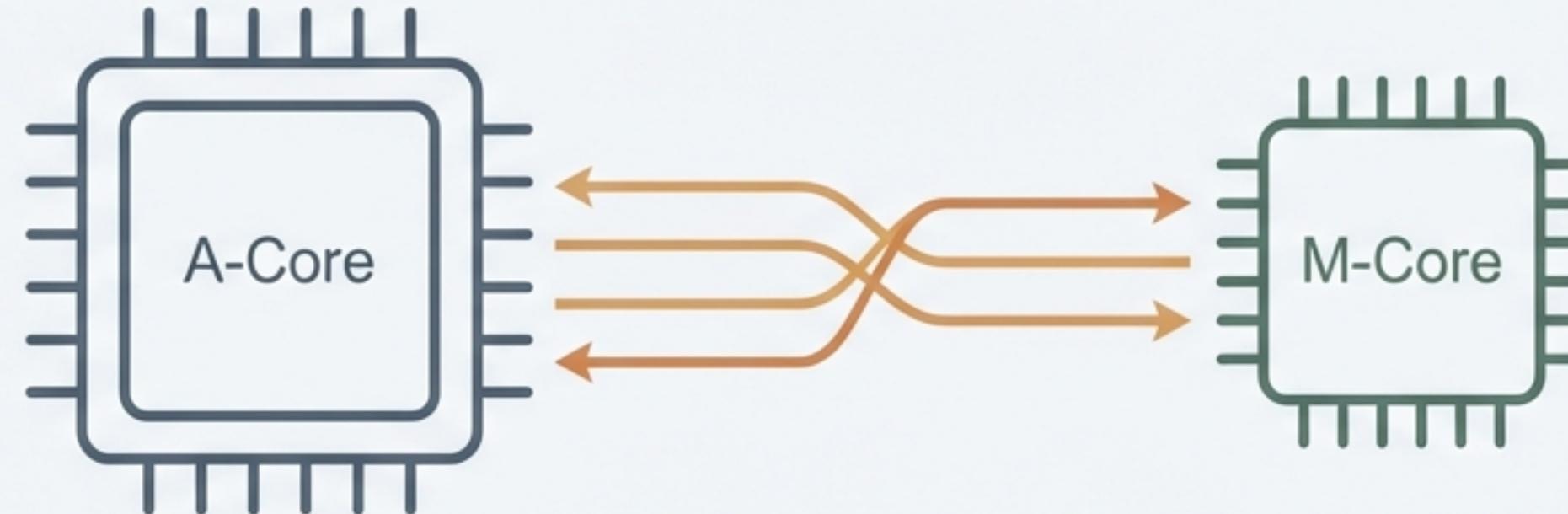


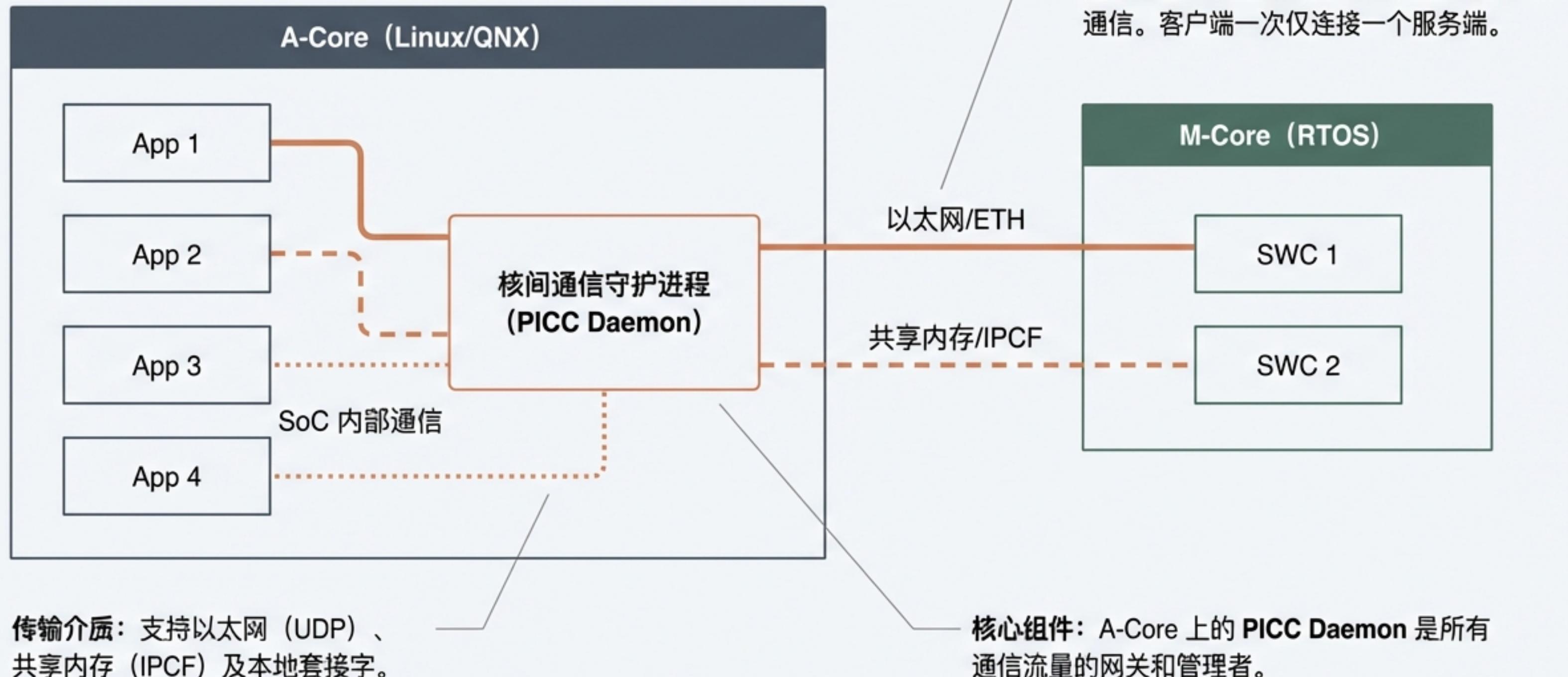
车云子系统核间通信 (Inter-Core Communication) 设计方案

为解决多核异构 SoC 内部的通信挑战，我们设计了一套高性能、高可靠的核间通信组件。



- **目标：**在 A-Core（应用处理器）与 M-Core（微控制器）之间建立标准化的数据交换通道。
- **设计原则：**采用面向服务 (SOA) 的通信语义，支持多种物理传输介质，并提供灵活的开发者 API。
- **适用场景：**覆盖从底层控制到上层应用的全方位车内通信需求。

系统架构：连接 A-Core 与 M-Core 的通信枢纽



通信的语言：基于 SOA 的服务语义

我们将复杂的通信抽象为两种核心的 SOA 服务类型：事件（Event）和方法（Method）。

事件（Event）



服务端（Server）向客户端（Client）的单向通知。

方法（Method）



客户端（Client）对服务端（Server）的请求/调用。

子类型 1：Notify With Ack (带确认的通知)

Server 发送通知，Client 中间件自动回复 ACK 报文以确认接收。应用层无需处理 ACK。



子类型 2：Notify Without Ack (无确认的通知)

Server 发送通知，Client 无需回复。



子类型

类型	客户端行为	服务端行为	核心特征
With Response	发送请求 (Request)	回复响应 (Response)	经典的请求-应答模式 (RR)
Without Response Without Ack	发送请求 (Request)	不响应，不确认	即发即忘模式 (FF)
Without Response With Ack	发送请求 (Request)	回复确认 (ACK)	确认请求已收到，但不返回业务数据

