讲师介绍--专业来自专注和实力

Darren老师

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。

重点内容

1. Tars微服务框架
2. Tars开发部署常见问题
3. Tars RPC原理

1 Tars微服务框架

1.1 Tars文档和文章
1.2 Tars的解决什么问题
1.3 TARS目标
1.4 TARS 设计思路
1.5 部署框架
1.6 服务主要交互流程
1.7 后台代码结构(C++)
1.8 TARS协议
1.9 代码自动生成
1.10 开发步骤
1.11 轻量化
1.12 负载均衡
1.13 容错
1.14 超时切换——(降低网络影响)
1.15 过载——
1.16 GITHUB——目录结构

1.1 文档

官方文档：
https://newdoc.tarsyun.com/#/markdown/TarsCloud/TarsDocs/README.md

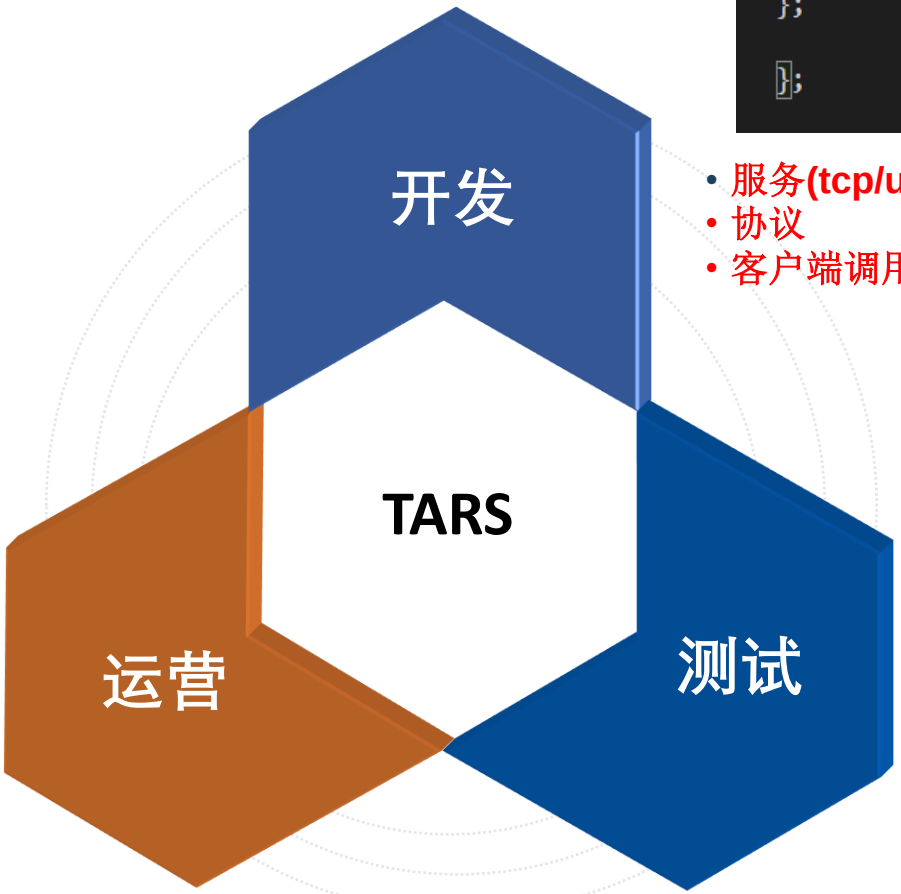
Github文档：https://tarscloud.github.io/TarsDocs/SUMMARY.html

Tars基金会文章： https://segmentfault.com/u/tarsfoundation

1.2 Tars的解决什么问题

```
module TestApp
{

interface Hello
{
    int test();
    int testHello(string sReq, out string sRsp);
};

};
```

TarsdocGitHub在线文档 (tarsyun.com)



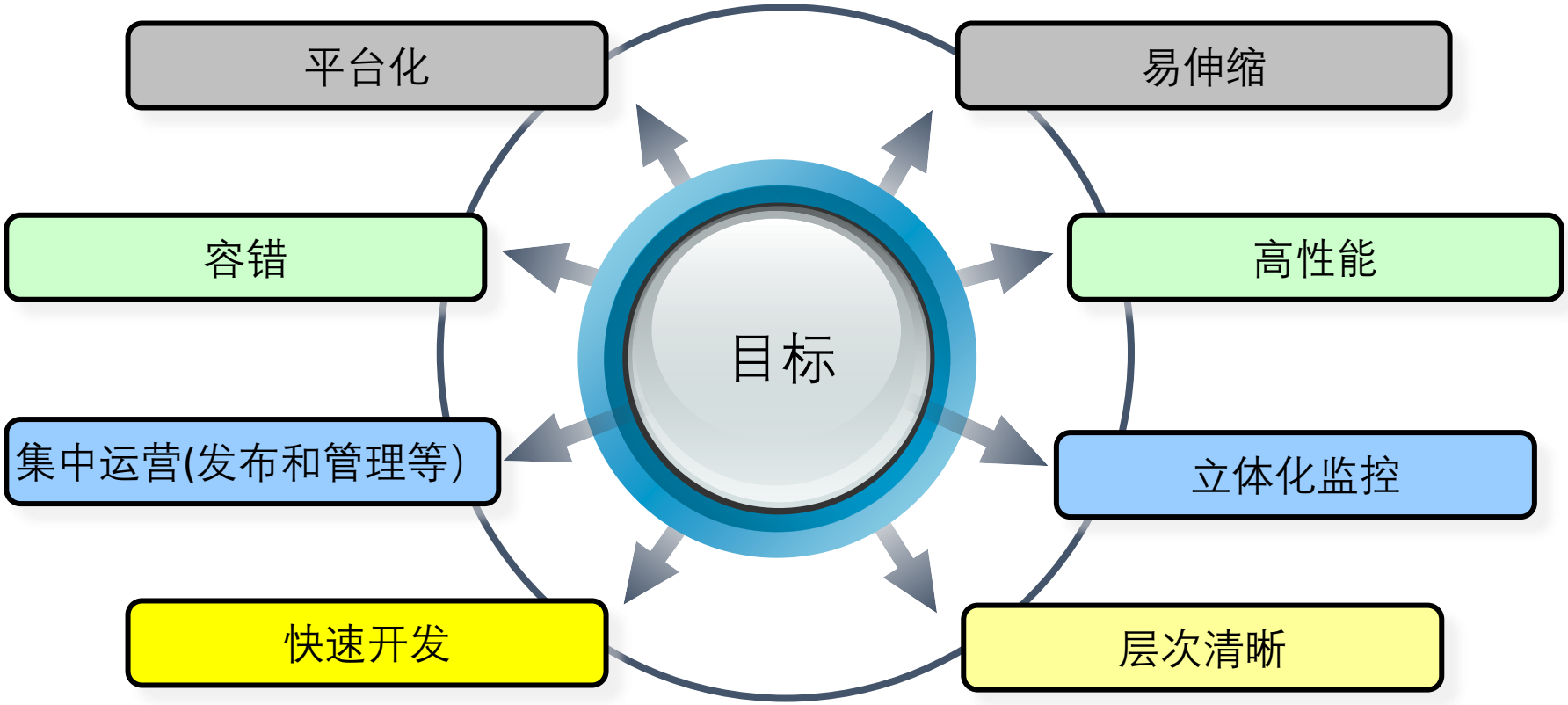
- 服务(tcp/udp, epoll/select)
- 协议
- 客户端调用(同步,异步)

- 容错/容灾
- 部署/发布
- 监控(异常，流量等)
- 集中日志/集中配置
- 服务管理(启停等)

