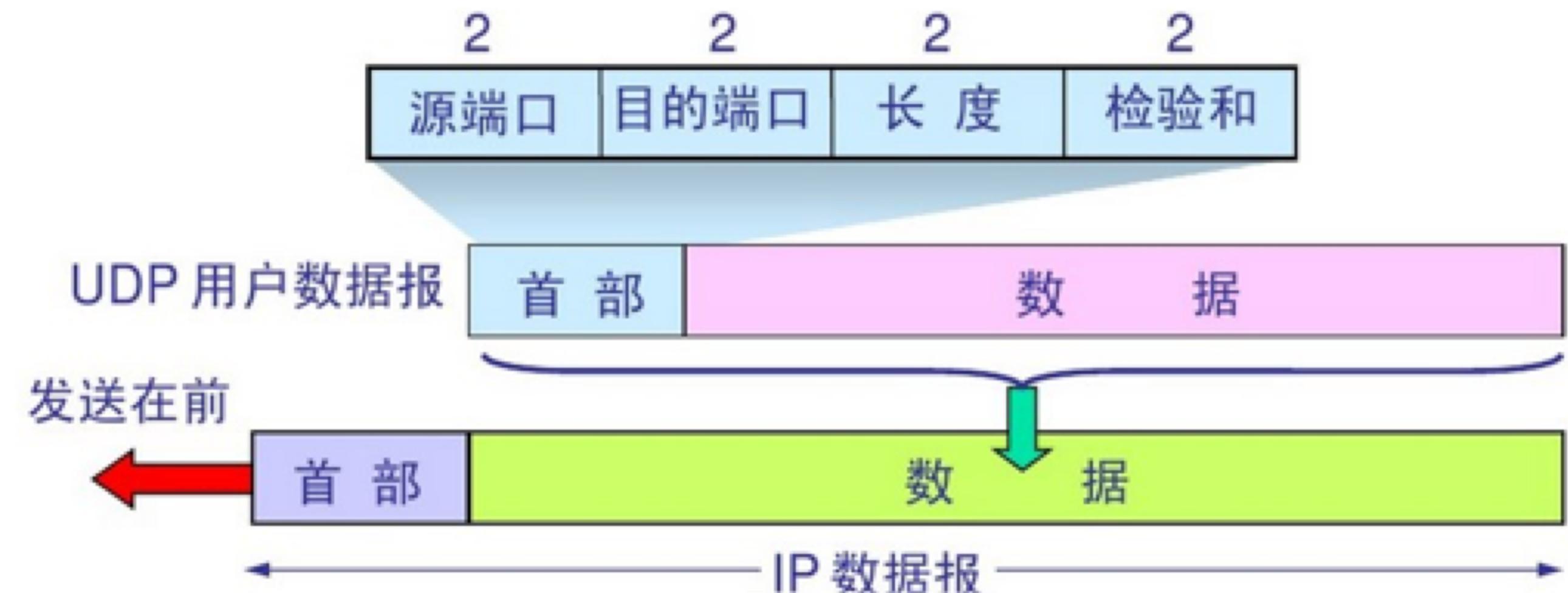
若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接

A 收到连接释放报文段后，必须发出确认，在确认报文段中 $ACK = 1$ ，确认号 $ack=w + 1$ ，自己的序号 $seq = u + 1$

2MSL --- 为了保证 A 发送的最后一个 ACK 报文段能够到达 B，防止“已失效的连接请求报文段”出现在本连接中。



UDP 数据格式



- (1) 源端口 (Source Port) : 16 位的源端口域包含初始化通信的端口号。源端口和IP地址的作用是标识报文的返回地址。
 - (2) 目的端口 (Destination Port) : 6 位的目的端口域定义传输的目的。这个端口指明报文接收计算机上的应用程序地址接口。
 - (3) 封包长度 (Length) : UDP 头和数据的总长度。
 - (4) 校验和 (Check Sum) : 和 TCP 和校验和一样，不仅对头数据进行校验，还对包的内容进行校验。



