xserver 使用说明

by spinlock9

2015 年 6 月 23 日

目　录

第一章： 启动 xserver

1.1 参数说明

在命令行上运行如下命令，能够获得 xserver 支持的全部参数列表：

```
$ ./xserver --help
  2014/06/03 19:05:34 [compile]: 2014-06-03 19:05:16 +0800 by go version ...
  2014/06/03 19:05:34 [version]: 2014-06-03 18:58:15 +0800 @a98c0a1
  Usage:
    -apps="": application names, separated by comma
    -debug=false: send log to stdio
    -heartbeat=60: keep alive message from server, in [1, 60] seconds
    -http="": default http port
    -listen="": RPC listen port
    -manage=500: session management interval, in [100, 10000] milliseconds
    -ncpu=1: maximum number of CPUs, in [1, 1024]
    -parallel=32: number of parallel worker-routins per connection, in [1, 1024]
    -remote="": RPC remote port
    -retrans="500,500,1000,1500,1500,2500,3000,4000,5000,7500,10000,15000":
        retransmission intervals, in [100, 30000] milliseconds
    -rtmfp="1935": rtmfp ports list, for example, '1935,1936,1937'
```

参数名	缺省值	参数说明
-ncpu	1	**并发使用 CPU 个数** 例如：8 核机器 -ncpu=8，24 核机器 -ncpu=16 等；如果考虑到 cache 对多线程程序的影响，应该配合使用 taskset 来启动
-rtmfp	1935	**监听 UDP 端口号列表** 多个端口用逗号分隔，例如：-rtmfp=1935,1936,1937
-parallel	32	**每个 UDP 端口并发 routine 个数** parallel × rtmfp 的值控制在 ncpu 的 2 到 4 倍为好，太大了没有意义
-apps		**接受的 app 名称列表** 多个 app 用逗号分隔，例如：-apps=introduction,test

参数名	缺省值	参数说明
-http		**状态监控端口** 建议打开，通过该端口可以监控程序运行状态和调试，例如：-http=6000
-listen		**RPC 监听端口** 例如：-listen=3000，表示监听 3000 端口
-remote		**RPC 发送目标 ip:port** 例如：-remote=127.0.0.1:2000，表示将所有 RPC 消息发送到对应服务器，如果服务器不存在或者连接异常，数据将会被丢弃
-retrans	备注[1]	**丢包重传延迟列表 (毫秒)** 参数指定了丢包重传的时间间隔，当重传次数超过参数个数时将按照最后一个参数指定的间隔进行重传；合法区间为 [100ms,30000ms]
-manage	500	**重传策略扫描间隔 (毫秒)** 服务器以 manage 为周期扫描所有 session 的发送队列，根据 retrans 指定的重传策略决定数据包是否重传；合法区间为 [100ms,10000ms]
-heartbeat	60	**超时检测间隔 (秒)** 服务器超过 6× heartbeat(s) 未收到客户端任何数据时，就会认定连接丢失，在此之前服务器会通过发送心跳包来挽救；合法区间 [1s,60s]
-debug	false	**调试信息输出到 stdout** 该参数的作用是将打开的 debug 信息平行输出到 stdout，仅仅方便开发过程的调试

表 1.1: xserver 参数说明

1.1.1 重传策略

xserver 的重传策略是在旧 xserver 的基础上改进而来，2 点具体如下：

A. 程序以 manage 为周期扫描每一个 NetStream，找到序列号最大的未确认的数据包（旧的实现是找到所有未确认的数据包），如果该数据包满足 retrans 所约定的重传间隔，则对该数据包进行重传；
B. 客户端收到数据包会给服务器发送 ack，其中包含了客户端的缺包的信息，服务端根据这些信息找到对应的数据包，如果数据包距离上次发送时间间隔超过 100ms，则对该数据包进行冲传；

这个改进（即只重传最后一个，而不是全部未确认的数据包）的好处是，能够做到在最小数据重传的前提下，尽快恢复连接的稳定性，同时获得更低的带宽浪费和 CPU 开销。尤其是针对 P2P 过程这种大连接数应用场景，效果会更加明显。

但是缺点是，如果同一个 NetStream 丢包数量不止 1 个，那么服务器至少要 2 次才能将所有未确认的数据包再次发送给前端，特别是如果客户端发送给服务器的 ack 也存在一定概

[1] -retrans=500,500,1000,1500,1500,2500,3000,4000,5000,7500,10000,15000

率的丢包的话，这种策略消耗的时间会更长。不过通过合理的设置 retrans 值，能够将这种情况带来的影响降到最小。

下面给出一个例子：服务端未确认的数据包序号为 {23,24,25}。旧 xserver 策略下（图1.1），每次重传过程都会将三个数据包同时重传，直到收到客户端的确认为止；新 xserver 策略下（图1.2），每次只会重传序号最大的数据包，这样服务端能够从客户端的 ack 中获取尚未确认的数据包序号，并由此驱动进一步重传。

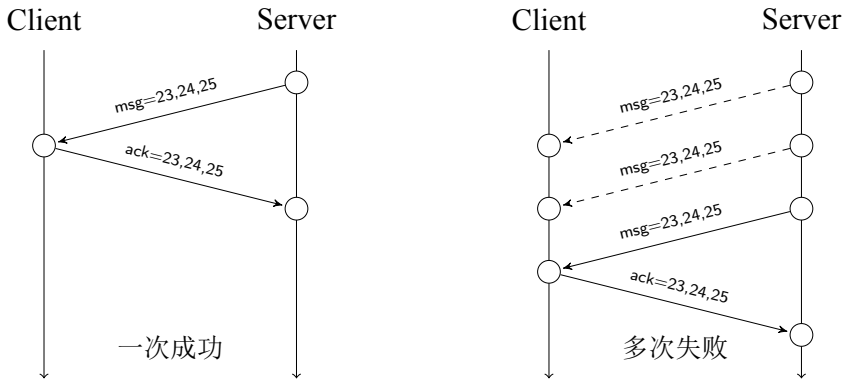

图 1.1: 旧 xserver 重传策略示意图

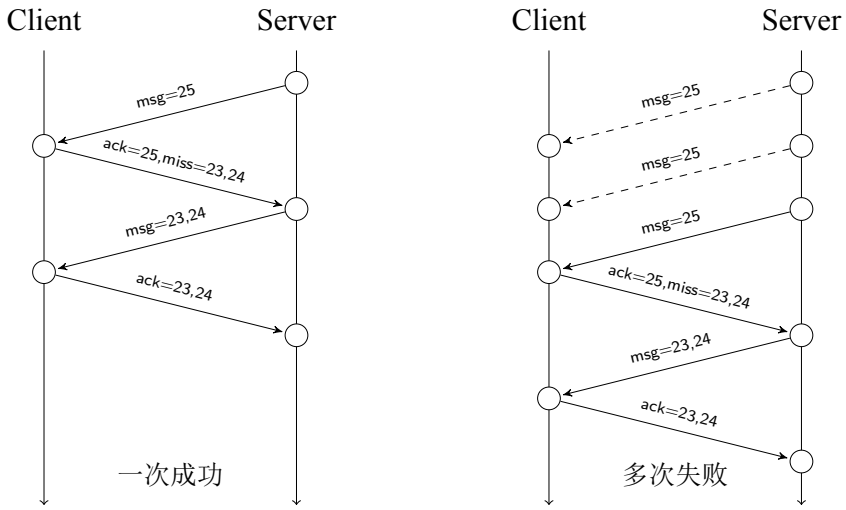

图 1.2: 新 xserver 重传策略示意图

重传策略的选择上，主要受 retrans,manage,heartbeat 这三个参数影响。这些参数会影响服务器的性能以及连接的稳定性。

-retrans

该参数决定了两次数据重传之间的时间间隔。以 -retrans=300,500,10000 为例，重传状态一共有三个，分别为 {0,1,2}。其中，状态转移 A,B 是服务器主动完成的（以 manage 规定的周期进行扫描并判断）；状态转移 C 受客户端发送的 ack 驱动。具体的描述如下：

A. 距离上一次重传时间满足重传间隔，重传数据包并则增加状态序号；
B. 距离上一次重传时间小于重传间隔，或者没有需要重传的数据包，什么都不做；
C. 收到来自客户端的 ack 消息，对缺失的并且发送间隔至少 100ms 的数据包重传，并重置重传状态计数器；

