xserver 简介

23rd June 2015

目录

什么是 RTMFP

- RTMFP(Real Time Media Flow Protocol) 服务器
  - 基于 UDP
    - 低延迟
    - 可控的可靠性
    - 协议内丢包容错
  - 支持动态 IP
  - 节省带宽：支持客户端 P2P

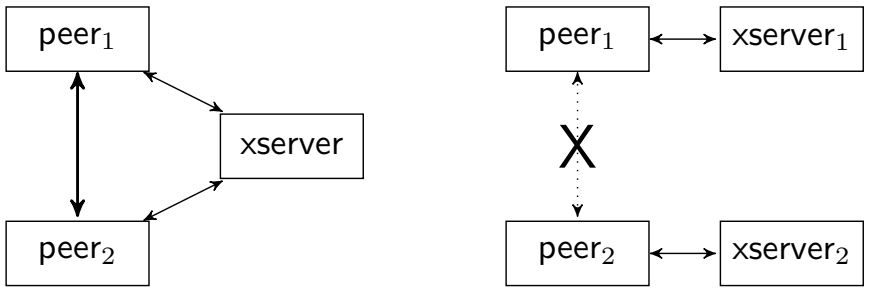
什么是 xserver

- OpenRTMFP@github
  - xserver = 130+ 个文件/21k+ 行 C++ 代码

什么是 xserver

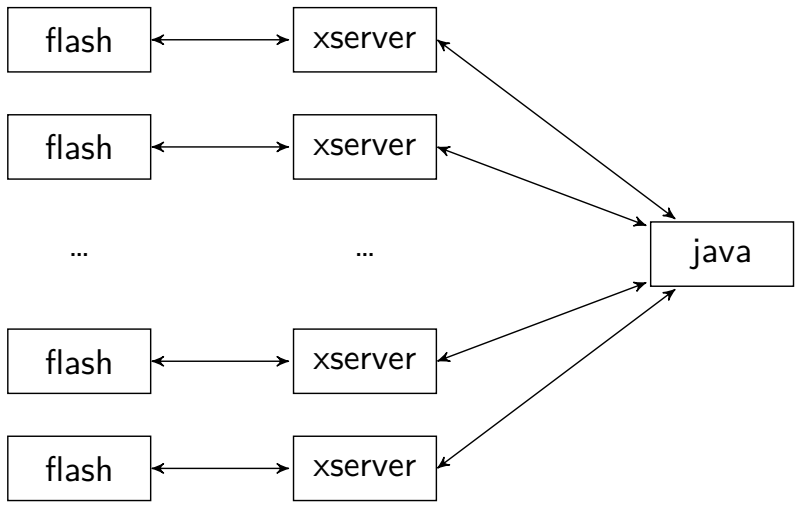
- P2P 应用场景
  - 视频数据片
  - 游戏内部分状态、消息
- 两个客户端 P2P 的条件：

  当且仅当两个客户端与同一个 RTMFP 服务器通信



什么是 xserver

- RPC 应用场景
  - 视频内弹幕等
  - 游戏内状态、动作等等



为什么重构

- 项目需求：需要深入了解 RTMFP 协议
- 可靠性差：
  - 可攻击漏洞导致程序 crash
  - 内存泄漏或者 BUG 导致吃光 CPU、内存等资源
  - 程序僵死：服务无响应
- 性能差：单线程易过载
  - P2P 过程 2–3w 连接数
  - 100 人/秒进入速度
- 不易维护、不易监控、不宜调试等等

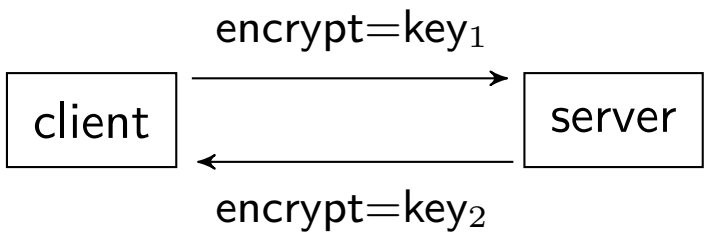
xserver

- 多线程：设计目标是 P2P 20–30w 并发连接无压力
  - 实际情况：
    - 27w 连接;1200CPU(MAX=1600)
    - 单核 250+ 人/秒的进入速度
- 可靠性
- 可读性
  - 尽保留 P2P/RPC 的最小实现
  - 数据结构的提炼
  - xserver = 50+ 文件/7k+ 行代码

目录

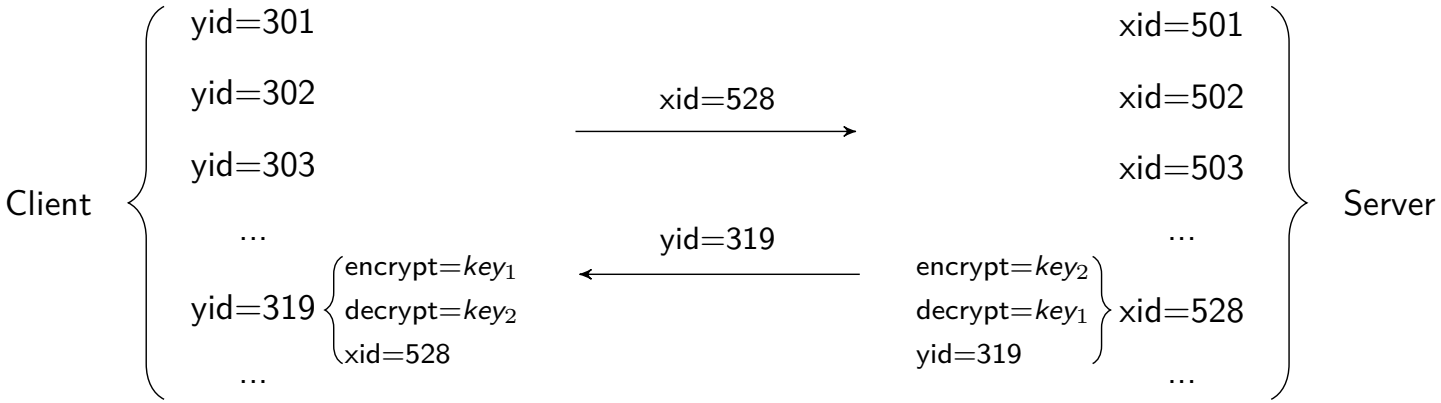
基本数据包

- 基于 UDP 通信，单个数据包大小不超过 1 个 MTU
- 数据内容通过 AES128 加密
  - 对称加密算法：密钥同时用于加密、解密过程
  - flash 与 xserver 协商生成一对密钥分别用于上下行加密



基本数据包

- 基于 UDP 通信，单个数据包大小不超过 1 个 MTU
- 数据内容通过 AES128 加密
  - 对称加密算法：密钥同时用于加密、解密过程
  - flash 与 xserver 协商生成一对密钥分别用于上下行加密
  - 两端分别维护数据结构保存密钥：数据包中记录解密 id