DHCP 数据格式

操作代码 (1字节)	硬件类型 (1字节)	硬件长度 (1字节)	跳数 (1字节)
事务ID (4字节)			
秒 (2字节)		标志 (2字节)	
客户端IP地址 (4字节)			
你的IP地址 (4字节)			
服务器IP地址 (4字节)			
网关IP地址 (4字节)			
客户端硬件地址 (16字节)			
服务器名 (64字节)			
引导文件名 (128字节)			
选项 (64字节)			

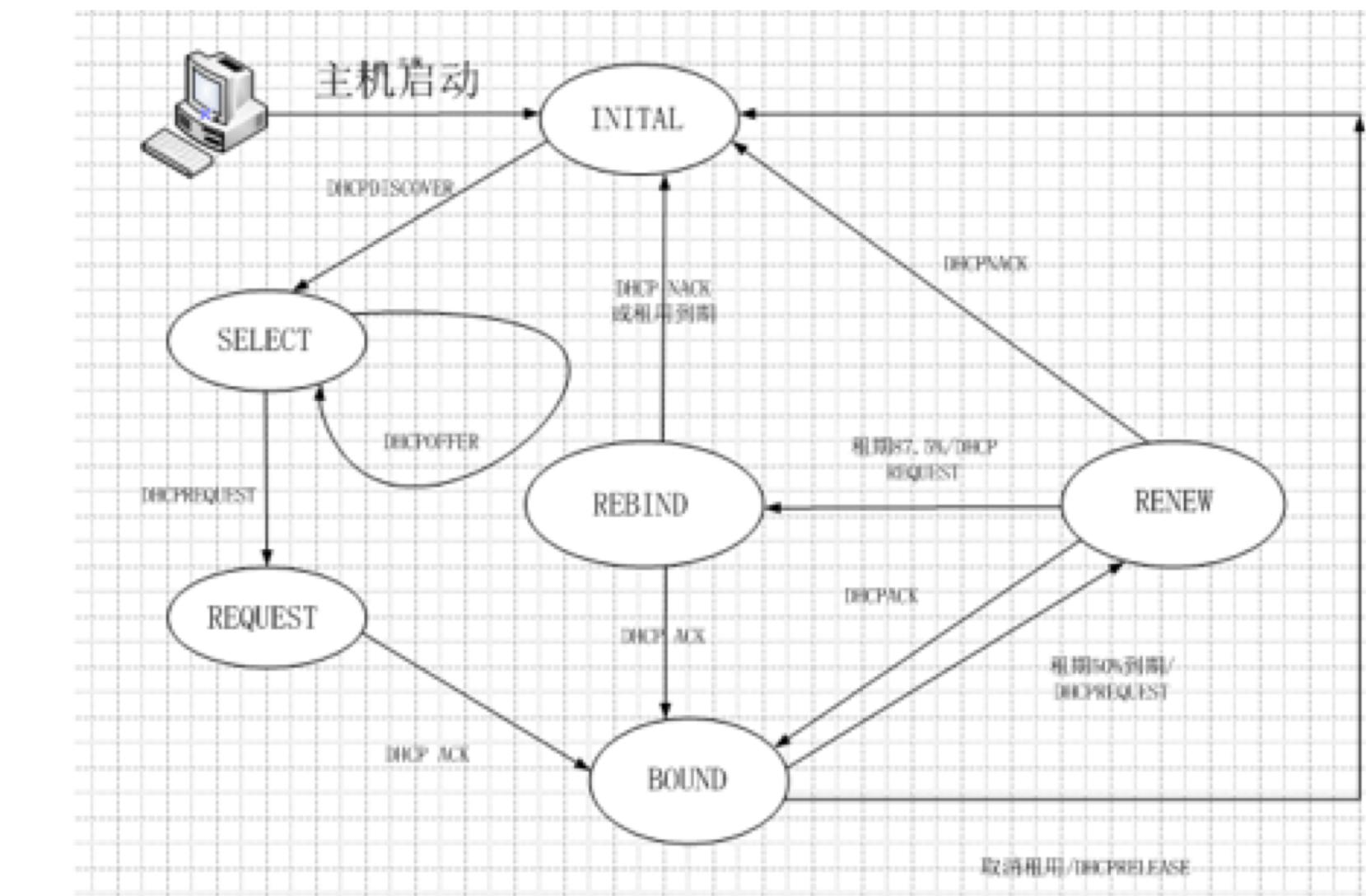
- OP : 若是 client 送给 server 的封包 , 设为 1 , 反向为 2 ;
- Htype : 硬件类别 , ethernet 为 1 ;
- Hlen : 硬件长度 , ethernet 为 6 ;
- Hops : 若数据包需经过 router 传送 , 每站加 1 , 若在同一网内 , 为 0 ;
- Transaction ID : 事务 ID , 是个随机数 , 用于客户和服务器之间匹配请求和相应消息 ;
- Seconds : 由用户指定的时间 , 指开始地址获取和更新进行后的时间 ;
- Flags : 从 0 – 15 bits , 最左 1 bit 为 1 时表示 server 将以广播方式传送封包给 client , 其余尚未使用 ;
- Ciaddr : 用户 IP 地址 ;
- Yiaddr : 客户 IP 地址 ;
- Siaddr : 用于 bootstrap 过程中的 IP 地址 ;
- Giaddr : 转发代理 (网关) IP 地址 ;
- Chaddr : client 的硬件地址 ;
- Sname : 可选 server 的名称 , 以 0x00 结尾 ;
- File : 启动文件名 ;
- Options : 厂商标识 , 可选的参数字段



