

2.通过平台发布应用(APP)

0 网页版本链接

0 基本概念

1. APP

2. Server

3. Servant

4. module

5. Tars 文件目录规范

6. 服务端开发方式

7. 客户端开发方式

8 模板配置

9 业务配置

10 日志系统

11 开发调试发布（这个自己去试，我们只讲web发布的）

1. 环境搭建

2. 服务命名

3. Tars管理系统

4. 服务部署

5. 服务开发

5.1. 创建服务

5.1.1. 运行tars脚本

5.1.2. tars接口文件

5.1.3. HelloImp是Servant的接口实现类

5.1.4. HelloServer是服务的实现类

5.2. 服务编译

5.3. 扩展功能

5.4. 客户端同步/异步调用服务

5.4.1 同步方式

5.4.2 异步方式

5.4.3 编译客户端程序

6. 服务发布

6.1 服务部署

6.1.1 添加应用

6.1.2 添加服务

6.2 服务发布

6.2.1 选中节点上传发布包

6.2.2 发布发布包

6.3 客户端测试

6.3.1 修改main.cpp

6.3.2 重新编译和执行程序

6.3.3 异常情况

6.3.4 如果不直连server呢?

FAQ

返回错误值

日志

调用栈

0 网页版本链接

腾讯课堂 零声教育 <https://ke.qq.com/course/420945?tuin=137bb271>

Darren QQ326873713

网页版本: <https://www.yuque.com/docs/share/2acc9fcd-14a4-4b05-959f-33b01b566a80?#>

《2.通过平台发布应用(APP)》

本文档基于官方教程<https://tarscloud.github.io/TarsDocs/hello-world/tarscpp.html>做补充。

本文实践环境: Ubuntu 18.04/16.04

0 基本概念

1. APP

APP 即**应用名**，标识一组服务的一个小集合，开发者可以根据需要自己定义，通常表示实现某个业务系统名称。

- 在 Tars 系统中，应用名必须唯一，例如：TestApp, ImApp
- 通常应用名对应代码中的**某个名字空间**
- tars 这个应用名是框架使用的，业务服务请不要使用

2. Server

Server 即**服务名**，提供服务的进程名称

- Server 名字根据业务服务功能命名，它会在 TARS web 平台上左边服务树上展示
- 一个 Server 必须属于某个 App, App 下的 Server 名称都具备唯一性
- 一般命名为：XXServer，例如 LogServer, TimerServer 等
- 一个 Server 代表一个独立的程序，绑定至少一个 ip，实现一组相关的接口

3. Servant

Servant 即服务提供者，提供了一个多个具体的接口(interface)，提供给客户端调用

- Servant 对应服务代码中一个类，继承于 tars 协议文件中的 interface(内涵多个具体的函数)，由业务开发者实现
- 一个 Servant 必须属于某个 Server, Server 下的 **Servant 名称都具备唯一性**
- Servant 需要一个名称，比如：HelloObj，当提供给客户端使用的，全称是：App.Server.Servant，比如：Test.HelloServer.HelloObj
- 客户端调用 Server 时，只需要指定 Servant 的名称即可完成远程通信(如何实现后续会介绍)

Tars 采取这种三层结构，尽可能的避免不同业务开发者开发的服务名称和 Servant 名称冲突

4. module

module 是 tars 协议文件中的关键字，定义了协议的空间，也对应了各语言名字空间(c++)或者包名(java, go)或模块(nodejs,php)

5. Tars 文件目录规范

Tars 文件是 TARS 服务的协议通信接口，尤其某 Tars Server 的客户端调用 Server 时都需要依赖该 Server 的 tars protocol 文件，因此非常重要，在管理上我们推荐按照如下方式管理(当然你可以不采取该模式，构建你自己合适的开发方式)：

- tars 文件原则上和相应的 server 放在一起；
- 每个 server 在开发机上建立 **/home/tarsproto/[namespace]/[server]**子目录；

- 所有 tars 文件需要更新到 `/home/tarsproto` 下相应 server 的目录；

```
root@ubuntu: /home/tarsproto/TestApp/HelloServer#  
Hello.h  Hello.tars
```

- 使用其他 server 的 tars 文件时，需要到 `/home/tarsproto` 中使用，不能 copy 到本目录下；
- 如此，在相同服务器上开发服务时，你只需要将 tars 文件 release 到该目录下，就能方便其他调用方使用
- tars 的接口原则上只能增加，不能减少或修改；
- 各语言提供的 Tars 服务框架，都提供了快速 release tars 文件到 `/home/tarsproto/[namespace]/[server]` 下的工具

6. 服务端开发方式

任何 Tars 服务端和客户端的开发方式都基本一样：

- 确定 APP, Server, Servant 名称；
- 编写 tars 文件，定义服务对外提供的 interface 以及 interface 下面的函数，主要 interface 必须有一个，interface 下面的函数可以有多个，tars 的文件的语法请参考 <https://tarscloud.github.io/TarsDocs/base/tars-protocol.html>
- 使用 tars2xxx 工具(不同语言的工具不同)，将 tars 文件生成不同的语言的代码，tars2xx类似 protobuf的对应的protoc工具；
- 实现 Tars 服务(请参考不同语言的文档)，继承生成的文件中的 Servant 类，实现 Servant 的 interface；
- 编译服务，发布在管理平台上(在管理平台需要配置 App, Server, Servant Obj 名称等)，可以参考后续章节；
- 当然你的服务也可以本地运行，可以在平台上启动服务后，`ps -ef`看到服务的启动方式后，放在本地执行即可(注意可能有配置文件，需要修改端口等信息)。

正常情况下，服务最终都通过 tarsweb 发布运行在 tars 平台上的各个节点服务器上，但是在调试过程中希望在本机运行，该如何处理？

服务启动实际上无非是一条命令行，比如 c++服务是：

```
HelloServer --config=xxxxx.conf
```

这里配置 config 表示服务启动的配置文件，在 tars 平台上是由 tarsnode 通过拉取模板配置生成的，并拉起 HelloServer，如果你想本地运行服务，就必须本地具备这个配置文件。

注意：

- 建议弄清楚 config 的主要配置项含义, 参考开发指南
- config 中的 ip 要注意提供成本机的
- `node=tars.tarsnode.ServerObj@xxxx`, 表示连接的 tarsnode 的地址, 如果本地没有 tarsnode, 这项配置可以去掉
- `local=...`, 表示开放的本机给 tarsnode 连接的端口, 如果没有 `tarsnode`, 可以掉这项配置
- `locator=...`, 表示主控中心的地址(框架地址), 用于根据服务名字获取 ip list 的
- 如果你是独立的客户端, 有这项配置, 就可以不用指定其他服务地址, 进行访问

注意这个配置文件不是业务配置, 而是服务框架的配置, 对应 tars 平台上的模板!

如何获取这个配置文件呢?

你可以先将服务发布到平台的某个节点上, 然后登陆节点服务器, 运行:

```
ps -efww | grep ${your server name}
```

你可以看到服务启动的命令行, 将对应的配置文件 copy 到本地, 并且打开配置文件, 修改配置文件里面对应 ip port 以及相关路径, 然后使用相同的命令行本地运行即可!

其他语言方式类似!

7. 客户端开发方式

完成服务端编写和启动后, 即可编写客户端, 通过引用 tars 文件生成的客户端代码, 并构建通信器, 使用通信器并根据 Servant 名称获取到对应的服务的代理对象, 使用代理对象完成通信.

