**单机版rest\_rpc与brpc性能对比测试**

目录

[一 测试环境 2](#_Toc18320_WPSOffice_Level1)

[二 测试方法 2](#_Toc24835_WPSOffice_Level1)

[三 测试代码编写及分析 3](#_Toc8272_WPSOffice_Level1)

[四 测试数据汇总 10](#_Toc13407_WPSOffice_Level1)

**一 测试环境**

IP：192.168.97.30

OS：18.04.1-Ubuntu SMP Fri Mar 15 15:27:12 UTC 2019

CPU：6Core，Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz

内存：16G

g++版本：g++ (Ubuntu 8.2.0-1ubuntu2~18.04) 8.2.0

机器类别：实体机

**二 测试方法**

1. server与client均为单进程并且部署在同一台实体机；
2. server开启多线程处理，线程数是CPU核数；
3. client是单线程；
4. client循环向server发送一定字节的字符串，比如由‘A’构成的1K字符串，server收到后不做业务处理，立即返回给client，client收到server返回的字符串也不做业务处理；
5. 当循环结束后，client统计延时；
6. server每隔1s会向标准输出设备打印出收到的请求数；
7. 采用rest\_rpc与brpc默认的编译方式（cmake -DCMAKE\_BUILD\_TYPE=Release）。

**三 测试代码编写及分析**

1. rest\_rpc测试代码
2. 下载源码：<https://github.com/qicosmos/rest_rpc>
3. 解压后进入examples/server目录，修改main.cpp，增加函数：std::string echo(connection\* conn, const std::string& orignal)

std::string echo(connection\* conn, const std::string& orignal) {

g\_qps.increase();

return orignal;

}

echo函数将客户端从conn连接传过来的字符串original原封不动得返回，并调用increase函数进行请求计数。

在main函数增加server.register\_handler("echo", echo);用于向server注册消息处理函数。

1. examples/client目录，修改main.cpp，在test\_performance1()函数中进行请求发送：

void test\_performance1() {

rpc\_client client("127.0.0.1", 9000);

bool r = client.connect();

if (!r) {

std::cout << "connect timeout" << std::endl;

return;

}

**string str(1024, 'A');**

auto begin = high\_resolution\_clock::now();

for (size\_t i = 0; i < LOOP; i++) {

auto future = client.async\_call("echo", str);

auto status = future.wait\_for(std::chrono::seconds(2));

if (status == std::future\_status::deferred) {

std::cout << "deferred\n";

}

else if (status == std::future\_status::timeout) {

std::cout << "timeout\n";

}

else if (status == std::future\_status::ready) {

}

}

auto end = high\_resolution\_clock::now();

cout << "elapse time = " << duration\_cast<seconds>(end - begin).count() << "s" << endl;

std::cout << "finish\n";

}

1. 代码修改完毕后，执行cmake -DCMAKE\_BUILD\_TYPE=Release），make clean，make，然后启动server与client进行测试。
2. rbpc测试代码
3. 下载源码：<https://github.com/apache/incubator-brpc.git>
4. 按官网编译指令下载相应包后进行编译

进入目录examples/echo\_c++，修改server.cpp，在virtual void Echo(google::protobuf::RpcController\* cntl\_base,

const EchoRequest\* request, EchoResponse\* response, google::protobuf::Closure\* done)

注释相应LOG日志：

virtual void Echo(google::protobuf::RpcController\* cntl\_base,

const EchoRequest\* request,

EchoResponse\* response,

google::protobuf::Closure\* done) {

// This object helps you to call done->Run() in RAII style. If you need

// to process the request asynchronously, pass done\_guard.release().

brpc::ClosureGuard done\_guard(done);

brpc::Controller\* cntl =

static\_cast<brpc::Controller\*>(cntl\_base);

// The purpose of following logs is to help you to understand

// how clients interact with servers more intuitively. You should

// remove these logs in performance-sensitive servers.

/\*LOG(INFO) << "Received request[log\_id=" << cntl->log\_id()

<< "] from " << cntl->remote\_side()

<< " to " << cntl->local\_side()

<< ": " << request->message()

<< " (attached=" << cntl->request\_attachment() << ")";\*/

// Fill response.

response->set\_message(request->message());

**g\_qps.increase();**

// You can compress the response by setting Controller, but be aware

// that compression may be costly, evaluate before turning on.

// cntl->set\_response\_compress\_type(brpc::COMPRESS\_TYPE\_GZIP);

if (FLAGS\_echo\_attachment) {

// Set attachment which is wired to network directly instead of

// being serialized into protobuf messages.

cntl->response\_attachment().append(cntl->request\_attachment());

}

}

代码行response->set\_message(request->message());将将客户端的请求原封不动发回客户端。

1. 修改client.cpp，增加：

const static size\_t BUF\_SIZE = 1024;

const static int LOOP = 1000000;

char buf[BUF\_SIZE] = ""

修改main.函数：：

int main(int argc, char\* argv[]) {

memset(buf, 'A', BUF\_SIZE);

// Parse gflags. We recommend you to use gflags as well.