DHCP 状态机

客户端从 dhcpserver 获取 ip 的过程，主要是在如下 6 种状态：

- 初始状态
 - 选择状态
 - 请求状态
 - 绑定状态
 - 重新获取状态
 - 重新绑定状态中切换



- 1) DHCPDISCOVER (0x01) : Client 开始 DHCP 过程的第一个报文
 - 2) DHCPOFFER (0x02) : Server 对 DHCPDISCOVER 报文的响应
 - 3) DHCPREQUEST (0x03) : Client 开始 DHCP 过程中对 server 的 DHCPOFFER 报文的回应，或者是 Client 续延 IP 地址租期时发出的报文
 - 4) DHCPDECLINE (0x04) : 当 Client 发现 Server 分配给它的 IP 地址无法使用，如 IP 地址冲突时，将发出此报文，通知 Server 禁止使用 IP 地址
 - 5) DHCPACK (0x05) : Server 对 Client 的 DHCPREQUEST 报文的确认响应报文，Client 收到此报文后，才真正获得了 IP 地址和相关的配置信息
 - 6) DHCPNAK (0x06) : Server 对 Client 的 DHCPREQUEST 报文的拒绝响应报文，Client 收到此报文后，会重新开始新的 DHCP 过程
 - 7) DHCPRELEASE (0x07) : Client 主动释放 server 分配给它的 IP 地址的报文，当 Server 收到此报文后，就可以回收这个 IP 地址，能够分配给其他的 Client
 - 8) DHCPINFORM (0x08) : Client 已经获得了 IP 地址，发送此报文，只是为了从 DHCP SERVER 处获取其他的一些网络配置信息，如 route ip , DNS Ip 等，这种报文的应用非常少见



socket 编程

accept
bind
shutdown
getpeername
getsockname
setsockopt
getsockopt
close
connect
listen
recv
recvfrom
send
sendmsg
sendto
socket
select
ioctl
read
write
writev
close
fcntl