通信的语法：私有协议报文结构



- **ProviderID**: 服务提供者唯一标识。
- **MethodID**: 服务 (Event/Method) 的唯一标识，在 Provider 内唯一。
- **ConsumerID**: 服务消费者 (客户端) 唯一标识。
- **SessionID**: 用于匹配异步请求/响应和带 ACK 的通知。

- **MessageType**: 标识报文的具体类型 (如请求、响应、连接等)。
- **Length**: Payload 的长度 (2 字节)。
- **ReturnCode**: 响应或错误报文中的状态码。
- **Payload**: 应用层自定义的数据载荷。

协议核心要素：ID 与 Session

Provider/Consumer ID

作用：系统范围内的唯一标识符，由系统工程师根据功能域统一分配。

ID 范围	功能	M核占用	A核占用
01 - 10	电源管理	01 - 05	06 - 10
11 - 20	OTA	11 - 15	16 - 20
...
81 - 254	预留	-	-

Method ID

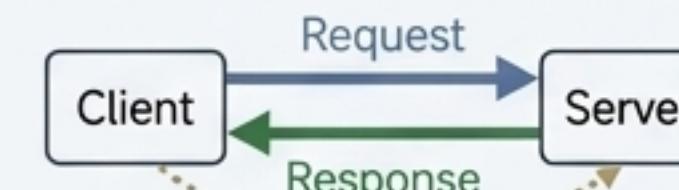
作用：标识一个 Provider 提供的具体服务。

关键规则：在同一个 Provider 内，Event ID 和 Method ID 必须唯一，不能重复。

Session ID

作用：核心的状态跟踪机制，范围 0x01~0xFF。

使用场景：



异步 Method 调用：用于精确匹配 Request 和后续的 Response 回调。



带 Ack 的 Event：用于匹配 Event 通知和对应的 Ack 报文。

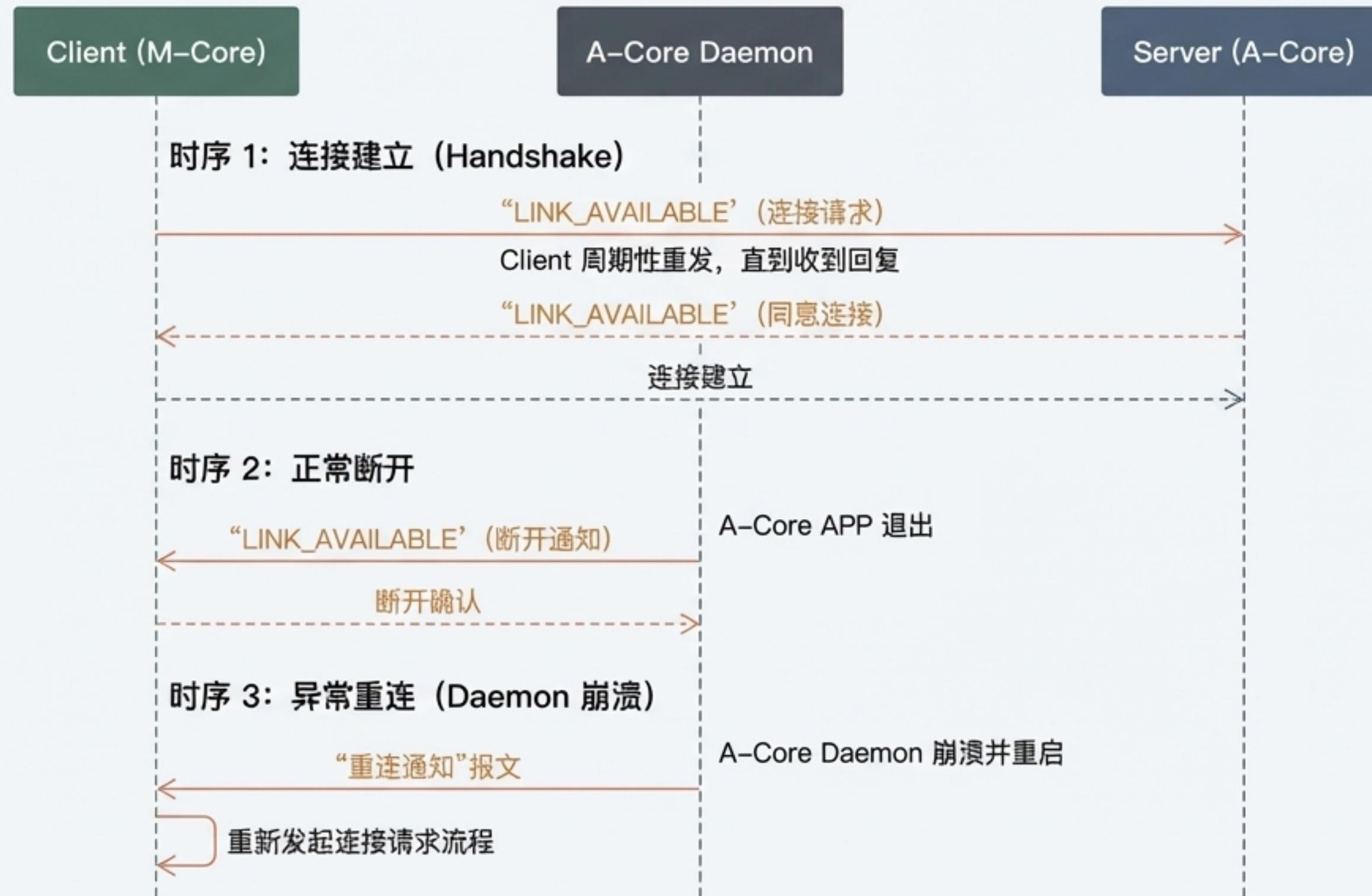
注解：在其他同步或无需回复的场景下，此字段为 0x00。

报文意图解析：MessageType 详解

值	名称	分类	描述
0x00	`LINK_AVAILABLE`	连接管理	用于建立和断开连接的握手报文。
0x99	`ERROR`	连接管理	内部错误通知，如 ID 重复、IPCF 发送失败。
