IPC性能极致优化方案-RPAL落地实践

×

谢正尧

字节跳动
研发工程师

目 录

第一部分

方案诞生的背景

方案诞生的背景

几种常见的同机通信场景：

1. 微服务合并部署（亲和性部署、sidecar 部署）

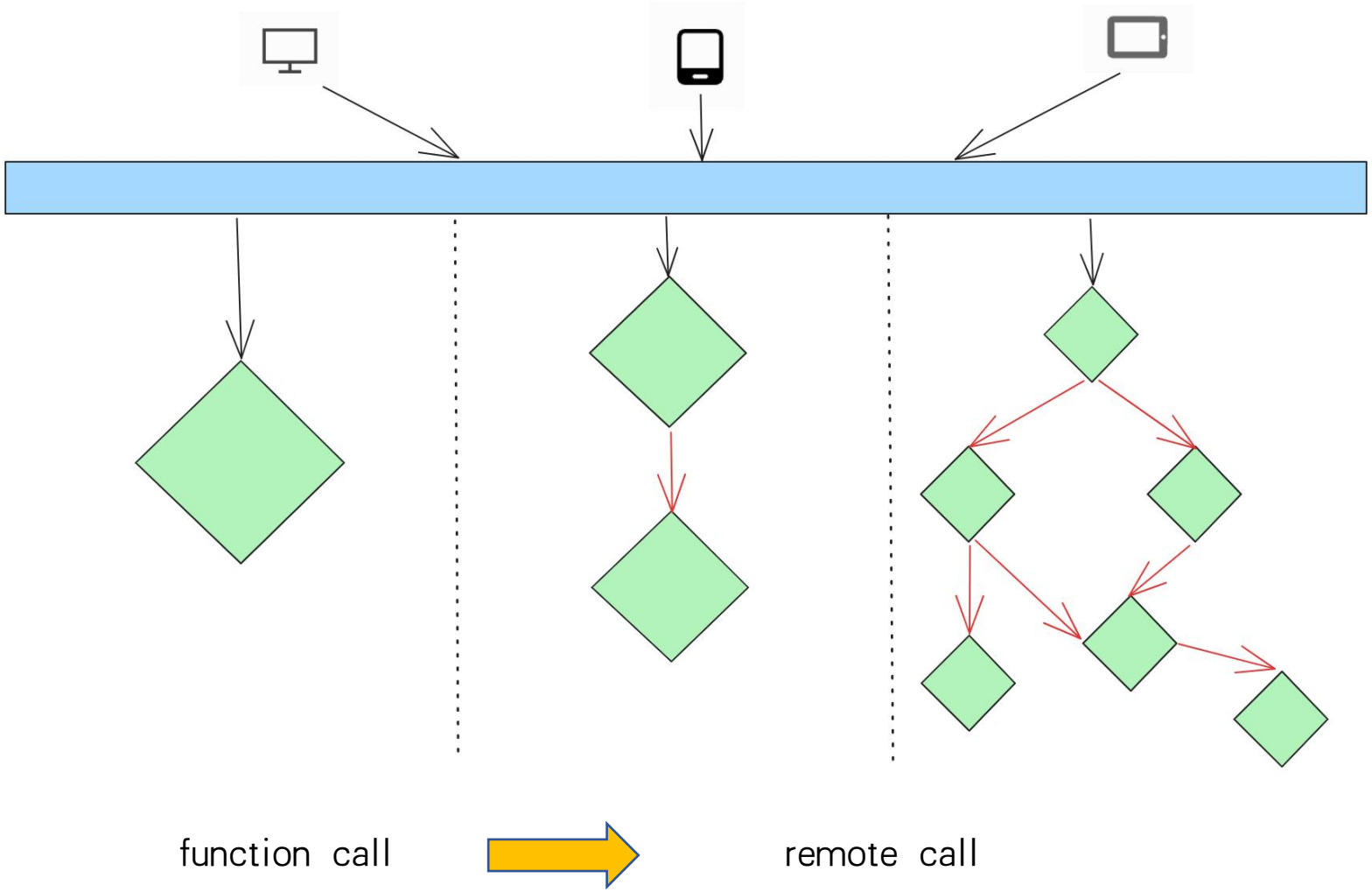
2. 本地基础组件：mesh sidecar、风控 sidecar、分布式网关...