- 接口级测试
- 集成测试

业务统一开发、运营、监控框架, 提升研发效率

1.3 TARS目标



1.4 TARS 设计思路

透明部署 自动发布 集中配置/LOG 调度分析

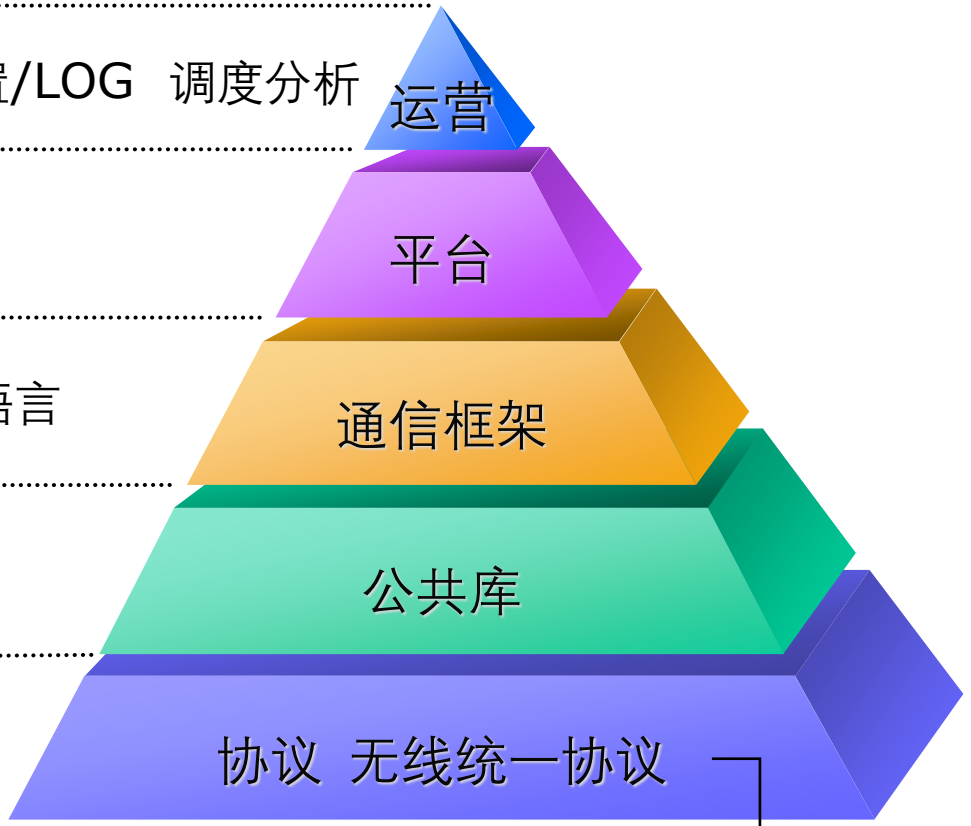
管理 容错 负载均衡 灰度

RPC 高性能 稳定 过载 多语言

框架以及业务使用

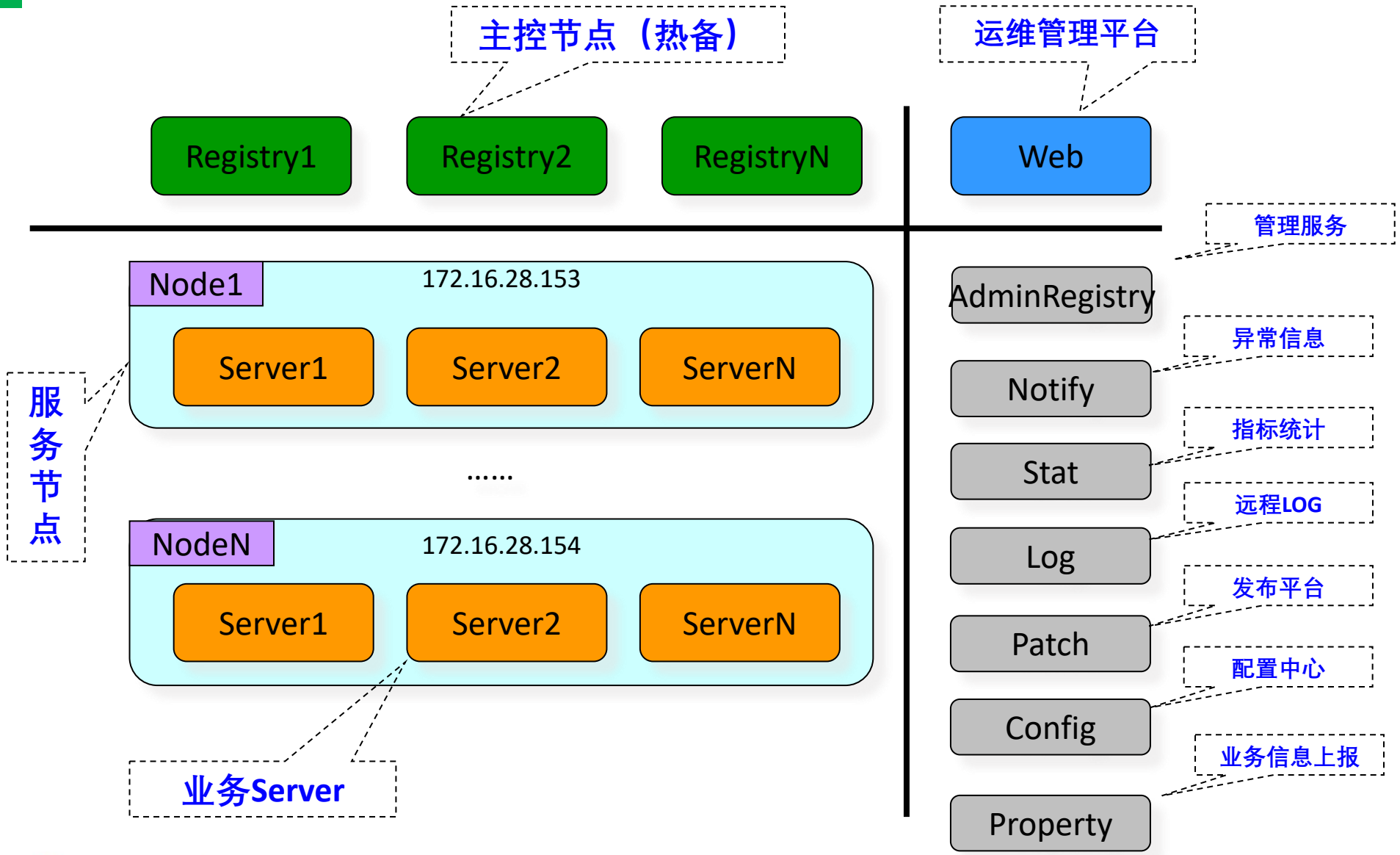
可扩展 自动生成 多语言
RPC protobuf

.proto文件
.tars文件

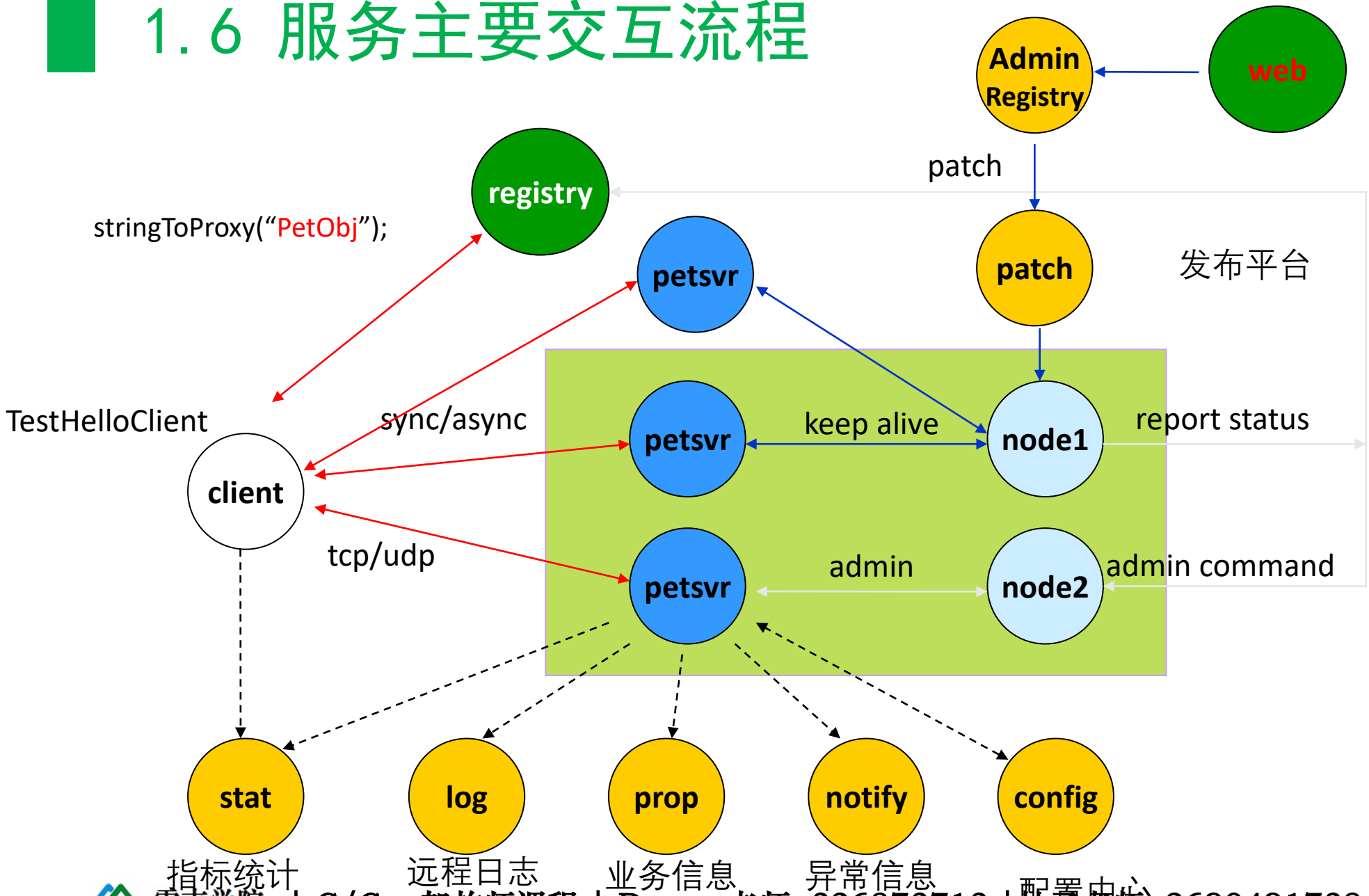


打通终端到后台以及后台之前的协议

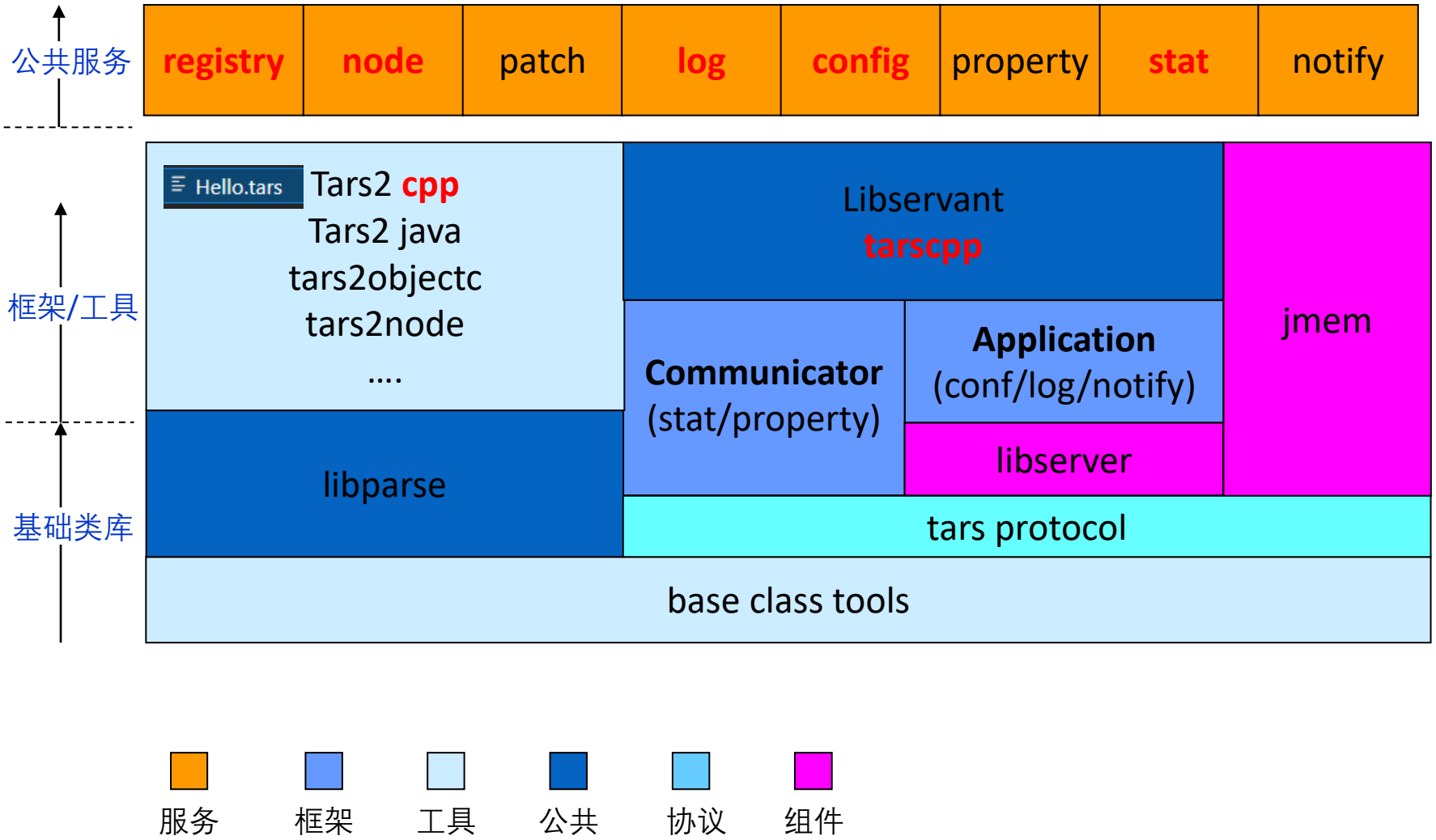
1.5 部署框架


主控节点（热备）
运维管理平台
Registry1
Registry2
RegistryN
Web
管理服务
Node1
172.16.28.153
Server1
Server2
ServerN
AdminRegistry
异常信息
Notify
指标统计
Stat
远程LOG
Log
发布平台
Patch
配置中心
Config
业务信息上报
Property
服务节点
.....
NodeN
172.16.28.154
Server1
Server2
ServerN
业务Server

1.6 服务主要交互流程


Admin
Registry
web
patch
registry
stringToProxy(“PetObj”);
petsvr
patch
发布平台
TestHelloClient
sync/async
petsvr
keep alive
node1
report status
client
tcp/udp
petsvr
admin
node2
admin command
stat
log
prop
notify
config
指标统计
远程日志
业务信息
异常信息
配置中心

1.7 后台代码结构(C++)


公共服务
registry
node
patch
log
config
property
stat
notify
框架/工具
Hello.tars
Tars2 cpp
Tars2 java
tars2objectc
tars2node
....
Libservant
tarscpp
jmem
Communicator
(stat/property)
Application
(conf/log/notify)
libserver
基础类库
libparse
tars protocol
base class tools
服务
框架
工具
公共
协议
组件

1.8 TARS协议

tag

可选字段

必选字段

```
struct TestInfo
{
    1  require  vector<string> vs;
    2  require  map<string, string> m;
    3  optional vector<map<string, string>> vm;
    4  require  map<vector<string>, vector<string>> mv;
    5  optional bool   b   = true;
    7  optional short  si  = 0;
    6  optional byte   by  = 0;