0x03	`SUBSCRIBE_SERVICE`	服务管理	(A-Core 内部) APP 向 Daemon 订阅 Event。
0x04	`STOP_SUBSCRIBE_SERVICE`	服务管理	(A-Core 内部) APP 取消订阅 Event。
0x05	`REQUEST`	Method 调用	对应 `Method With Response`。
0x06	`REQUEST_NO_RETURN_WITH_ACK`	Method 调用	对应 `Method Without Response With Ack`。
0x07	`REQUEST_NO_RETURN_WITHOUT_ACK`	Method 调用	对应 `Method Without Response Without Ack`。
0x08	`NOTIFICATION_WITH_ACK`	Event 通知	对应 `Notify With Ack`。
0x09	`NOTIFICATION_WITHOUT_ACK`	Event 通知	对应 `Notify Without Ack`。
0x80	`RESPONSE`	响应	对 `REQUEST` 的业务数据回复。
0x81	`ACK`	响应	对 `REQUEST_NO_RETURN_WITH_ACK` 的中间件确认。
0x82	`EVENT_ACK`	响应	对 `NOTIFICATION_WITH_ACK` 的中间件确认。

注解：此表格是理解所有通信时序和交互逻辑的关键。

交互流程 (1) : 连接生命周期管理



交互流程 (2) : 一次完整的 Method 调用

场景描述: A-Core (Client, ID: 0xd2) 调用 M-Core (Server, ID: 0xce) 提供的 Method (ID: 0x03) , 类型为 `REQUEST` (带响应)。



真实日志摘录 (Log Example)

请求报文:

ce 03 d2 00 05 00 02 xxxx

ProviderID=0xce

MessageType=0x05 (REQUEST)