方案诞生的背景

微服务合并部署

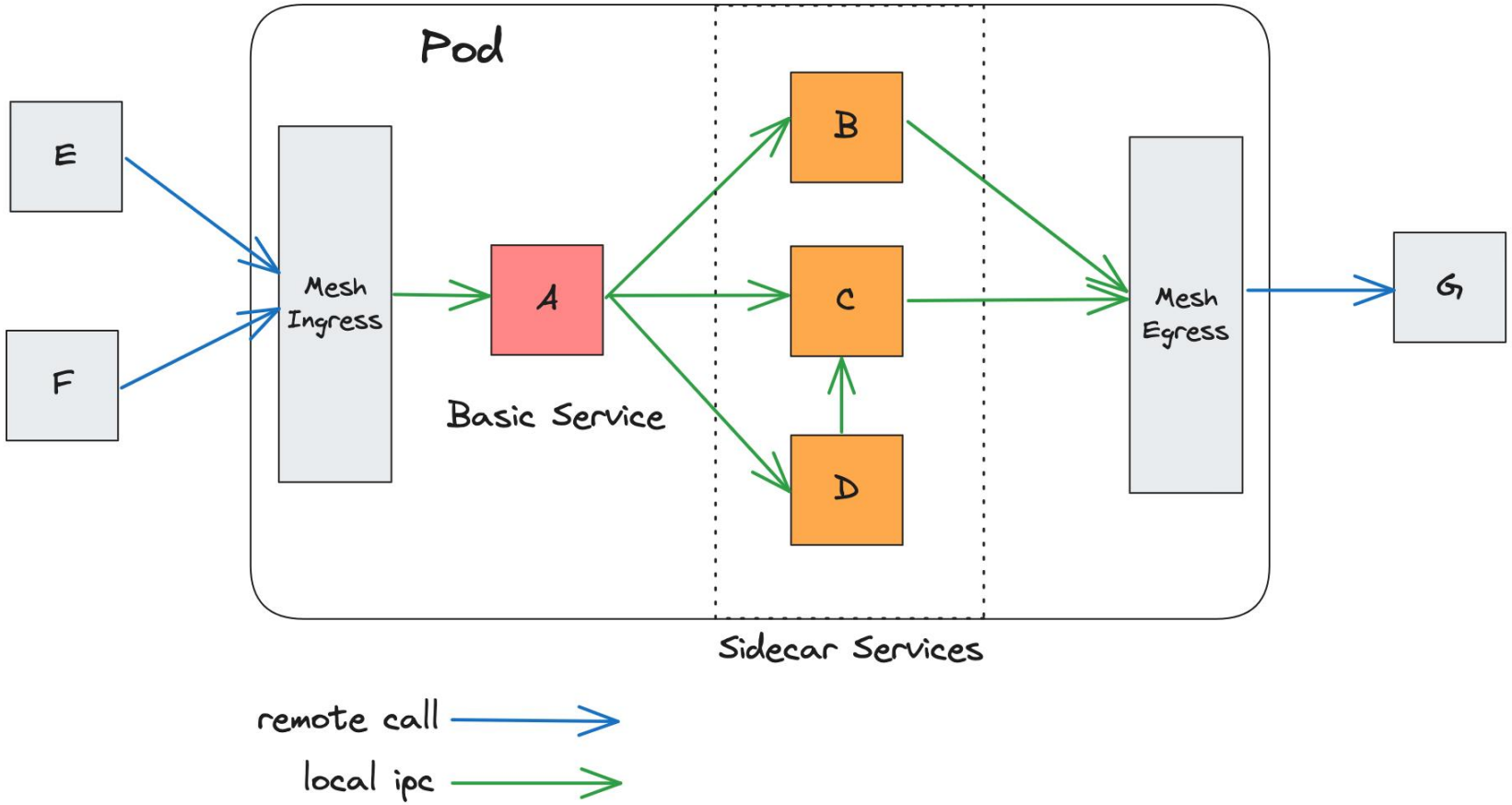
微服务化拆分：

1. 序列化
2. 网络开销
3. 服务治理



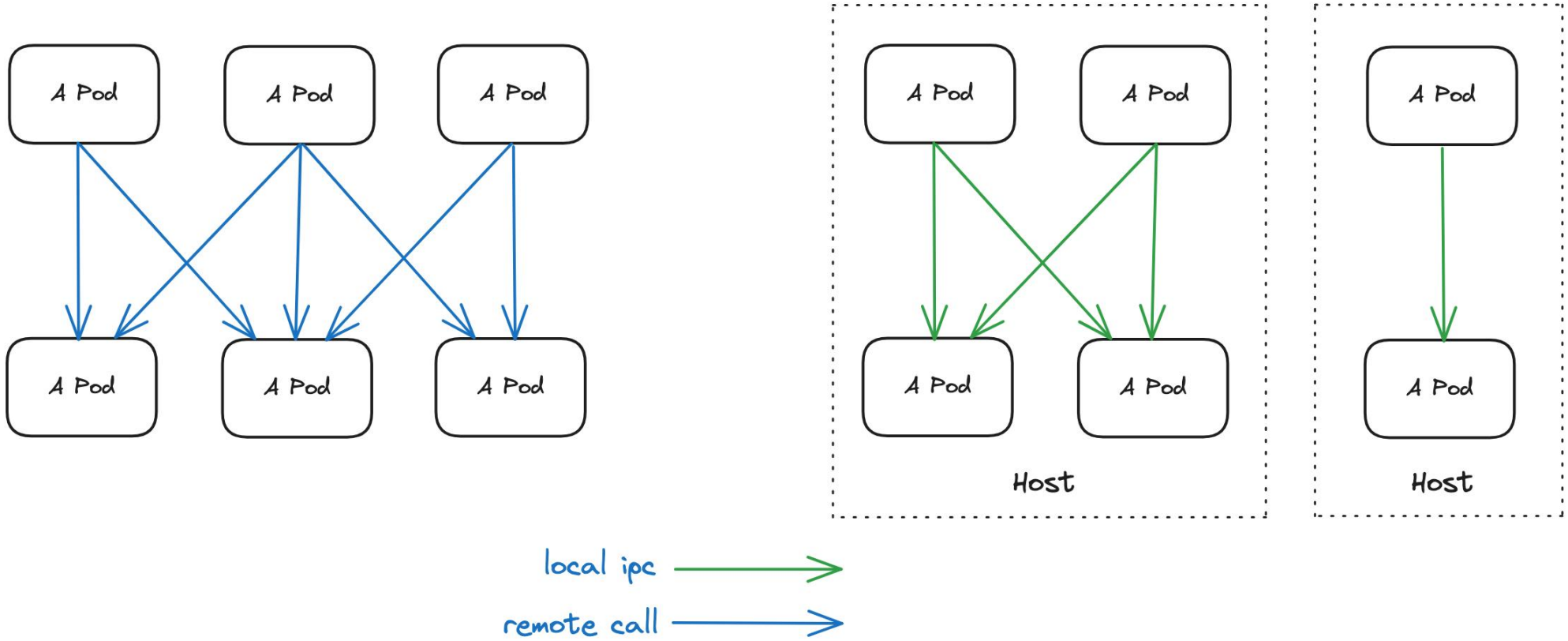
方案诞生的背景

微服务合并形态：sidecar 进程通信



方案诞生的背景

微服务合并形态：亲和性部署



方案诞生的背景

怎么放大本地通信的优势？

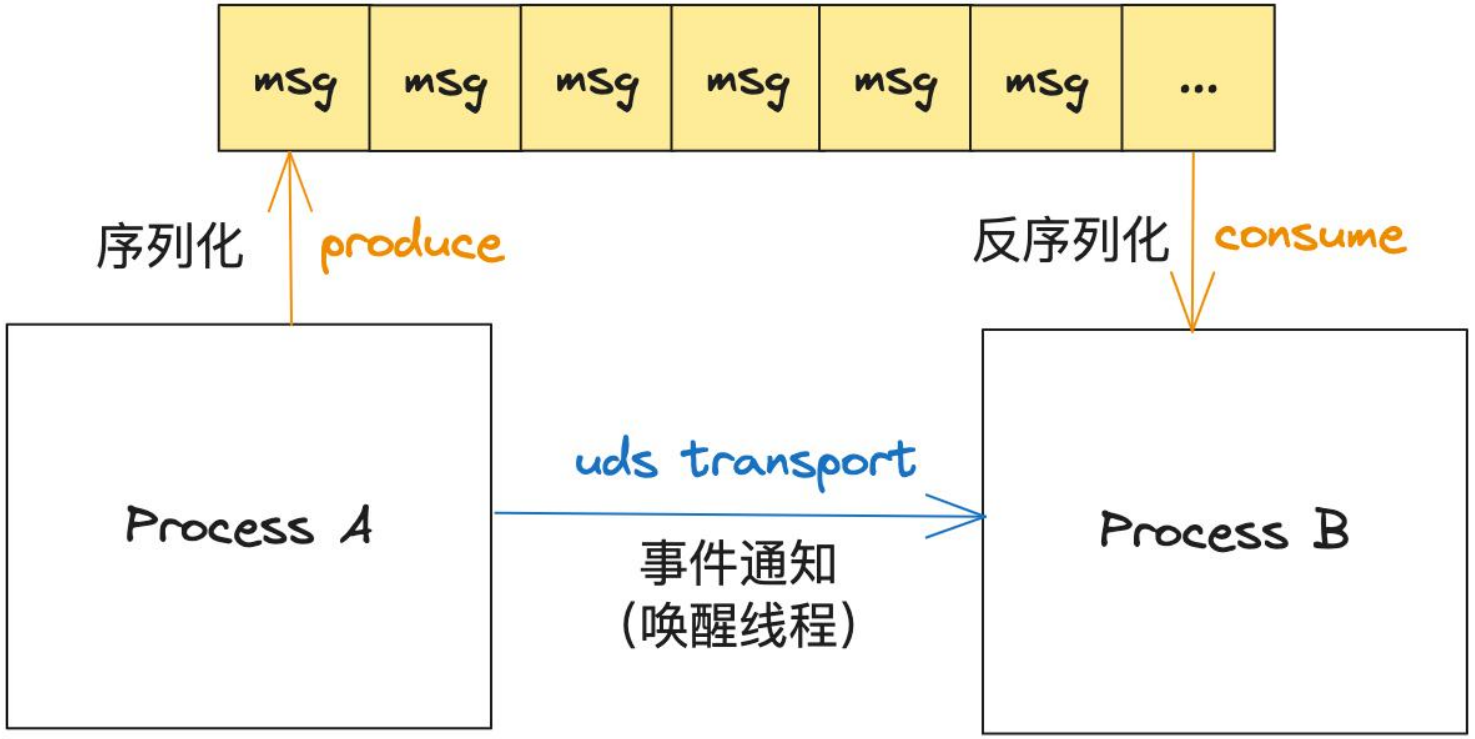
低延迟  提升用户体验

低开销  降低计算成本

常见的本地通信方案：回环 IP、UDS、共享内存IPC

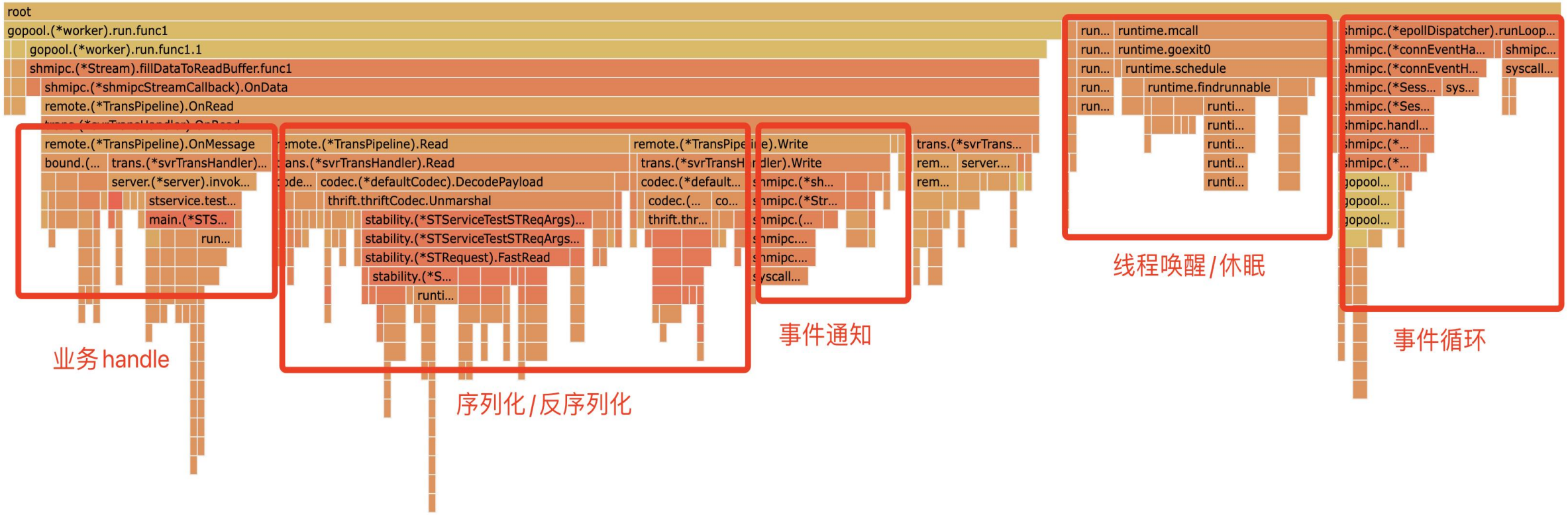
方案诞生的背景

以性能较优的 IPC 方案 share memory ipc 为例分析性能瓶颈：



注：方案 github 地址：https://github.com/cloudwego/shmipc-go

方案诞生的背景


root
gopool.(*worker).run.func1
gopool.(*worker).run.func1.1
shmipc.(*Stream).fillDataToReadBuffer.func1
shmipc.(*shmipcStreamCallback).OnData
remote.(*TransPipeline).OnRead
remote.(*TransPipeline).OnMessage
bound.(...
trans.(*svrTransHandler)...
server.(*server).invok...
stservice.test...
main.(*STS...
run...
remote.(*TransPipeline).Read
ans.(*svrTransHandler).Read
ode...
codec.(*defaultCodec).DecodePayload
thrift.thriftCodec.Unmarshal
stability.(*STServiceTestSTReqArgs)...
stability.(*STServiceTestSTReqArgs...
stability.(*STRequest).FastRead
stability.(*S...
runti...
remote.(*TransPipe
trans.(*svrTransH
codec.(*default...
codec.(...
co...
thrift.thr...
e).Write
dler).Write
shmipc.(*sh...
shmipc.(*Str...
shmipc.(...
shmipc....
shmipc....
syscall...
trans.(*svrTrans...
rem...
server....
rem...
run...
runtime.mcall
run...
runtime.goexit0
run...
runtime.schedule
run...
runtime.findrunnable
run...
runti...
runti...
runti...
runti...
runti...
shmipc.(*epollDispatcher).runLoop...
shmipc.(*connEventHa...
shmipc...
shmipc.(*connEventH...
syscall...
shmipc.(*Sess...
sys...
shmipc.(*Ses...
shmipc.handl...
shmipc.(*...
shmipc.(*...
gopool...
gopool...
gopool...
业务 handle
序列化/反序列化
事件通知
线程唤醒/休眠
事件循环