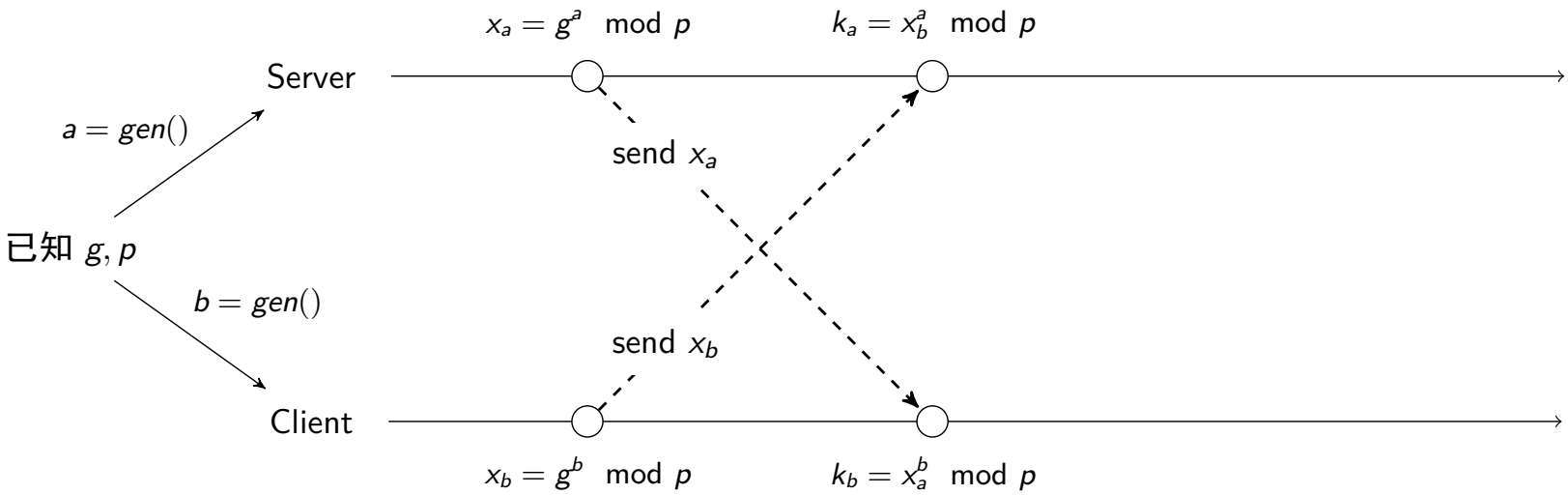


基本数据包

- 基于 UDP 通信，单个数据包大小不超过 1 个 MTU
- 数据内容通过 AES128 加密
  - 对称加密算法：密钥同时用于加密、解密过程
  - flash 与 xserver 协商生成一对密钥分别用于上下行加密
  - 两端分别维护数据结构保存密钥：数据包中记录解密 id
  - 握手过程使用 id=0 以及默认密钥 “Adobe Systems 02”