Payload Length=2

ConsumerID=0xd2

响应报文:

ce 03 d2 00 80 00 02 yyyy

PayloadID=0x03

服务端返回数据

MessageType=0x80 (RESPONSE)

ReturnCode=0x00 (成功)

可靠性保障：心跳健康与数据完整性

心跳健康监测 (Heartbeat)

机制：A-Core 和 M-Core 之间在每个通道上双向周期性发送 Ping/Pong 报文。

流程：

- 一方发送 Ping。
- 对端立即回复 Pong。

异常处理：连续 3 次未收到对端的 Pong 报文，则认为该通道失效，并通知上层应用对端已断开。

周期：2 秒。

日志示例：

```
Ping: ff 00 ff 00 ff 00 00 01 00  
Pong: ff 00 ff 00 ff 00 00 01 01
```

数据完整性 (Counter & CRC)

机制：在 ETH/IPCF 通道上发送堆叠数据前，增加计数器和 CRC 校验。

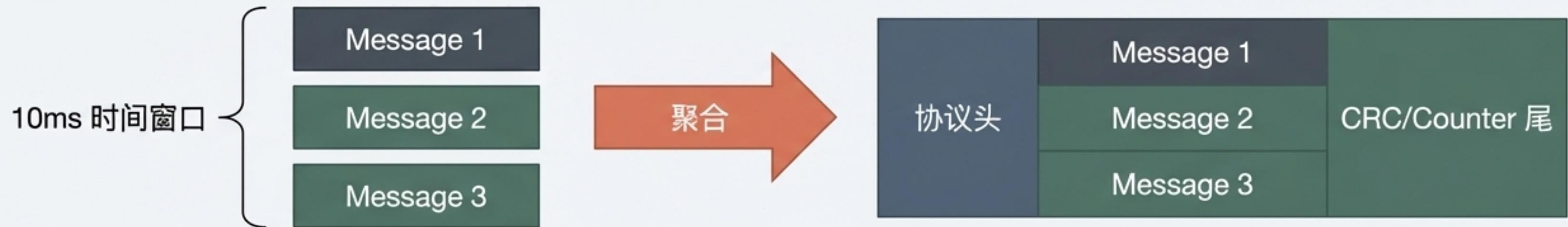
流程：

- **计数器 (Counter)**：使用 2 字节记录当前通道的发送次数。
- **CRC 校验**：对包括计数器在内的整个数据包进行 CRC16 校验。
- **封装**：将 2 字节的 CRC 校验码附加在数据包末尾。



性能优化：消息堆叠机制

为了降低系统调用（发送/读取接口）的频率，并提高网络或总线利用率，我们对短时间内产生的多条消息进行打包（堆叠），然后单次发送。



触发条件：

- 时间阈值：达到 10ms 发送周期。
- 大小阈值：
 - ETH 通道：堆叠后的消息大小接近 MTU。
 - IPCF 通道：堆叠后的消息大小接近 IPCF 驱动配置的最大 Buffer。

优势：

- 减少中断和上下文切换。
- 提高吞吐量。

开发者视角：A-Core API 编程模型

A-Core 提供了同步和异步两种 API 模式，以适应不同的应用场景需求。

关键区别：由于实时操作系统（RTOS）的特性，M-Core 的任务不能被长时间阻塞，因此其 API 行为本质上都是异步的。A-Core 运行在功能更强大的操作系统上，开发者可以根据需求选择合适的模

同步 API (Synchronous)

描述：API 调用会阻塞当前线程，直到收到对端的响应（Response 或 ACK）或超时。

- **优点：**编程模型简单，逻辑线性。
- **缺点：**可能阻塞线程，不适用于对实时性要求高的 UI 线程或主循环。

线程安全：同步发送函数非线程安全。

异步 API (Asynchronous)