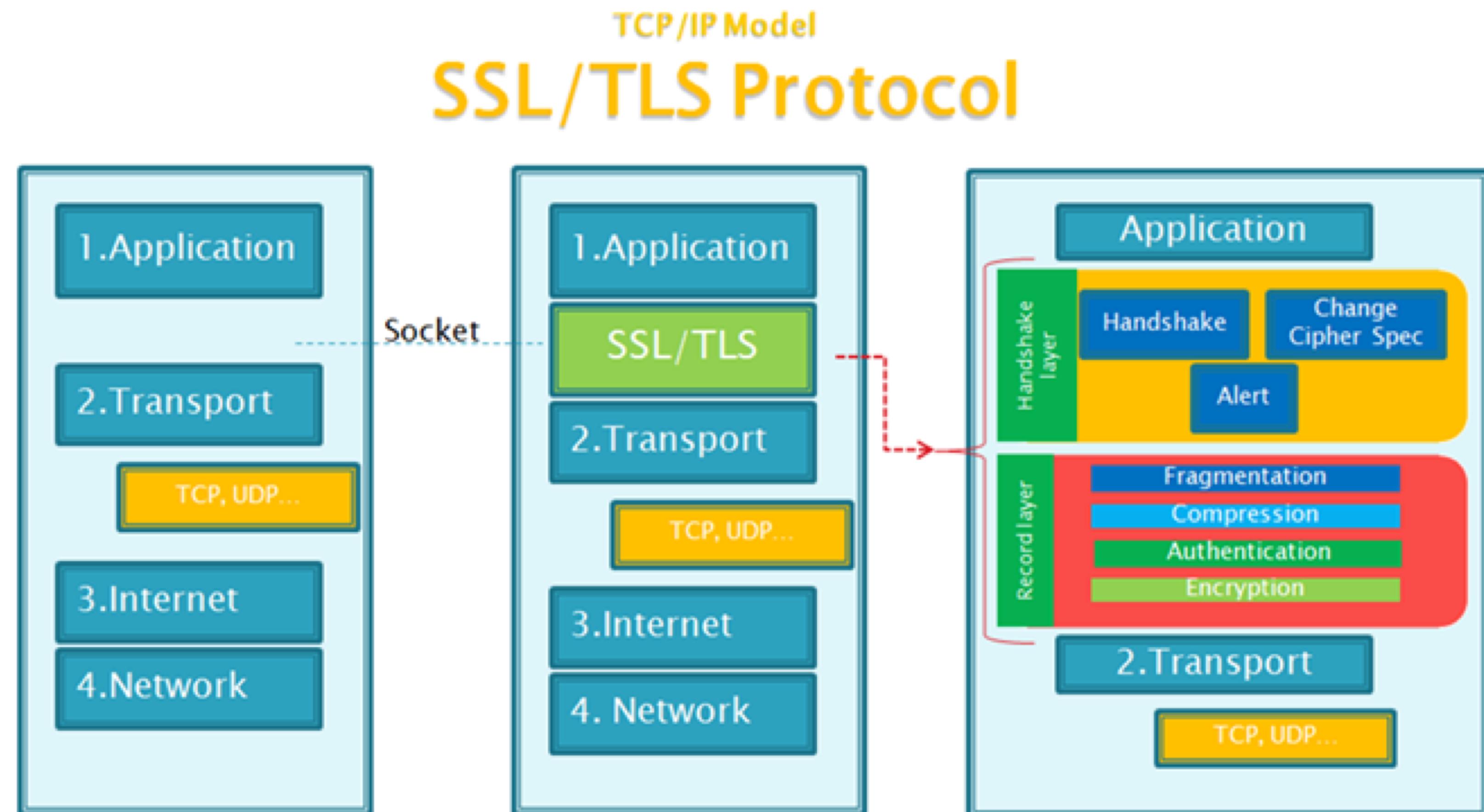
TCP_NODELAY	0x01
TCP_KEEPALIVE	0x02
TCP_KEEPIDLE	0x03
TCP_KEEPINTVL	0x04
TCP_KEEPCNT	0x05

SO_DEBUG	0x0001
SO_ACCEPTCONN	0x0002
SO_DONTROUTE	0x0010
SO_USELOOPBACK	0x0040
SO_LINGER	0x0080
SO_DONTLINGER	((int)(
SO_OOBINLINE	0x0100
SO_REUSEPORT	0x0200
SO_SNDBUF	0x1001
SO_RCVBUF	0x1002
SO SNDLOWAT	0x1003
SO RCVLOWAT	0x1004
SO_SNDTIMEO	0x1005
SO_RCVTIMEO	0x1006
SO_ERROR	0x1007
SO_TYPE	0x1008
SO_CONTIMEO	0x1009
SO_NO_CHECK	0x100a