GFLAGS\_NS::ParseCommandLineFlags(&argc, &argv, true);

// A Channel represents a communication line to a Server. Notice that

// Channel is thread-safe and can be shared by all threads in your program.

brpc::Channel channel;

// Initialize the channel, NULL means using default options.

brpc::ChannelOptions options;

options.protocol = FLAGS\_protocol;

options.connection\_type = FLAGS\_connection\_type;

options.timeout\_ms = FLAGS\_timeout\_ms/\*milliseconds\*/;

options.max\_retry = FLAGS\_max\_retry;

if (channel.Init(FLAGS\_server.c\_str(), FLAGS\_load\_balancer.c\_str(), &options) != 0) {

LOG(ERROR) << "Fail to initialize channel";

return -1;

}

// Normally, you should not call a Channel directly, but instead construct

// a stub Service wrapping it. stub can be shared by all threads as well.

example::EchoService\_Stub stub(&channel);

// Send a request and wait for the response every 1 second.

//int log\_id = 0;

example::EchoRequest request;

example::EchoResponse response;

request.set\_message(buf);

auto begin = high\_resolution\_clock::now();

for (int i = 0; i < LOOP; i++)

{

brpc::Controller cntl;

cntl.request\_attachment().append(FLAGS\_attachment);

stub.Echo(&cntl, &request, &response, NULL);

if (!cntl.Failed()) {

/\*LOG(INFO) << "Received response from " << cntl.remote\_side()

<< " to " << cntl.local\_side()

<< ": " << response.message() << " (attached="

<< cntl.response\_attachment() << ")"

<< " latency=" << cntl.latency\_us() << "us";\*/

}

else {

LOG(WARNING) << cntl.ErrorText();

}

}

auto end = high\_resolution\_clock::now();

cout << "elapse time = " << duration\_cast<seconds>(end - begin).count() << "s" << endl;

std::cout << "finish\n";

/\*while (!brpc::IsAskedToQuit()) {

// We will receive response synchronously, safe to put variables

// on stack.

example::EchoRequest request;

example::EchoResponse response;

brpc::Controller cntl;

//request.set\_message("hello world");

request.set\_message(buf);

cntl.set\_log\_id(log\_id ++); // set by user

// Set attachment which is wired to network directly instead of

// being serialized into protobuf messages.

cntl.request\_attachment().append(FLAGS\_attachment);

// Because `done'(last parameter) is NULL, this function waits until

// the response comes back or error occurs(including timedout).

stub.Echo(&cntl, &request, &response, NULL);

if (!cntl.Failed()) {

LOG(INFO) << "Received response from " << cntl.remote\_side()

<< " to " << cntl.local\_side()

<< ": " << response.message() << " (attached="

<< cntl.response\_attachment() << ")"

<< " latency=" << cntl.latency\_us() << "us";

} else {

LOG(WARNING) << cntl.ErrorText();

}

usleep(FLAGS\_interval\_ms \* 1000L);

}

\*/

//LOG(INFO) << "EchoClient is going to quit";

return 0;