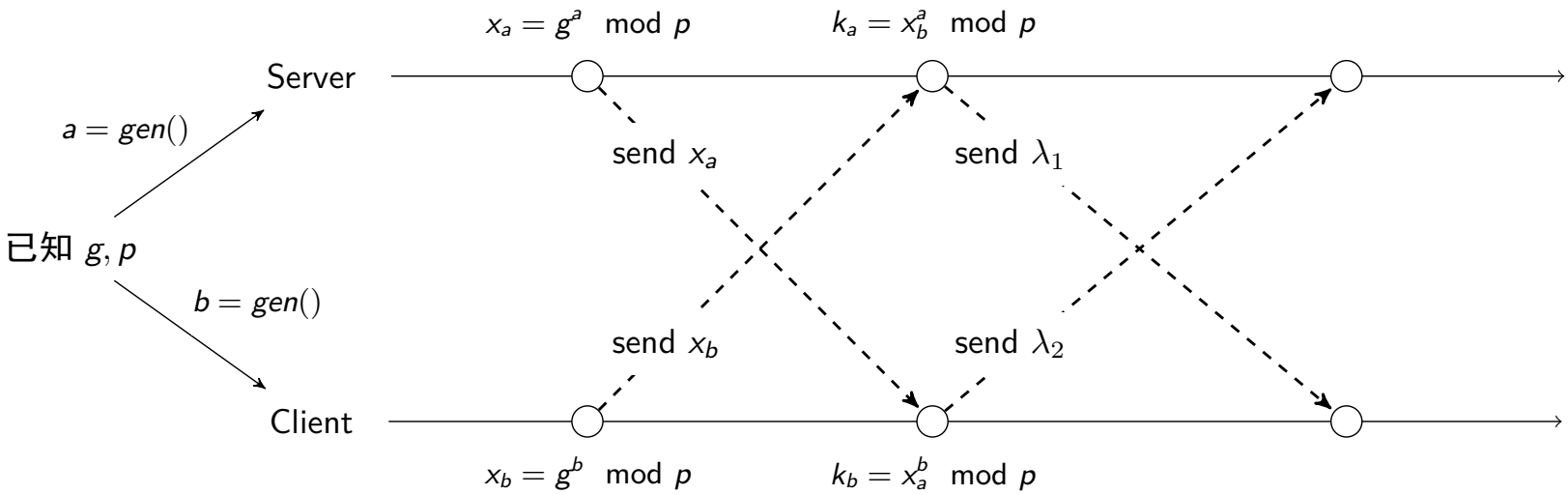
共享密钥生成

- 通过非对称加密 DH 算法生成共享 key



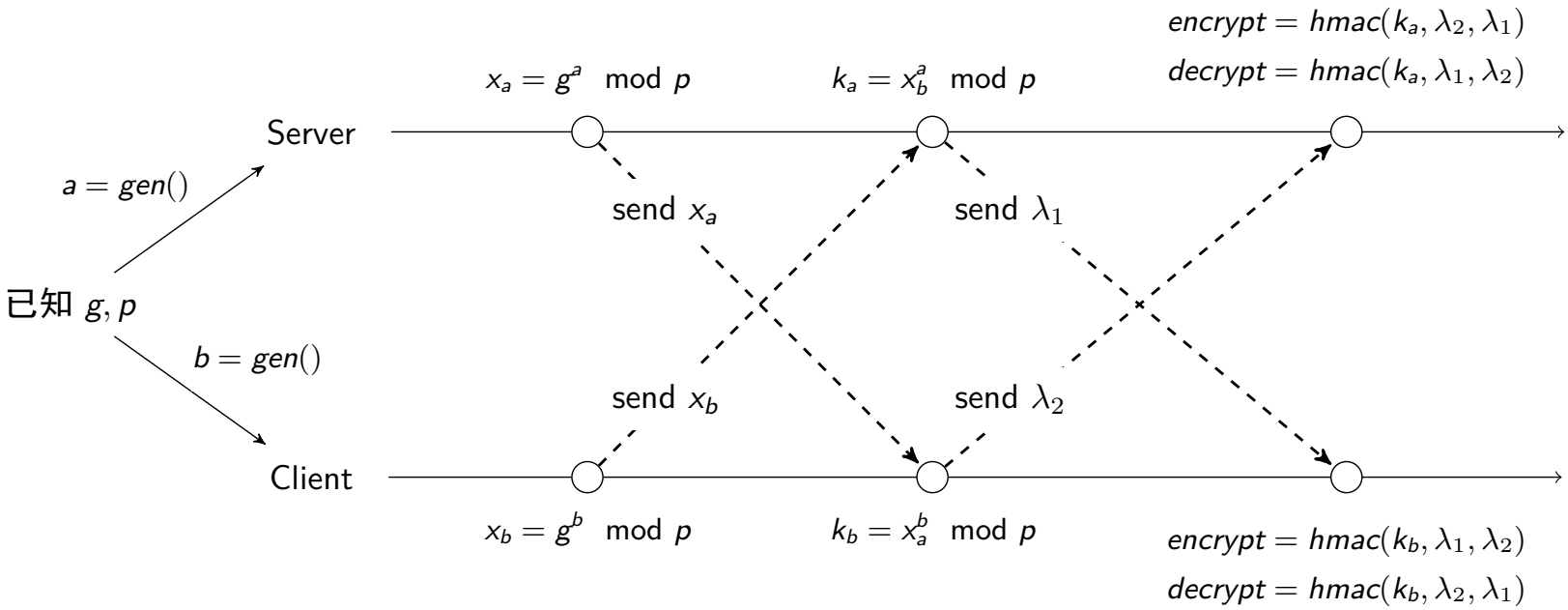
共享密钥生成

- 两端各生成一个随机串并交换



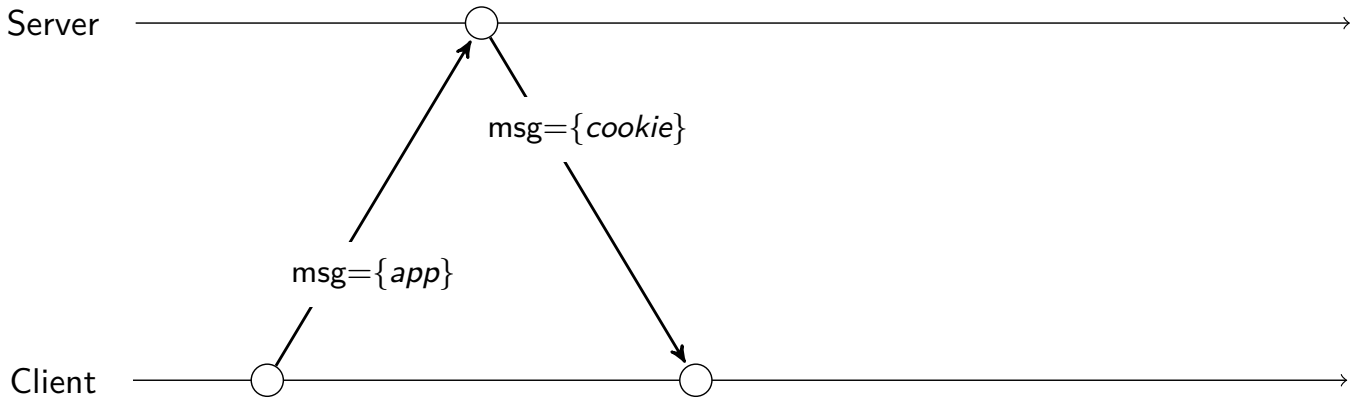
共享密钥生成

- 两端分别用共享 key 对随机串进行 hash 的到一组共享密钥



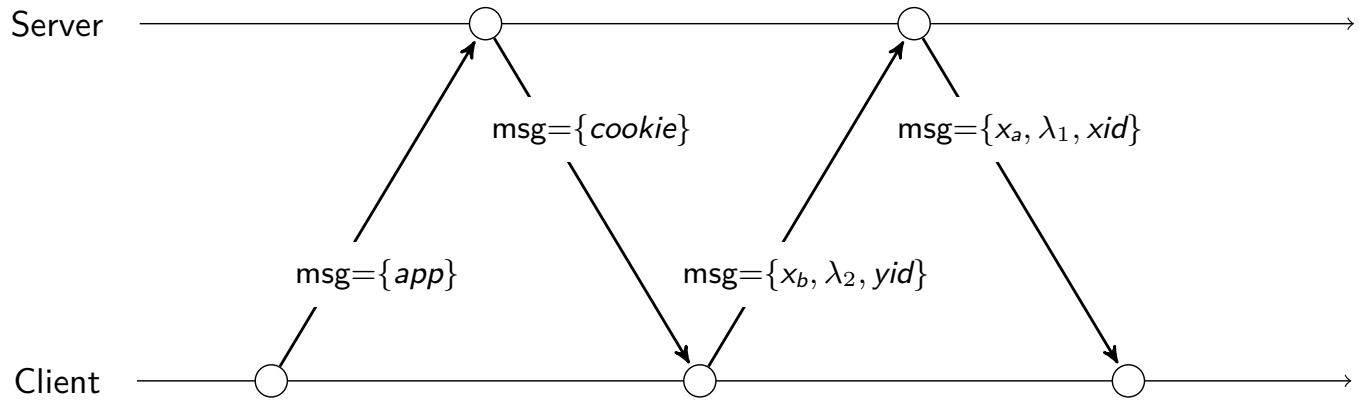
握手过程

- 探测对方并对应用进行验证



握手过程

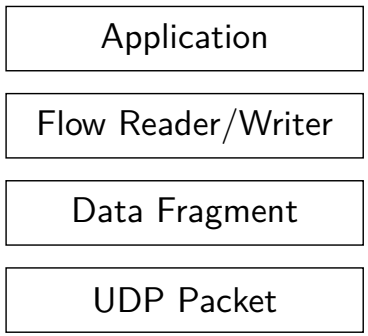