注意:

- 通信器(**Communicator**)是客户端管理线程和网络的一个类, 它在各大语言中都有对应的类, 通常服务框架中会提供一个初始化过的给你使用(通常你使用这个对象即可), 如果是存客户端, 那么你就需要自己创建通信器
- 如果你的 client 是另外一个服务, 并部署在框架上, 那么你的通信器可以使用框架提供好的通信器(参考各语言文档), 此时调用 Server 时, 不需要指定 ip port, 只需要 Servant 的 Obj 名称即可, 框架会自动寻址并完成调用
- 如果你的 client 是独立的客户端程序, 并不部署在框架上, 那么你可以自己创建通信器, 此时调用服务有两种方式:
 - 直接指定服务端的 ip 端口的方式(你可以指定多个, 框架会自动容灾切换), 各语言基本相同, 比如 c++语言:

```

1 Communicator *communicator = new Communicator();
2 HelloPrx helloPrx = communicator->stringToProxy<HelloPrx>
  ("Test.HelloServer.HelloObj@tcp -h xxx -p yyy:tcp -h www -p zzz");
3 helloPrx->call();

```

- 也可以通信器指定到对应的框架的主控中, 这样就不需要指定对应的 ip port 了, 各语言基本相同, 比如 c++语言:

```

1 Communicator *communicator = new Communicator();
2 communicator->setProperty("locator", "tars.tarsregistry.QueryObj@tcp -h
  xxxx -p 17890");
3 HelloPrx helloPrx = communicator->stringToProxy<HelloPrx>
  ("Test.HelloServer.HelloObj");
4 helloPrx->call();

```

这种客户端调用方式, 虽然服务可以寻址和容灾, 但是没有上报信息, 如果需要上报信息还需要指定其他相关属性, 比如:

```

1 communicator->setProperty("stat", "tars.tarsstat.StatObj");
2 communicator->setProperty("property", "tars.tarsproperty.PropertyObj");

```

当然你可以直接用配置文件来初始化通信器, 参考 web 平台模板配置中的 client 部分. 另外上报服务这里类同, 如果没有 locator 指定框架的主控地址, 你就需要自己指定上报的 ip port.

8 模板配置

web 平台上, 运维管理中有模板配置, 模板配置对于框架非常重要, 需要去理解模板配置的作用.

每一个部署在 TARS 框架上的服务, 最终其实被框架发布到对应的节点了(tarsnode), 那么 tarsnode 在拉起这个服务时如何知道服务绑定的端口, 启动的线程数等信息呢? 答案就是通过: 模板配置

tarsnode 会去平台拉取服务对应的模板(服务部署时配置好的), 然后根据模板生成服务对应的配置, 并用户这个配置启动服务.

注意不同语言使用的模板配置文件不同, 可以参考后续各语言的文档.

强烈建议你不需要修改框架自带的模板, 因为后续框架升级可能会修改这些模板内容, 如果你需要修改, 你可以继承该模板, 让你的服务使用继承的模板

9 业务配置

上一节讲到模板配置, 模板配置实际是 rpc server 用到的公用配置, 它是 tarsnode 生成的. 但是对于业务服务而言, 一般都有自己的配置信息, 这种配置我们称之为业务配置.

业务配置一般都在 web 平台上配置中心来管理, 使用方式如下:

- 在 web 平台上, 对应服务的配置中心下, 添加业务配置文件
- 在服务代码中, 加载配置文件(各语言的 rpc 库都提供了加载配置文件的 api)
- 如果有需要, 也可以在 web 平台上 push 配置文件到具体的 server(PHP 的 swoole 不支持)

业务配置具体使用可以参见业务配置

10 日志系统

业务服务中通常也需要打印日志, 框架也提供了集中的日志中心.

通常日志分两类:

- 一类是 debug 日志(循环日志/单个文件控制大小/多个日志文件循环),
- 一类是按天/小时等的日志, 这类日志通常记录重要的日志信息, 这类日志可以输出到远程日志中心上.

通过日志 api, 你可以将日志直接输出到远程的日志中心上, 即 tarslog 所在机器上(日志路径为: /usr/local/app/tars/remote_app_log)

所有语言的框架都提供了远程日志的 api.

另外在 web 平台上, 点击服务详情界面, 点击服务名称, **也可以快速查看到服务的本机日志.**

11 开发调试发布 (这个自己去试, 我们只讲web发布的)

如果开发过程中, 每次都需要手工发布到 web 平台调试, 调试效率是非常低, 因此 Tars 平台提供了一个方式, 能够一键发布服务到 Tars 框架上.

使用方式如下:

- 这需要 web >= 2.0.0, tarscpp>=2.1.0 的版本才能支持.
- 完成框架安装后, 登录用户中心, 创建一个 token
- linux 上使用 curl 命令即可完成服务的上传和发布, 以 Test/HelloServer 为例, 参考 cmake 管

```
1 curl http://${your-web-host}/api/upload_and_publish?ticket=${token} -  
   Fsuse=@HelloServer.tgz -Fapplication=Test -Fmodule_name=HelloServer -  
   Fcomment=dev
```

注意替换你的 token

c++版本的 cmake 已经内嵌了命令行在服务的 CMakeLists.txt 中, 比如用
cmake_tars_server.sh 创建服务之后, 只需要:

```
1 cd build  
2 cmake .. -DTARS_WEB_HOST=${WEB_HOST} -DTARS_TOKEN=${TOKEN}  
3 make HelloServer-tar  
4 make HelloServer-upload
```

即可完成服务的上传和发布(提前需要在 web 平台配置好)

注意:

- 替换 WEB_HOST 和 token
- HelloServer.tgz 是 c++的发布包, java 对应是 war 包, 其他语言类似, 对应你上传到 web 平台的发布包

1. 环境搭建

Tars C++环境搭建参考tars_install.md

请务必先阅读 concept and spec

2. 服务命名

使用Tars框架的服务, 其的服务名称有三个部分:

- **APP：应用名**，标识一组服务的一个小集合，在Tars系统中，应用名必须唯一。例如：
TestApp；
- **Server：服务名**，提供服务的进程名称，Server名字根据业务服务功能命名，一般命名为：
XXServer，例如HelloServer；
- **Servant：服务者**，提供具体服务的接口或实例。例如:HelloImp；

说明：

一个Server可以包含多个Servant，系统会使用服务的App + Server + Servant，进行组合，来定义服务在系统中的路由名称，称为路由Obj，其名称在整个系统中必须是唯一的，以便在对外服务时，能唯一标识自身。

因此在定义APP时，*需要注意APP的唯一性*。

例如：`TestApp.HelloServer.HelloObj`。

3. Tars管理系统

用户登录成功后，会进入Tars管理系统，如下图



TARS管理系统的菜单树下，有以下功能：

- 业务管理：包括已部署的服务，以及服务管理、发布管理、服务配置、服务监控、特性监控等；
- 运维管理：包括服务部署、扩容、模版管理等；

4. 服务部署

服务部署，其实也可以在服务开发后进行，不过建议先做。

如下图：

TARS 服务管理 运维管理

服务部署

部署申请

应用 *
TestApp

服务名称 *
HelloServer

服务类型 *
tars_cpp

模版 *
tars.default

节点 *
10.120.129.226

Set分组
☐ 启用Set
Set名 Set地区 Set组名

OBJ名称*	OBJ绑定IP*	端口*	端口类型*	协议*	线程数*	最大连接数*	队列最大长度*	队列超时时间(ms)*	操作
HelloObj	10.120.129.226	20001	*TCP *UDP	*TARS *ITARS	5	200000	10000	60000	