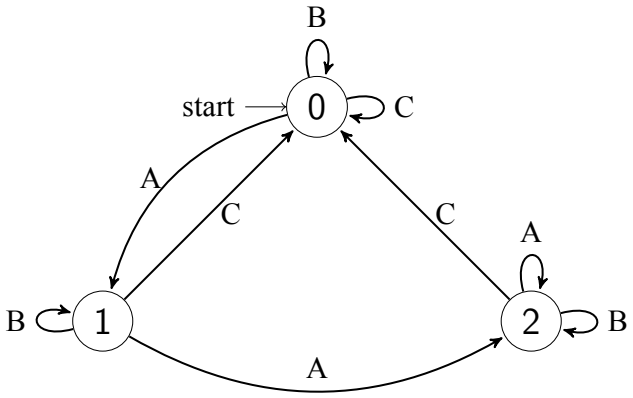

图 1.3: retrans 状态转移过程

除此之外，设置 retrans 的过程还需要考虑的是，客户端的丢包重传策略是 2 分钟内进行大概 20 次，与 retrans 的默认值近似。

-manage

该参数影响的是服务器对丢包扫描的频率。

如果 manage 的值大于 retans 的最小值，那么后者实际上是失效的；反之，如果 manage 过小，那么 retrans 又会起作用，大部分时间服务线程都在进行无效的扫描。所以一个合理的值应该略小于 retrans 最小值。

对于 P2P 的大连接数过程，连接数比数据到达的时间更重要，所以通常 manage 设置的会比较大，比如 1 秒，以保证更大的吞吐；但是对于 RPC 过程，数据的到达时间更重要，因此 manage 会设置的比较小。

-heartbeat

当服务器发现长时间没有收到来自客户端的数据包时，服务器会主动发送一个心跳包来探测客户端状态。这个时间间隔由 heartbeat 参数决定。

服务器会对客户端进行 6 次这样的试探，如果连续 6 次试探都没有返回的话，服务器则认为该客户端连接丢失，随即会补发一个主动断开连接的数据包。也就是说，-heartbeat=5 意味着：如果客户端丢失，服务器一定能在 30 秒内作出响应。

1.1.2 状态监控

设置 http 参数能够打开 xserver 的状态监控接口，以服务器上 xserver.test 运行的，参数 -http=6000 的服务为例：

xserver 状态

下图中提供了查询 xserver 基本运行状态的 url，可以通过 **xids** 和 **cookies** 两个值计算出当前服务器的进入速度，从而监控服务器压力。

```
$ curl http://xserver.test:6000/summary
  {
      "build": {                        # 程序编译时间以及版本控制信息
          "compile": "2014-06-03 18:58:18 +0800 by go version go1.2.2 linux/amd64",
          "version": "2014-06-03 18:58:15 +0800 @a98c0a1"
      },
      "cookies": 8,                     # handshake 过程未过期的 cookie 数量（5 分钟过期）
      "counts": {                       # 内部事件计数器（每分钟更新）
          "cookie.commit": 134,
          "cookie.new": 196,
          "cookie.notfound": 12,
          "cookie.timeout": 63,
            ... ...
      },
      "heap": {                         # 参见 godoc runtime/MemStats（单位 MB）
          "alloc": 881,                 # - 分配并使用的内存
          "idle": 602,                  # - idle span
          "inuse": 982,                 # - non-idle span
          "objects": 0,                 # - 分别的 object 个数
          "sys": 1585                   # - 从系统申请的内存
      },
      "runtime": {
          "nproc": 16,                  # 使用 CPU 个数
          "routines": 397               # 全部 routine 个数
      },
      "session": {
          "aids": 2277,                 # 用 ip:port 区分的 session 数（可能冲突）
          "pids": 2277,                 # 由 sha256 计算生成 pid 区分的 session 数（可能冲突）
          "xids": 2277,                 # 分配的 unique 的 xid 数（不会冲突）
          "z": {
              "closed": 0,              # 标记为 closed 并等待清理的 session 数
              "manage": [
                  1820,                 # 发送心跳数为 0 的 session 数（不严格准确）
                  67,                   # 发送心跳数为 1 的 session 数（不严格准确）
                  86,                   # 发送心跳数为 2 的 session 数（不严格准确）
                  89,                   # 发送心跳数为 3 的 session 数（不严格准确）
```

```
                75,                 # 发送心跳数为 4 的 session 数（不严格准确）
                70,                 # 发送心跳数为 5 的 session 数（不严格准确）
                70                  # 发送心跳数为 6 的 session 数（不严格准确）
            ]
        }
    },
    "streams": 0,                   # 系统中存在的 stream 个数
    "time": {
        "boot": 1401793822,         # 服务启动时间（单位秒）
        "current": 1402039273       # 机器当前时间（单位秒）
    }
}
```

hashmap 信息

注意：该接口对程序有一定压力，不建议频繁使用。

xserver 的最初设计需求是单一程序 30 万左右的连接数。所以实现中要考虑到高并发情况下的 hash 表的效率：

A. 第一层采用固定桶数的 hash，来减小锁的力度；
B. 第二层采用内置 hashmap 加 RWLock 的实现来提高并发行；

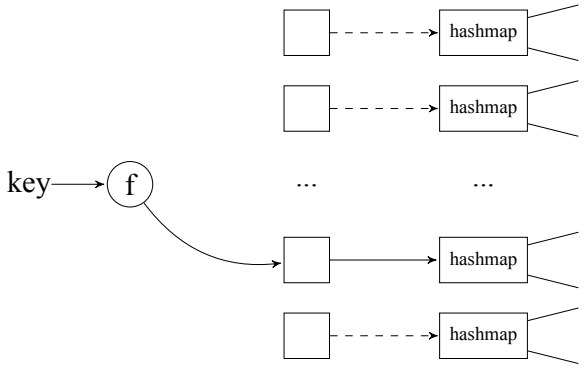

图 1.4: hash 表优化

```
$ curl http://xserver.test:6000/mapsize
{
  "aids":[108,90,96,98,87,90,98,87,...,101,119,101,71,93,101,90,93,111],
  "pids":[97,101,100,91,90,101,90,85,...,104,83,89,110,96,89,101,83,96,92],
  "xids":[85,82,98,98,101,102,80,99,...,97,88,83,102,109,93,84,107,79,90]
}
```

其中，三个数组分别表示按照不同意义组织的 hash 表第二层的大小。分布越均匀越好。

session 信息

注意：该接口对程序压力巨大，当服务器连接数很大是不要使用。

```
$ curl http://xserver.test:6000/dumpall
  [
      {
          "addrs": [                                    # 客户端内网地址列表
              {
                  "IP": "192.168.1.104",
                  "Port": 2473,
                  "Zone": ""
              }
          ],
          "aid": "221.235.22.194:10680",
          "closed": false,                              # 连接是否已关闭
          "manage": {
              "cnt": 0,                                 # 服务端 heartbeat 状态
              "lasttime": 1402117152384136280           # 收到最后一条消息的时间戳
          },
          "pid": "a248afcb8b4196c5756202bd1cba9af4b0eb76f39b1eac5ab675ad0490c5d4ca",
          "raddr": "221.235.22.194:10680",              # 客户端外网地址
          "xid": 1,                                     # session 在服务端 id
          "yid": 33554432                               # session 在客户端 id
      }
     ... ...
```

stream 信息

xserver 也提供了接口来获取全部 stream 信息，因为线上通常不用它来做客户端之间的直接消息通信，所以返回结果一般为空。

```
$ curl http://xserver.test:6000/streams
  {}
```

1.1.3 日志监控

xserver 在运行过程中会监控当前目录下的特殊文件，如果文件存在，则向文件中追加调试日志；如果文件不存在，对应的日志服务则会关闭。对照如下：

a. log/outlog 记录各种状态调试信息，由其会打印大量加密前的 rtmfp 协议包；
b. log/errlog 记录运行时的各种错误，例如 rtmfp 数据包错误等等；
c. log/ssslog 记录 session 建立、关闭以及强制关闭等信息；

d. log/tcplog 记录与后端 RPC 通信过程中的发送、接受以及主动抛弃的数据包；

在线下调试过程中，配合 -debug 参数，能够将这些日志信息打印到 stdout 中，方便调试。

1.2 典型参数

注意：多线程程序不是 CPU 越多越好，因为线程间同步以及 cache 一致性的开销也会变大。例如在 32 核机器上运行 8 个相同实例，建议使用 taskset 启动：将全部 CPU 分成 8 组，并确保每组 CPU 在同一个 node/socket 上。以减少竞争，增加 cache 局部性。