方案诞生的背景

IPC 的性能瓶颈有哪些：

1. 系统特权级切换；

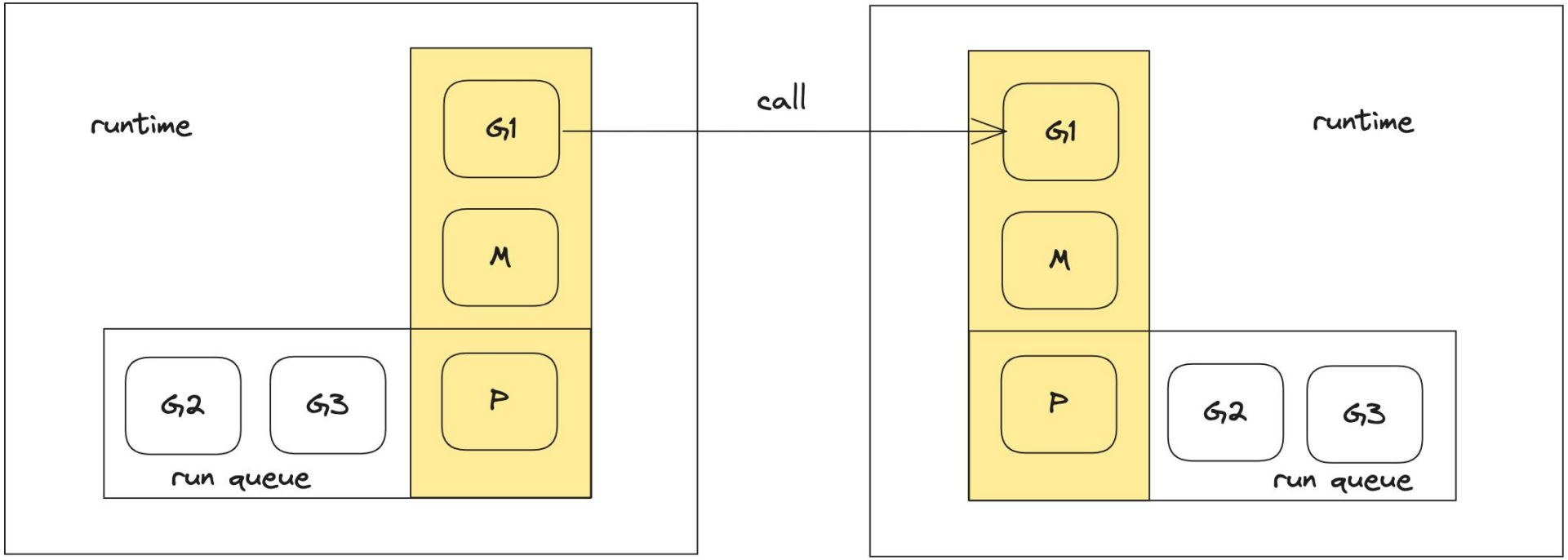
2. 异步线程唤醒/休眠（事件通知）；

3. 数据拷贝（序列化/反序列化）；

方案诞生的背景

能不能把库函数调用的高性能优势做到 IPC 里面，降低进程间的事件通知和数据拷贝开销？

以go-go微服务 RPC 通信场景为例，该问题可以抽象为，如何高效地在两个 go runtime 间进行函数调用？



方案诞生的背景

从性能瓶颈的两点分析：

1．异步线程唤醒：

关键在于如何最低限度降低线程唤醒的开销，非必要不通知事件。

2．数据序列化/反序列化

需要做到跨进程的虚拟地址空间共享，通过传递指针来传递一切数据。

基于以上问题，我们最终引入了 RPAL（Run Process As Library） 方案，基于跨进程虚拟地址共享，复用 epoll 网络模型，实现了纯用户态的事件轮询和无拷贝的指针读写接口。

第二部分

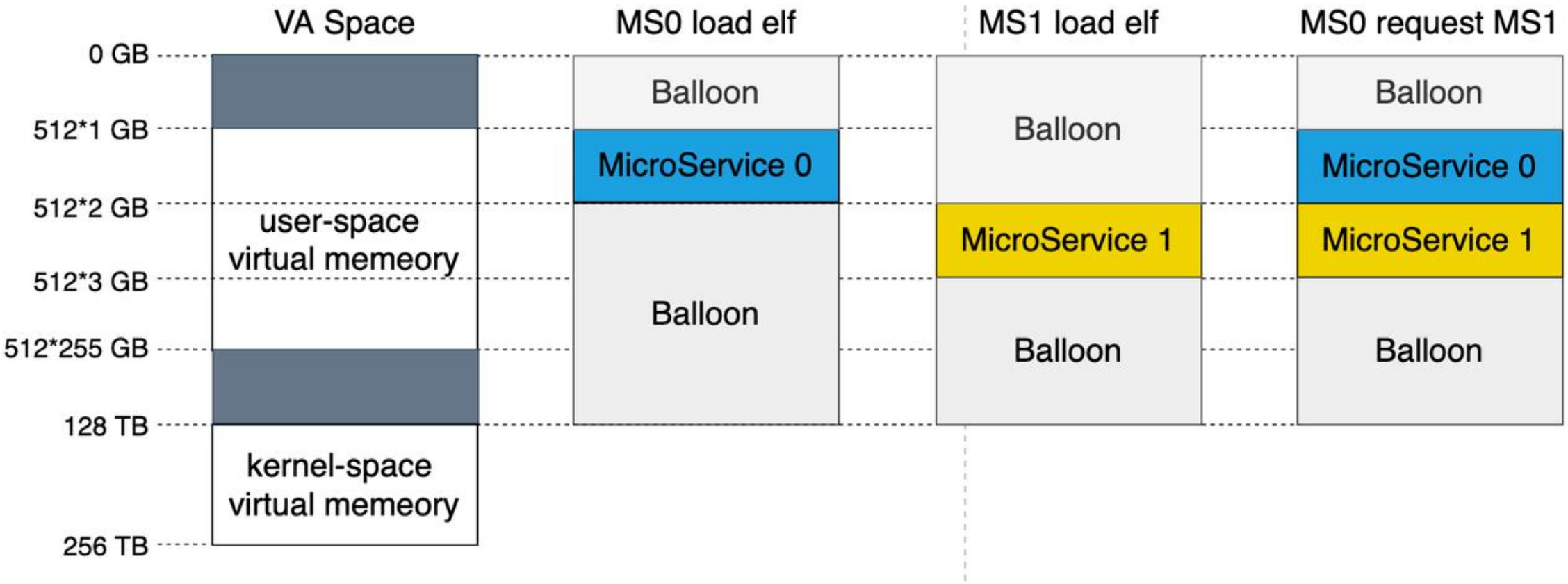
全进程地址空间共享与保护

全进程地址空间共享与保护

模拟插件/动态链接库等方案的用户态上下文切换和虚拟地址访问，需要解决：

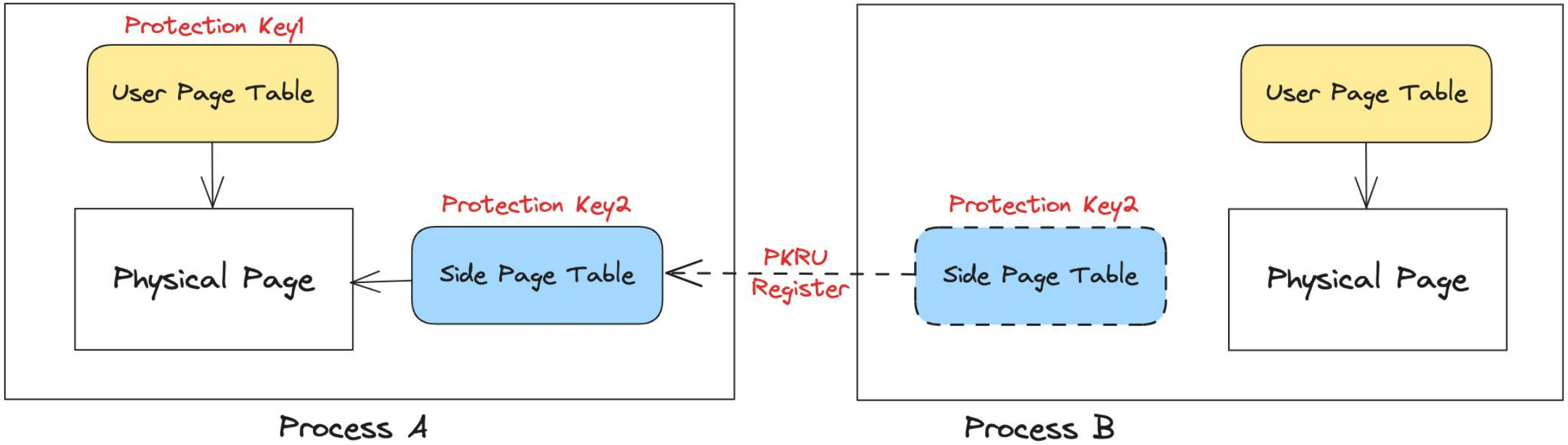
1. 虚拟地址冲突问题；
2. 页表隔离问题；
3. 内存安全性问题；

全进程地址空间共享与保护



地址空间气泡方案

全进程地址空间共享与保护



1. Intel x86 中，每个 leaf page table 的页表项的第59–62位称为 Protection Key，这 4 bits 可以将页表项划分为 16 个域，从而可以给每一个域单独赋予一个权限；

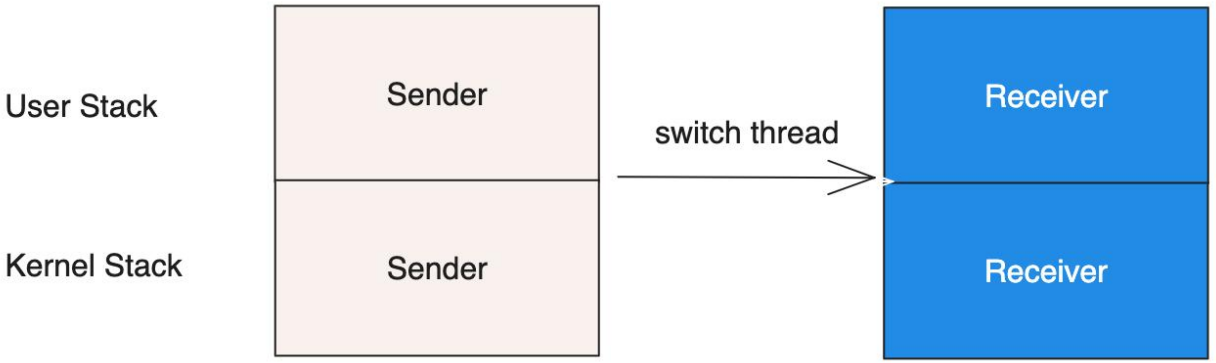
2. Intel x86 为每个线程提供了一个寄存器 PKRU (User Page Key Register)，其长度为 32 bits，每 2–bit 对应页表中的一个 Protection Key，分别为 WD 位和 AD 位，用于控制所在域的内存访问权限。