SSL/TLS

SSL 是一个介于 HTTP 协议与 TCP 之间的一个可选层，其位置大致如下：





SSL/TLS

SSL (Secure Socket Layer , 安全套接字层)

为 Netscape 所研发 , 用以保障在 Internet 上数据传输之安全 , 利用数据加密 (Encryption) 技术 , 可确保数据在网络上之传输过程中不会被截取。当前版本为 3.0 。它已被广泛地用于 Web 浏览器与服务器之间的身份认证和加密数据传输。

SSL 协议位于 TCP/IP 协议与各种应用层协议之间 , 为数据通讯提供安全支持。

SSL 协议可分为两层 :

- SSL 记录协议 (SSL Record Protocol) : 它建立在可靠的传输协议 (如 TCP) 之上 , 为高层协议提供数据封装、压缩、加密等基本功能的支持。
- SSL 握手协议 (SSL Handshake Protocol) : 它建立在 SSL 记录协议之上 , 用于在实际的数据传输开始前 , 通讯双方进行身份认证、协商加密算法、交换加密密钥等。

TLS : (Transport Layer Security , 传输层安全协议)

用于两个应用程序之间提供保密性和数据完整性。

TLS 1.0 是 IETF (Internet Engineering Task Force , Internet 工程任务组) 制定的一种新的协议 , 它建立在 SSL 3.0 协议规范之上 , 是 SSL 3.0 的后续版本 , 可以理解为 SSL 3.1 , 它是写入了 [RFC](#) 的。

TLS 协议由两层组成 : TLS 记录协议 (TLS Record) 和 TLS 握手协议 (TLS Handshake) 。较低的层为 TLS 记录协议 , 位于某个可靠的传输协议 (例如 TCP) 上面。