1.2.1 游戏参数

```
$ nohup ./xserver -ncpu=4 -parallel=32 -rtmfp=2000 -http=5000 \
    -listen=3000 -remote=127.0.0.1:4000 \
    -heartbeat=5 -manage=200 \
    -retrans=200,300,500,700,1000,1500,2000,4000,5000,7500,10000,15000 \
    -apps=introduction,test &
```

1.2.2 P2P 参数

```
$ nohup ./xserver -ncpu=16 -parallel=32 -rtmfp=2000 -http=5000 \
    -apps=introduction,test &
```

1.2.3 调试过程

```
$ nohup ./xserver -rtmfp=2000 -http=5000 \
    -listen=3000 -remote=127.0.0.1:4000 \
    -apps=introduction,test -debug=true &
```

第二章： Java 通信接口

2.1 消息格式

xserver 与 java 之间的 RPC 通信建立在 tcp 之上，并通过 protobuf 编码的。结构如图：

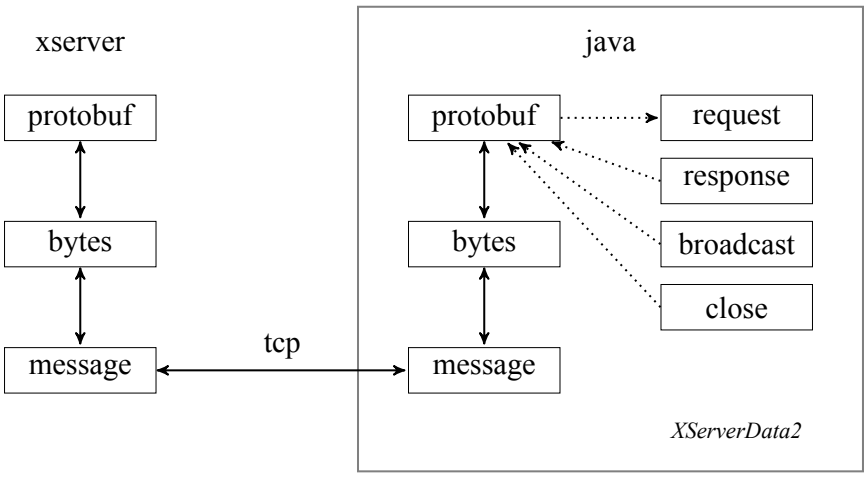

图 2.1: RPC 过程数据包处理示意图

tcp 层数据包格式

在 tcp 连接上传输的数据包由三部分组成，以发送序列化后长度为 13565=0x34fd 个字节的 protobuf 数据包为例：

a. 先是 4 字节魔数，值为 0xdeadbeaf；
b. 再是 4 字节数据包长度，本例为 0x34fd；
c. 最后是将全部数据包内容；

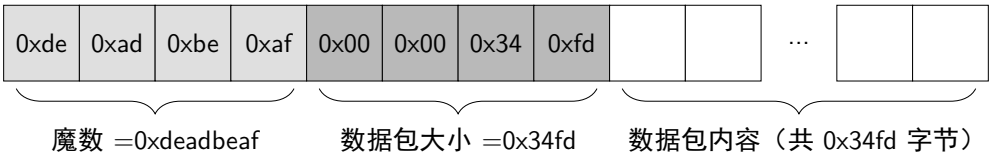

图 2.2: RPC 过程数据包格式

2.2 消息类型

2.2.1 request 消息

java 程序从 xserver 收到的消息格式定义为 request，所有字段含义如下：

```
public class XServerData {
    public static class Request {
        public int port;            # xserver listen 端口号
        public String code;         # request 过程操作类型：只有 {join,exit,call} 三种取值
        public int xid;             # 对应的 session id
        public String addr;         # 客户端外网地址
        public boolean reliable;    # 消息是否为 reliable
        public double callback;     # 客户端 callback
        public Amf0Input input;     # request 调用内容
    }
```

使用时只需要阻塞的调用如下静态函数即可，具体实现请参见源代码：

```
    public static Request parseRequest(InputStream is) throws Exception;
```

join 消息

当客户端与 xserver 建立连接，无论有无进一步数据通信，xserver 都会向后端 RPC 服务发送 “join” 消息。则此时 request 只有如下四个字段有意义：

```
    public static class Request {
        public int port;
        public String code = "join";
        public int xid;
        public String addr;
    }
```

exit 消息

当客户端与 xserver 断开连接，包括客户端主动断开或者被 xserver 被动踢掉，xserver 都会向后端 RPC 服务发送 “exit” 消息。

```
public static class Request {
    public int port;
    public String code = "exit";
    public int xid;
    public String addr;
}
```

call 消息

该过程是真正的客户端到后端服务的 RPC 过程。过程中全部字段都有意义：

```
public static class Request {
    public int port;
    public String code = "call";
    public int xid;
    public String addr;
    public boolean reliable;    # false 表示数据由 proxySend2 发出；否则为 true
    public double callback;     # ! = 0 表示客户端有有效 callback handler；否则为 0
    public Amf0Input input;     # ! = null 表示有有效 content 数据；否则为 null
}
```

2.2.2 response 消息

当服务端需要调用客户端的 callback handler 时，例如：服务端收到 callback!=0 的 reqeust，并完成服务，现在将要返回调用结果给客户端时，需要通过 response 接口来完成：

```
public static byte[] newResponse(int xid, byte[] data, double callback, boolean
    reliable) throws Exception;
```

该函数生成完整的 response 消息：魔数 + 消息大小 + 消息内容，参见图2.2。例如：

```
OutputStream os = socket.getOutputStream();
os.write(XServerData2.newResponse(xid, data, callback, reliable));
os.flush();
```

此外额外的参数说明如下：

a. data 表示发送的数据，例如序列化后的 amf 数据；可以为空，即 null

b. callback 表示客户端 callback handler，通常来自 request
c. reliable 表示消息是否由 xserver 保证消息可靠，如果为 false，消息可能会由于网络问题造成丢失

注意：reliable 和 unreliable 消息不要频繁交替发送，否则客户端可能会由于之前 reliable 消息的丢包重传造成 unreliable 消息的 deliver 延迟。

2.2.3 broadcast 消息

当服务器需要向单个或者多个客户端广播消息时，需要使用 broadcast 消息接口：

```
public static byte[] newBroadcastMesssage(List<Integer> xids, byte[] data,
    boolean reliable) throws Exception;
```

使用方法以及注意事项与 response 过程相同。不同的是，该过程消息会被 deliver 到 NetConnection 的回调函数上，如果是 reliable 消息，回调函数为 recvPull，否则为 recvPull2。

2.2.4 close 消息

当后端 java 主动断开客户端的连接时可以使用该接口：

```
public static byte[] newCloseMessage(List<Integer> xids) throws Exception;
```

xserver 收到该函数请求后，会主动断开与客户端的连接。并想后端 java 发送 code 为 “exit” 的 request 消息。

第 三 章： RTMFP 协议

xserver 仅仅实现了与 P2P 和 RPC 有关的最小协议集合。按照实现先后，主要包括：握手、Session 数据以及 P2P 三个过程。

3.1 基本数据包

数据包大小

RTMFP 协议基于 UDP，通信过程中，服务器要保证每个 UDP 包大小不超过 1400 字节。

生成密钥过程

RTMFP 通信过程中，数据是通过 AES128 加密的，因此在建立连接的最初阶段，客户端和服务去通过 DH 加密算法协商产生两组密钥，分别用于两个方向上的通信。