    8  optional int    ii  = 34;
    9  optional long   li  = 0;
    10 optional float  f   = 45.34f;
    11 optional double d   = 0;
    12 optional string s   = "a\"bc";
    13 require  ETest t;
    14 optional map<int,string> mi;
};
```

序列化

反序列化

```
JceOutputStream<BufferWriter> os;
ti.writeTo(os);
```

```
JceInputStream<BufferReader> is;
is.setBuffer(os.getBuffer(), os.getLength());
tii.readFrom(is);
```

```
tars::TarsOutputStream<tars::BufferWriterVector> _os;
```

tars协议采用接口描述语言（Interface description language，缩写IDL）来实现，它是一种二进制、可扩展、代码自动生成、支持多平台的协议

1.9 代码自动生成

tars 文件

```
interface Pay
{
    int doPayCoin(UserInfo tUserInfo, bool mode, out string operId);
};
```

tars2cpp

/usr/local/tars/cpp/tools/tars2cpp Pay.tars

客户端

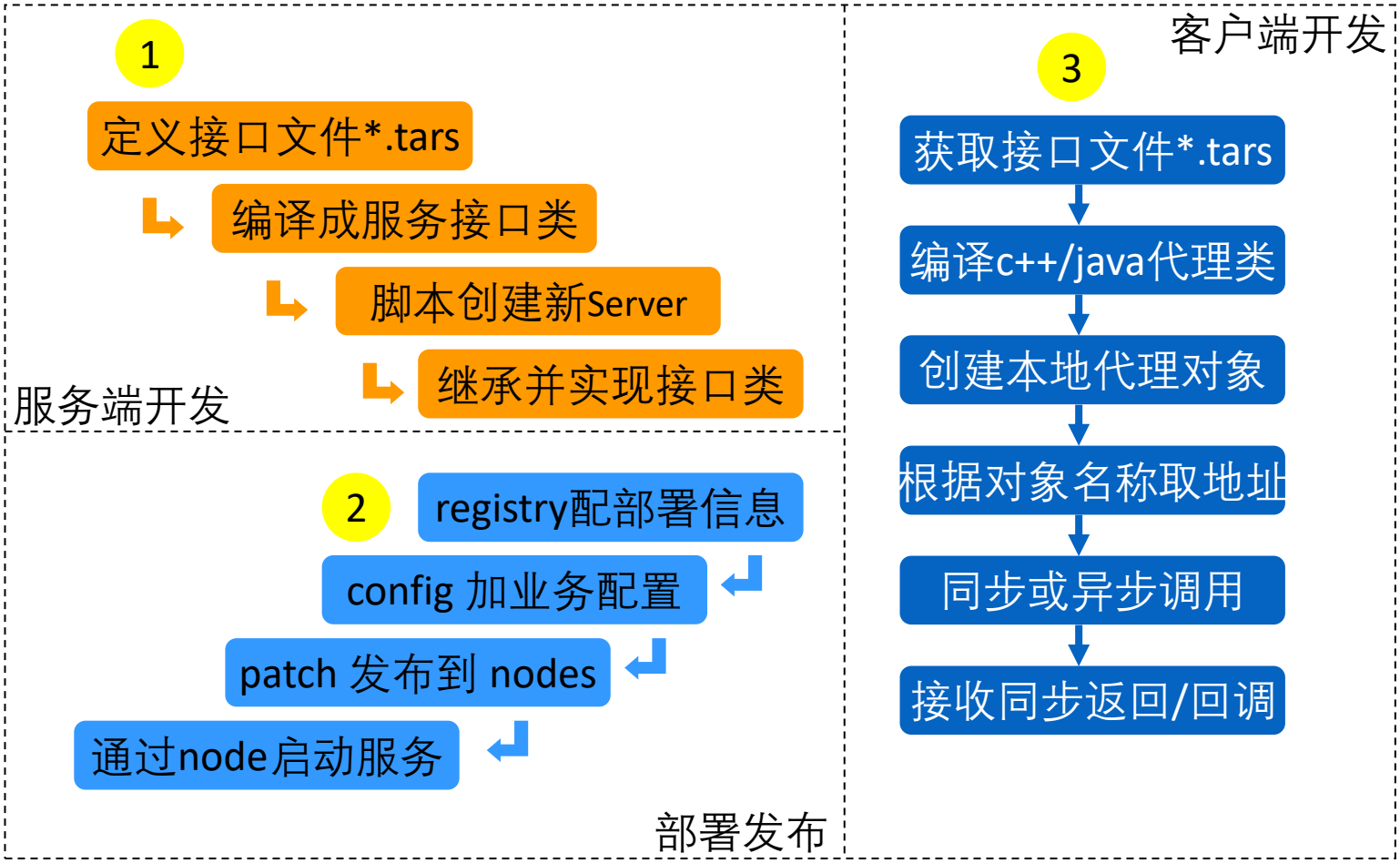
```
class PayProxy
{
    int doPayCoin(const UserInfo& tUserInfo,
                  bool mode, string& operId,
                  const map<string, string>& context);

    void async_doPayCoin(PayProxyCallbackPtr callback,
                         const UserInfo& tUserInfo,
                         bool mode, string& operId,
                         const map<string, string>& context);
};
```

服务端

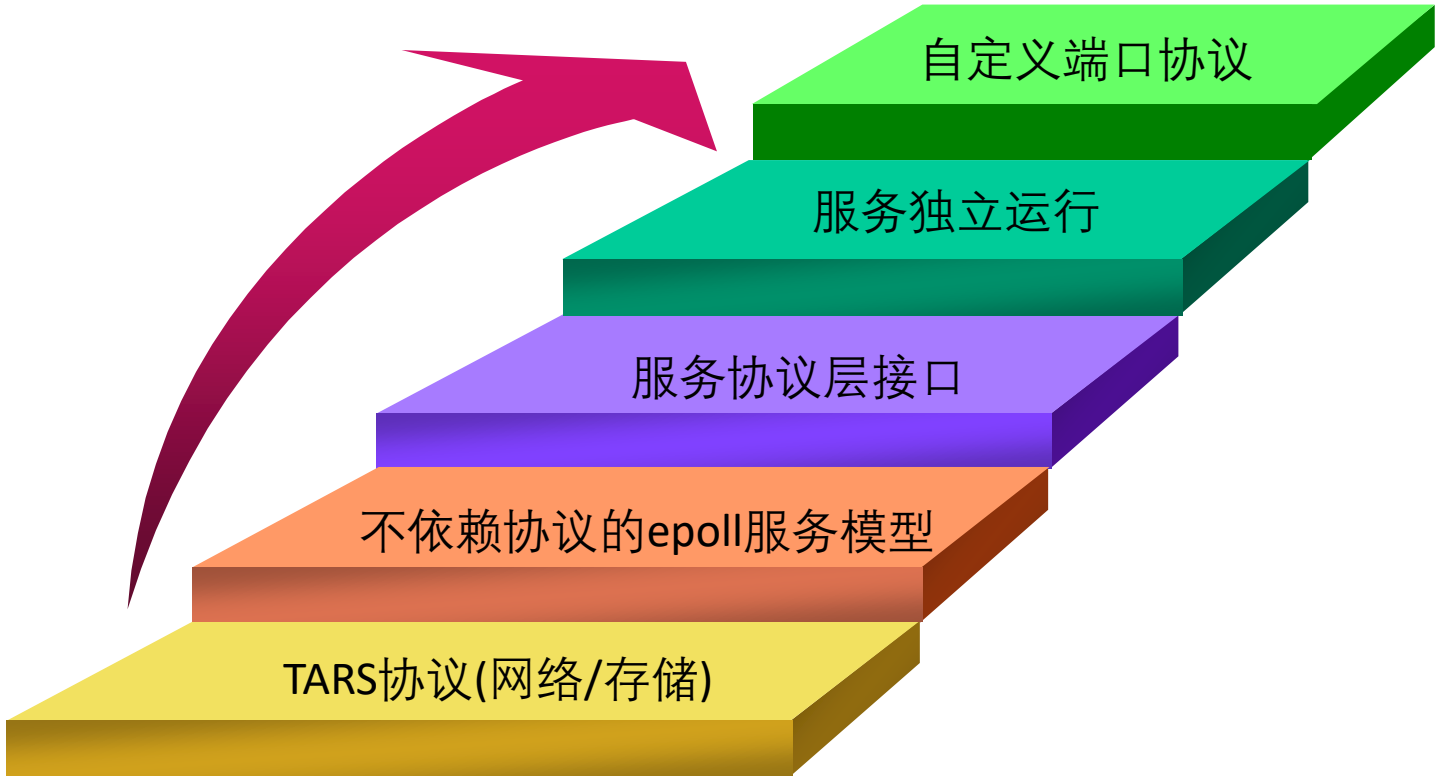
```
class PayServant
{
    virtual int doPayCoin(const UserInfo& tUserInfo,
                          bool mode, string& operId,
                          JceCurrentPtr current) = 0;
};
```

```
onDispatch()
{
    switch(funcName)
    {
        case "doPayCoin": doPayCoin(...);
        ......
    }
}
```