提交

- “应用”指你的服务程序归在哪一个应用下，例如：“TestApp”。
- “服务名称”指你的服务程序的标识名字，例如：“HelloServer”。
- “服务类型”指你的服务程序用什么语言写的，例如：c++的选择“tars_cpp”。
- “模版“ 指你的服务程序在启动时，设置的配置文件的名称，默认用”tars.default“即可。
- “节点“ 指服务部署的机器IP。
- “Set分组“ 指设置服务的Set分组信息，Set信息包括3部分：Set名、Set地区、Set组名。
- “OBJ名称“ 指Servant的名称。
- “OBJ绑定IP“ 指服务绑定的机器IP，一般与节点一样。
- “端口“ 指OBJ要绑定的端口。
- “端口类型“ 指使用TCP还是UDP。
- “协议“ 指应用层使用的通信协议，Tars框架默认使用tars协议。
- “线程数“ 指业务处理线程的数目。
- “最大连接数“ 指支持的最大连接数。
- “队列最大长度“ 指请求接收队列的大小（请求数据在队列中最大个数多了就过载了, 会丢弃）。
- “队列超时时间“ 指请求接收队列的超时时间（请求在队列中等待了当前设置的时间没有处理, 会丢弃）。

点击“提交“，成功后，菜单数下的TestApp应用将出现HelloServer名称，同时将在右侧看到你新增的服务程序信息，如下图：

TARS 服务管理 运维管理

Test
TestApp
HelloServer
tars
tarsconfig
tarslog
tarsnotify
tarspatch
tarsproperty
tarsqueryproperty
tarsquerystat
tarsstat

服务管理 发布管理 服务配置 服务监控 特性监控

服务列表

服务	节点	启用Set	Set名	Set区	Set组	设置状态	当前状态	进程ID	版本	发布时间	操作
HelloServer	10.120.129.226	否				inactive	inactive	0		0000:00:00 00:00:00	编辑 下线 重启 停止 管理Servant 更多命令

服务实时状态

时间	服务ID	线程ID	结果
No matching records found			

在管理系统上的部署暂时先到这里，到此为止，只是使你的服务在管理系统上占了个位置，真实程序尚未发布。

5. 服务开发

5.1. 创建服务

5.1.1. 运行tars脚本

▼ Shell | 复制代码

```
1 /usr/local/tars/cpp/script/cmake_tars_server.sh [App] [Server] [Servant]
```

本例中执行：`/usr/local/tars/cpp/script/cmake_tars_server.sh TestApp HelloServer Hello`
命令执行后，会在当前目录的TestApp/HelloServer/src 目录下，生成下面文件：

HelloServer.h HelloServer.cpp Hello.tars HelloImp.h HelloImp.cpp CMakeLists

这些文件，已经包含了最基本的服务框架和默认测试接口实现。

比如

```
root@ubuntu:/home/lqf/tars# /usr/local/tars/cpp/script/cmake_tars_server.sh TestApp HelloServer Hello
```

```
root@ubuntu:/home/lqf/tars# ll
total 52
drwxrwxr-x  4 lqf  lqf   4096 Aug 23 21:04 ./
drwxr-xr-x 47 lqf  lqf  36864 Aug 23 20:20 ../
drwxr-xr-x  5 root root  4096 Aug 23 21:04 HelloServer/
drwxrwxr-x 25 lqf  lqf   4096 Aug 23 17:44 tarsFramework/
```

5.1.2. tars接口文件

定义tars接口文件的语法和使用，参见tars_tup.md。

如下：

Hello.tars：

```
1  module TestApp
2  {
3
4  interface Hello
5  {
6      int test();
7  };
8
9  };
```

采用tars2cpp工具自动生成c++文件：/usr/local/tars/cpp/tools/tars2cpp Hello.tars会生成Hello.h文件，里面包含客户端和服务端的代码。

5.1.3. HelloImp是Servant的接口实现类

实现服务定义的tars件中的接口，如下：

HelloImp.h

```

1  #ifndef _HelloImp_H_
2  #define _HelloImp_H_
3
4  #include "servant/Application.h"
5  #include "Hello.h"
6
7  /**
8   * HelloImp继承hello.h中定义的Hello对象
9   *
10  */
11  class HelloImp : public TestApp::Hello
12  {
13  public:
14      /**
15       *
16       */
17      virtual ~HelloImp() {}
18
19      /**
20       * 初始化, Hello的虚拟函数, HelloImp初始化时调用
21       */
22      virtual void initialize();
23
24      /**
25       * 析构, Hello的虚拟函数, 服务析构HelloImp退出时调用
26       */
27      virtual void destroy();
28
29      /**
30       * 实现tars文件中定义的test接口
31       */
32      virtual int test(tars::TarsCurrentPtr current) { return 0;};
33
34  };
35  //////////////////////////////////////
36  #endif

```

HelloImp.cpp:

```
1 ▾ #include "HelloImp.h"
2   #include "servant/Application.h"
3
4   using namespace std;
5
6   //////////////////////////////////////
7   void HelloImp::initialize()
8   ▾ {
9       //initialize servant here:
10      //...
11  }
12
13  //////////////////////////////////////
14  void HelloImp::destroy()
15  ▾ {
16      //destroy servant here:
17      //...
18  }
```

5.1.4. HelloServer是服务的实现类

如下:

HelloServer.h:

```
1  #ifndef _HelloServer_H_
2  #define _HelloServer_H_
3
4  #include <iostream>
5  #include "servant/Application.h"
6
7  using namespace tars;
8
9  /**
10   * HelloServer继承框架的Application类
11   **/
12  class HelloServer : public Application
13  {
14  public:
15      /**
16       *
17       **/
18      virtual ~HelloServer() {};
19
20      /**
21       * 服务的初始化接口
22       **/
23      virtual void initialize();
24
25      /**
26       * 服务退出时的清理接口
27       **/
28      virtual void destroyApp();
29  };
30
31  extern HelloServer g_app;
32
33  //////////////////////////////////////
34  #endif
```

HelloServer.cpp

```

1  ▾ #include "HelloServer.h"
2  #include "HelloImp.h"
3
4  using namespace std;
5
6  HelloServer g_app;
7
8  ///////////////////////////////////////////////////////////////////
9  void
10 HelloServer::initialize()
11 ▾ {
12     //initialize application here:
13
14     //添加Servant接口实现类HelloImp与路由Obj绑定关系
15     addServant<HelloImp>(ServerConfig::Application + "." +
16 ServerConfig::ServerName + ".HelloObj");
17 }
18 ///////////////////////////////////////////////////////////////////
19 void
20 HelloServer::destroyApp()
21 ▾ {
22     //destroy application here:
23     //...
24 }
25 ///////////////////////////////////////////////////////////////////
26 int
27 main(int argc, char* argv[])
28 ▾ {
29     try
30     ▾ {
31         g_app.main(argc, argv);
32         g_app.waitForShutdown();
33     }
34     catch (std::exception& e)
35     ▾ {
36         cerr << "std::exception:" << e.what() << std::endl;
37     }
38     catch (...)
39     ▾ {
40         cerr << "unknown exception." << std::endl;
41     }
42     return -1;
43 }
44 ///////////////////////////////////////////////////////////////////

```


5.2. 服务编译

进入代码目录,首先做

```
1 cd build
2 cmake ..
3 make -j4
4 # 打包服务, 打包后为HelloServer.tgz
5 make HelloServer-tar
```