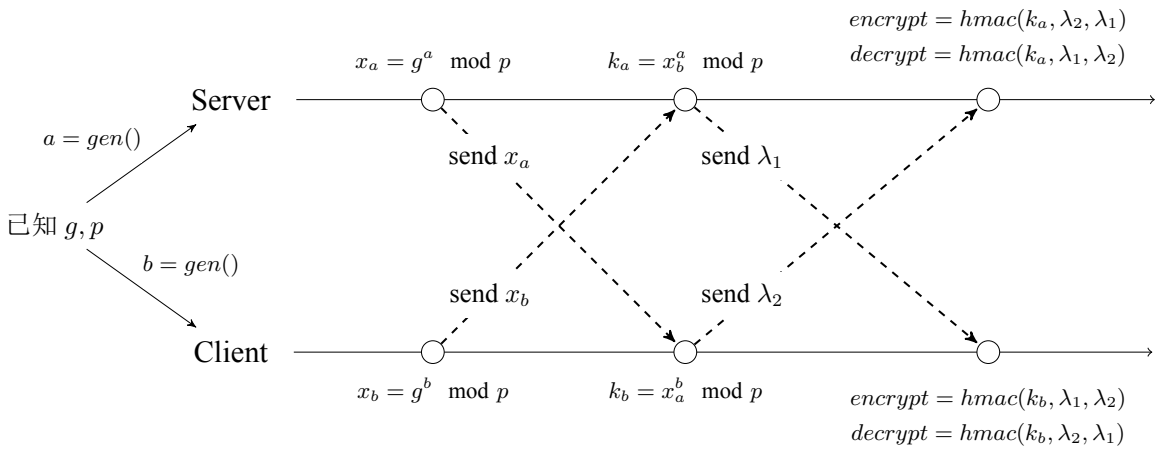

图 3.1: 密钥生成过程

数据包格式

协议中，客户端与服务端通信是无链接的，因此通信过程中，需要在数据包中保存 session 有关的信息，如图所示：

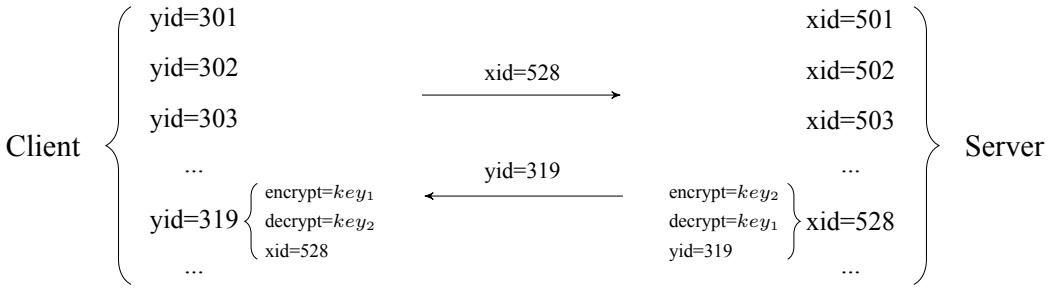

图 3.2: 对称的通信过程

简单描述数据包生成和接收过程如下：

a. Client 与 Server 之间都维护 session 信息的映射关系；
b. 生成数据包时，使用加密密钥进行加密，并将对应的 id 写入数据包；
c. 读取数据包时，先通过 id 找到正确的 session 信息，用解密密钥进行解密后才能的到真正的数据；

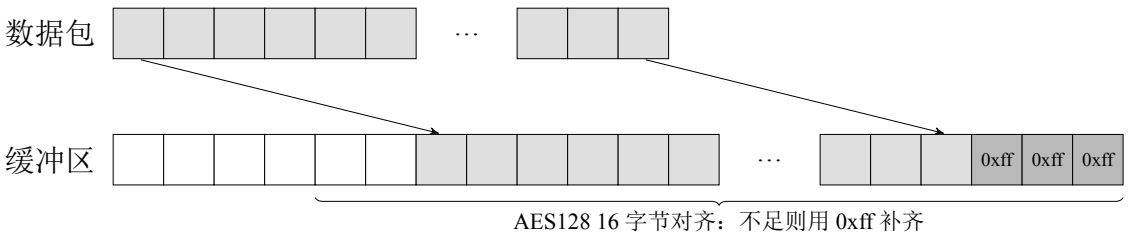

图 3.3: 数据包生成：构建缓冲区

构建缓冲区过程（图3.3）：分配缓冲区时，要空出前 6 个字节，然后将要发送的数据拷贝到缓冲区；同时保证待加密数据长度满足 16 字节对齐，长度不足则用 0xff 补全。

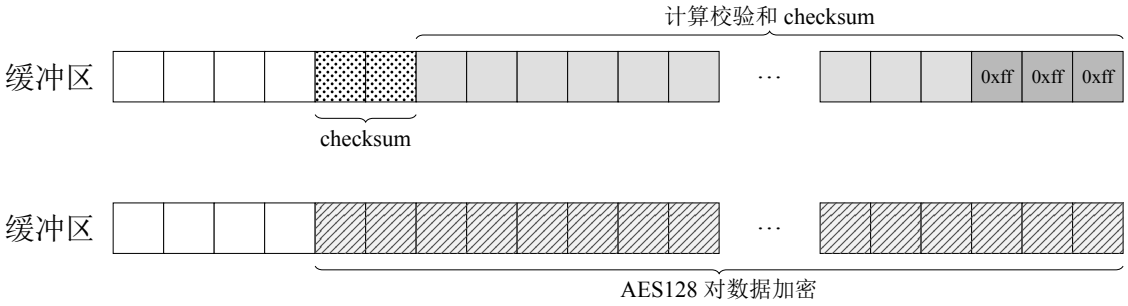

图 3.4: 数据包生成：数据包加密

数据包加密过程（图3.4）：先计算全部数据的 16 位校验和，并将其写入到缓冲区第 4、5 字节位置；然后对包括校验、数据在内的待加密数据进行 AES128 加密。

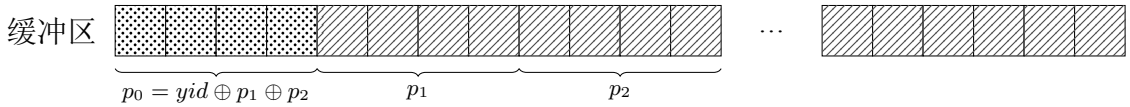

图 3.5: 数据包生成：写入 session 信息

写入 session 信息过程（图3.5）：假设该过程是服务端 xid 向客户端 yid 发送数据包的过程，那么需要将服务端 yid 写入到数据包中，以便客户端收到该数据包能够找到对应的解密密钥等各种 session 信息。但是 yid 不是直接写到缓冲区的，而是通过 yid 与加密后的缓冲区的第 2、3 个 32 位整数进行异或操作，然后将的到的结果写入到缓冲区的前 4 个字节。

数据片格式

一个 RTMFP 数据包是由多个数据片直接拼接而成，其中每个数据片格式如下：

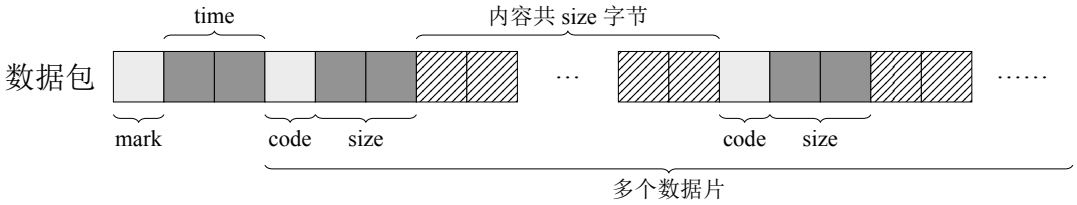

图 3.6: 数据片格式

其中 code≠0xff，因此当数据片提取过程中遇到 code 为 0xff 时，即可认为遇到数据包结束。这种设计主要避免 AES128 加密过程中的缓冲区补全操作引入的额外字节。

3.2 握手过程

如图3.5所示，每一个通信数据包都包含了一组 session 信息。因此协议选择 id=0 作为握手过程，并用默认密钥进行加解密操作。

协议握手过程一共分成 4 个步骤：

A. 客户端 -服务端：客户端向服务端发起建立连接请求，服务端对请求进行验证；
B. 服务端 -客户端：如果服务端接受请求，则返回动态生成的 cookie 进行响应；
C. 客户端 -服务端：客户端开始 DH 算法，发送 $\{x_b, \lambda_2$,yid≠0$\}$ 给服务端；
D. 服务端 -客户端：服务端响应 DH 算法，返回 $\{x_a, \lambda_1$,xid≠0$\}$ 给客户端；