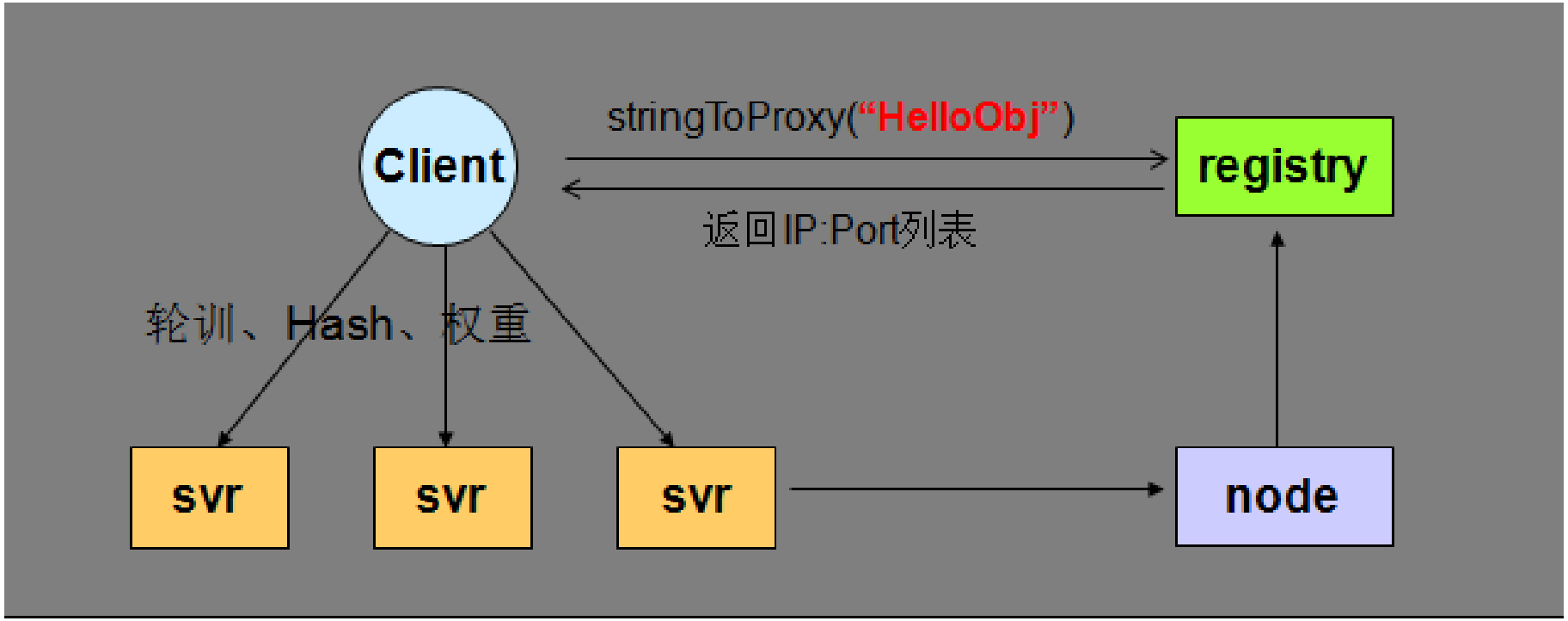
1.10 开发步骤


1
定义接口文件*.tars
编译成服务接口类
脚本创建新Server
继承并实现接口类
服务端开发
2
registry配部署信息
config 加业务配置
patch 发布到 nodes
通过node启动服务
部署发布
客户端开发
3
获取接口文件*.tars
编译c++/java代理类
创建本地代理对象
根据对象名称取地址
同步或异步调用
接收同步返回/回调

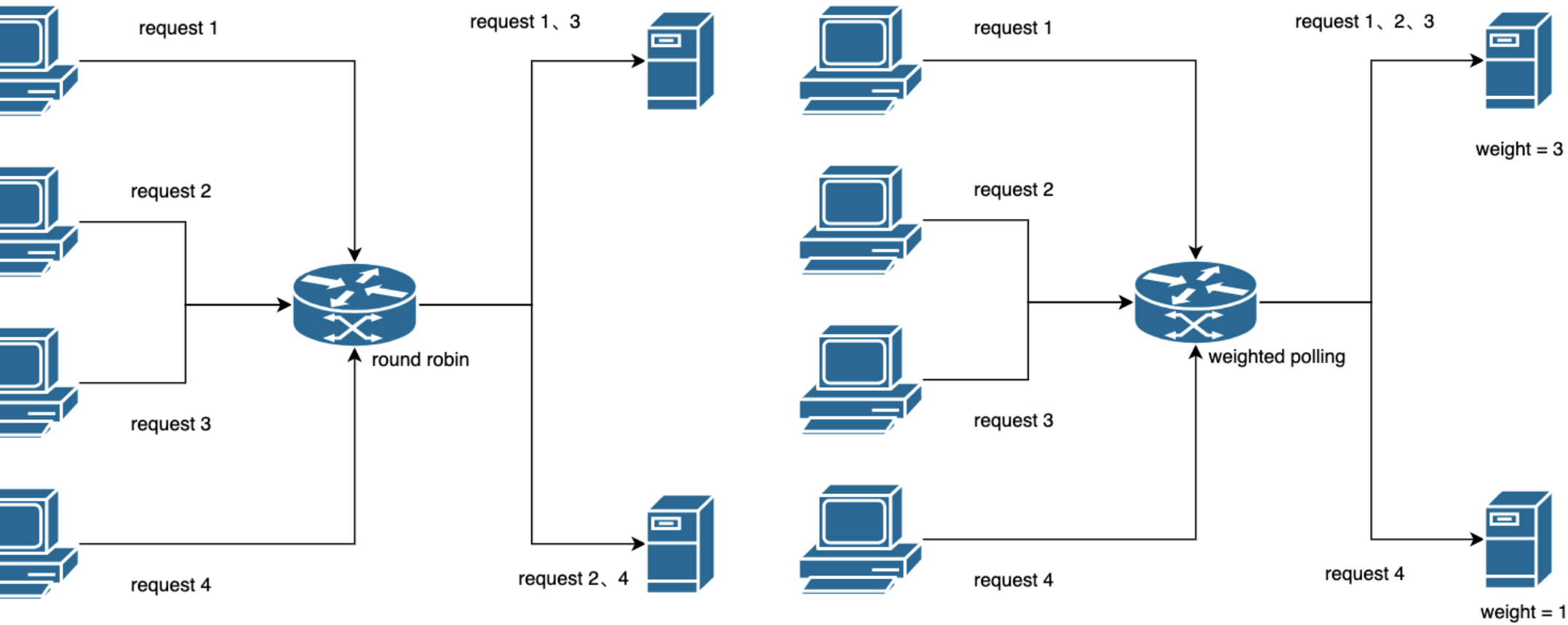
1.11 轻量化——(轻重结合，更多的选择)