C++ | 复制代码

5.3. 扩展功能

Tars框架提供了接口定义语言的功能,可以在tars文件中,增加一下接口和方法,扩展服务的功能。

1. 修改xxx.tars文件

可以修改由cmake_tars_server.sh生成的tars文件,以下3个接口方法中,test是默认生成的,testHello是新增加的接口。修改Hello.tars:

```
1 module TestApp
2 {
3
4 interface Hello
5 {
6     int test();
7     int testHello(string sReq, out string sRsp);
8 };
9
10 };
11
```

C++ | 复制代码

使用/usr/local/tars/cpp/tools/tars2cpp Hello.tars,重新生成Hello.h。

修改HelloImp.h/HelloImp.cpp,实现新的接口代码。

2. 修改对应的实现接口

(1) 其中HelloImp.h中继承Hello类的testHello方法:

C++ | 复制代码

```
1 virtual int testHello(const std::string &sReq, std::string &sRsp,  
    tars::TarsCurrentPtr current);
```

(2) HelloImp.cpp实现testHello方法:

C++ | 复制代码

```
1 int HelloImp::testHello(const std::string &sReq, std::string &sRsp,  
    tars::TarsCurrentPtr current)  
2 {  
3     TLOGDEBUG("HelloImp::testHellosReq:"<<sReq<<endl);  
4     sRsp = sReq;  
5     return 0;  
6 }
```

重新

Bash | 复制代码

```
1 make clean all  
2 make  
3 make tar
```

会重新生成HelloServer.tgz发布包。

```

Built target tar
root@ubuntu:/home/lqf/tars/HelloServer/build# ll
total 860
drwxr-xr-x 6 root root 4096 Aug 23 21:32 ./
drwxr-xr-x 5 root root 4096 Aug 23 21:04 ../
drwxr-xr-x 2 root root 4096 Aug 23 21:32 bin/
-rw-r--r-- 1 root root 12089 Aug 23 21:04 CMakeCache.txt
drwxr-xr-x 10 root root 4096 Aug 23 21:32 CMakeFiles/
-rw-r--r-- 1 root root 1543 Aug 23 21:04 cmake_install.cmake
-rw-r--r-- 1 root root 794237 Aug 23 21:32 HelloServer.tgz 生成的压缩包
-rw-r--r-- 1 root root 10915 Aug 23 21:04 Makefile
-rw-r--r-- 1 root root 163 Aug 23 21:04 run-k8s-upload.cmake
-rw-r--r-- 1 root root 610 Aug 23 21:04 run-k8s-upload-HelloServer.cmake
-rw-r--r-- 1 root root 479 Aug 23 21:04 run-k8s-upload-tars-HelloServer.cmake
-rw-r--r-- 1 root root 170 Aug 23 21:04 run-release.cmake
-rw-r--r-- 1 root root 102 Aug 23 21:04 run-tar.cmake
-rw-r--r-- 1 root root 837 Aug 23 21:04 run-tar-HelloServer.cmake
-rw-r--r-- 1 root root 168 Aug 23 21:04 run-upload.cmake
-rw-r--r-- 1 root root 170 Aug 23 21:04 run-upload-tars.cmake
drwxr-xr-x 3 root root 4096 Aug 23 21:04 src/
drwxr-xr-x 3 root root 4096 Aug 23 21:32 tmp/

```

5.4. 客户端同步/异步调用服务

在开发环境上，创建/home/tarsproto/[APP]/[Server]目录。

例如：/home/tarsproto/TestApp/HelloServer在刚才编写服务器的代码目录下，

执行 make HelloServer-release 这时会在/home/tarsproto/TestApp/HelloServer目录下生成h、tars和mk文件。

```

root@ubuntu:/home/lqf/tars/HelloServer/build# make HelloServer-release
[ 20%] Built target tars-HelloServer
[ 80%] Built target HelloServer
Scanning dependencies of target HelloServer-release
[100%] call /home/lqf/tars/HelloServer/build/src/run-release-HelloServer.cmake
cp -rf /home/lqf/tars/HelloServer/src/Hello.h /home/tarsproto/TestApp/HelloServer
cp -rf /home/lqf/tars/HelloServer/src/Hello.tars /home/tarsproto/TestApp/HelloServer
[100%] Built target HelloServer-release

```

这样在有某个服务需要访问HelloServer时，就直接引用HelloServer服务make release的内容，不需要把HelloServer的tars拷贝过来（即代码目录下不需要存放HelloServer的tars文件）。

建立客户端代码目录，如TestHelloClient/。

编写main.cpp，创建实例并调用刚编写的接口函数进行测试。

5.4.1 同步方式

```
1  ▾ #include <iostream>
2  #include "servant/Communicator.h"
3  #include "Hello.h"
4
5  using namespace std;
6  using namespace TestApp;
7  using namespace tars;
8
9  int main(int argc, char ** argv)
10 ▾ {
11     Communicator comm;
12
13     try
14 ▾ {
15         HelloPrx prx;
16         comm.stringToProxy("TestApp.HelloServer.HelloObj@tcp -h
10.120.129.226 -p 20001" , prx);    // -h xxx 填对应的地址
17
18         try
19 ▾ {
20             string sReq("hello world");
21             string sRsp("");
22
23             int iRet = prx->testHello(sReq, sRsp); // 同步调用
24             cout<<"iRet:"<<iRet<<" sReq:"<<sReq<<" sRsp:"<<sRsp<<endl;
25
26         }
27         catch(exception &ex)
28 ▾ {
29             cerr << "ex:" << ex.what() << endl;
30         }
31         catch(...)
32 ▾ {
33             cerr << "unknown exception." << endl;
34         }
35     }
36     catch(exception& e)
37 ▾ {
38         cerr << "exception:" << e.what() << endl;
39     }
40     catch (...)
41 ▾ {
42         cerr << "unknown exception." << endl;
43     }
44 }
```

```
45     return 0;  
46 }
```

5.4.2 异步方式

```
1  #include <iostream>
2  #include "servant/Communicator.h"
3  #include "Hello.h"
4
5  using namespace std;
6  using namespace TestApp;
7  using namespace tars;
8
9  class HelloCallBack : public HelloPrxCallback
10 {
11 public:
12     HelloCallBack(){}
13
14     virtual ~HelloCallBack(){}
15
16     virtual void callback_testHello(tars::Int32 ret, const std::string&
sRsp)
17     {
18         cout<<"callback_testHello ret:"<< ret << "|sRsp:" << sRsp <<endl;
19     }
20
21     virtual void callback_testHello_exception(tars::Int32 ret)
22     {
23         cout<<"callback_testHello_exception ret:"<< ret <<endl;
24     }
25 };
26
27 int main(int argc,char ** argv)
28 {
29     Communicator comm;
30
31     try
32     {
33         HelloPrx prx;
34         comm.stringToProxy("TestApp.HelloServer.HelloObj@tcp -h
10.120.129.226 -p 20001" , prx);
35
36         try
37         {
38             string sReq("hello world");
39             HelloPrxCallbackPtr cb = new HelloCallBack();
40             prx->async_testHello(cb, sReq);
41             cout<<" sReq:"<<sReq<<endl;
42         }
43         catch(exception &ex)
```

```

44     {
45         cerr<<"ex:"<<ex.what() <<endl;
46     }
47     catch(...)
48     {
49         cerr<<"unknown exception."<<endl;
50     }
51 }
52 catch(exception& e)
53 {
54     cerr<<"exception:"<<e.what() <<endl;
55 }
56 catch (...)
57 {
58     cerr<<"unknown exception."<<endl;
59 }
60
61 getchar();
62
63 return 0;
64 }