第三部分

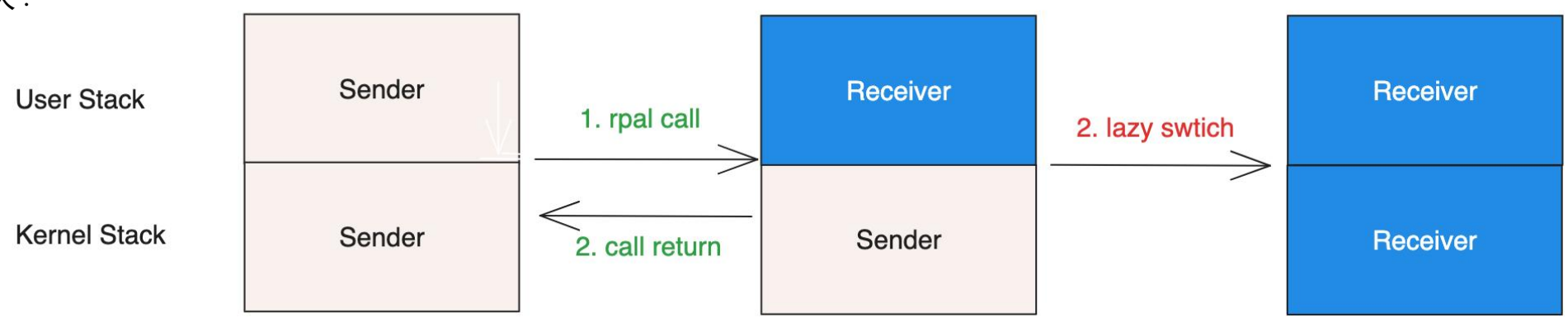
用户态进程切换

用户态进程切换

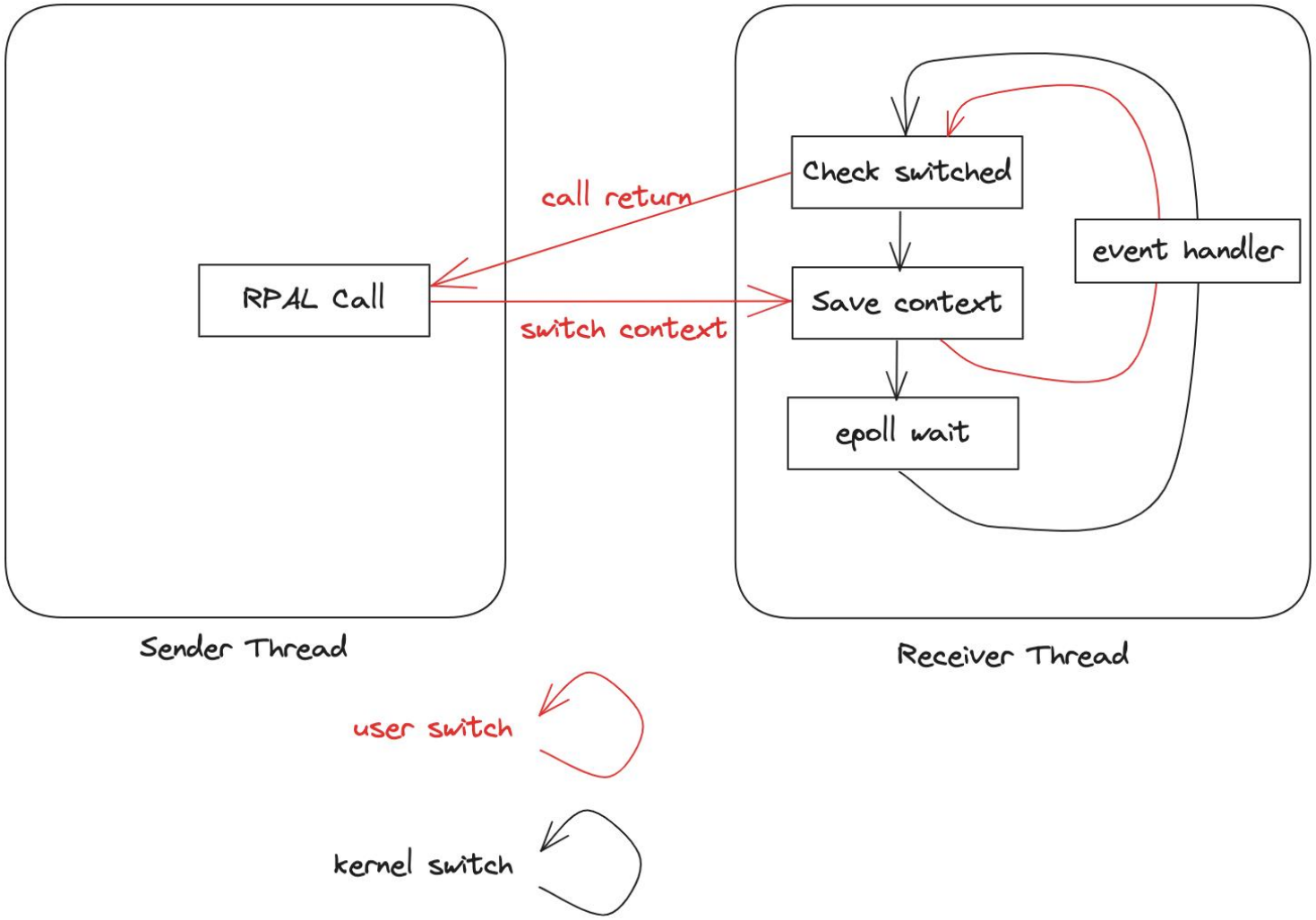
传统线程切换



rpal线程切换：



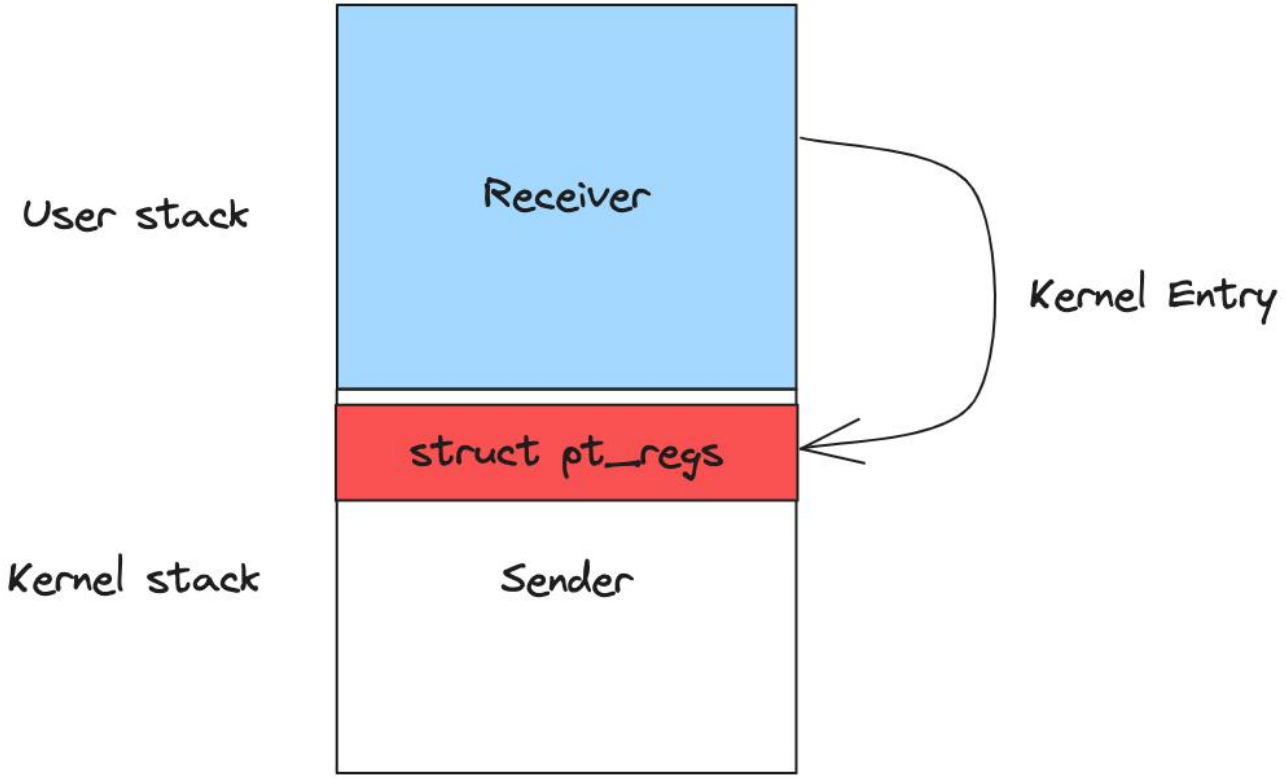
用户态进程切换


call return
RPAL Call
switch context
Check switched
Save context
epoll wait
event handler
Sender Thread
Receiver Thread
user switch
kernel switch