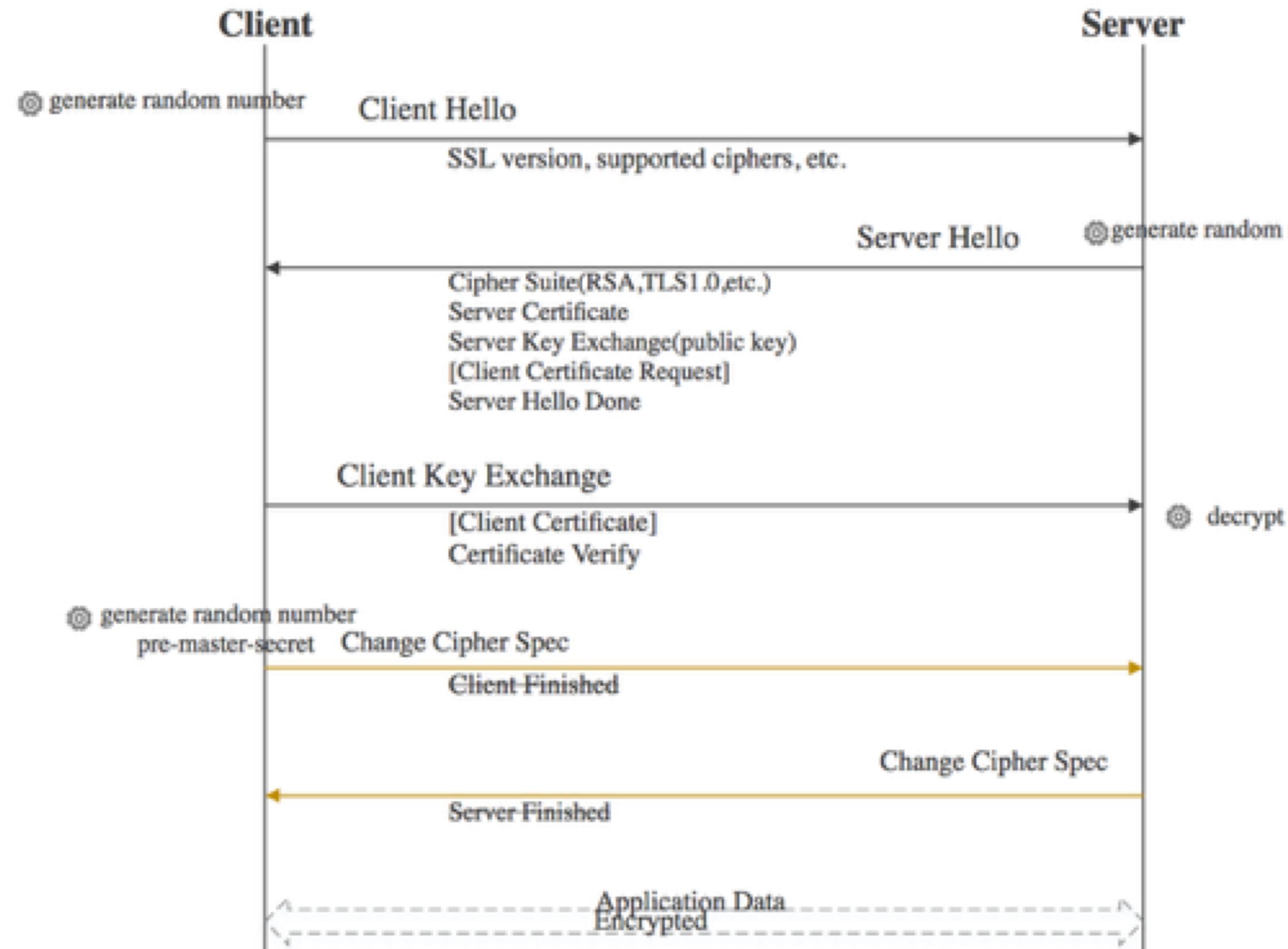


SSL/TLS

1. 版本号：TLS 记录格式与 SSL 记录格式相同，但版本号的值不同，TLS 的版本 1.0 使用的版本号为 SSLv3.1。
 2. 报文鉴别码：SSLv3.0 和 TLS 的 MAC 算法及 MAC 计算的范围不同。TLS 使用了 RFC-2104 定义的 HMAC 算法。SSLv3.0 使用了相似的算法，两者差别在于 SSLv3.0 中，填充字节与密钥之间采用的是连接运算，而 HMAC 算法采用的是异或运算。但是两者的安全程度是相同的。
 3. 伪随机函数：TLS 使用了称为PRF的伪随机函数来将密钥扩展成数据块，是更安全的方式。
 4. 报警代码：TLS 支持几乎所有的 SSLv3.0 报警代码，而且 TLS 还补充定义了很多报警代码，如解密失败 (decryption_failed)、记录溢出 (record_overflow)、未知CA (unknown_ca)、拒绝访问 (access_denied) 等。
 5. 密文族和客户证书：SSLv3.0 和 TLS 存在少量差别，即 TLS 不支持 Fortezza 密钥交换、加密算法和客户证书。
 6. certificate_verify 和 finished 消息：SSLv3.0 和 TLS 在用 certificate_verify 和 finished 消息计算 MD5 和 SHA-1 散列码时，计算的输入有少许差别，但安全性相当。
 7. 加密计算：TLS 与 SSLv3.0 在计算主密值 (master secret) 时采用的方式不同。
 8. 填充：用户数据加密之前需要增加的填充字节。在 SSL 中，填充后的数据长度要达到密文块长度的最小整数倍。而在 TLS 中，填充后的数据长度可以是密文块长度的任意整数倍（但填充的最大长度为 255 字节）。



SSL



SSL 协议分为两部分：

- Handshake Protocol : 用来协商密钥，协议的大部分内容就是通信双方如何利用它来安全的协商出一份密钥。
 - Record Protocol : 定义了传输的格式。

由于非对称加密的速度比较慢，所以它一般用于密钥交换，双方通过公钥算法协商出一份密钥，然后通过对称加密来通信，当然，为了保证数据的完整性，在加密前要先经过HMAC的处理。