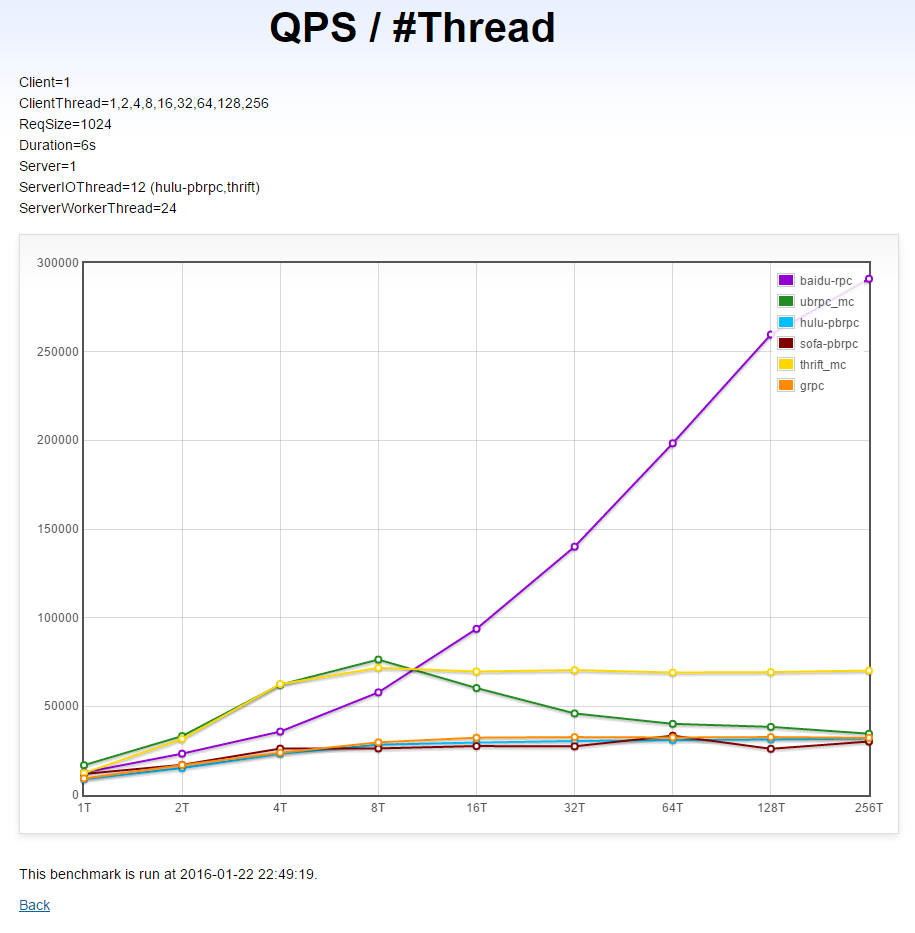
}

去除日志，并统计循环发送延迟，在循环体内只是向server发送报文，并未执行任何业务。

1. 编译方法与rest\_rpc类似。

**四 测试数据汇总**

brpc性能测试数据（<https://github.com/apache/incubator-brpc/blob/master/docs/cn/benchmark.md>）



本次测试使用相同环境与方法对rest\_rpc与brpc进行性能测试，最终测试数据取多次测试数据的平均值，测试数据如下：

|  |  |  |
| --- | --- | --- |
| **消息字节** | **rest\_rpc QPS** | **brpc QPS** |
| 1K | 52631 | 41667 |
| 8K | 42863 | 34483 |
| 16K | 37037 | 29412 |
| 64K | 16949 | 8982 |

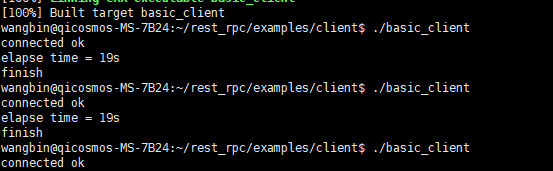
从相关数据可以看出，rest\_rpc QPS要高于brpc，特别是当消息字节达到64K时，rest QPS几乎是brpc的2倍。