用户态进程切换

延迟进程切换

1.发生 Kernel Entry 时，sender 线程将 pt_regs（保存 Kernel 返回到用户态的上下文信息）压入 sender 线程内核栈

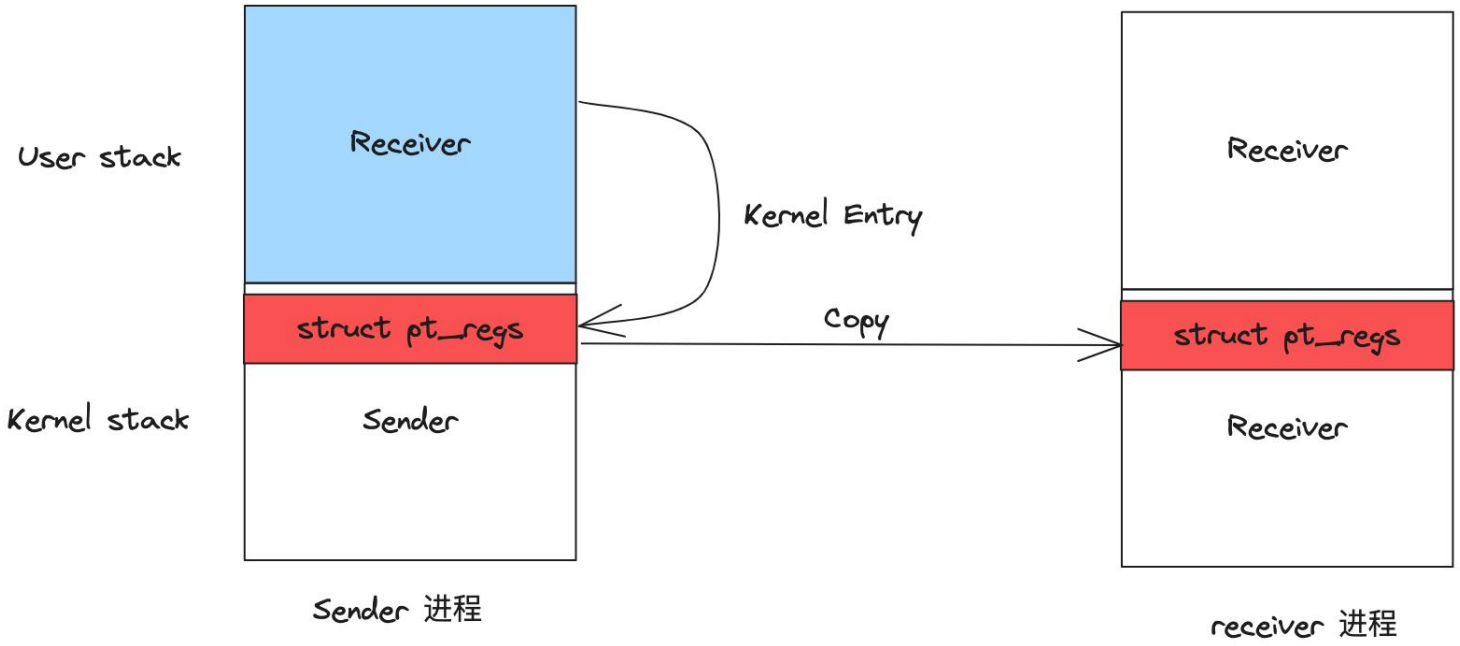


用户态进程切换

延迟进程切换

2．判断 fsbase 寄存器保存的地址是否在 kernel current task 的 512GB 地址空间内？

> 若不是，代表当前在RPAL Call，将 pt_regs 拷贝并覆盖掉之前处于 epoll_wait 上下文的 receiver 线程的内核栈中



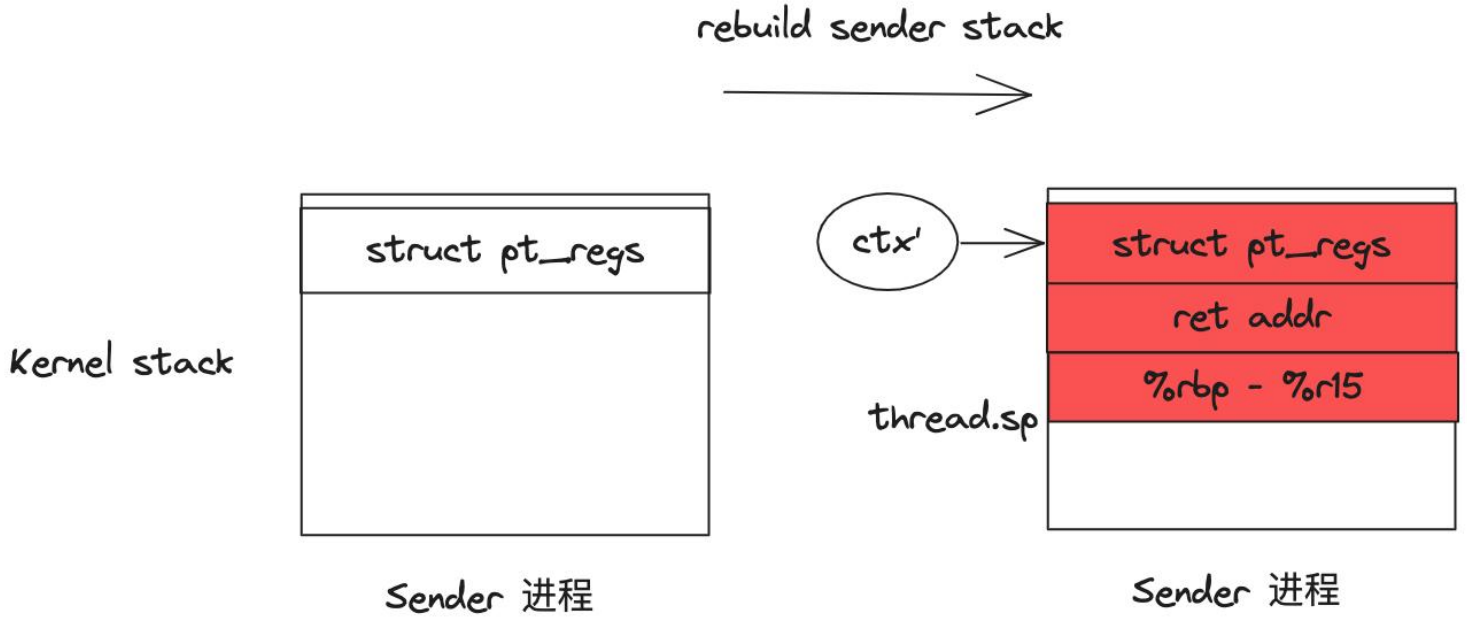
用户态进程切换

延迟进程切换（lazy switch）

用户态切换时还需要保留一个操作：

> sender线程将自身线程上下文拷贝一个副本，并允许kernel访问该副本。

3．sender 线程在 lazy_switch 过程中，恢复 receiver 线程的内核栈后，将保存好的 sender 线程上下文拷贝到 sender 线程内核栈 pt_regs 处内存。