握手过程每个数据包只包含一个数据片。在此过程之后，服务端和客户端都完成了 AES128 密钥的生成和计算以及 session 的建立，之后所有的通信过程都将在 session 的基础上完成。

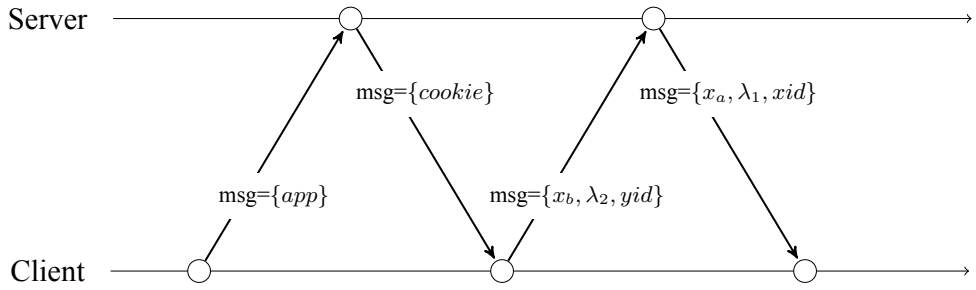

图 3.7: 握手过程

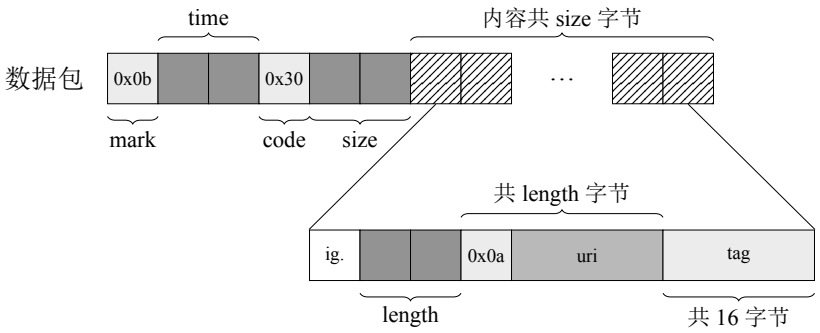

图 3.8: 握手过程：过程 A 数据包

过程 A 中（图3.8）：数据片中 uri 包含了客户端访问的 app 名称，服务端对其进行权限验证；由于此时客户端与服务端都没有创建 session 有关的数据结构，所以此时客户端生成临时 tag 作为标识，这与过程 B 中的 cookie 在服务端的作用一样。

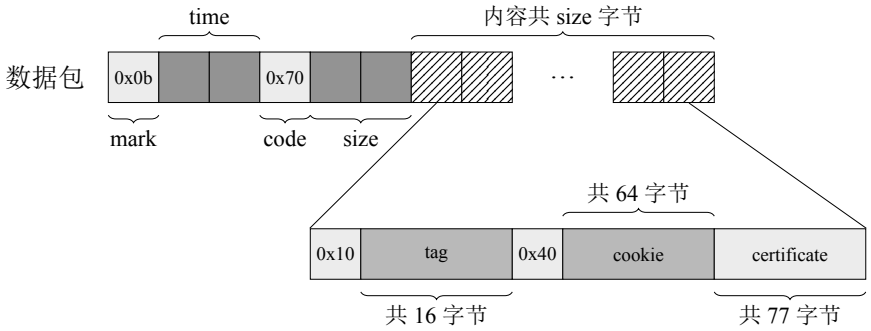

图 3.9: 握手过程：过程 B 数据包

过程 B 中（图3.9）：服务器生成响应过程 A 的数据包，其中包含客户端发来的标识符 tag、服务端生成的 cookie 以及服务端证书 certificate。

客户端收到该数据包之后，会判断 tag 的合法性。如果合法，那么客户端会做好进入过程 C 的准备：创建客户端 session 数据结构，分配好 yid、准备 DH 算法公钥 x_b 以及 λ_2，并在过程 C 中联通服务端发来的 cookie 一起发送给服务端。

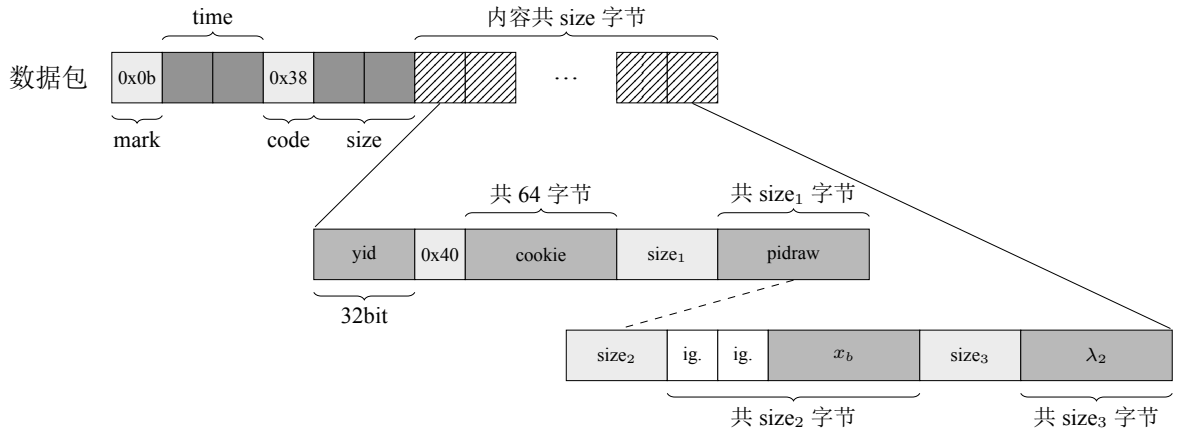

图 3.10: 握手过程：过程 C 数据包

过程 C 中（图3.10）：数据包中包含了客户端为建立连接所准备的所有数据结构。

服务端收到该消息之后，需要判断 cookie 的合法性。如果合法，则服务端也立即建立相应的服务端 session 数据结构，并分配好对应的 xid、生成服务端 DH 算法公钥 x_a 和 λ_1，并在下一过程 D 中将这些信息返回给客户端。除此之外，pidraw 是由客户端生成的一段随机串，服务端使用 pidraw.sha256 作为该连接的 pid。

其中 $size_1$、$size_2$、$size_3$ 均为 7bit 数（一种变长的整数表示方式）。

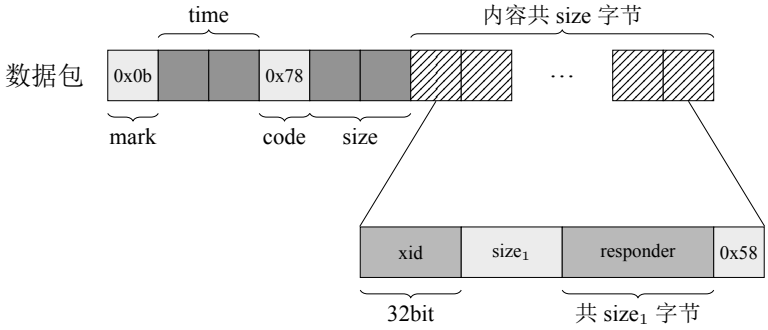

图 3.11: 握手过程：过程 D 数据包

过程 D 中（图3.11）：服务端将生成的 xid、DH 算法公钥 x_a 和 λ_1（包含在 responder 中）返回给客户端，完成 session 的建立。

客户端和服务端都需要将 xid、yid 写入自己的 session 数据结构中，并完成一对 AES128 密钥的生成操作。在握手过程结束以后，客户端和服务端之间的通信将基于 session 进行：生成完整 UDP 数据包时将需要使用正确的 xid 或者 yid，以及使用正确的密钥进行加解密。