```

5.4.3 编译客户端程序

编写Makefile, 引用刚才/home/tarsproto/APP/Server, 如下:

C++ | 复制代码

```

1  #-----
2  APP      :=TestApp
3  TARGET   :=TestHelloClient
4  CONFIG   :=
5  STRIP_FLAG := N
6
7  INCLUDE  += -I/home/tarsproto/TestApp/HelloServer/
8  LIB      +=
9  #-----
10 include /usr/local/tars/cpp/makefile/makefile.tars
11 #-----

```



```
root@ubuntu:/home/lqf/tars/TestHelloClient# ll
total 16
drwxr-xr-x 2 root root 4096 Aug 23 21:39 ./
drwxrwxr-x 5 lqf lqf 4096 Aug 23 21:35 ../
-rw-r--r-- 1 root root 929 Aug 23 21:38 main.cpp
-rw-r--r-- 1 root root 424 Aug 23 21:39 Makefile
```

make出目标文件，上传到能访问服务器的环境中进行运行测试即可。执行make后的目录

```
root@ubuntu:/home/lqf/tars/TestHelloClient# ll
total 4560
drwxr-xr-x 2 root root 4096 Aug 23 21:40 ./
drwxrwxr-x 5 lqf lqf 4096 Aug 23 21:35 ../
-rw-r--r-- 1 root root 929 Aug 23 21:38 main.cpp
-rw-r--r-- 1 root root 3599 Aug 23 21:40 .main.d
-rw-r--r-- 1 root root 1739688 Aug 23 21:40 main.o
-rw-r--r-- 1 root root 424 Aug 23 21:39 Makefile
-rwxr-xr-x 1 root root 2907872 Aug 23 21:40 TestHelloClient*
```

这里你也可以通过cmake来管理，也强烈建议你通过cmake管理！

6. 服务发布

下面截图的IP可能会有不同，对应的ip地址以自己机器为主，不需要手动修改。

6.1 服务部署

这个可以先做，也可以在开发完服务程序代码后再做，建议先做。步骤如下：

- 进入Tars管理界面，点击运维管理
- 根据页面内容，设定应用名、服务名称、Obj等（这里设定的名称要与后续开发代码时设定的一致，否则部署不成功。

6.1.1 添加应用

在运维管理添加应用，一定要添加应用（官方文档有误,没有写这个步骤），否则客户端找不到对应的应用。



互联网数据中心（Internet Data Center）简称IDC。



6.1.2 添加服务



- 创建完成，可在服务部署页面的右侧服务树看到（需要刷新）。

TARS

Cache

服务管理

运维管理

中文

请输入内容...

tars

TestApp

HelloServer

刷新后才有的

服务管理

发布管理

服务配置

服务监控

特性监控

接口调试

权限管理

更多命令

批量编辑

服务列表

<input type="checkbox"/>	服务	节点	启用Set	设置状态	当前状态	流量状态	进程ID	版本	发布人	发布时间	操作
<input checked="" type="checkbox"/>	HelloServer	192.168.206.128	不启用	Off	Off	Active	0			1970-01-01 00:00:00	编辑 重启

服务实时状态

时间	服务ID	线程ID	结果
没有数据			

添加服务后创建的配置文件

/usr/local/app/tars/tarsnode/data/TestApp.HelloServer/conf/TestApp.HelloServer.config.conf

6.2 服务发布

在管理系统的菜单树下，找到你部署的服务，点击进入服务页面。

选择“发布管理”，选中要发布的节点，点击“发布选中节点”，点击“上传发布包”，选择已经编译好的发布包，如下图。

6.2.1 选中节点上传发布包

TARS

Cache

服务管理

运维管理

网关配置

中文

请输入内容...

tars

TestApp

HelloServer

服务管理

发布管理

服务配置

服务监控

特性监控

接口调试

权限管理

调用链

服务列表

<input checked="" type="checkbox"/>	节点	启用Set	设置状态	当前状态
<input checked="" type="checkbox"/>	192.168.0.143	不启用	Inactive	Inactive

上传发布包

发布选中节点

历史记录

管理发布包

发布管理

×

应用·服务名

TestApp-HelloServer

待发布节点

192.168.0.143

备注

发布版本

请选择

▼

上传发布包

☐ 普通发布

上传发布包

发布包

请选择

选择

HelloServer.tgz

备注

6.2.2 发布发布包

上传好发布包后，点击“选择发布版本”下拉框就会出现你上传的服务程序，选择最上面的一个（最新上传的）。如下图：

发布管理



应用·服务名

TestApp-HelloServer

待发布节点

192.168.0.143

备注

发布版本

12 | PubTime: | UploadTime:2022-03-30 20:07:29 |admin|

上传发布包

☐ 普通发布

发布

管理发布包

点击“发布”，服务开始发布，发布成功后，切换到“服务管理”出现下面的界面，如下图：

Service Management Interface Screenshot:

- Navigation: tars, TestApp, HelloServer
- Service List Table:

服务	节点	启用Set	设置状态	当前状态	流量状态	进程ID	版本	发布人	发布时间	操作
HelloServer	192.168.206.128	不启用	Active	Active	Active	23194	24	admin	2021-08-24 15:24:27	编辑

若失败的话，可能是命名问题，上传问题，以及其他环境问题。

日志查看路径：

```
tail -f /usr/local/app/tars/app_log/tars/tarsAdminRegistry/tars.tarsAdminRegistry.log
```

```
tail -f /usr/local/app/web/log/20220330.info.log
```

异常

服务列表 C										查看服务状态	更多命令	批量编辑	批量重启	更多
<input type="checkbox"/>	服务	节点	节点状态	启用Set	设置状态	当前状态	流量状态	进程ID	版本	发布人	发布时间	操作		
<input type="checkbox"/>	HelloServer	192.168.0.143(node无效)	不启用	Active	Inactive	Active	40379				1970-01-01 00:00:00	编辑	重启	停止
												暂停	Servant管理	
												查看服务状态	查看模板	更多命令

6.3 客户端测试

6.3.1 修改main.cpp

修改TestHelloClient的main.cpp的

```
comm.stringToProxy("TestApp.HelloServer.HelloObj@tcp -h 10.120.129.226 -p 20001" , prx);
```

修改为对server应节点的IP地址加server对应的端口。

```
comm.stringToProxy("TestApp.HelloServer.HelloObj@tcp -h 192.168.206.128 -p 22785" , prx);
```

6.3.2 重新编译和执行程序

然后重新make编译。

运行执行程序，正常情况下如图所示。

```
root@ubuntu:/home/lqf/tars/TestHelloClient# ./TestHelloClient
iRet:0 sReq:hello world sRsp:hello world
```

6.3.3 异常情况

(1) TARSSERVERNOSERVANTERR = -4; //服务器端没有该Servant对象

主要原因是tars官方的文档没有添加应用TestApp的操作，参考《6.1.1节 添加应用》。

C++ | 复制代码

```
1  ex:server servant mismatch exception: ret:-4 msg:[ServantProxy::invoke
   errno:-4,info:,servant:TestApp.HelloServer.HelloObj,func:testHello,adapte
   r:tcp 192.168.206.128 -p 22785 -t 3000,reqid:1]
2
```