1.12 负载均衡

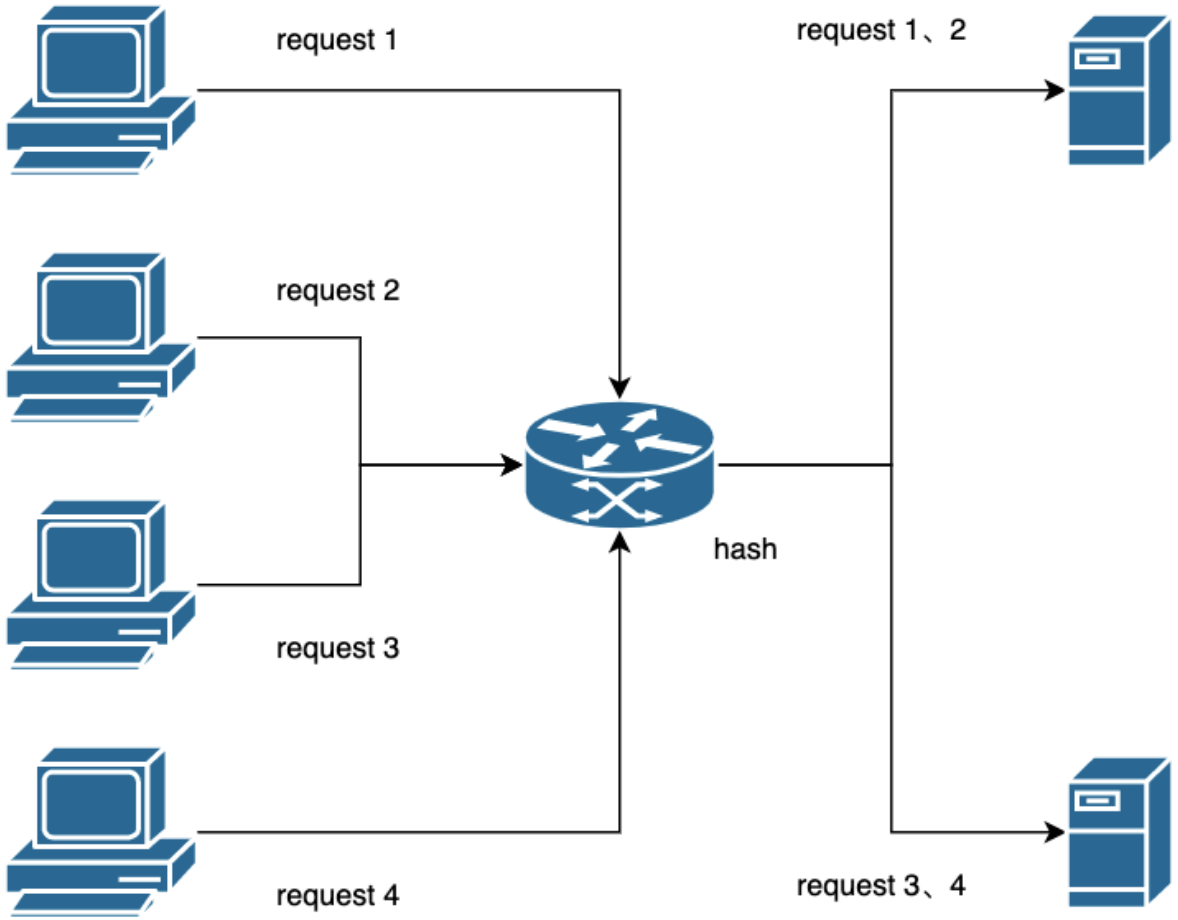

Client
stringToProxy("HelloObj")
registry
返回IP:Port列表
轮训、Hash、权重
svr
svr
svr
node

1.12 负载均衡-轮询和权重



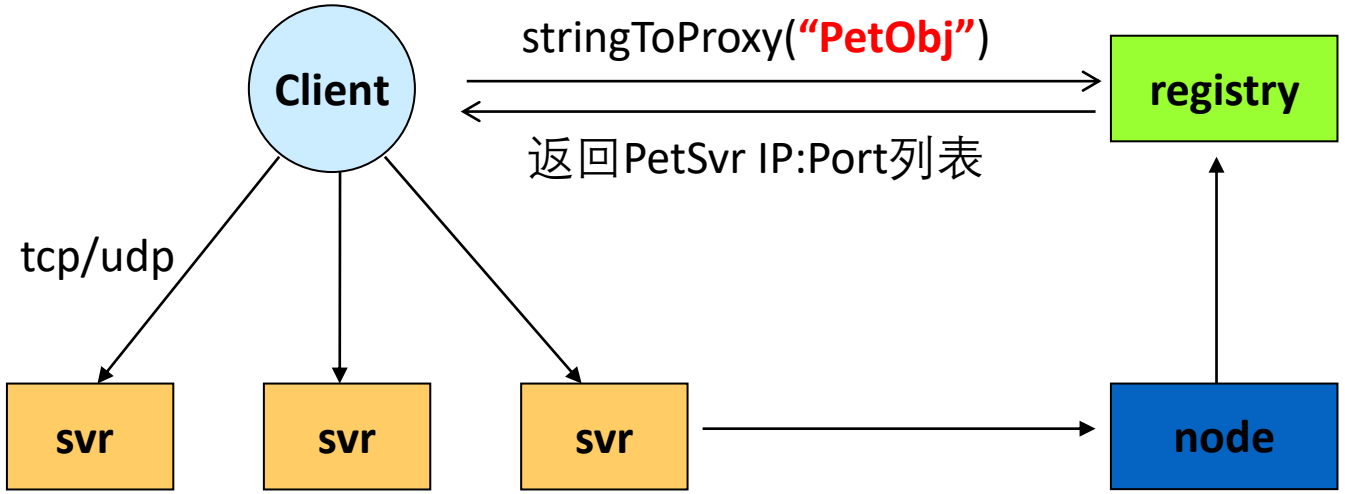
1.12 负载均衡--一致性哈希

很多时候在一些存在缓存的业务场景中，我们除了对流量平均分配有需求，同时也对同一个客户端请求应该尽可能落在同一个节点上有需求。



1.13 容错——(服务器挂掉不影响业务)

```
comm.setProperty("locator", "tars.tarsregistry.QueryObj@tcp -h 192.168.0.143 -t 60000 -p 17890");
```

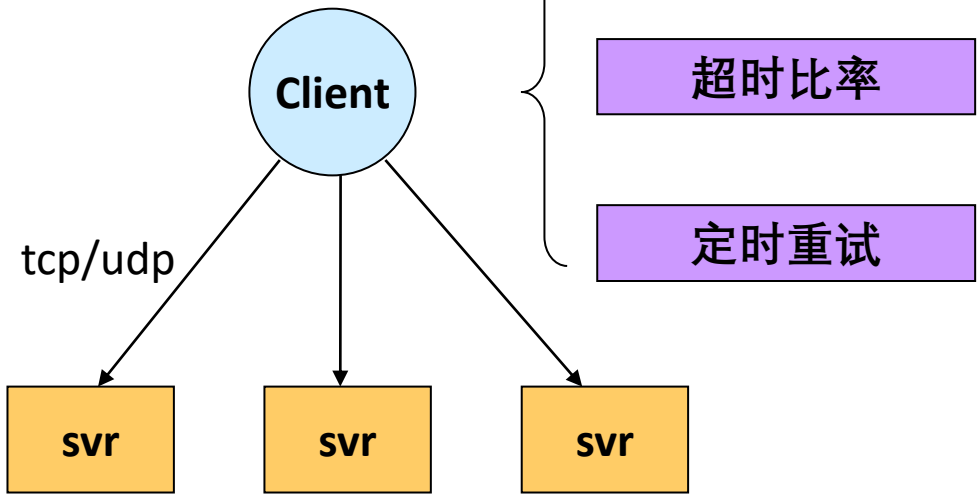


1. 缺省到每个server(ip:port) 一个连接
2. 多个连接竞争消息队列
3. 一个连接异常后影响最后一个请求
4. 定时从registry异步刷新服务列表，实现动态加入/移除Svr
5. Registry 部署多台，通过db共享数据，实现热备
6. 缺省轮循选择服务节点，支持HASH方式
7. 某个连接或者节点失效后，会定时重连（10秒）

```
comm.stringToProxy("TestApp.HelloServer.HelloObj@tcp -h 192.168.0.143  -p 22785" , prx);
```

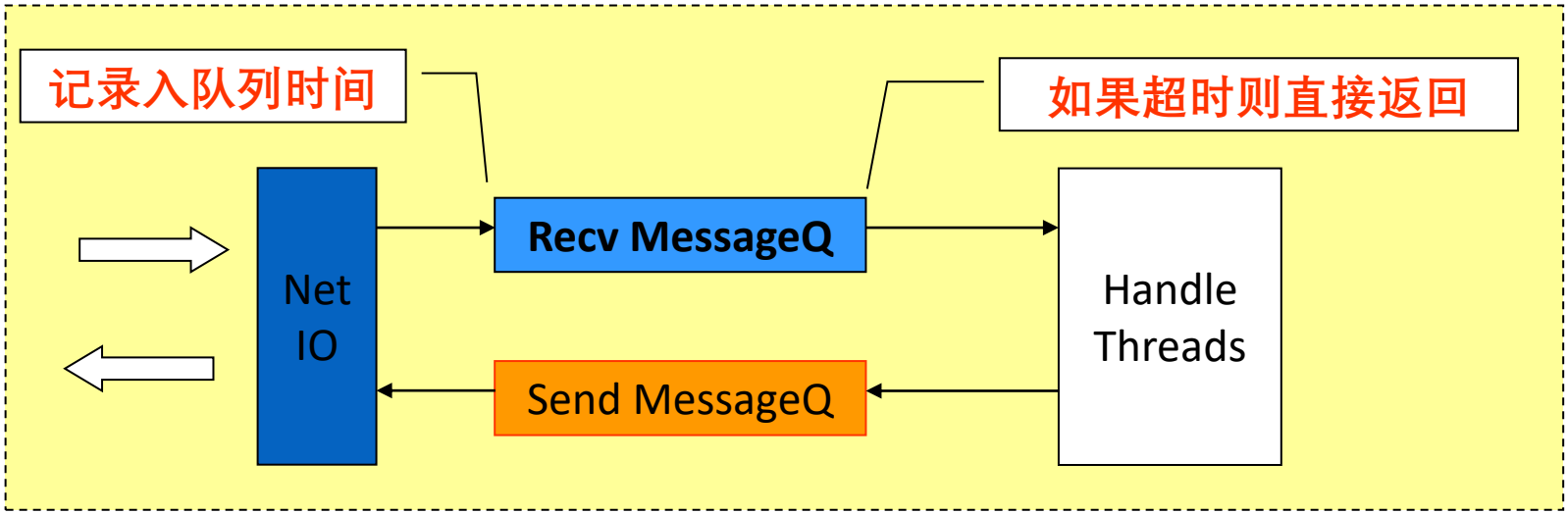
1.14 超时切换——(降低网络影响)

```
tcp -h 192.168.206.128 -p 25769 -t 3000,reqid:0]
```



1. 客户端内嵌超时切换逻辑
2. 连续超时次数大于某值
3. 超时比率大于某值
4. 定时重试

1.15 过载——



服务端接收队列达到某个阀值后——

1. 拒绝新请求
2. 监测每条消息在队列中的时间，已超时的消息不做业务逻辑处理
3. 超时时长由客户端控制

1.16 GITHUB——目录结构

https://github.com/TarsCloud/Tars

目录	说明
cpp	c++源码
framework	核心框架服务源码, 依赖cpp
go	Go源码
Java	Java源码
nodejs	Nodejs源码
php	Php源码
tup	Tup协议源码(https://tarscloud.github.io/TarsDocs/kai-fa/tarstup.html)
web	Web管理平台
docker	Docker制作脚本
docs	文档(https://tarscloud.github.io/TarsDocs/)

2 Tars开发部署常见问题

2.1 Tars部署
2.2 TarsWeb启动和停止
2.3 Tars框架服务启动和停止
2.4 常见问题定位

2.1 Tars部署

官方网址:
https://newdoc.tarsyun.com/#/markdown/TarsCloud/TarsDocs/installation/source.md

基于官方修改的部署网址
https://www.yuque.com/docs/share/9bd772dc-475b-417e-8669-1c6642557402?# 《1.tars源码安装》

开发范例
https://www.yuque.com/docs/share/2acc9fcd-14a4-4b05-959f-33b01b566a80?# 《2.通过平台发布应用(APP)》

2.2 TarsWeb启动和停止

启动模块： pm2 start tars-node-web

停止模块： pm2 stop tars-node-web

重启模块： pm2 start tars-node-web

注意, 如果你发现没有pm2命令, 一般都是node环境遍历没有生效, 你可以执行:

source /etc/profile

node的环境变量是在安装Tars时自动安装的, 注意用户是root.