- 探测对方并对应用进行验证
- 进行 DH 算法生成密钥



目录

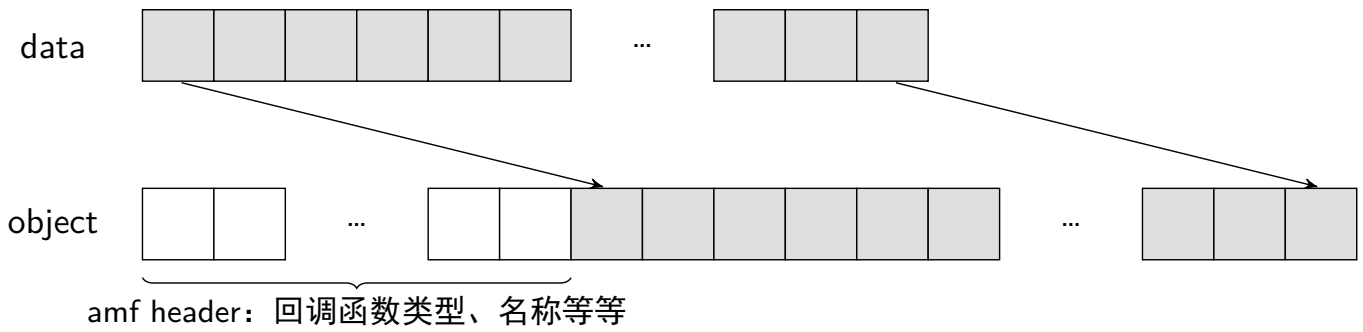
多层结构

- 应用层：产生数据，并对数据进行封装
- 数据流：缓存应用层数据，维护传输可靠性
- 数据片：数据发送最小单元



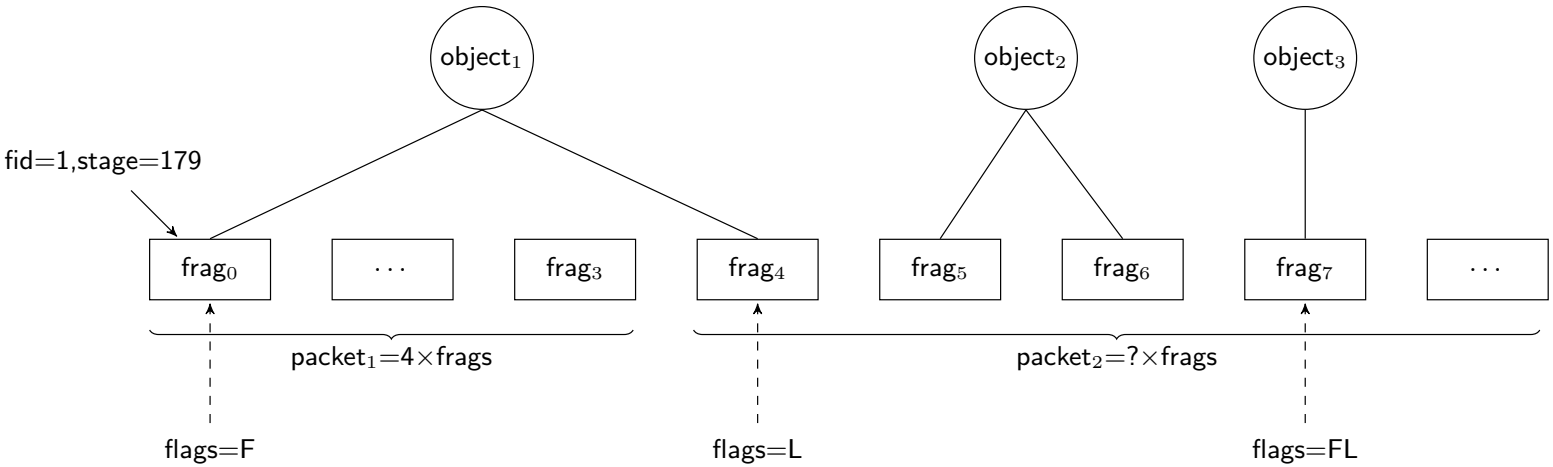
多层结构

- 应用层：产生数据，并对数据进行封装



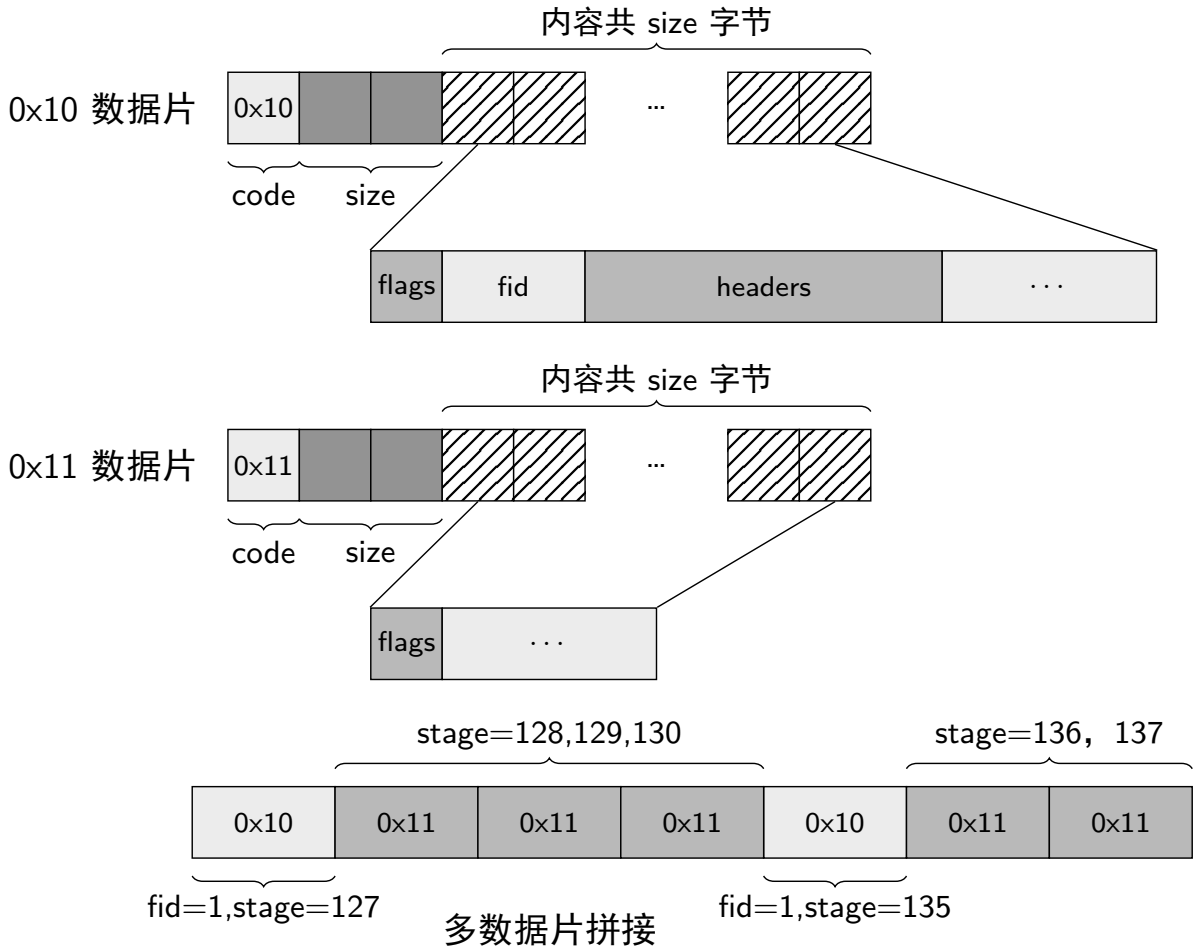
多层结构

- 数据流：缓存应用层数据，维护传输可靠性
  - 对数据进行切片并分配序号
  - 切片大：切片个数少，减少传输成本
  - 切片小：UDP 包数量少，组装灵活