(2) TARSASYNCCALLTIMEOUT = -7; //异步调用超时

设置的server ip地址或者端口不对。参考《6.1.2节 添加服务》的IP地址和端口。

▼ C++ 复制代码

```
1 ex:server unknown exception: ret:-7 msg:[ServantProxy::invoke  
  errno:-7,info:,servant:TestApp.HelloServer.HelloObj,func:testHello,adapte  
  r:tcp -h 127.0.0.1 -p 2785 -t 3000,reqid:0]
```

6.3.4 如果不直连server呢？

tars.tarsregistry.QueryObj

```

1  ▾ #include <iostream>
2  #include "servant/Communicator.h"
3  #include "Hello.h"
4
5  using namespace std;
6  using namespace TestApp;
7  using namespace tars;
8  static string helloObj = "TestApp.HelloServer.HelloObj";
9  int main(int argc, char ** argv)
10 ▾ {
11     Communicator comm;
12
13     try
14     {
15         HelloPrx prx;
16         comm.setProperty("locator", "tars.tarsregistry.QueryObj@tcp -h
17 192.168.206.128 -t 60000 -p 17890"); // 去注册中心查找对应的服务程序
18         prx = comm.stringToProxy<HelloPrx>(helloObj);
19         try
20         {
21             string sReq("hello world");
22             string sRsp("");
23
24             int iRet = prx->testHello(sReq, sRsp);
25             cout<<"iRet:"<<iRet<<" sReq:"<<sReq<<" sRsp:"<<sRsp<<endl;
26
27         }
28         catch(exception &ex)
29         {
30             cerr << "ex:" << ex.what() << endl;
31         }
32         catch(...)
33         {
34             cerr << "unknown exception." << endl;
35         }
36     }
37     catch(exception& e)
38     {
39         cerr << "exception:" << e.what() << endl;
40     }
41     catch (...)
42     {
43         cerr << "unknown exception." << endl;
44     }

```



```
45     return 0;
46 }
```

FAQ

返回错误值

▼ C++ 复制代码

```
1 //定义TARS服务给出的返回码
2 const int TARSSERVERSUCCESS = 0; //服务器端处理成功
3 const int TARSSERVERDECODEERR = -1; //服务器端解码异常
4 const int TARSSERVERENCODEERR = -2; //服务器端编码异常
5 const int TARSSERVERNOFUNCERR = -3; //服务器端没有该函数
6 const int TARSSERVERNOSERVANTERR = -4; //服务器端没有该Servant对象
7 const int TARSSERVERRESETGRID = -5; //服务器端灰度状态不一致
8 const int TARSSERVERQUEUEETIMEOUT = -6; //服务器队列超过限制
9 const int TARSASYNCCALLTIMEOUT = -7; //异步调用超时
10 const int TARSINVOKETIMEOUT = -7; //调用超时
11 const int TARSProxyCONNECTERR = -8; //proxy链接异常
12 const int TARSSERVEROVERLOAD = -9; //服务器端超负载,超过队列长度
13 const int TARSADAPTERNULL = -10; //客户端选路为空,服务不存在或者
    所有服务down掉了
14 const int TARSINVOKEBYINVALIDESET = -11; //客户端按set规则调用非法
15 const int TARSCLIENTDECODEERR = -12; //客户端解码异常
16 const int TARSSERVERUNKNOWNERR = -99; //服务器端位置异常
```

在initializeServer()中会填充ServerConfig里面的各个静态成员变量，留待需要的时候取用。可以看到有_epollServer = new TC_EpollServer(iNetThreadNum)，服务端TC_EpollServer被创建出来，而且网络线程NetThread也被建立出来了：

```
TC_EpollServer::TC_EpollServer(unsigned int iNetThreadNum) { if(_netThreadNum < 1) {
    _netThreadNum = 1; } //网络线程的配置数目不能15个 if(_netThreadNum > 15) {
    _netThreadNum = 15; } for (size_t i = 0; i < _netThreadNum; ++i) {
    TC_EpollServer::NetThread* netThreads = new TC_EpollServer::NetThread(this);
    _netThreads.push_back(netThreads); }}
```

此后，其实有一个AdminAdapter被建立，但其与一般

日志

```
tail -f -n 100 xx.log
```

```
root@ubuntu:/usr/local/app# find -name *.log
```

```
./tars/app_log/UNKNOWN/tarspatch/UNKNOWN.tarspatch.log
```

```
./tars/app_log/UNKNOWN/tarslog/UNKNOWN.tarslog.log
```

```
./tars/app_log/TestApp/HelloServer/TestApp.HelloServer.log
```

```
./tars/app_log/tars/tarsnotify/tars.tarsnotify_EX_20220330.log
```

```
./tars/app_log/tars/tarsnotify/tars.tarsnotify.log
```

```
./tars/app_log/tars/tarsquerystat/tars.tarsquerystat.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsquerystat.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsAdminRegistry.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.KeepAliveThread.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsproperty.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarspatch.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.ReportMemThread.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsnotify.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsconfig.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarslog.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsstat.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.TestApp.HelloServer.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsqueryproperty.log
```

```
./tars/app_log/tars/tarsnode/tars.tarsnode.tars.tarsregistry.log
```

```
./tars/app_log/tars/tarsAdminRegistry/tars.tarsAdminRegistry.log
```

```
./tars/app_log/tars/tarsproperty/tars.tarsproperty.log
```

```
./tars/app_log/tars/tarsconfig/tars.tarsconfig.log
```

```
./tars/app_log/tars/tarspatch/tars.tarspatch.log
```

```
./tars/app_log/tars/tarsqueryproperty/tars.tarsqueryproperty.log
```

```
./tars/app_log/tars/tarsqueryproperty/tars.tarsqueryproperty_inout_20220330.log
```

```
./tars/app_log/tars/tarslog/tars.tarslog_20220330.log
```

```
./tars/app_log/tars/tarslog/tars.tarslog.log
```

```
./tars/app_log/tars/tarsstat/tars.tarsstat.log
```

```
./tars/app_log/tars/tarsstat/tars.tarsstat_CountStat_20220330.log
```

```
./tars/app_log/tars/tarsregistry/tars.tarsregistry.log
```

```
./tars/app_log/tars/tarsregistry/tars.tarsregistry.ReapThread.log
./web/log/20220330.error.log
./web/log/20220330.info.log
./web/log/20220330.warn.log
./web/node_modules/cluster/test/logs/nested/workers.access.log
./web/node_modules/cluster/test/logs/nested/workers.error.log
./web/node_modules/cluster/test/logs/nested/master.log
root@ubuntu:/usr/local/app#
```

```
tail -f -n 100 ./tars/app_log/TestApp/HelloServer/TestApp.HelloServer.log
2022-03-30 21:46:38|139869234452352|DEBUG|[TARS]accept [192.168.0.143:59298] [19]
incoming
2022-03-30 21:46:38|139868957042432|DEBUG|HelloImp::testHellosReq:hello world
```