3.3 Session 通信过程

在握手过程之后，所有的数据通信都建立在 session 的基础上。与握手过程略有不同的是，此时的通信过程一个数据包中可以包含连续的多个数据片。

如图3.12所示 RTMFP 三个协议层，此时通信的数据片可以分成两个集合：

a. session 粒度上的控制数据片；
b. flow 有关的数据片；

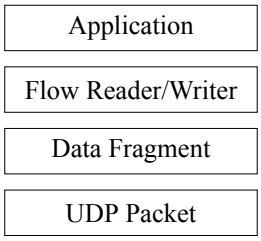

图 3.12: RTMFP 协议层

与握手过程一样，Session 通信过程的数据包也满足3.6的格式，所不同的是，服务端收发过程的数据包中的 mark 值不同，具体数值可以参考服务端对 request 和 response 处理的代码。

3.3.1 控制数据片

心跳

RTMFP 心跳数据片分成两种，请求和响应。

a. 客户端在与服务端连接成功之后，xserver 会将客户端心跳周期设置成 20 秒：即客户端每 20 秒向服务端发送一个心跳请求，服务端回复一个心跳响应。
b. 服务端会周期的检查客户端的通信情况，如果一定时间内（参见参数设置中的 heartbeat）没有收到客户端的数据请求，服务端则会主动发送一个心跳请求，然后等待客户端的心跳响应。

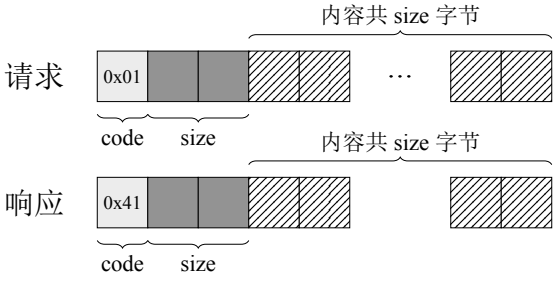

图 3.13: Session 通信：心跳

关闭连接

RTMFP 关闭连接的数据片也分成两种，请求和响应。主动关闭的端发送请求消息，另一端收到请求片之后发送响应消息进行响应。

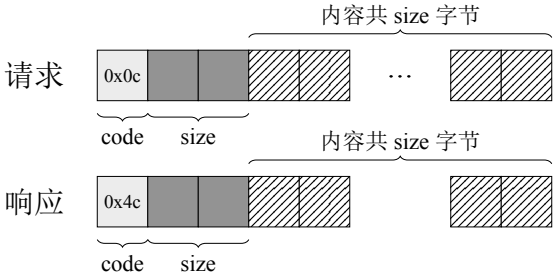

图 3.14: Session 通信：关闭连接

内部错误

在处理 flow 的过程中，如果出现无法恢复的严重错误，可以发送请求主动关闭该 flow。数据片结构如下图：

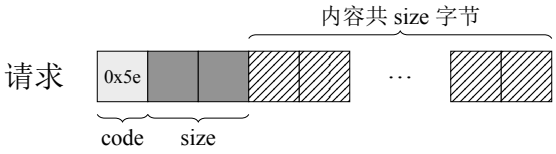

图 3.15: Session 通信：内部错误

数据片中包含一个 7bit 数表示出错的 flow 的 fid。

其他

xserver 仅仅实现了 RTMFP 协议的子集，对于未知的 code，xserver 会主动发起关闭连接请求来断开与客户端的连接。

3.3.2 flow 中的数据片

应用层数据片切割

RTMFP 协议中，尽可能要求一个 UDP 包大小不超过一个 MTU。因此应用层发送数据之前，需要将数据切割成满足该条件的一个或者多个数据片（这里只 flow 中的数据碎片

fragment），并将数据片放入发送缓存中。最后，需要对缓存中的数据进行 flush 或者丢包重传时，需要将一个或者多个数据片按照图3.6中的方法组装成一个数据包。

为了既满足这些限制，又要尽可能的减少 UDP 包的个数，xserver 采用一个折中方案，即预先将所有应用层的数据切割成大小不超过 256 字节的数据片：

a. 数据片越小，发送过程数据片装包越灵活，但是考虑到每个片都要维护一些信息，这样无效的开销也越大；
b. 数据片越大，装包越不灵活，发送的 UDP 包个数也可能越多；

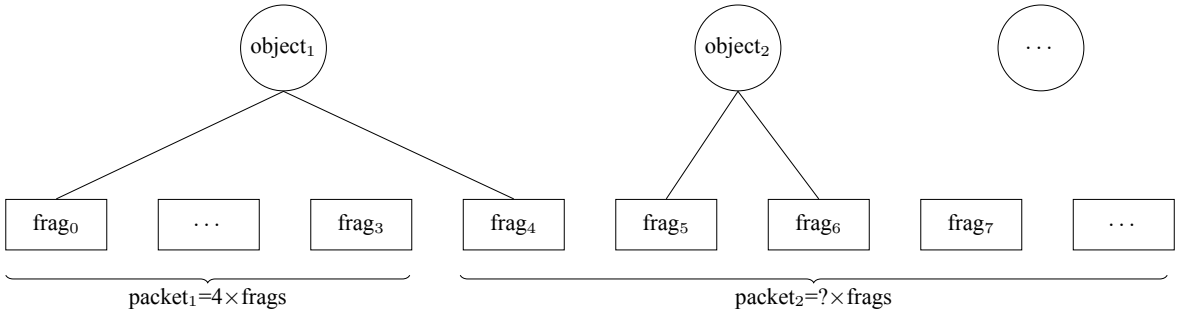

图 3.16: Session 通信：数据切片与装包

fragment 格式与组织

将应用层的数据进行切片，得到的数据结构如下：

```
public class Fragment {
    public long stage;          # 数据片序号
    public int flags;           # 一些标志
    public byte[] data;         # 数据内容
};
```

在每一个 flow 中，都分别独立的维护了收、发两个缓冲区，并利用这些缓冲区，来保证收发数据的 FIFO 和 Reliable 性质。

其中 flags 的不同位可以表示如下意义，具体实现可以参见代码：

a. 应用层数据的第一个数据片；
b. 应用层数据的最后一个数据片；
c. 当前 flow 的最后一个数据片；
d. 当前数据片内容无意义；

例如：对于应用层数据切片，第一个片都具有标志 a，最后一个片都具有标志 b；如果应用层数据不满 256 字节，即只切割成一个数据片，那么该片同时具有 a、b 两个标志位；如果数据片具有标志 c，则该数据片可丢弃；如果数据片具有标志 d，那么该数据片为当前 flow 最后一个数据片，xserver 会关闭该 flow。

具体操作的理解，请参考 flow reader/writer 的实现，:P。

ack 数据包

RTMFP 协议中，客户端每收到一个服务端或者服务端每收到一个客户端的数据包时，都会生成一个 ack 来告知对方已经收到数据，用以删除缓冲区中的缓存。

在 xserver 中，ack 还有一个功能是驱动被动丢包重传。这比主动重传响应时间更短。

实现中，每个 ack 的可以表示为一组连续的不相邻的闭区间的并，表示当前已经收到的数据片的 stage 集合，例如：

$$[s_0, s_1] \cup [s_2, s_3] \cup [s_4, s_5] \cup \cdots \cup [s_{2n}, s_{2_n+1}]，\text{其中 } s_0 = 0, s_{2i} \leq s_{2i+1}, s_{2i+1} + 1 < s_{2i+2}$$

那么又知道 7bit 数数值越小，序列化后越节省空间。那么可以定义另外一组数，如下：

$$\begin{cases} b_0 = s_0 = 0 \\ b_{2i+1} = s_{2i+1} - s_{2i} \\ b_{2i+2} = s_{2i+2} - s_{2i+1} - 2 \end{cases}$$

而序列 $b_1, b_2, \ldots, b_{2n+1}$ 即为 ack 序列化之后的结果，如图：

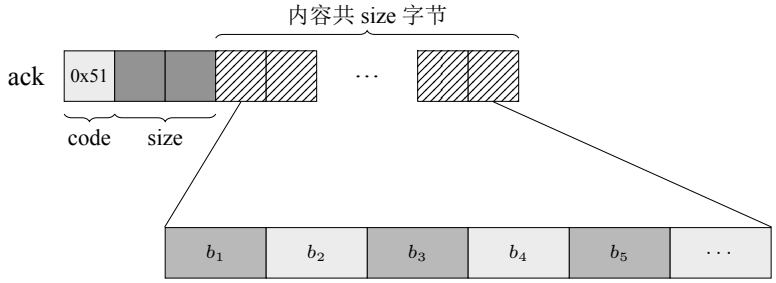

图 3.17: Session 通信过程：ack 数据片

0x10 数据包

在将多个数据片装如一个数据包时，除了写入数据片信息，还要写入数据片所属 flow 的信息。客户端和服务端对应 flow 信息确认如下：

a. 如果 writer 如果从未收到过来自 reader 的该 flow 的 ack，那么 writer 需要写完整的 flow 信息，以便 reader 能够建立正确的 flow 数据结构和映射关系；
b. 如果 writer 曾经收到过来自 reader 的属于该 flow 的 ack，那么 writer 只需要写如 flow 对应的 fid 即可；