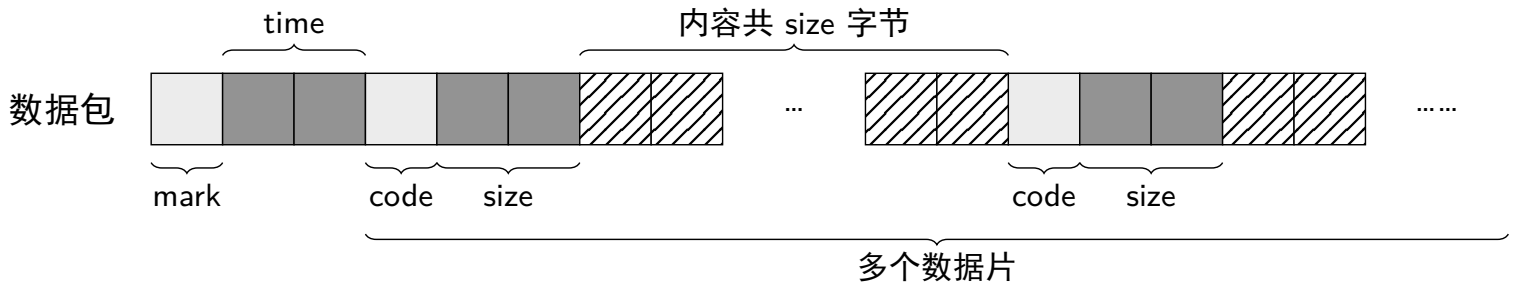
多层结构

- 数据片：数据发送最小单元



多层结构

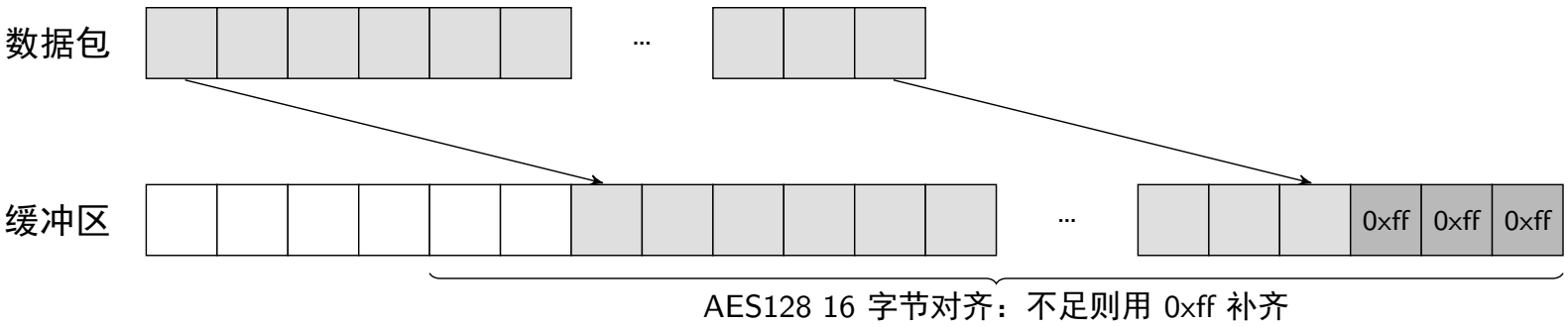
- 数据包：包含一个或者多个数据片，并加密



  - 控制片：心跳请求与响应、关闭连接请求、数据包 ack 等等
  - 流数据：0x10 数据片、0x11 数据片

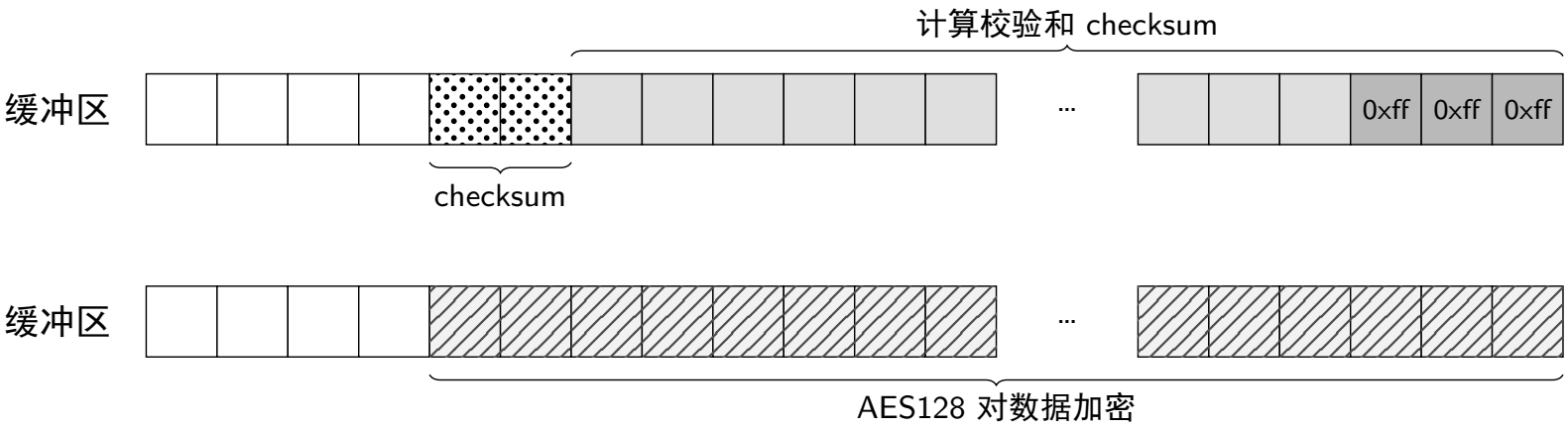
UDP 数据包生成

- 拷贝数据包内容到缓冲区并填充 0xff 补全



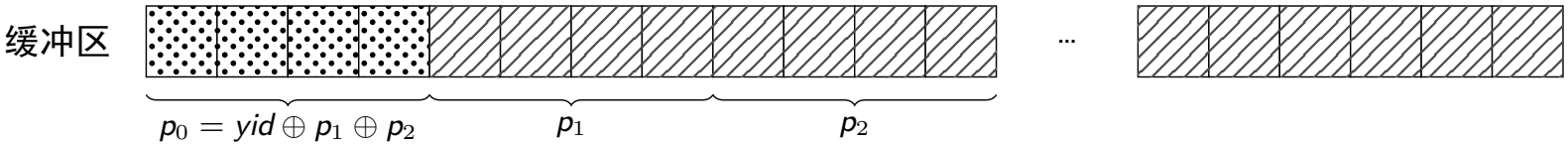
UDP 数据包生成

- 拷贝数据包内容到缓冲区并填充 0xff 补全
- 生成数据校验和并使用加密密钥加密



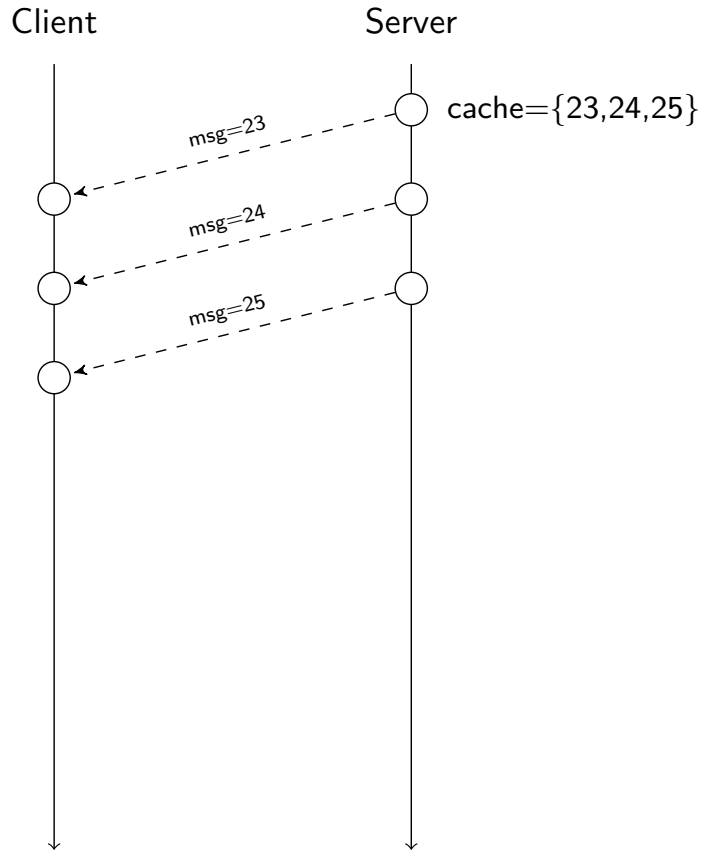
UDP 数据包生成

- 拷贝数据包内容到缓冲区并填充 0xff 补全
- 生成数据校验和并使用加密密钥加密
- 写入对端 id 信息