2.3 Tars框架服务启动和停止

框架包括tarsAdminRegistry tarsregistry tarsnotify tarsconfig tarsnode tarslog tarspatch tarsproperty tarsqueryproperty tarsquerystat tarsstat web等服务

- 启动框架: /usr/local/app/tars/tars-start.sh
- 停止框架: /usr/local/app/tars/tars-stop.sh

- 控制某个组件:
  - /usr/local/app/tars/xxxx/util/start.sh
  - /usr/local/app/tars/xxxx/util/stop.sh

```
/usr/local/app/tars/${var}/util/start.sh
```

2.4 常见问题定位

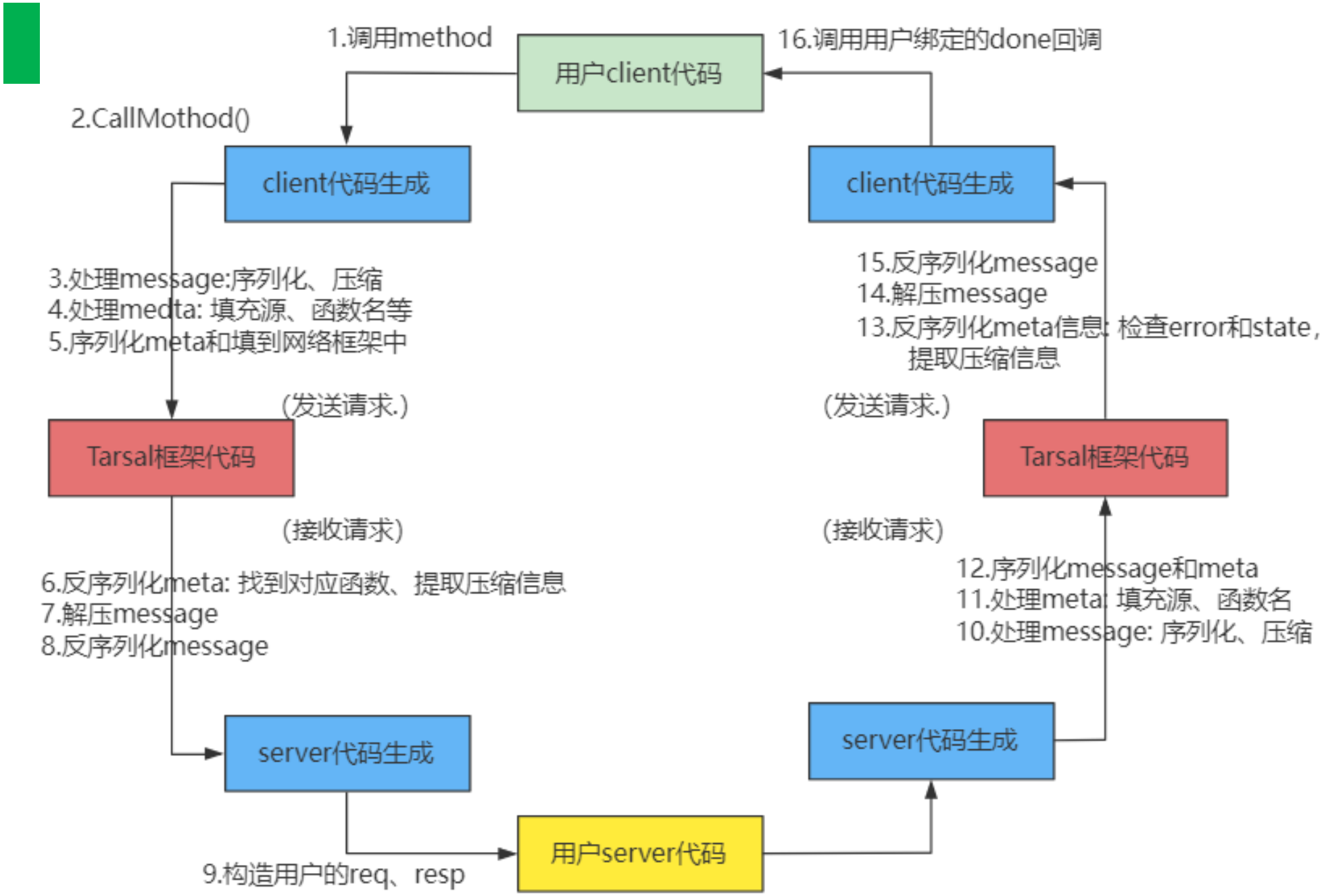
主要目录

1. 框架服务目录: /usr/local/app/tars/xxx
2. 业务服务: /usr/local/app/tars/tarsnode/data/
3. 服务日志: /usr/local /app/tars/app_log/xxx/xxxserver/*.log
4. Web日志: /usr/local/app/web/log

Web查看日志: http://192.168.0.143:3000/static/logview/logview.html

3 Tars Rpc

3.1 RPC原理
3.2 Tars RPC底层协议
3.3 客户端与服务端结构（多线程模型）
3.4 以Hello为例


1.调用method
用户client代码
16.调用用户绑定的done回调
2.CallMothod()
client代码生成
client代码生成
3.处理message:序列化、压缩
4.处理medta: 填充源、函数名等
5.序列化meta和填到网络框架中
15.反序列化message
14.解压message
13.反序列化meta信息: 检查error和state，提取压缩信息
(发送请求.)
(发送请求.)
Tarsal框架代码
Tarsal框架代码
(接收请求)
(接收请求)
6.反序列化meta: 找到对应函数、提取压缩信息
7.解压message
8.反序列化message
12.序列化message和meta
11.处理meta: 填充源、函数名
10.处理message: 序列化、压缩
server代码生成
server代码生成
用户server代码
9.构造用户的req、resp

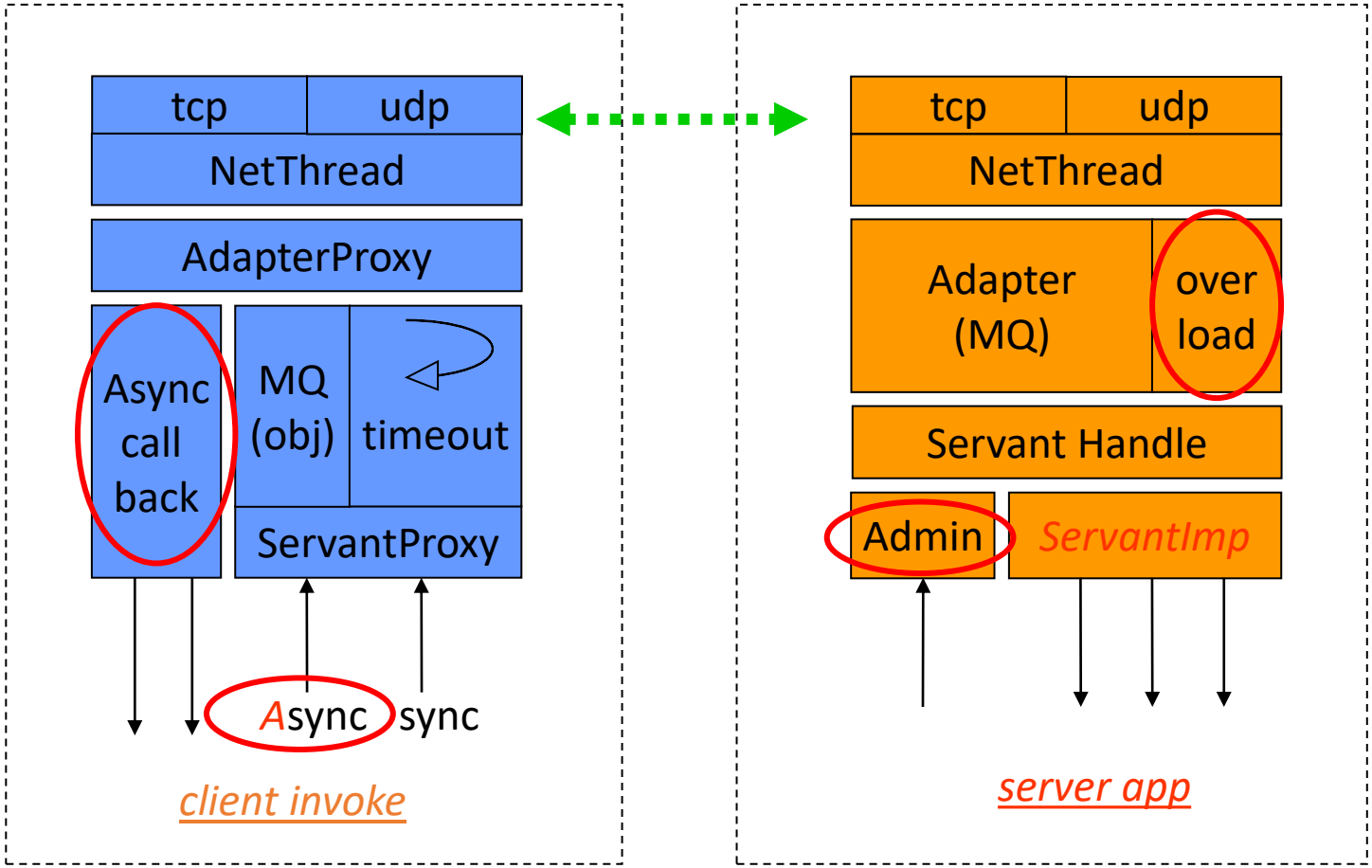
3.2 Tars RPC底层协议

```
module tars
{
    //请求包体
    struct RequestPacket
    {
        1  require short       iVersion;
        2  require byte        cPacketType  = 0;
        3  require int         iMessageType = 0;
        4  require int         iRequestId;
        5  require string      sServantName = "";
        6  require string      sFuncName    = "";
        7  require vector<byte> sBuffer;
        8  require int         iTimeout     = 0;
        9  require map<string, string> context;
        10 require map<string, string> status;
    };

};
```

```
//响应包体
  struct ResponsePacket
  {
      1 require short       iVersion;
      2 require byte        cPacketType  = 0;
      3 require int         iRequestId;
      4 require int         iMessageType = 0;
      5 require int         iRet         = 0;
      6 require vector<byte>  sBuffer;
      7 require map<string, string> status;
      8 optional string       sResultDesc;
      9 optional map<string, string> context;
  };
```

https://newdoc.tarsyun.com/#/markdown/TarsCloud/TarsDocs/dev/tarscpp/tars-guide.md