该数据片中还有一个重要的数据结构，stageack。它表示 writer 收到的来自与 reader 的 ack 中，最大的连续的 stage 序号，即上一节中的 s_1 的值。

如果 reader 收到一个比发送给 writer 的 s_1 还要大的 stageack 值时，表示 writer 并不关心 reader 是否全部收到满足区间 [s_1,stageack] 的数据片，也就是 writer 不会对这区间的数据片进

行丢包重传。所以此时 reader 会放弃这中间不完整的数据，而向前跳。利用这种性质，能够巧妙的在服务端实现上行或者下行的高效的 unreliable 数据通道。

数据片格式如下：

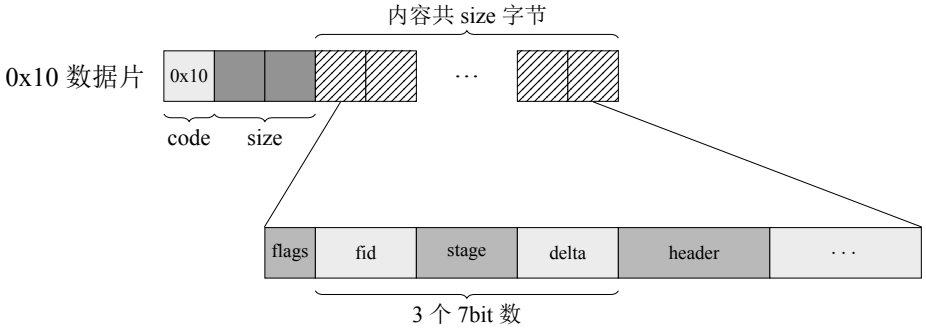

图 3.18: Session 通信过程：0x10 数据片

其中 stageack=stage-delta，同样由于这样编码使得数据包体积更小；header 以空字符串作为结束标志；剩余字节中，出去 header 的内容，其他都为数据片中的数据。剩余字节中，出去 header

0x11 数据包

RTMFP 协议中通过 0x11 数据片来进一步节省空间：如果即将写入的数据片与前一个数据片来自与同一个 flow，并且 stage 是连续的下一个，那么实际上可以省略这些 stage 的空间。RTMFP 协议中通过 0x11 数据包来实现这样的功能。

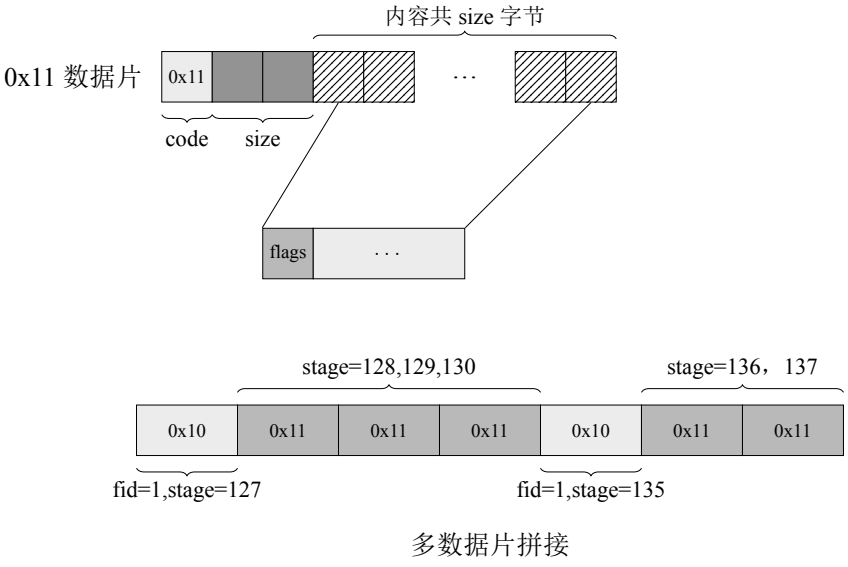

图 3.19: Session 通信过程：0x11 数据片

unreliable 消息

所谓 unreliable 消息是针对应用层而言的。xserver 通过修改 stageack 来达到这个目的。

服务端 -客户端：服务端作为 writer，客户端作为 reader 的过程。假设服务端发送先后发送数据 $\{o_1,o_2\}$ 到客户端，并且发送 o_1 过程中出现丢包，那么即便客户端完全收到 o_2 消息，也需要等待 o_1 的丢包重传。而 xserver 所做的优化是在发送 o_2 的过程中，主动修改 stageack 的值为 o_1 最后一个数据片的 stage，使得客户端一旦收到 o_2 的数据片便会直接忽略尚在等待的 o_1 数据。假设，$o_1 = \{m_{197}, m_{198}\}$ 以及 $o_2 = \{m_{199}, m_{200}\}$，过程如下：

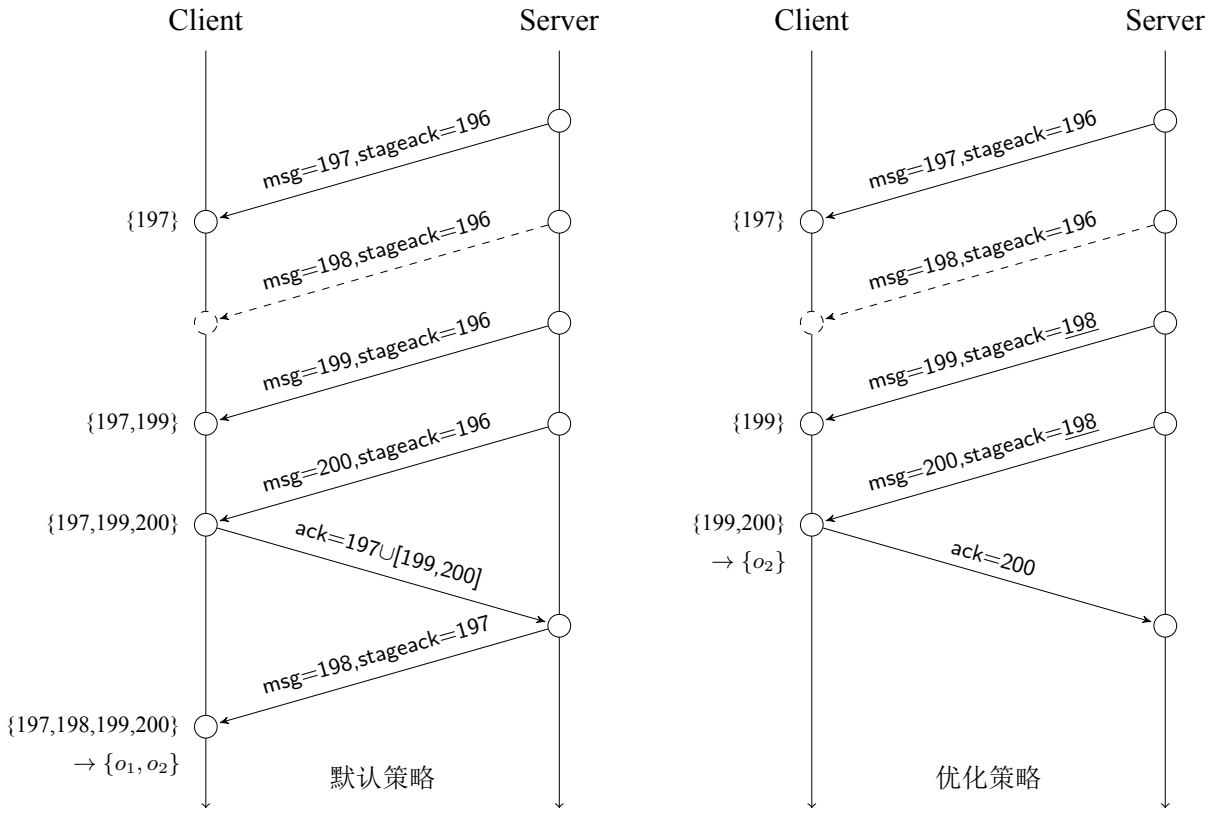

图 3.20: 服务端 -客户端：unreliable 消息

客户端 -服务端：客户端作为 writer，服务端作为 reader 的过程。客户端是 flash 实现，当客户端发送的 unreliable 数据发生丢包的时候，需要通过服务端的 ack 进行判断，并根据情况发送一些特殊的空包来驱动服务端跳过这些未收到的数据。