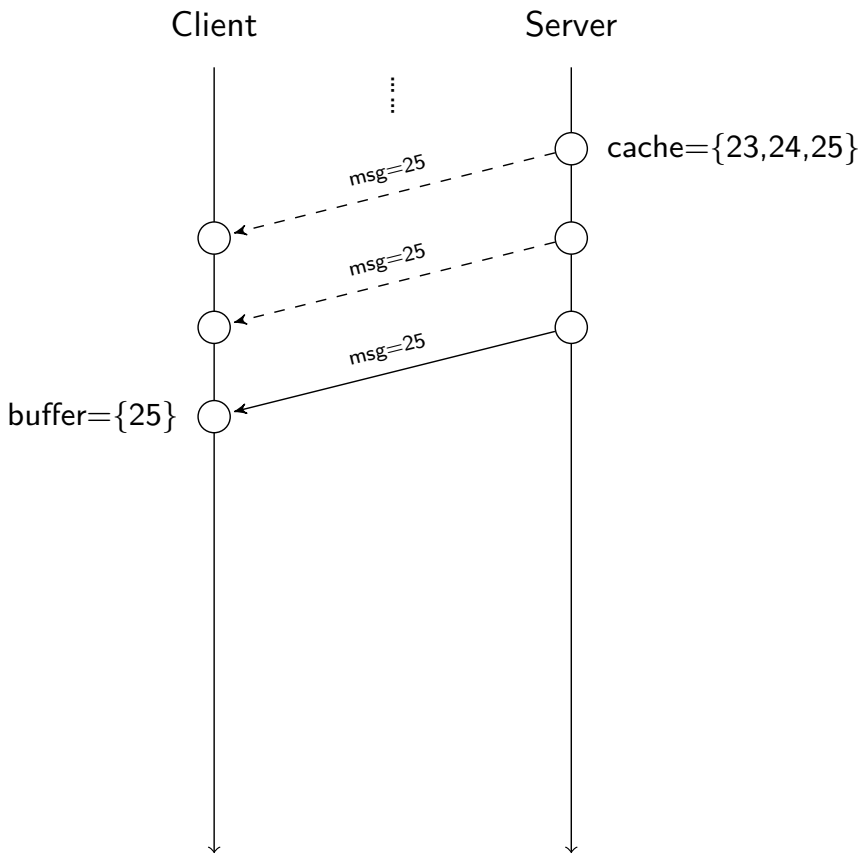
Reliable 通信过程

- 写入数据 obj=$\{m_{23},m_{24},m_{25}\}$



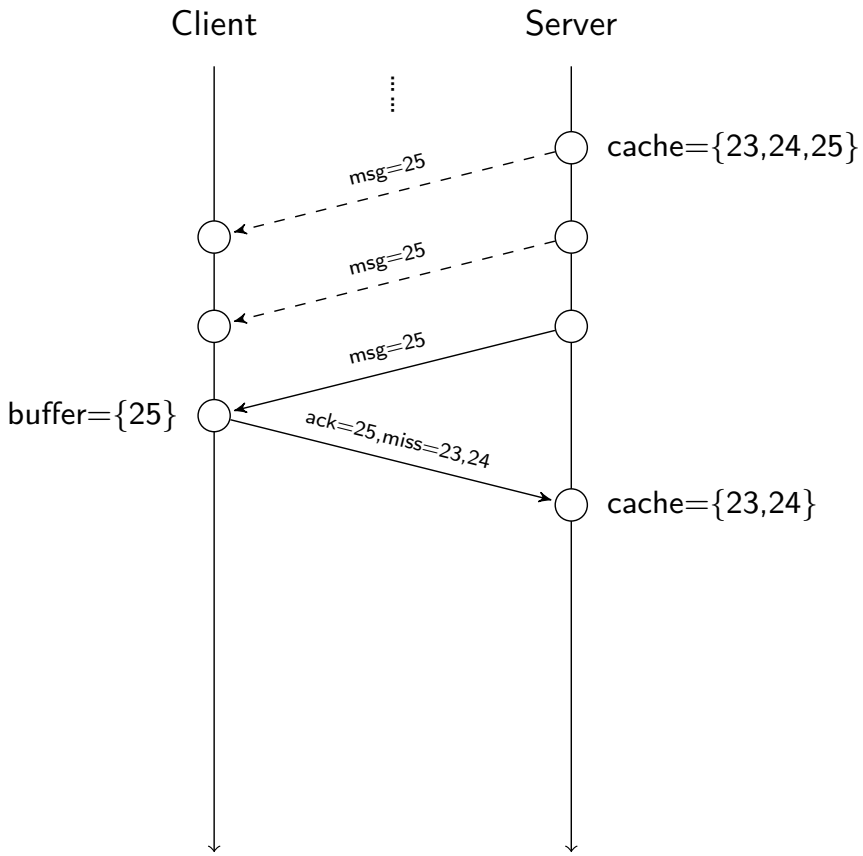
Reliable 通信过程

- 主动对最后一个数据包进行丢包重传



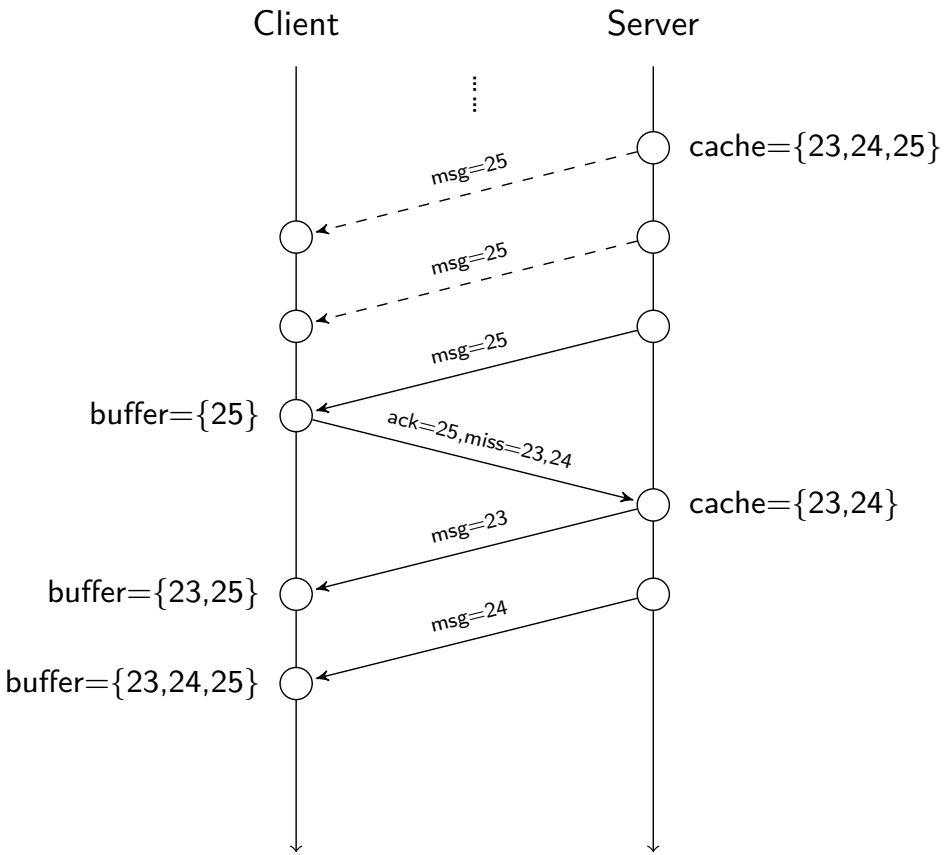
Reliable 通信过程

- 收到消息立即返回 ack



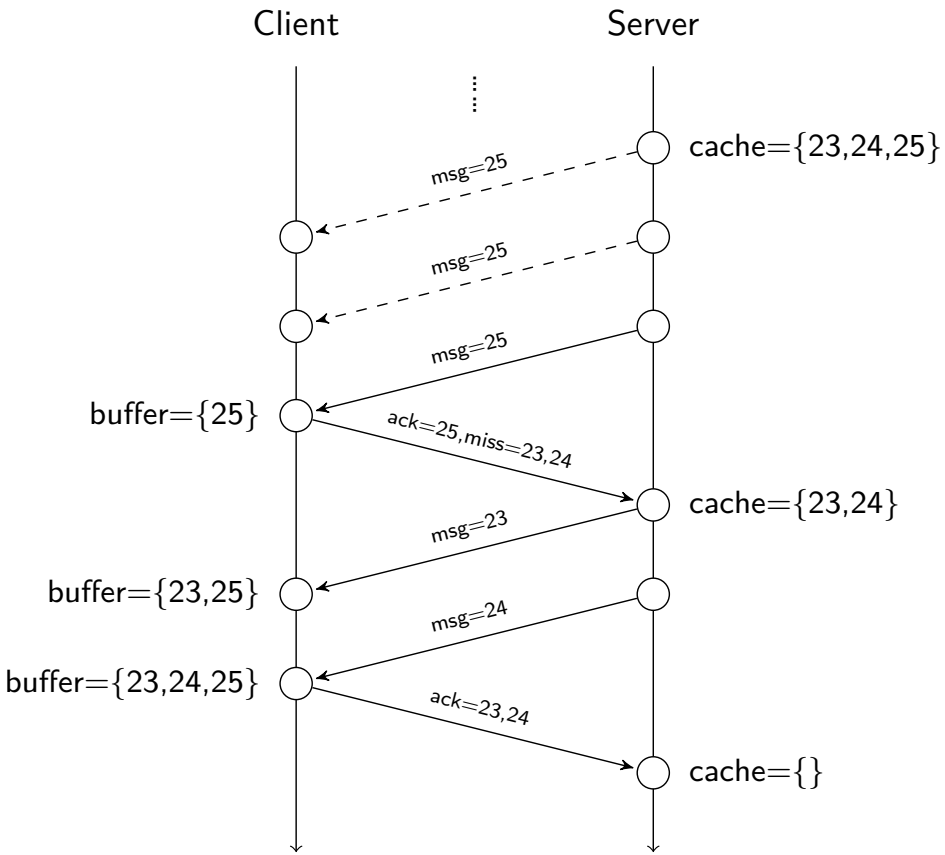
Reliable 通信过程

- ack 驱动丢包重传



Reliable 通信过程

- 收到数据包 deliver 消息并返回 ack



Unreliable 通信过程

- 协议中，Sender 在发送数据时，还返回从 Receiver 发来的 ack 的 ack*!!

$$\text{ack}=[0,\text{ack}^*] \cup [s_0, s_1] \cup [s_2, s_3] \cup \cdots \cup [s_{2n}, s_{2_n+1}]$$