描述：API 调用后立即返回，通过注册回调函数来处理对端的响应。

- **优点：**非阻塞，性能高，适用于高并发场景。
- **缺点：**编程模型稍复杂，需要处理回调逻辑。

线程安全：异步发送函数线程安全，可多线程调用。

A-Core API 详解 (1) : 同步调用

同步 API 适用于逻辑简单、需要立即获取结果的场景。

代码示例 1:
发送 Event (带 Ack)

```
int EventSend(uint8_t event_id, uint8_t consumer_id,  
             const vector<uint8_t>& tx_buffer,  
             MethodType en_method_type = EN_notificationWithAck,  
             uint16_t wait_millisecond = 500);
```

设为 EN_notificationWithAck
时，函数将同步等待 ACK。

代码示例 2:
调用 Method
(带 Response)

```
int MethodCall(uint8_t remote_id, uint8_t method_id,  
              const std::vector<uint8_t>& tx_buffer,  
              ReturnCode& return_code,  
              std::vector<uint8_t>& rx_buffer,  
              MethodType type = EN_methodWithResponse,  
              uint16_t u16_wait_milliseconds = 500);
```

[出参] 用于接收服务端返回的
Response 数据。

[出参] 服务端的处理结果状态码
(0x00 代表成功)。

设为 EN_methodWithResponse 或
EN_methodWithoutResponseWith
Ack 时，函数会同步等待。

A-Core API 详解 (2) : 异步调用

异步调用通过“请求-回调”机制实现，并使用 Session ID 关联请求与响应。



注册回调函数：

```
// 回调函数类型定义
typedef std::function<void(..., uint8_t session_id)>
AsynMethodResponseCallbackFunction;

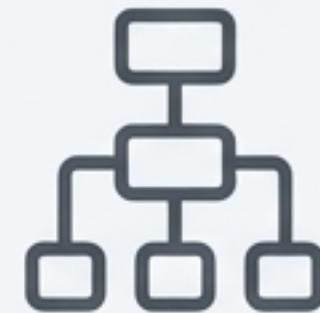
// 注册接口
int RegisterAsynMethodResponseHandler(AsynMethodResponseCallbackFunction callback);
```

发起异步请求：

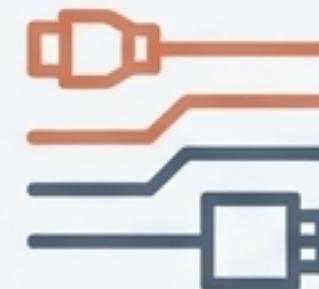
```
// 异步请求接口
int AsynMethodCall(uint8_t remote_id, uint8_t method_id,
const std::vector<uint8_t>& tx_buffer,
uint8_t& session_id, // [出参] 关键的 Session ID
MethodType type = EN_methodWithResponse);
```

设计总结：一个灵活、健壮、高效的通信基石

核心特性回顾



标准化的 SOA 语义：
基于 Event 和 Method，提供清晰的交互模型。



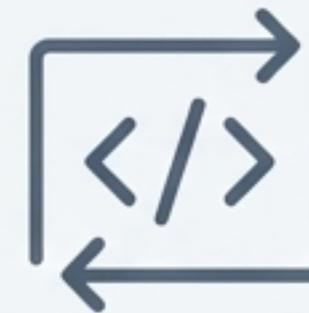
多介质支持：灵活适配以太网、共享内存等不同物理通道。



端到端可靠性：通过连接管理、心跳、CRC校验确保通信的稳定与数据完整。



高性能设计：消息堆叠机制有效提升了传输效率和系统性能。



开发者友好的 API：提供同步/异步两种模式，满足不同应用场景的开发需求。

设计哲学

分层与抽象

将复杂的底层传输细节与上层业务逻辑解耦。

明确的协议

私有协议结构清晰，字段定义明确，易于调试和扩展。

兼顾实时与非实时

API 设计充分考虑了 M-Core 的实时性约束和 A-Core 的灵活性需求。