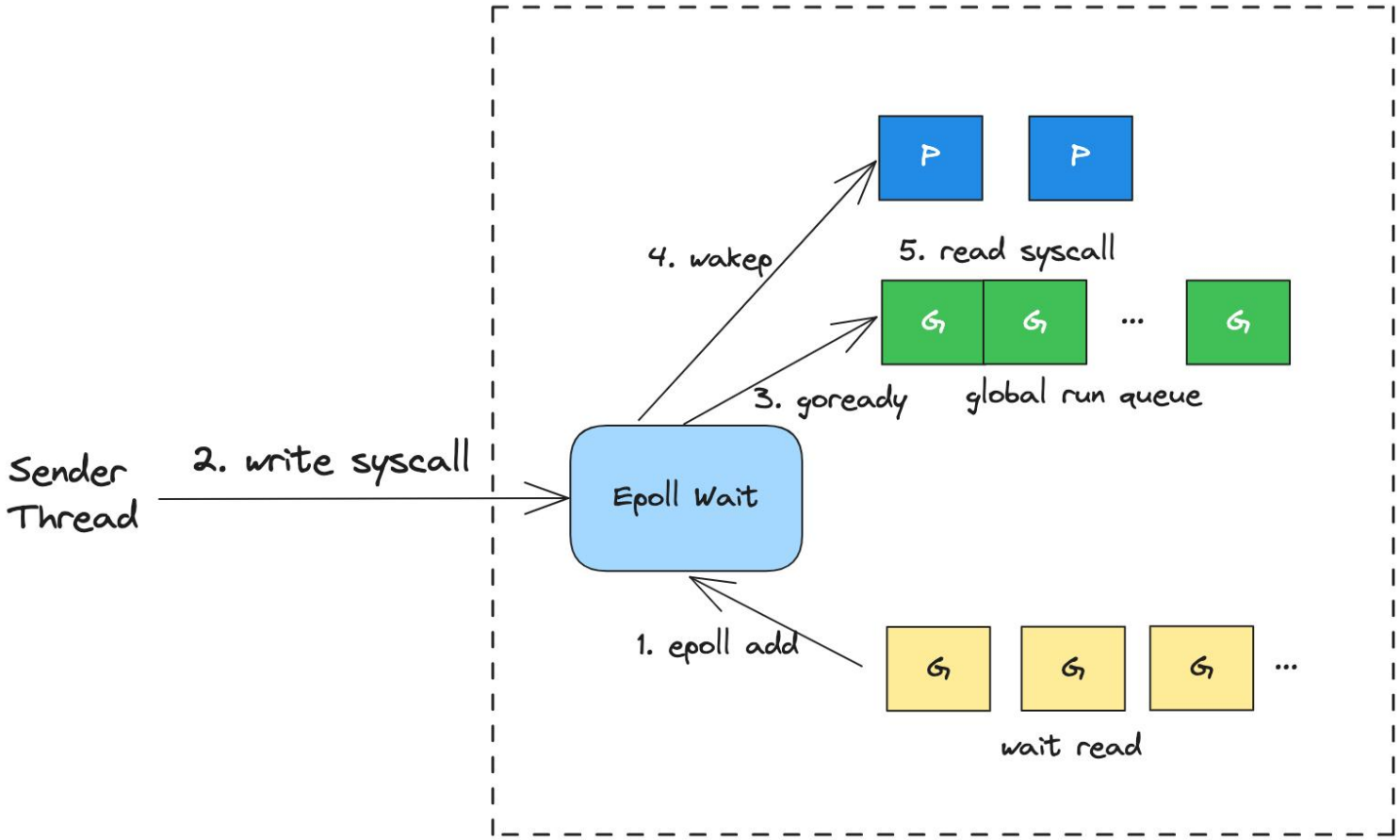
第四部分

高效的Go Event Poller

高效的Go Event Poller

Go 原生 epoll 模型

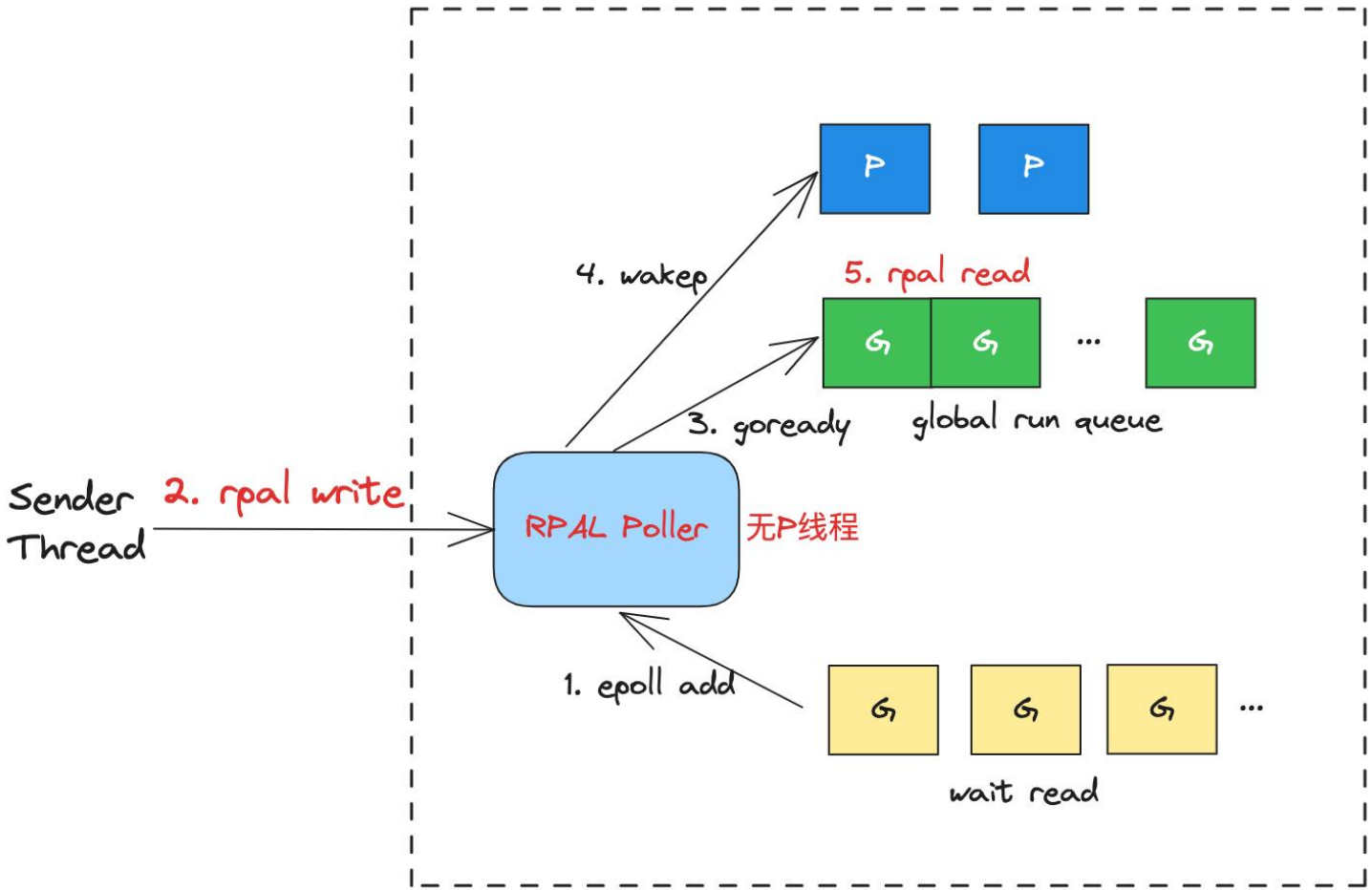
1. writev syscall
2. epoll wait
3. readv syscall
4. 可能还有 futex_wake



高效的Go Event Poller

纯用户态 poller 实现

1. ~~writev syscall~~
2. ~~epoll wait~~
3. ~~readv syscall~~
4. 仅在必要时调用 futex_wake（没有自旋线程 && 有 idle P）