其中 $s_{2i+1} + 1 < s_{2i+2}, i \geq 0$

  - ack* 表示 Receiver 收到的最大连续 stage 序号

Unreliable 通信过程

- 协议中，Sender 在发送数据时，还返回从 Receiver 发来的 ack 的 ack*!!

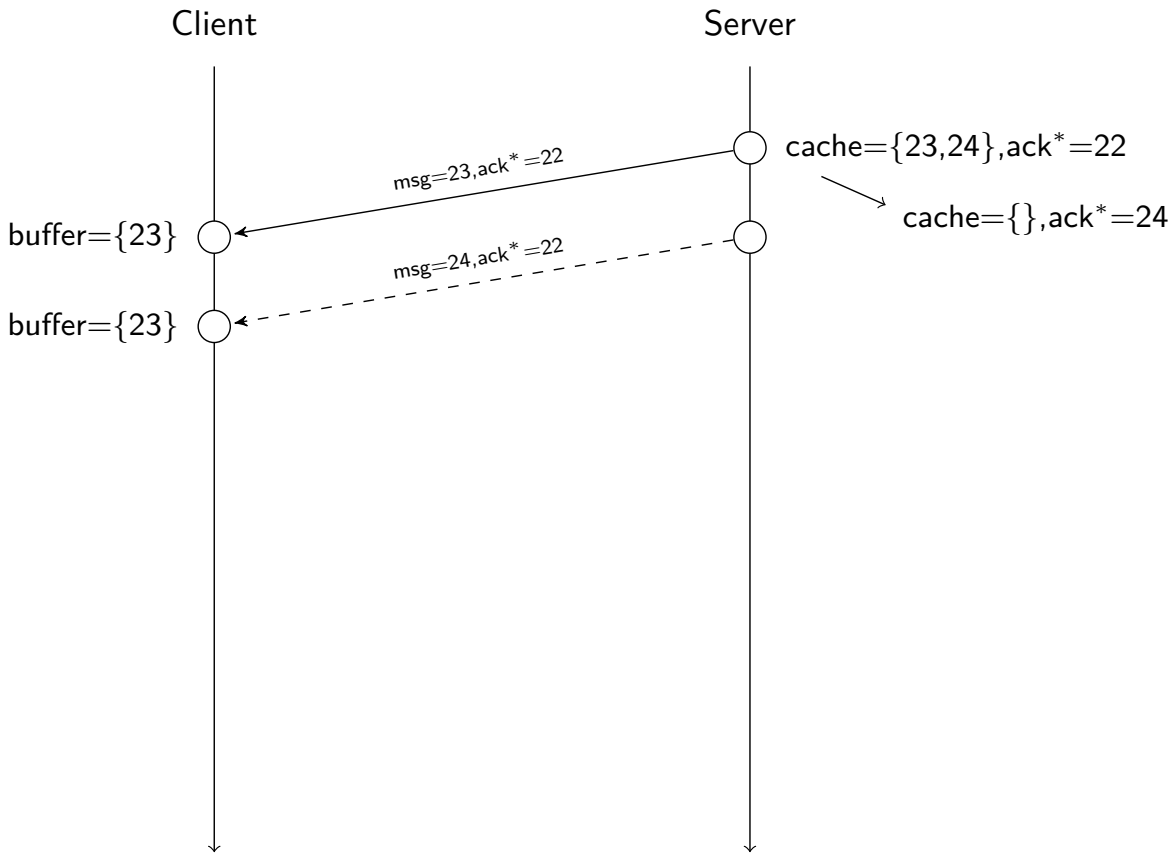
$$\text{ack}=[0,\text{ack}^*] \cup [s_0, s_1] \cup [s_2, s_3] \cup \cdots \cup [s_{2n}, s_{2_n+1}]$$

$$\text{其中 } s_{2i+1} + 1 < s_{2i+2}, i \geq 0$$

  - ack* 表示 Receiver 收到的最大连续 stage 序号
  - 实践中发现：ack* 对于 Receiver 还表示 stage≤ack* 的消息 Sender 都不再缓存!!