例如，假设客户端发送消息 $\{o_1, o_2, \dots\} = \{m_{197}, m_{198}, \dots\}$ 到服务端，如下图：

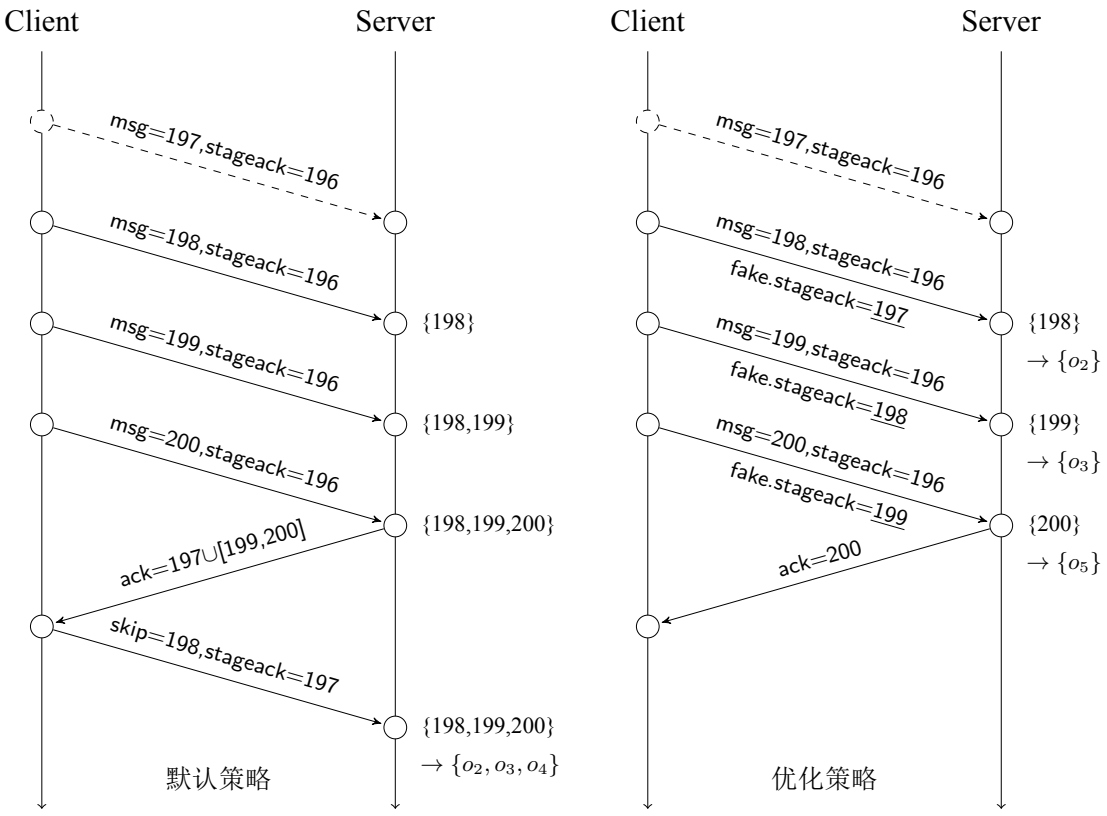

图 3.21: 客户端 -服务端：unreliable 消息

3.4 P2P 过程

假设客户端 P_1 主动发起与客户端 P_2 的 P2P 过程，流程如下：

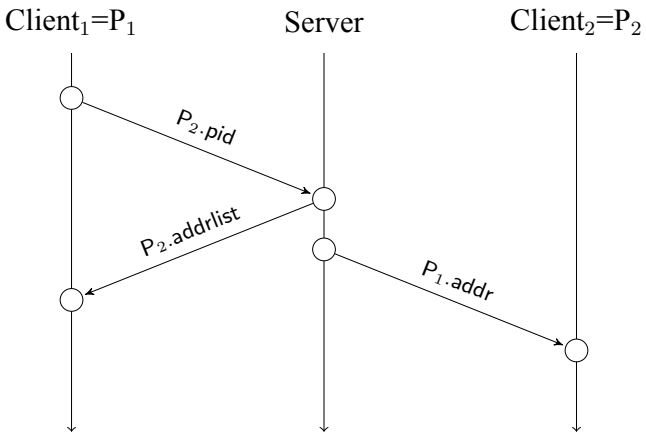

图 3.22: P2P 过程

具体过程如下：

a. P_1 发送 P2P 请求给服务端，其中消息中包好 P_2 的 pid；
b. 服务端向 P_1 发送 P_2 的全部地址，包括外网地址以及内网地址；

c. 服务端向 P_2 发送 P_1 的一个地址，通常是外网地址，用于 P_2 在 NAT 上打洞；

d. P_1 和 P_2 尝试建立 P2P 连接；

P_1 发向 xserver 的数据包与握手过程 A（图3.8）的数据包几乎完全一样，请求包与响应包结构如下：

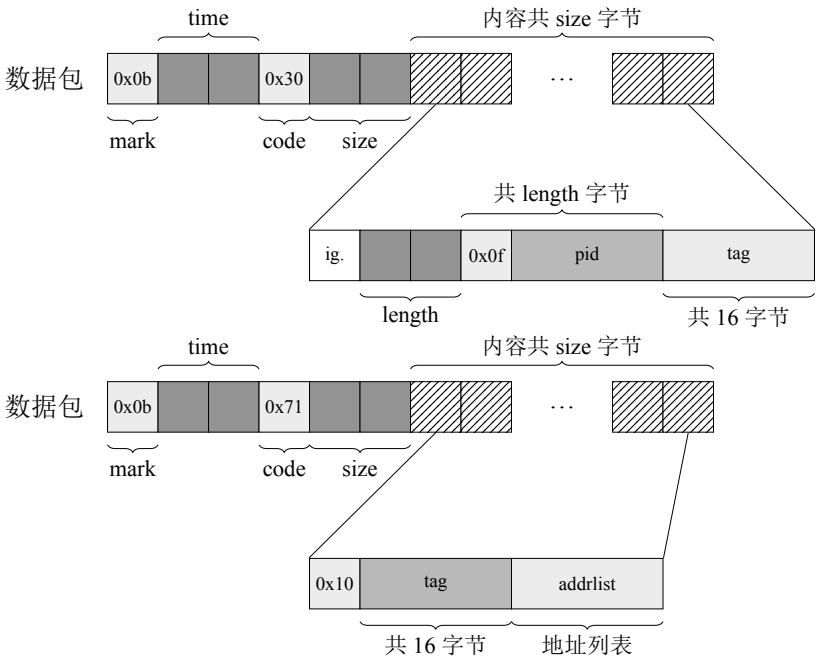

图 3.23: P2P 过程：P_1 请求与响应数据包

xserver 发送给 P_2 的数据片通过的是已经 xserver 与 P_2 已经建立好的 flow 连接，与控制数据片相似，数据片格式如下：

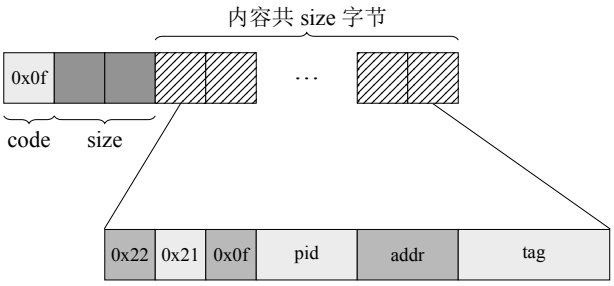

图 3.24: P2P 过程：P_2 收到数据片

表格索引

插图索引