调用栈

```
#0 tars::TC_TCPTransceiver::send (this=0x90d040, buf=0x7fff0005170, len=77, flag=0)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_transceiver.cpp:1016
#1 0x00000000005e0e4f in tars::TC_Transceiver::sendRequest (this=0x90d040, buff=..., addr=...)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_transceiver.cpp:741
#2 0x00000000004ea0cf in tars::AdapterProxy::invoke_connection_parallel (this=0x90ca70,
msg=0x90dc30)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/servant/libservant/AdapterProxy.cpp:382
#3 0x00000000004eaded in tars::AdapterProxy::invoke (this=0x90ca70, msg=0x90dc30)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/servant/libservant/AdapterProxy.cpp:476
#4 0x000000000048bd71 in tars::ObjectProxy::prepareConnection (this=0x90b0c0,
adapterProxy=0x90ca70)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/servant/libservant/ObjectProxy.cpp:234
#5 0x000000000048bdf7 in tars::ObjectProxy::onConnect (this=0x90b0c0,
adapterProxy=0x90ca70)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/servant/libservant/ObjectProxy.cpp:241
#6 0x00000000004e8607 in tars::AdapterProxy::onConnectCallback (this=0x90ca70,
trans=0x90d040)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/servant/libservant/AdapterProxy.cpp:162
#7 0x00000000004f9821 in std::__invoke_impl<void, void (tars::AdapterProxy::*&)
(tars::TC_Transceiver*), tars::AdapterProxy*&, tars::TC_Transceiver*> (__f=
@0x90d490: (void (tars::AdapterProxy::*)(tars::AdapterProxy * const, tars::TC_Transceiver *))
0x4e8530 <tars::AdapterProxy::onConnectCallback(tars::TC_Transceiver*>, __t=@0x90d4a0:
0x90ca70) at /usr/local/gcc-11.2/include/c++/11.2.0/bits/invoke.h:74
```

```

#8 0x00000000004f8c94 in std::__invoke<void (tars::AdapterProxy::*)(tars::TC_Transceiver*),
tars::AdapterProxy*&, tars::TC_Transceiver*> (__fn=
    @0x90d490: (void (tars::AdapterProxy::*)(tars::AdapterProxy * const, tars::TC_Transceiver *))
0x4e8530 <tars::AdapterProxy::onConnectCallback(tars::TC_Transceiver*)>) at /usr/local/gcc-
11.2/include/c++/11.2.0/bits/invoke.h:96
#9 0x00000000004f7bd7 in std::_Bind<void (tars::AdapterProxy::*(tars::AdapterProxy*,
std::_Placeholder<1>))(tars::TC_Transceiver*)>::__call<void, tars::TC_Transceiver*&&, 0ul, 1ul>
(std::tuple<tars::TC_Transceiver*&&>&&, std::_Index_tuple<0ul, 1ul>) (
    this=0x90d490, __args=...) at /usr/local/gcc-11.2/include/c++/11.2.0/functional:420
#10 0x00000000004f6673 in std::_Bind<void (tars::AdapterProxy::*(tars::AdapterProxy*,
std::_Placeholder<1>))(tars::TC_Transceiver*)>::operator()<tars::TC_Transceiver*, void>
(tars::TC_Transceiver*&&) (this=0x90d490)
    at /usr/local/gcc-11.2/include/c++/11.2.0/functional:503
#11 0x00000000004f501b in std::__invoke_impl<void, std::_Bind<void (tars::AdapterProxy::*
(tars::AdapterProxy*, std::_Placeholder<1>))(tars::TC_Transceiver*)>&, tars::TC_Transceiver*>
(std::__invoke_other, std::_Bind<void (tars::AdapterProxy::*(tars::AdapterProxy*--Type <RET>
for more, q to quit, c to continue without paging--
, std::_Placeholder<1>))(tars::TC_Transceiver*)>&, tars::TC_Transceiver*&&) (__f=...)
    at /usr/local/gcc-11.2/include/c++/11.2.0/bits/invoke.h:61
#12 0x00000000004f36a0 in std::__invoke_r<void, std::_Bind<void (tars::AdapterProxy::*
(tars::AdapterProxy*, std::_Placeholder<1>))(tars::TC_Transceiver*)>&, tars::TC_Transceiver*>
(std::_Bind<void (tars::AdapterProxy::*(tars::AdapterProxy*, std::_Placeholder<1>))
(tars::TC_Transceiver*)>&, tars::TC_Transceiver*&&) (__fn=...) at /usr/local/gcc-
11.2/include/c++/11.2.0/bits/invoke.h:154
#13 0x00000000004f1ee4 in std::_Function_handler<void (tars::TC_Transceiver*),
std::_Bind<void (tars::AdapterProxy::*(tars::AdapterProxy*, std::_Placeholder<1>))
(tars::TC_Transceiver*)> >::_M_invoke(std::_Any_data const&, tars::TC_Transceiver*&&) (
    __functor=..., __args#0=@0x7ffff56c64d0: 0x90d040) at /usr/local/gcc-
11.2/include/c++/11.2.0/bits/std_function.h:291
#14 0x000000000005e3989 in std::function<void (tars::TC_Transceiver*)>::operator()
(tars::TC_Transceiver*) const (this=0x90d2a0,
    __args#0=0x90d040) at /usr/local/gcc-11.2/include/c++/11.2.0/bits/std_function.h:560
#15 0x000000000005dff23 in tars::TC_Transceiver::onSetConnected (this=0x90d040)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_transceiver.cpp:433
#16 0x000000000005dfd9e in tars::TC_Transceiver::setConnected (this=0x90d040)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_transceiver.cpp:421
#17 0x000000000005df274 in tars::TC_Transceiver::checkConnect (this=0x90d040)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_transceiver.cpp:293
#18 0x000000000005e0bcd in tars::TC_Transceiver::doRequest (this=0x90d040)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_transceiver.cpp:645

```

```

#19 0x000000000046f402 in tars::CommunicatorEpoll::handleOutputImp (this=0x906580,
data=...)
    at /home/lqf/tars/TarsFramework-
v3.0.5/tarscpp/servant/libservant/CommunicatorEpoll.cpp:388
#20 0x0000000000482771 in std::__invoke_impl<bool, bool (tars::CommunicatorEpoll::*&)
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&), tars::CommunicatorEpoll*&,
std::shared_ptr<tars::TC_Epoller::EpollInfo> const&> (__f=
    @0x7ffff00050c0: (bool (tars::CommunicatorEpoll::*)(tars::CommunicatorEpoll * const, const
std::shared_ptr<tars::TC_Epoller::EpollInfo> &)) 0x46f36c
<tars::CommunicatorEpoll::handleOutputImp(std::shared_ptr<tars::TC_Epoller::EpollInfo>
const&)>,
    __t=@0x7ffff00050d0: 0x906580) at /usr/local/gcc-11.2/include/c++/11.2.0/bits/invoke.h:74
#21 0x0000000000481d77 in std::__invoke<bool (tars::CommunicatorEpoll::*&)
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&), tars::CommunicatorEpoll*&,
std::shared_ptr<tars::TC_Epoller::EpollInfo> const&> (__fn=
    @0x7ffff00050c0: (bool (tars::CommunicatorEpoll::*)(tars::CommunicatorEpoll * const, const
std::shared_ptr<tars::TC_Epoller::EpollInfo> &)) 0x46f36c
<tars::CommunicatorEpoll::handleOutputImp(std::shared_ptr<tars::TC_Epoller::EpollInfo>
const&)>)
    at /usr/local/gcc-11.2/include/c++/11.2.0/bits/invoke.h:96
--Type <RET> for more, q to quit, c to continue without paging--
#22 0x0000000000480f43 in std::_Bind<bool (tars::CommunicatorEpoll::*
(tars::CommunicatorEpoll*, std::_Placeholder<1>))(std::shared_ptr<tars::TC_Epoller::EpollInfo>
const&)>::__call<bool, std::shared_ptr<tars::TC_Epoller::EpollInfo> const&, 0ul, 1ul>
(std::tuple<std::shared_ptr<tars::TC_Epoller::EpollInfo> const&>&&, std::_Index_tuple<0ul, 1ul>)
(this=0x7ffff00050c0, __args=...)
    at /usr/local/gcc-11.2/include/c++/11.2.0/functional:420
#23 0x0000000000480031 in std::_Bind<bool (tars::CommunicatorEpoll::*
(tars::CommunicatorEpoll*, std::_Placeholder<1>))(std::shared_ptr<tars::TC_Epoller::EpollInfo>
const&)>::operator()<std::shared_ptr<tars::TC_Epoller::EpollInfo> const&, bool>
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&) (this=0x7ffff00050c0) at /usr/local/gcc-
11.2/include/c++/11.2.0/functional:503
#24 0x000000000047eb75 in std::__invoke_impl<bool, std::_Bind<bool
(tars::CommunicatorEpoll::*)(tars::CommunicatorEpoll*, std::_Placeholder<1>))
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&)>&,
std::shared_ptr<tars::TC_Epoller::EpollInfo> const&>(std::__invoke_other, std::_Bind<bool
(tars::CommunicatorEpoll::*)(tars::CommunicatorEpoll*, std::_Placeholder<1>))
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&)>&,
std::shared_ptr<tars::TC_Epoller::EpollInfo> const&) (__f=...)
    at /usr/local/gcc-11.2/include/c++/11.2.0/bits/invoke.h:61