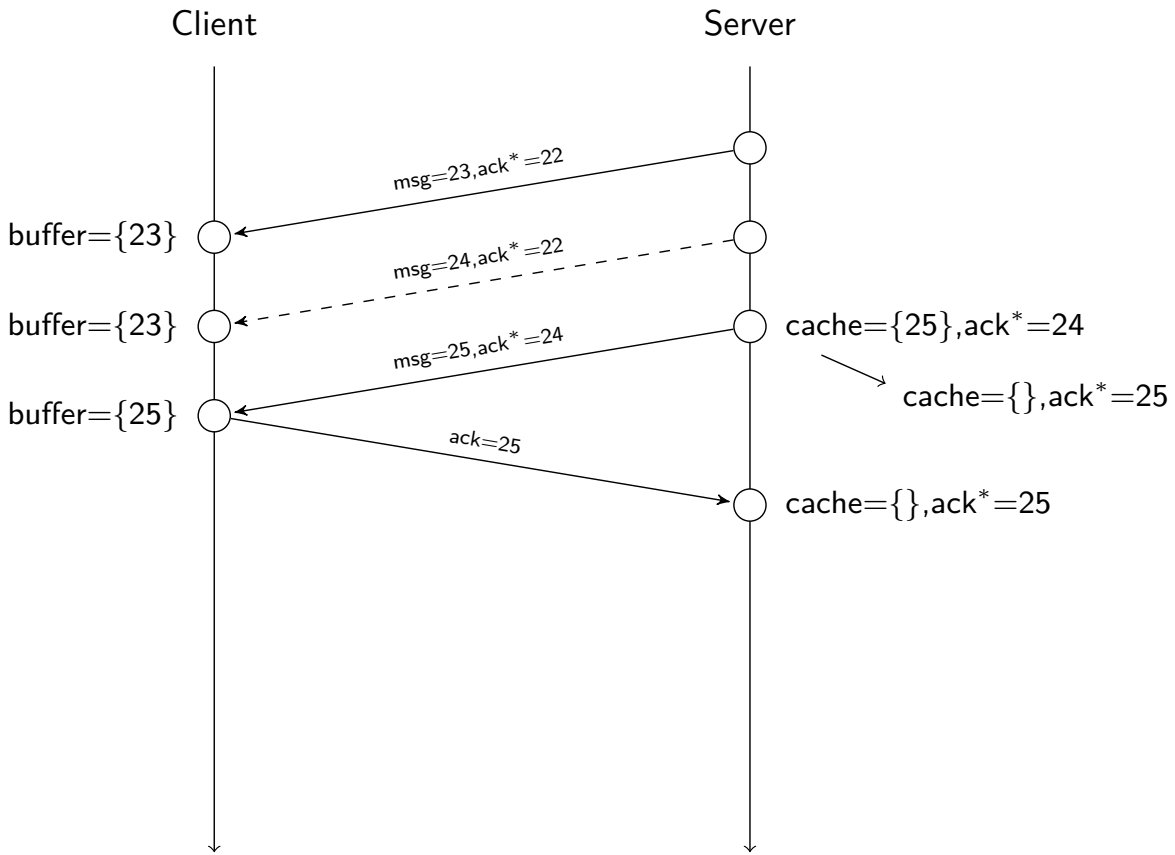
Unreliable 通信过程

- 写入数据 $\text{obj}_1 = \{m_{23}, m_{24}\}$



Unreliable 通信过程

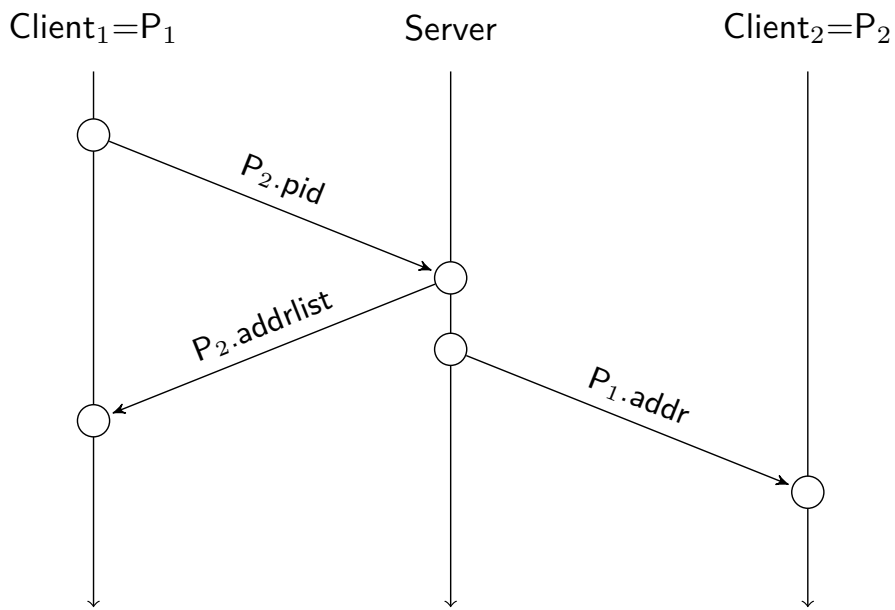
- 写入数据 $\mathrm{obj}_2=\{\mathrm{m}_{25}\}$



目录

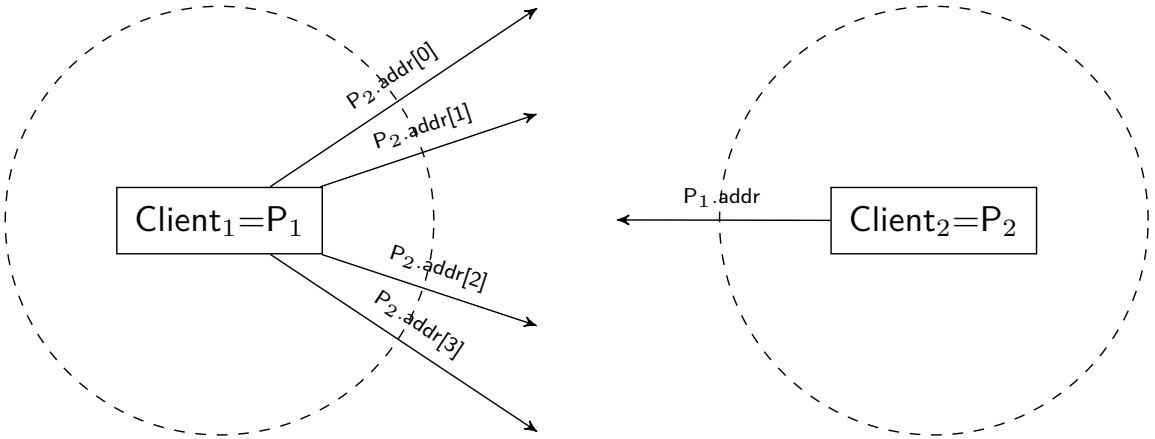
P2P 打洞

- P_1 发起与 P_2 的 P2P 过程
  - P_1 将 P_2.pid 发送给服务端
  - 服务端向两边返回对方的信息



P2P 打洞

- 双方打洞
  - P_1 向获取到的 P_2 的所有地址发送 request
  - P_2 向获取到的 P_1 的地址发送 response



目录

为什么选择 Go

- 编译执行
  - 编译速度快 - C 语言爱好者
  - 命令行开发、调试 +linux 工具链
- 运行效率
  - c > go > c++
  - 代码质量影响大于语言选择
- 开发效率
  - 自带内存 gc
  - 保留但是弱化指针

为什么选择 Go

- 语法简洁
  - 简单的类声明、初始化：编译器做最少的事情
    - c++ 的构造函数、析构函数、虚函数？
    - java 的构造函数执行顺序？
  - 空!= null：减少错误和参数检查
    - 空字符串 = ""
    - 空数组 = [] 类型

为什么选择 Go

- 语法简洁
  - 内置切片 slice
  - 多函数返回值
  - 错误处理：error ？ panic ？
  - defer 比 finally 更强大
  - 语法更灵活：闭包 + 函数类型

为什么选择 Go

- routine 概念
  - 轻量级线程：fiber?
- channel 概念
  - 管道?
  - 生产者 + 消费者 $\rightarrow$ 线程同步