3.3 客户端与服务端结构（多线程模型）



3.4 以Hello为例-客户端

Client

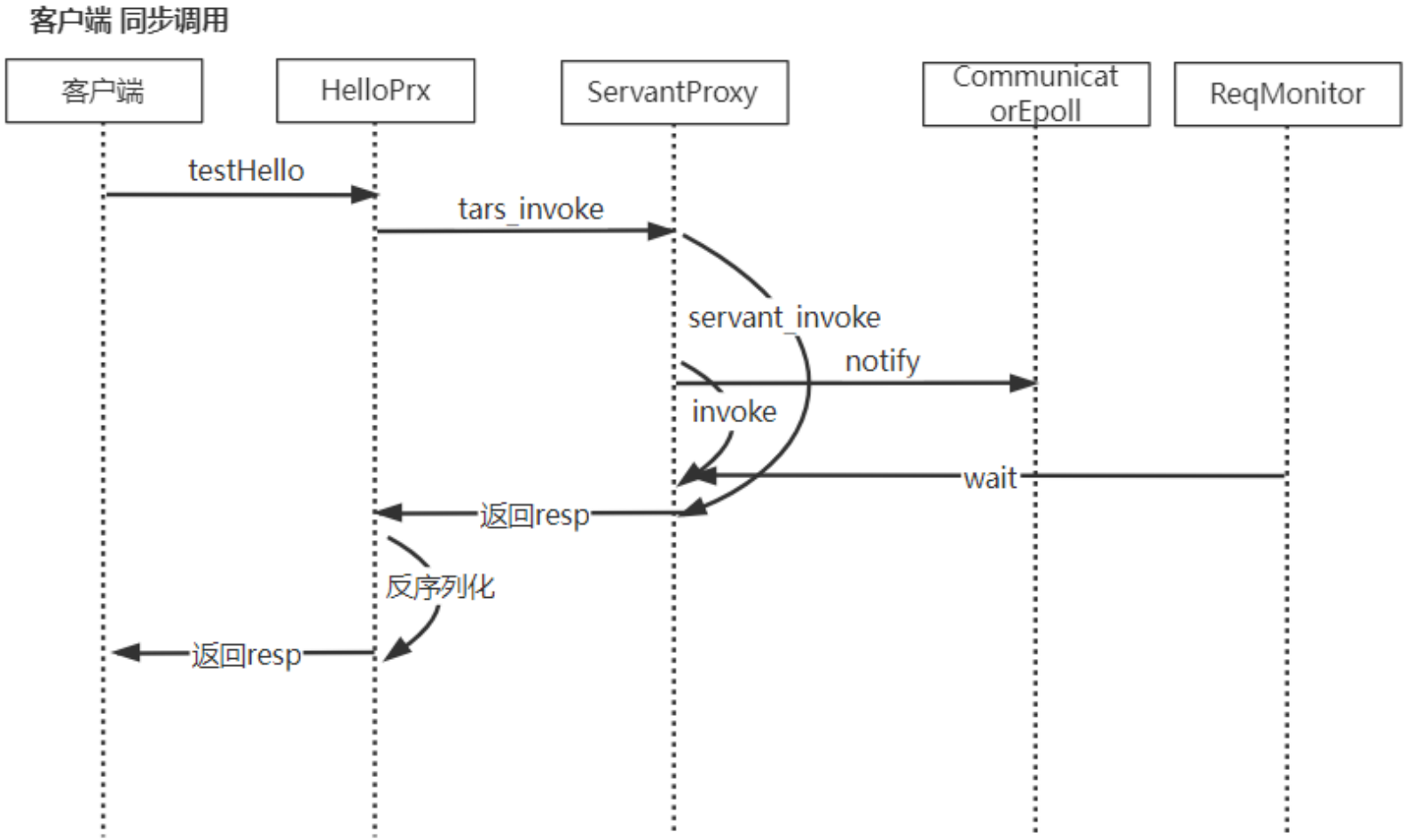
```
int iRet = prx->testHello(sReq, sRsp);
```

```
static string helloObj = "TestApp.HelloServer.HelloObj";
int main(int argc,char ** argv)
{
    Communicator comm;

    try
    {
        //HelloPrx prx;
        //comm.stringToProxy("TestApp.HelloServer.HelloObj@tcp -h 192.168.
        //comm.stringToProxy("TestApp.HelloServer.HelloObj@tcp -h 192.168.
        HelloPrx prx;
        comm.setProperty("locator", "tars.tarsregistry.QueryObj@tcp -h 192
        prx = comm.stringToProxy<HelloPrx>(helloObj);
```

```
rs::Int32 testHello(const std::string & sReq,std::string &sRsp,const map<string, string> &context = TARS_CONTEXT(),map<string

tars::TarsOutputStream<tars::BufferWriterVector> _os;
_os.write(sReq, 1);
_os.write(sRsp, 2);
std::map<string, string> _mStatus;
shared_ptr<tars::ResponsePacket> rep = tars_invoke(tars::TARSNORMAL,"testHello", _os, context,
if(pResponseContext)
{
    pResponseContext->swap(rep->context);
}

tars::TarsInputStream<tars::BufferReader> _is;
_is.setBuffer(rep->sBuffer);
tars::Int32 _ret;
_is.read(_ret, 0, true);
_is.read(sRsp, 2, true);
return _ret;
```

```
shared_ptr<ResponsePacket> ServantProxy::tars_invoke(char  cPacketType
                                                     const string& sFuncName,
                                                     const vector<char>& buf,
                                                     const map<string, string>& context,
                                                     const map<string, string>& status)
                                                    //   ResponsePacket& rsp)
{
    ReqMessage *msg = new ReqMessage();

    msg->init(ReqMessage::SYNC_CALL);

    msg->request.iVersion = TARSVERSION;
    msg->request.cPacketType = cPacketType;
    msg->request.sFuncName = sFuncName;
    msg->request.sServantName = (*_objectProxy)->name();

    msg->request.sBuffer = buf;
    msg->request.context      = context;
    msg->request.status       = status;
    msg->request.iTimeout     = _syncTimeout;
```

3.4 以Hello为例-客户端-同步调用



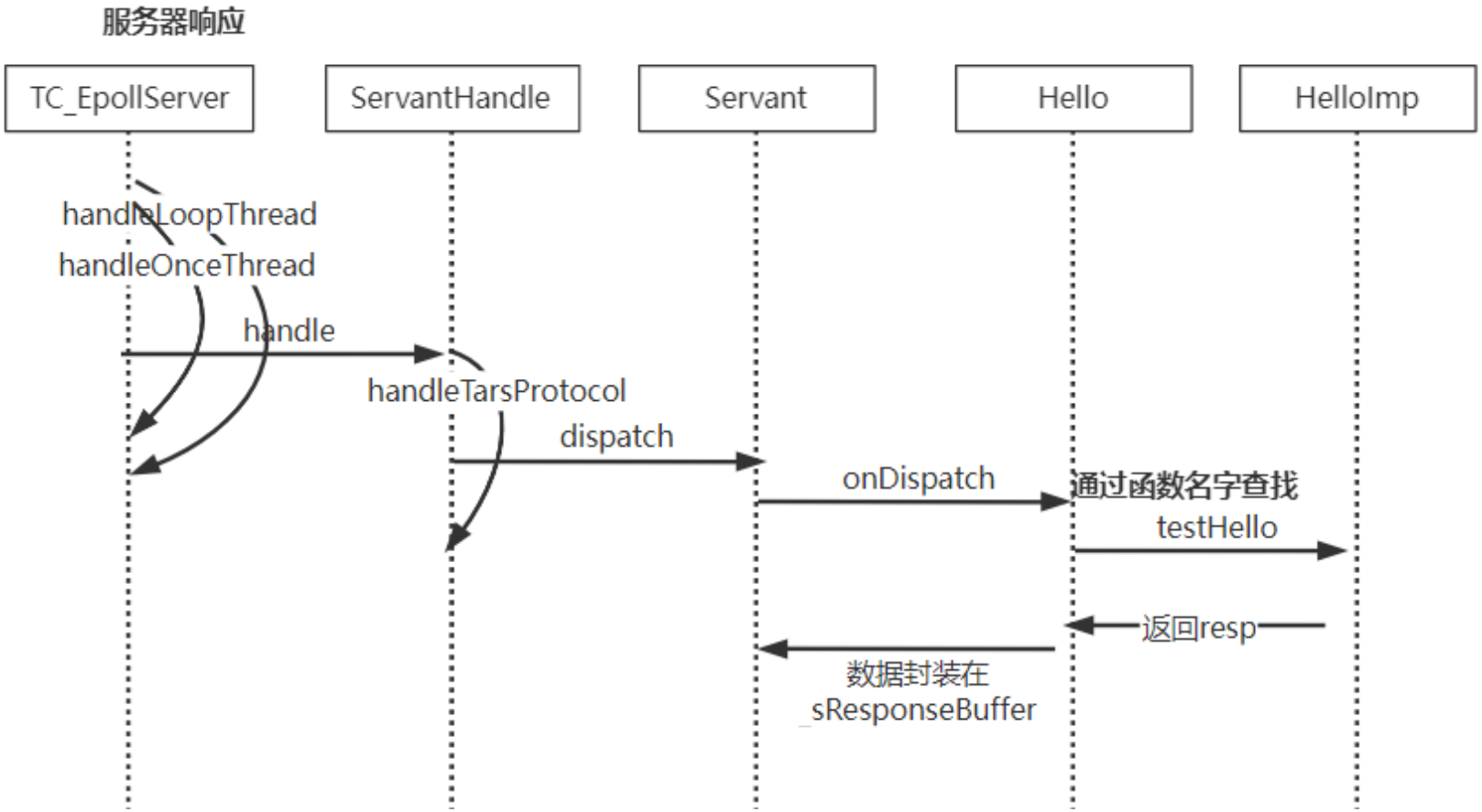
3.4 以Hello为例-服务端

HelloServer.cpp

```cpp
void
HelloServer::initialize()
{
    //initialize application here:
    //...

    addServant<HelloImp>(ServerConfig::Application + "." + ServerConfig::ServerName + ".HelloObj");
}
```