```

```

#25 0x000000000047cffa in std::__invoke_r<bool, std::_Bind<bool (tars::CommunicatorEpoll::*
(tars::CommunicatorEpoll*, std::_Placeholder<1>)))(std::shared_ptr<tars::TC_Epoller::EpollInfo>
const&>&, std::shared_ptr<tars::TC_Epoller::EpollInfo> const&>(std::_Bind<bool
(tars::CommunicatorEpoll::* (tars::CommunicatorEpoll*, std::_Placeholder<1>))
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&>&,
std::shared_ptr<tars::TC_Epoller::EpollInfo> const&) (__fn=...) at /usr/local/gcc-
11.2/include/c++/11.2.0/bits/invoke.h:142
#26 0x000000000047a3c7 in std::_Function_handler<bool
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&), std::_Bind<bool
(tars::CommunicatorEpoll::* (tars::CommunicatorEpoll*, std::_Placeholder<1>))
(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&>>::_M_invoke(std::_Any_data const&,
std::shared_ptr<tars::TC_Epoller::EpollInfo> const&) (__functor=..., __args#0=...)
    at /usr/local/gcc-11.2/include/c++/11.2.0/bits/std_function.h:291
#27 0x000000000058dc3b in std::function<bool (std::shared_ptr<tars::TC_Epoller::EpollInfo>
const&>::operator()(std::shared_ptr<tars::TC_Epoller::EpollInfo> const&) const
(this=0x7ffff0004d38, __args#0=...)
    at /usr/local/gcc-11.2/include/c++/11.2.0/bits/std_function.h:560
#28 0x000000000058b39f in tars::TC_Epoller::EpollInfo::fireEvent (this=0x7ffff0004cd0, event=4)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_epoller.cpp:110
#29 0x000000000058c548 in tars::TC_Epoller::done (this=0x7ffff0000f60, ms=1000)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_epoller.cpp:634
#30 0x0000000000549e00 in tars::TC_CoroutineScheduler::run (this=0x7ffff0000900)
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_coroutine.cpp:503
#31 0x00000000005c91c0 in tars::TC_Thread::coroutineEntry (pThread=0x906580, iPoolSize=3,
iStackSize=131072, autoQuit=false)
--Type <RET> for more, q to quit, c to continue without paging--
    at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_thread.cpp:178
#32 0x00000000005cb2dc in std::__invoke_impl<void, void (*) (tars::TC_Thread*, unsigned int,
unsigned long, bool), tars::TC_Thread*, unsigned int, unsigned long, bool> (
    __f=@0x90a9c8: 0x5c9048 <tars::TC_Thread::coroutineEntry(tars::TC_Thread*, unsigned int,
unsigned long, bool)>)
    at /usr/local/gcc-11.2/include/c++/11.2.0/bits/invoke.h:61
#33 0x00000000005cb13d in std::__invoke<void (*) (tars::TC_Thread*, unsigned int, unsigned
long, bool), tars::TC_Thread*, unsigned int, unsigned long, bool> (
    __fn=@0x90a9c8: 0x5c9048 <tars::TC_Thread::coroutineEntry(tars::TC_Thread*, unsigned int,
unsigned long, bool)>)
    at /usr/local/gcc-11.2/include/c++/11.2.0/bits/invoke.h:96
#34 0x00000000005cafad in std::thread::_Invoker<std::tuple<void (*) (tars::TC_Thread*,
unsigned int, unsigned long, bool), tars::TC_Thread*, unsigned int, unsigned long, bool>
>::_M_invoke<0ul, 1ul, 2ul, 3ul, 4ul> (this=0x90a9a8)

```

```

at /usr/local/gcc-11.2/include/c++/11.2.0/bits/std_thread.h:253
#35 0x00000000005caef6 in std::thread::_Invoker<std::tuple<void (*) (tars::TC_Thread*, unsigned
int, unsigned long, bool), tars::TC_Thread*, unsigned int, unsigned long, bool> >::operator()
(this=0x90a9a8)
at /usr/local/gcc-11.2/include/c++/11.2.0/bits/std_thread.h:260
#36 0x00000000005caeba in std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)
(tars::TC_Thread*, unsigned int, unsigned long, bool), tars::TC_Thread*, unsigned int, unsigned
long, bool> > >::_M_run (this=0x90a9a0)
at /usr/local/gcc-11.2/include/c++/11.2.0/bits/std_thread.h:211
#37 0x00007ffff7884e80 in std::execute_native_thread_routine (__p=0x90a9a0) at
.././.././.././libstdc++-v3/src/c++11/thread.cc:82
#38 0x00007ffff7bc16ba in start_thread () from /lib/x86_64-linux-gnu/libpthread.so.0
#39 0x00007ffff6fd251d in clone () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) info threads

```

Id	Target Id	Frame
1	Thread 0x7fff7fd0780 (LWP 42180) "TestHelloClient"	0x00007ffff7bc7360 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/x86_64-linux-gnu/libpthread.so.0
2	Thread 0x7fff6eca700 (LWP 42191) "TestHelloClient"	0x00007ffff7bc7709 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/x86_64-linux-gnu/libpthread.so.0
3	Thread 0x7fff66c9700 (LWP 42192) "TestHelloClient"	0x00007ffff7bc7709 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/x86_64-linux-gnu/libpthread.so.0
4	Thread 0x7fff5ec8700 (LWP 42193) "TestHelloClient"	0x00007ffff7bc7709 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/x86_64-linux-gnu/libpthread.so.0
* 5	Thread 0x7fff56c7700 (LWP 42194) "TestHelloClient"	tars::TC_TCPTransceiver::send (this=0x90d040, buf=0x7fff0005170, len=77, flag=0) at /home/lqf/tars/TarsFramework-v3.0.5/tarscpp/util/src/tc_transceiver.cpp:1016
6	Thread 0x7fff4ec6700 (LWP 42195) "TestHelloClient"	0x00007ffff7bcac1d in nanosleep () from /lib/x86_64-linux-gnu/libpthread.so.0
7	Thread 0x7ffeffff700 (LWP 42196) "TestHelloClient"	0x00007ffff7bc7709 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/x86_64-linux-gnu/libpthread.so.0