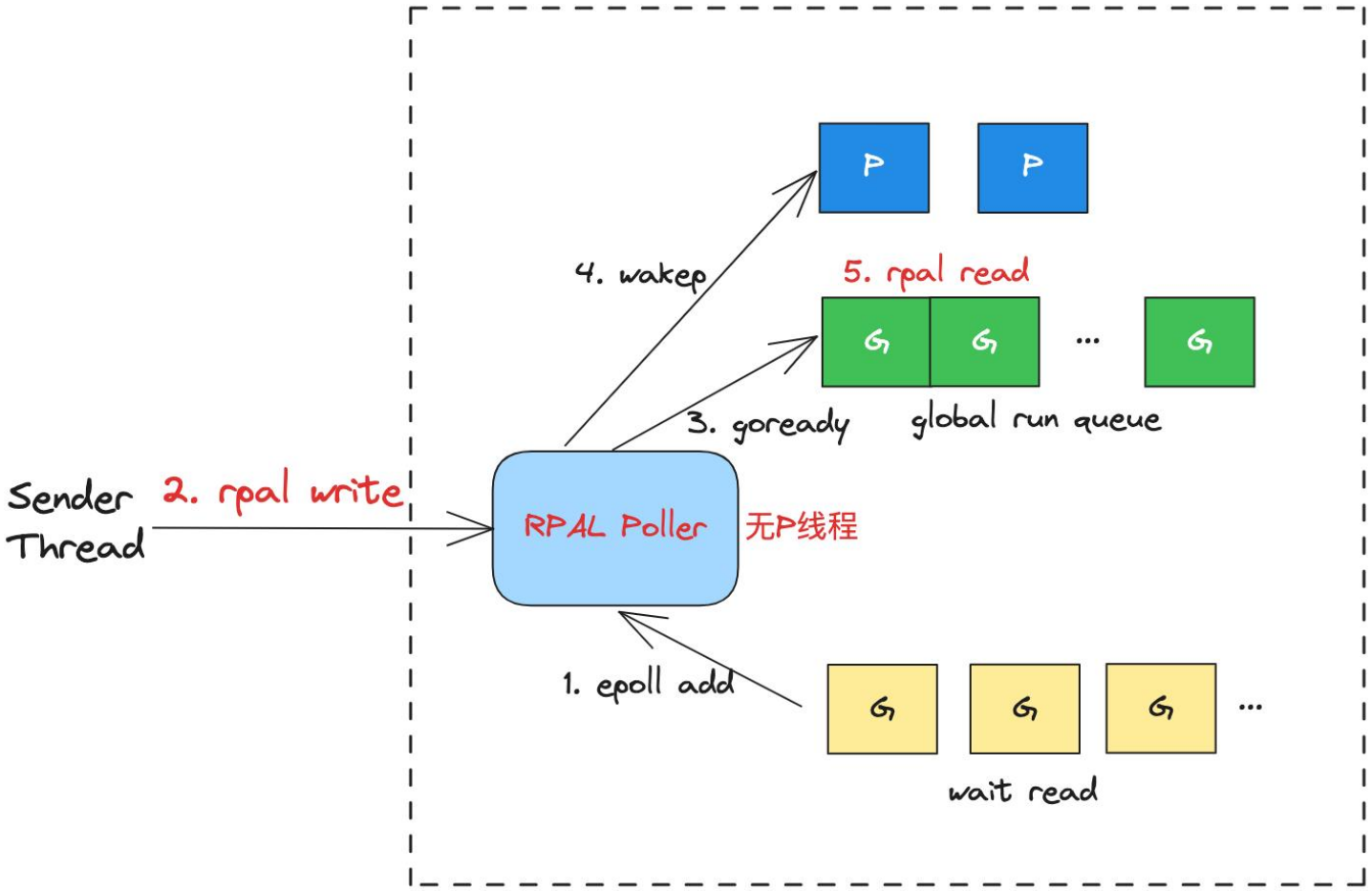


高效的Go Event Poller

思考：

1. 为什么要异步唤醒 M 处理？

2. 是否有同步的 Go 函数调用方案？



第五部分

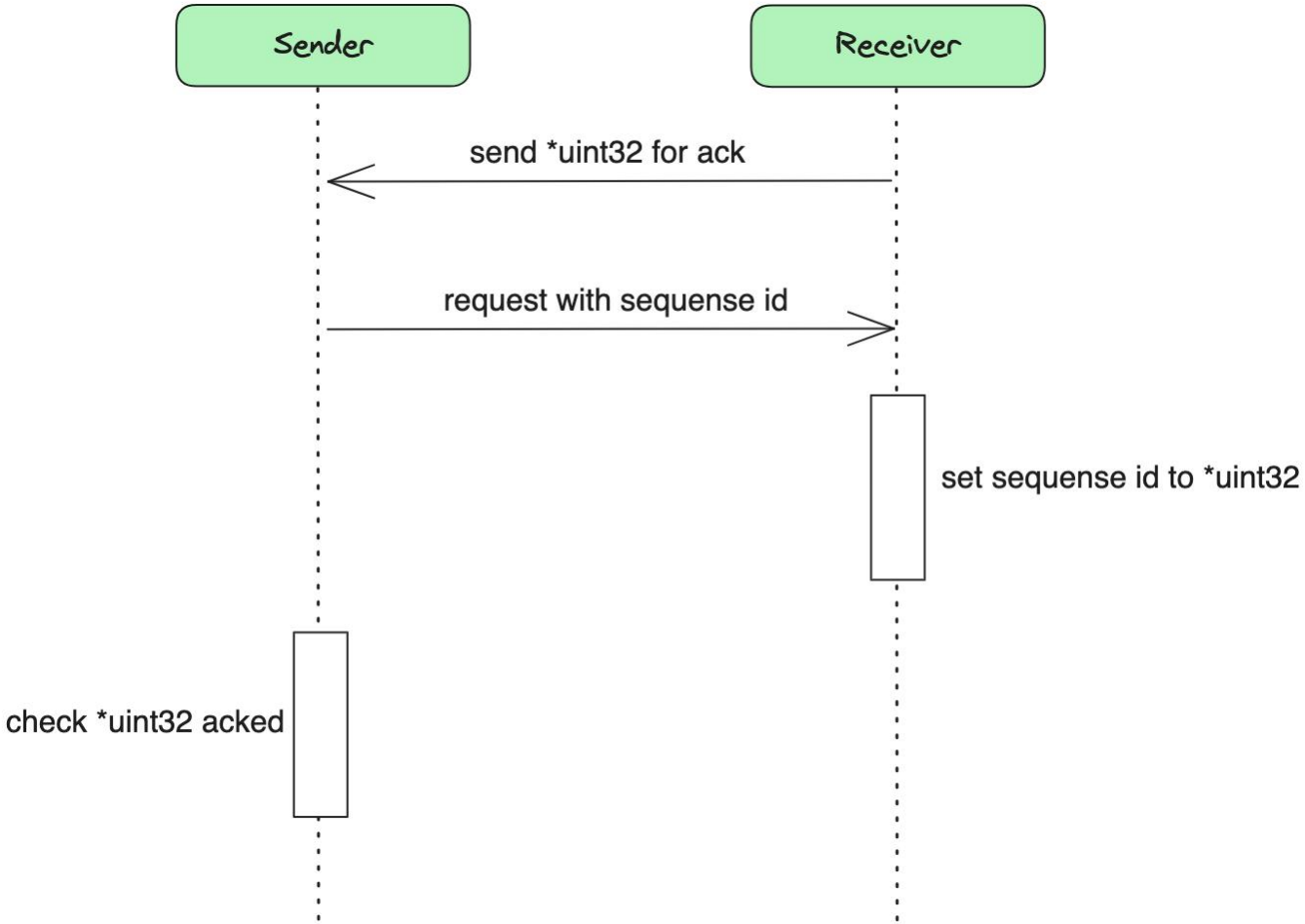
RPC 框架 Kitex 集成

RPC 框架 Kitex 集成

指针读写抽象接口：

```
type Conn interface {
    net.Conn
    RpalRead() (AckPointer, error)
    RpalPeek() (AckPointer, error)
    RpalWrite(p unsafe.Pointer) error
}

type AckPointer interface {
    Pointer() unsafe.Pointer
    Ack()
}
```