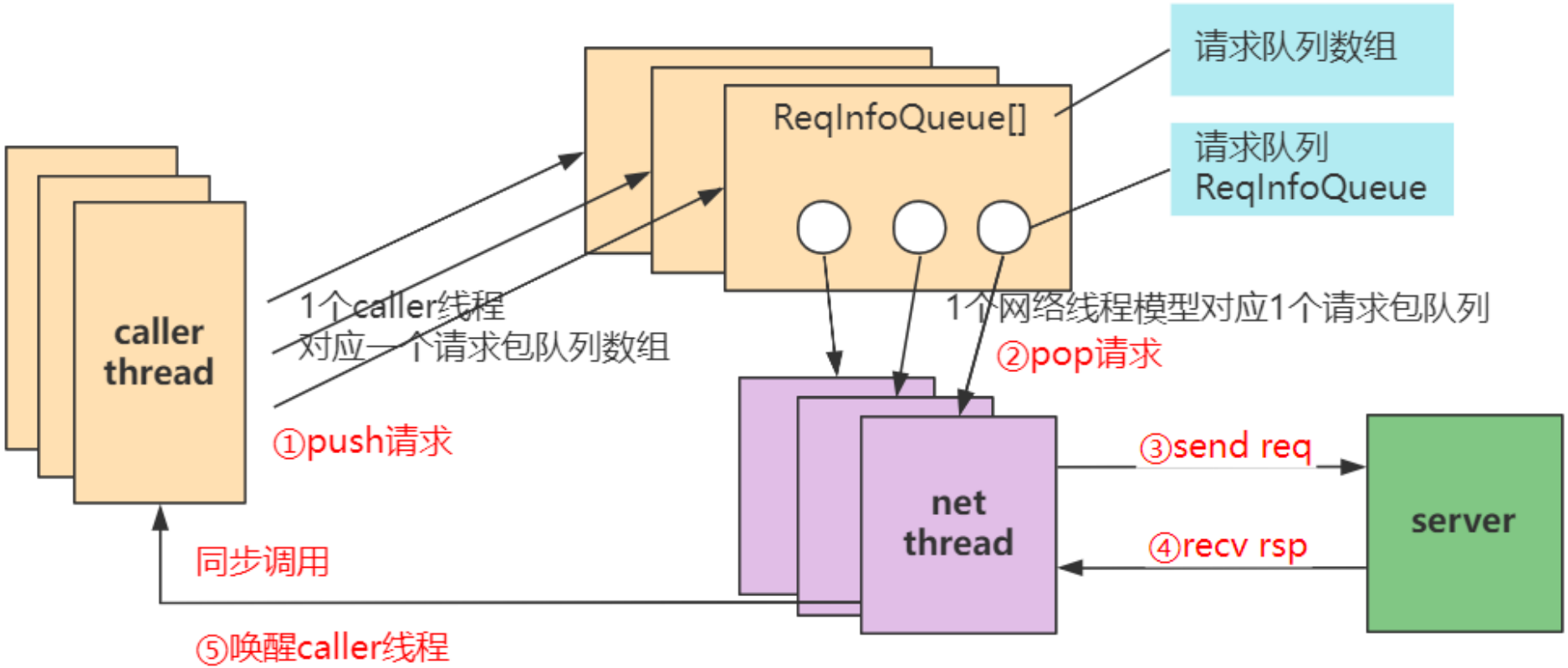
```cpp
int onDispatch(tars::TarsCurrentPtr _current, vecto
{
    static ::std::string __TestApp__Hello_all[]=
    {
        "test",
        "testHello"
    };

    pair<string*, string*> r = equal_range(__TestApp__Hello_all, __TestApp__Hello_all+2, _current->getFuncName());
    if(r.first == r.second) return tars::TARSSERVERNOFUNCERR;
    switch(r.first - __TestApp__Hello_all)
    {
        case 0:
        {
            tars::TarsInputStream<tars::BufferReader> _is;
            _is.setBuffer(_current->getRequestBuffer());
            if (_current->getRequestVersion() == TUPVERSION)
            {
                UniAttribute<tars::BufferWriterVector, tars::BufferReader>  tarsAttr;
                tarsAttr.setVersion(_current->getRequestVersion());
                tarsAttr.decode(_current->getRequestBuffer());
            }
            else if (_current->getRequestVersion() == JSONVERSION)
            {
                tars::JsonValueObjPtr _jsonPtr = tars::JsonValueObjPtr::dynamicCast(tars::TC_Json::getValue(_current->
```

3.4 以Hello为例-服务端-响应

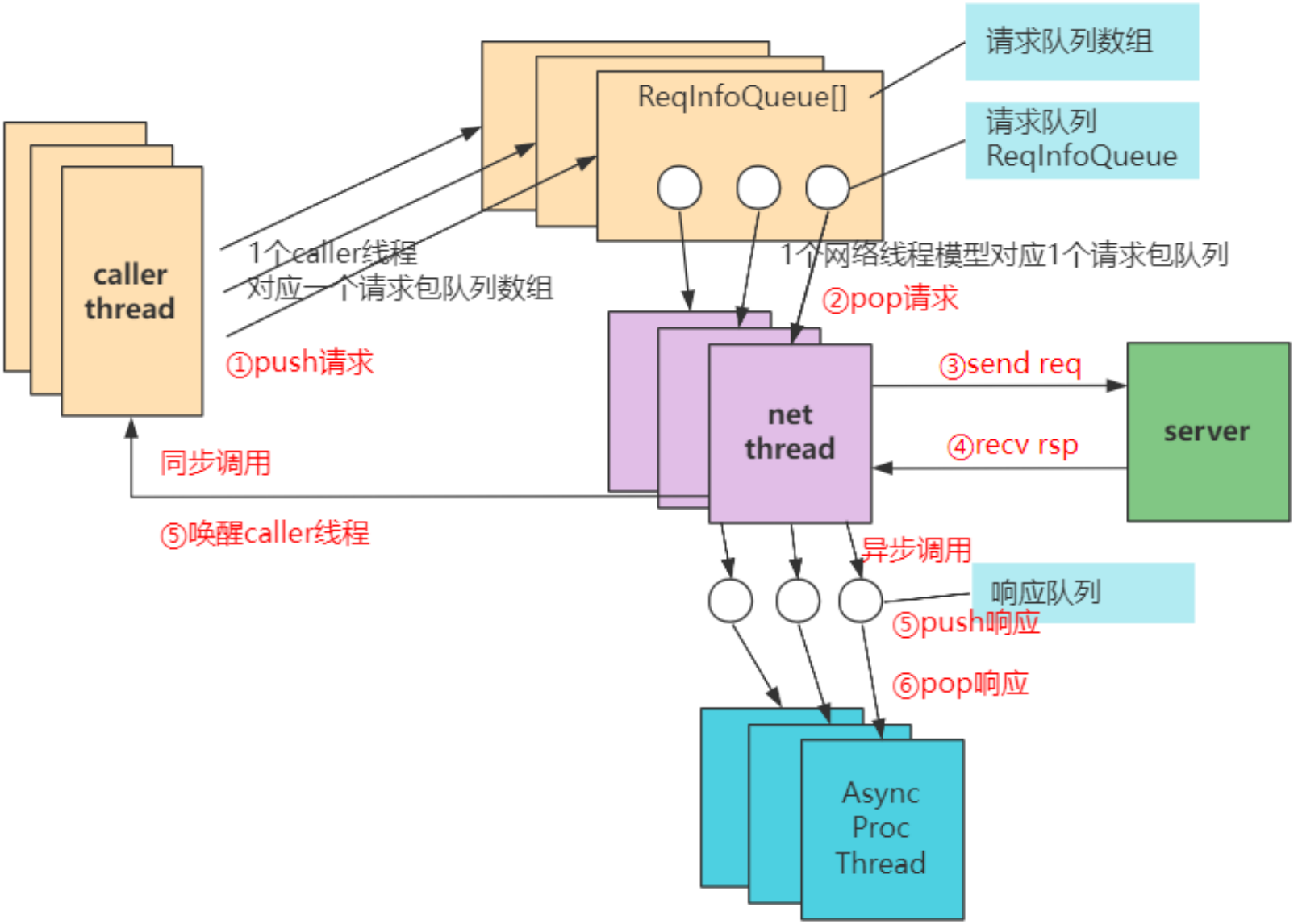


TestApp::Hello::onDispatch(tars::TC_AutoPtr<tars::Current>, std::vector<char, std::allocator<char> >&)

3.5 客户端同步调用


请求队列数组
ReqInfoQueue[]
请求队列
ReqInfoQueue
caller
thread
1个caller线程
对应一个请求包队列数组
1个网络线程模型对应1个请求包队列
②pop请求
①push请求
net
thread
③send req
server
④recv rsp
同步调用
⑤唤醒caller线程

3.5客户端异步调用


请求队列数组
ReqInfoQueue[]
请求队列
ReqInfoQueue
caller
thread
1个caller线程
对应一个请求包队列数组
1个网络线程模型对应1个请求包队列
②pop请求
①push请求
net
thread
③send req
server
④recv rsp
同步调用
⑤唤醒caller线程
异步调用
响应队列
⑤push响应
⑥pop响应
Async
Proc
Thread

4 参考

主要来自于Tars官方PPT、文章和范例