SSL缺省只进行server端的认证，客户端的认证是可选的。



SSL

- openssl
- mbedtls
- axTLS



mbedtls 基本使用

数据结构

- `mbedtls_net_context`：目前只有文件描述符，可以用于适配异步 I/O 库
- `mbedtls_ssl_context`：保存 SSL 基本数据
- `mbedtls_ssl_config`
- `mbedtls_ctr_drbg_context`
- `mbedtls_entropy_context`：保存熵配置
- `mbedtls_x509_crt`：保存认证信息

Connect 阶段

- `mbedtls_net_connect()`：参数是 server 和端口，均为字符串。Server 可以使域名或者 IP 字符串。最后一个参数使用 `MBEDTLS_NET_PROTO_TCP` 即可。端口号不仅仅可以传入数字字符串，也可以类似于 `get_addrinfo` 函数的 `protocol` 参数那样，传入类似于“HTTPS”这样的可读化字符串。



阶段

Init 阶段

下面是init阶段需要调用的各函数。函数的参数，在调用的时候按照上面的函数类型一个一个传入就行了

- mbedtls_net_init()
 - mbedtld_ssl_init()
 - mbedtld_ssl_config_init()
 - mbedtls_ctr_drbg_init()
 - mbedtld_x509_crt_init()
 - mbedtls_entropy_init()
 - mbedtls_ctr_drbg_seed()

其中`mebedtls_ctr_drbg_seed()`可以指定熵函数。如果回调使用默认的`mebedtls_entropy_func`的话，可以传入一个初始的熵 seed，也可以 NULL

Handshake 阶段

```
mbedtls_ssl_config_defaults()  
mbedtls_ssl_conf_authmode()  
mbedtls_ssl_conf_ca_chain()  
mbedtls_ssl_conf_rng()  
mbedtls_ssl_set_bio()  
mbedtls_ssl_setup()  
mbedtls_ssl_handshake()
```

Application 阶段

mbedtls_ssl_write()
mbedtls_ssl_read()



openSSL 基本使用

初始化

```
int SSL_Library_init (void);
```

选择会话协议和创建会话环境

目前支持的会话协议包括：`TLSv1.2, TLSv1.1, TLSv1.0, SSLv2, SSLv3, SSLv2/v3`。比如`SSLv2/v3`的client，则函数调用：const `SSL_METHOD *SSLv23_client_method`(void);OpenSSL中的会话环境称为“CTX”，申请函数：`SSL_CTX *SSL_CTX_new (SSL_METHOD *method)`;其中`method`就是会话协议。

设置CTX的属性

验证方式：

```
int SSL_CTX_set_verify (SSL_CTX *ctx,int mode,  
                      int (*verify_callback), int (X509_STROE_CTX *));
```

加载CA证书：

```
SSL_CTX_load_verify_location (SSL_CTX *ctx, const char  
*Cafile, const char *Capath);
```

加载用户私钥：

```
SSL_CTX_use_Private_file (SSL_CTX *ctx, const char *file, int  
type);
```

加载用户证书：

```
SSL_CTX_use_certificate_file (SSL_CTX *ctx, const char *file,  
int type);
```

建立SSL套接字

```
SSL *SSL_new (SSL_CTX *ctx);
```

绑定SSL套接字

```
int SSL_set_fd (SSL *ssl, int fd);
int SSL_set_rfd (SSL *ssl, int fd);      // 只读
int SSL_set_wfd (SSL *ssl, int fd);      // 只写
```

完成SSL握手

```
int SSL_connect (SSL *ssl);      // 用于client  
Int SSL_accept (SSL *ssl);     // 用于client
```

数据传输

```
int SSL_read (SSL *ssl, void *buf, int num);
int SSL_write (SSL *ssl, const void *buf, int num);
```

结束SSL通信

关闭SSL套接字

```
int SSL_shutdown (SSL *ssl);
```

释放SSL套接字

```
void SSL_free (SSL *ssl);
```

释放SSL会话

```
void SSL_CTX_free(SSL_CTX *ctx);
```