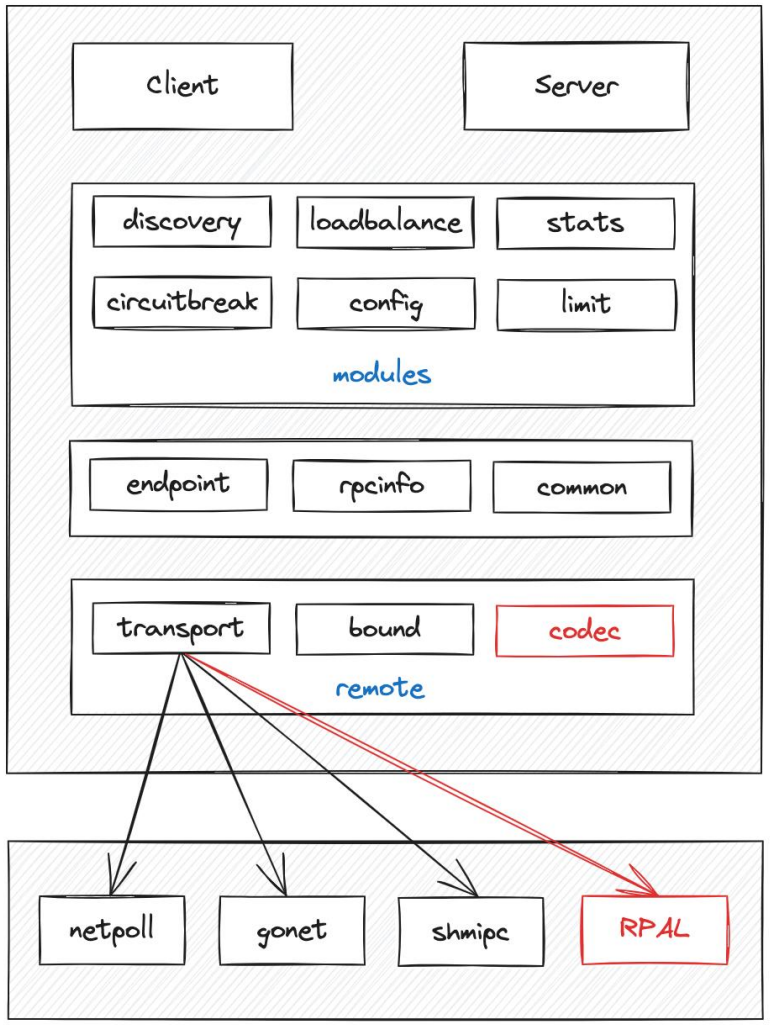


对象指针ACK

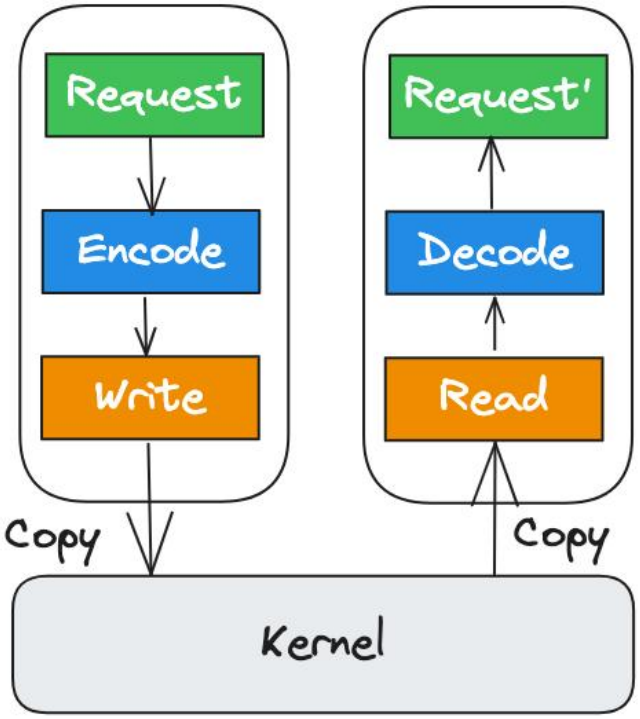
RPC 框架 Kitex 集成

新增 Transport：绕过 Kernel 传递指针

重写 Codec：绕过序列化/反序列化

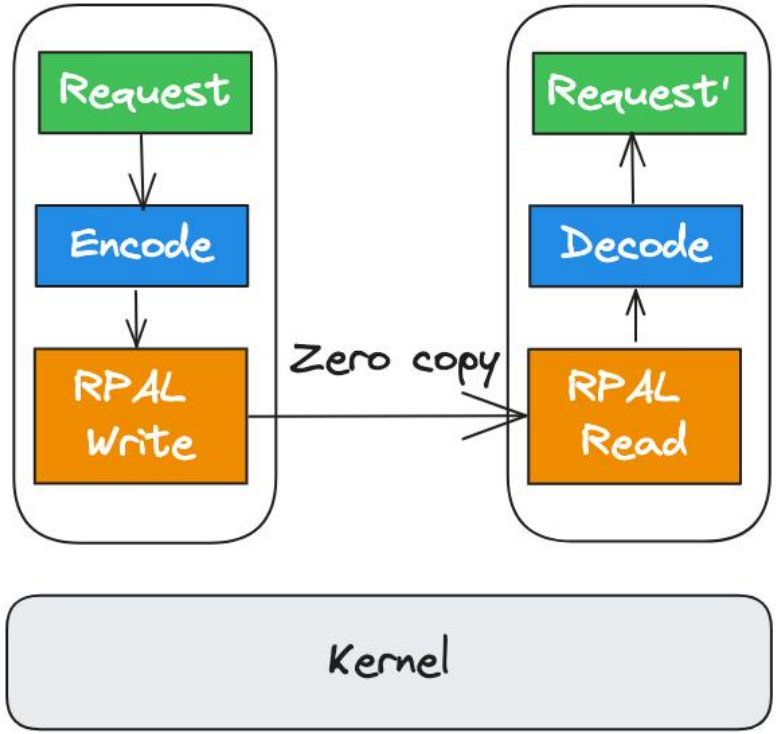


RPC 框架 Kitex 集成



传统 uds/tcp 读写

多次 syscall：epoll_wait/readv/writev



RPAL 读写

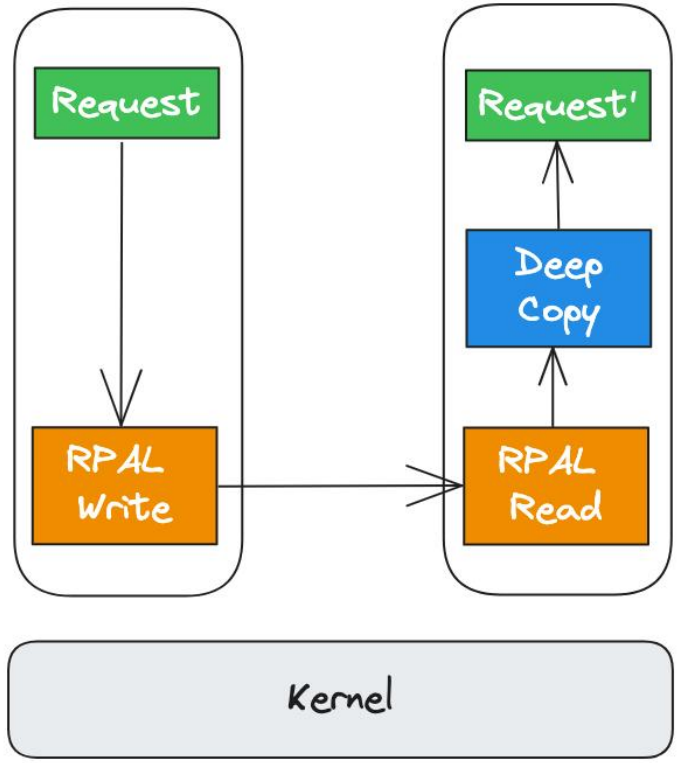
仅在必要时调用 wakep，全程无 syscall 无拷贝

RPC 框架 Kitex 集成

```go
type Client interface {
    Echo(ctx context.Context, req *echo.EchoRequest) (r *echo.EchoResponse, err error)
}
```

1. 一致的函数签名
2. 几乎一致的 IDL 定义

RPC 框架 Kitex 集成



使用深拷贝替代序列化/反序列化，两种方式：

1. 通过 RPC Method 进行类型断言（IDL 定义一致）；

2. 传递类型信息构造 reflection cache 加速深拷贝（IDL 定义可以不一致）

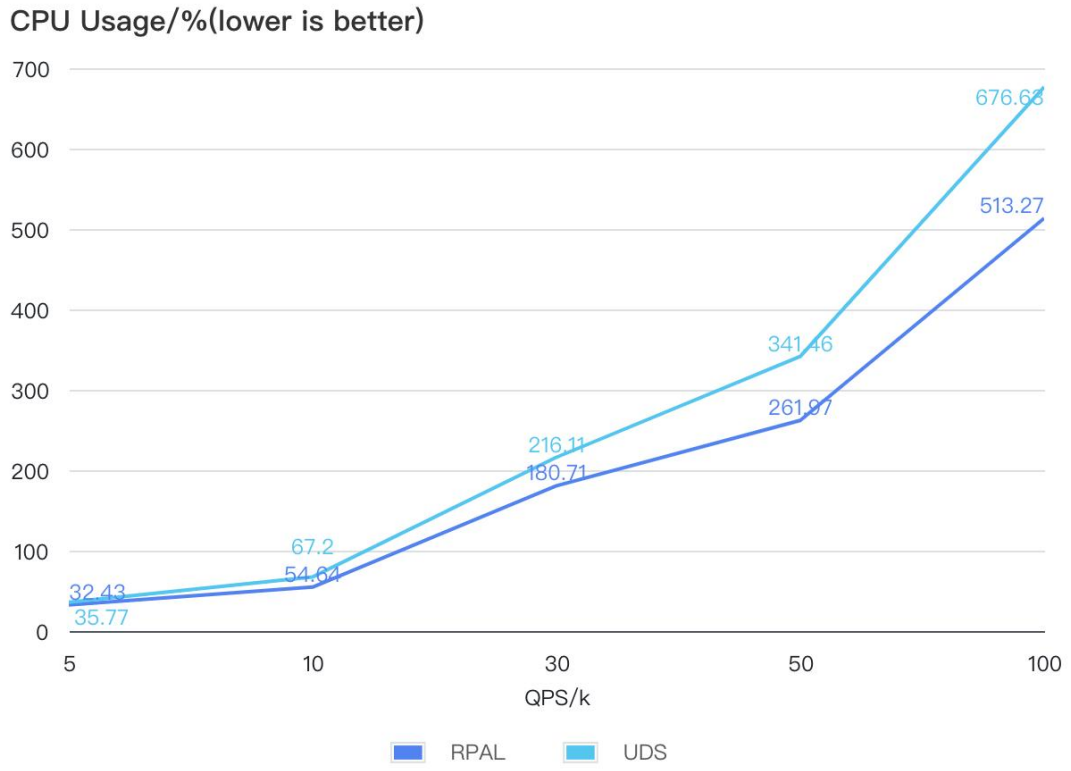
第六部分

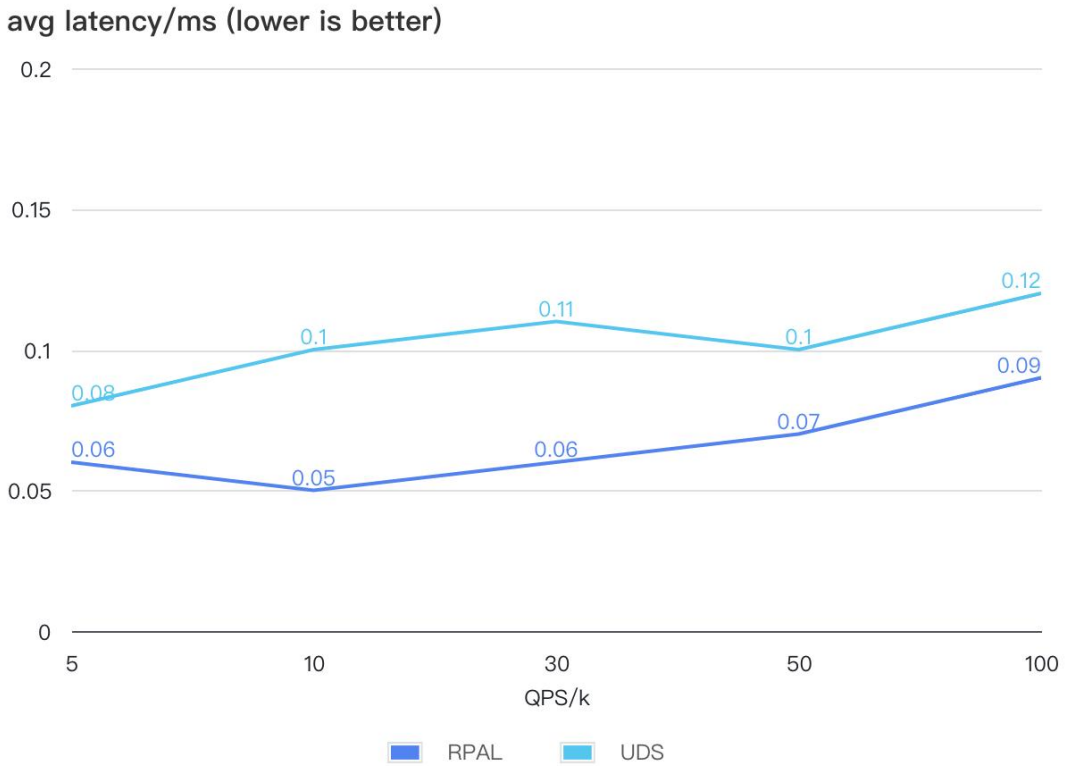
性能收益与业务展望

性能收益与业务展望

性能压测

1 kb 请求/响应下，以不同 QPS 在 Kitex 框架进行 benchmark 测试，对比 uds 和 rpal 的性能差异：





注：以上仅测试包含序列化开销的性能对比，benchmark测试受影响因素较多，实际收益需结合业务场景。

性能收益与业务展望

业务真实数据

1. 字节跳动微服务合并部署场景下，部分服务通过接入 RPAL 整体取得了 1—5% 的 CPU 收益，以及 RPC 链路 1—6ms 的 P99 延迟下降。

2. 将某项 Mesh 提供的治理功能进行同步 RPAL Call，对比同进程 Function Call 仅增加 200 ns 延迟。

性能收益与业务展望

业务展望

1. 定制化场景深度优化：

同步 RPAL Call；

请求/响应 Zero Copy；

2. 业务进程与服务网格 IPC 性能优化：

结合用户态协议栈，实现网络 IO 绕过内核

CloudWeGo 是一套由字节跳动基础架构**服务框架团队**开源的、可快速构建企业级云原生微服务架构 的中间件集合。

CloudWeGo 项目共同的特点是**高性能、高扩展性、高可靠**，专注于微服务通信与治理。

CloudWeGo 包括 Kitex、Hertz、Volo、Netpoll、Monoio、Sonic 等多个重点子项目，涵盖 Go 与 Rust 开发语言， 上至框架下至网络库、编解码库、序列化库均是自研，各个项目既可独立使用也可搭配使用，并围绕这些项目，构建了完整的上下游生态。

cloudwego公众号

Thank You!