附：

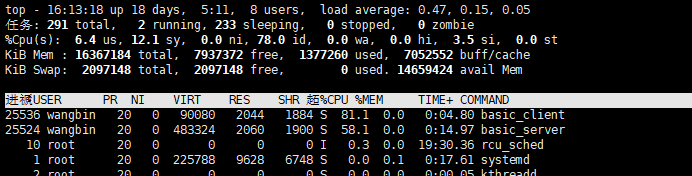
原始测试数据：

rest\_rpc 1k数据

QPS 52631.57

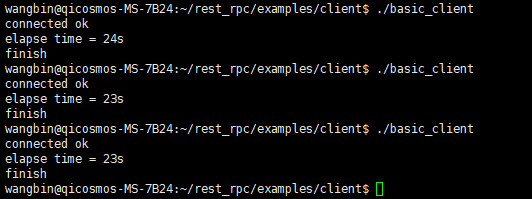


CPU

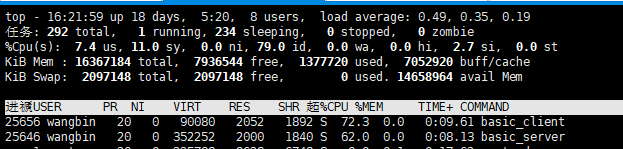


rest\_rpc 8k数据

QPS 42863.26

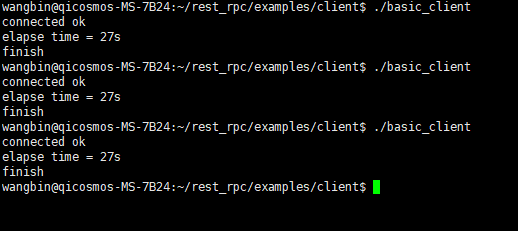


CPU

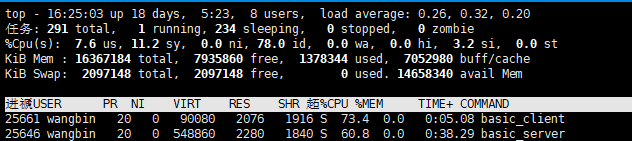


rest\_rpc 16k数据

QPS 37037.03

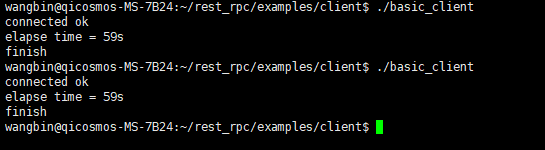


CPU

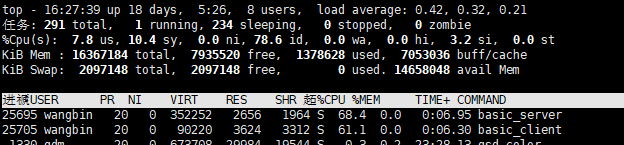


rest\_rpc 64k数据

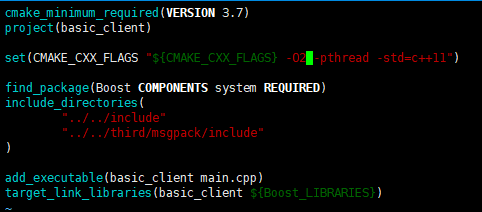
QPS 16949.15



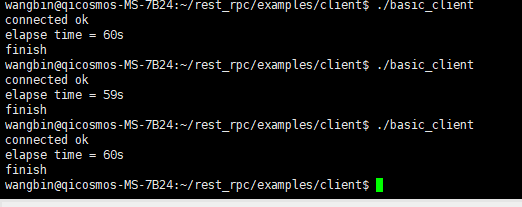
CPU



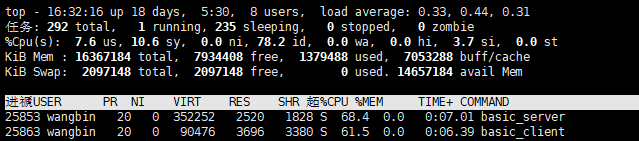
开启O2优化后测试：



QPS 16761.65



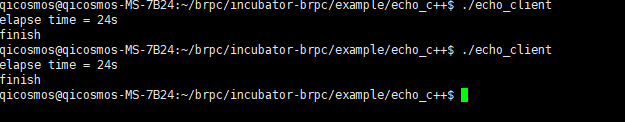
CPU



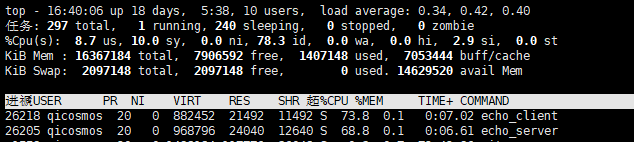
单机：

brpc 1k数据（开启O2）

QPS 41666.67

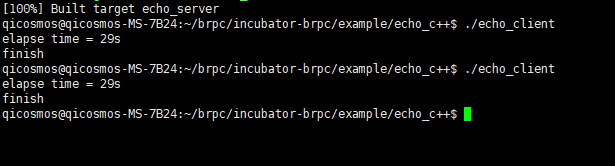


CPU

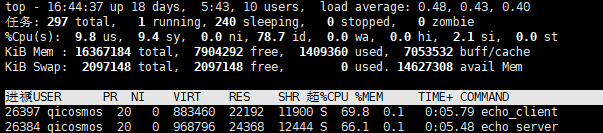


brpc 8k数据

QPS 34482.76

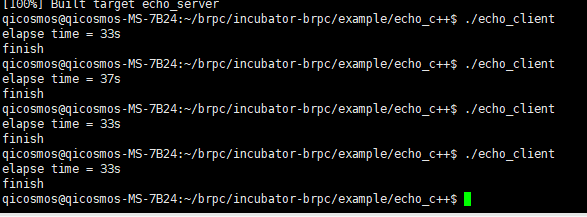


CPU

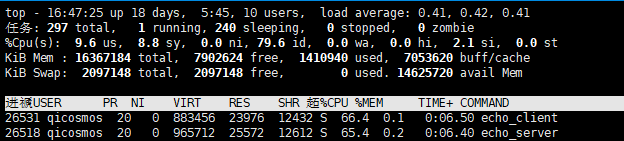


brpc 16k数据

QPS 29411.76

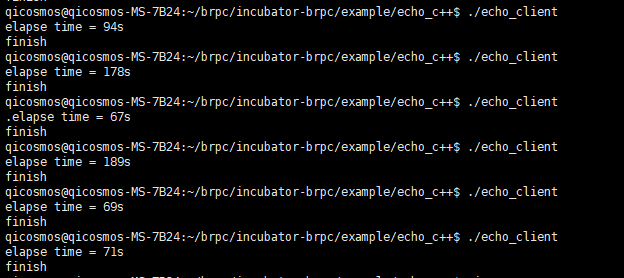


CPU



brpc 64k数据

QPS 8982.30



CPU

