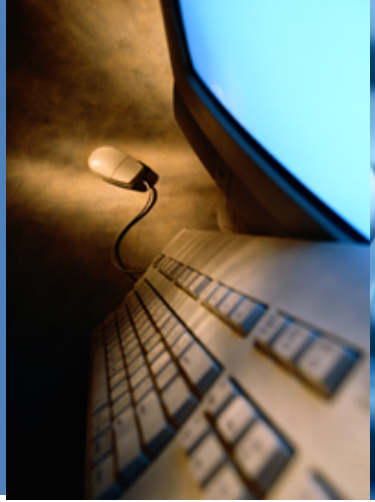
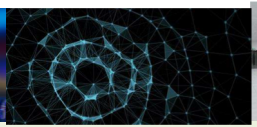
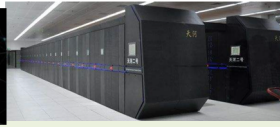


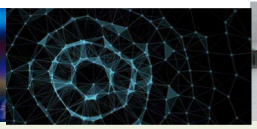

分布式系统
Distributed Systems

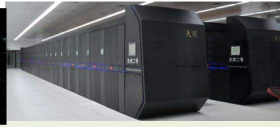
陈鹏飞
计算机学院
chchenpf7@mail.sysu.edu.cn
办公室：学院楼310
主页：http://sdcs.sysu.edu.cn/node/3747



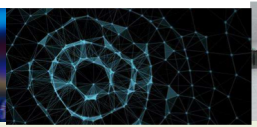


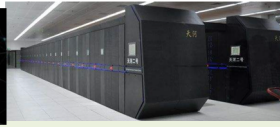
第四讲 — 分布式系统通信





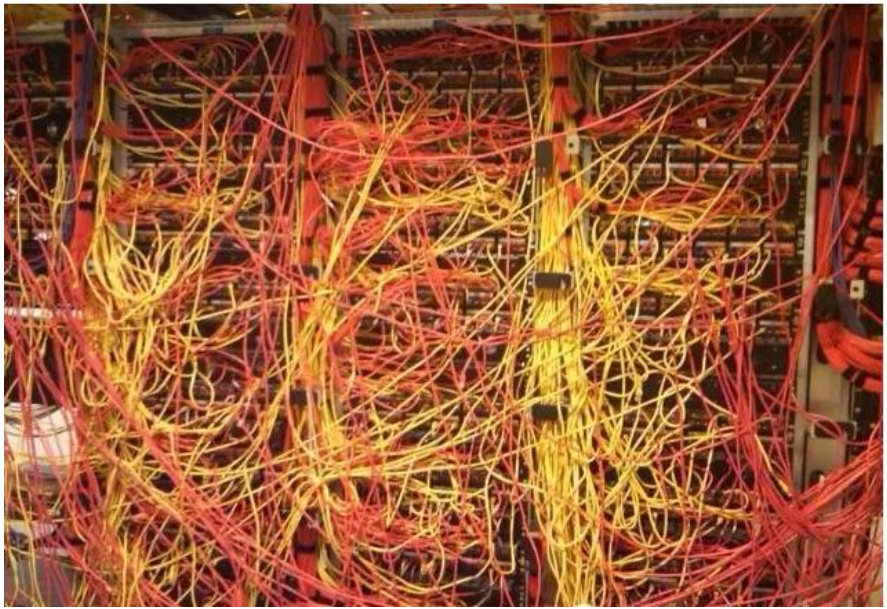
大纲





背景知识

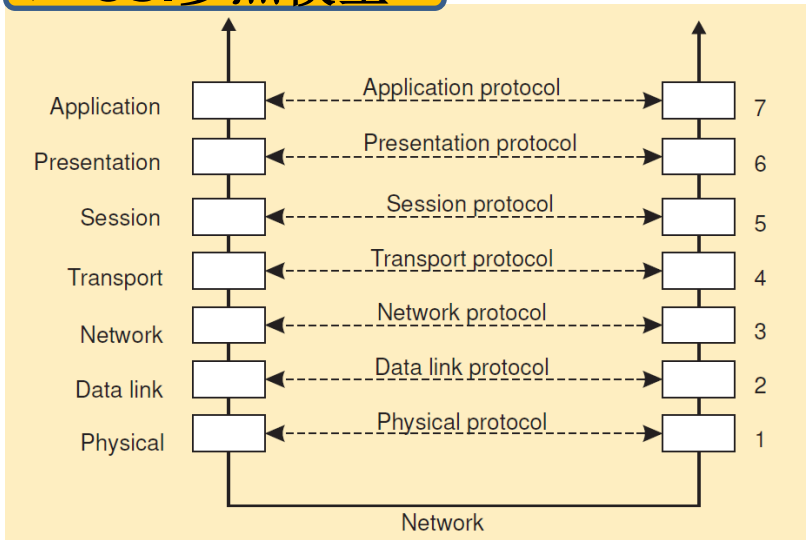
通信



进程间的通信是分布式系统的核心

基本的网络模型

➢ OSI参照模型



协议？

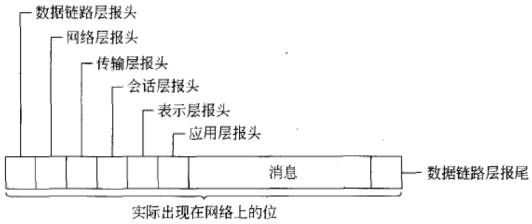


图 4.2 在网络上传输的典型消息

➢ 缺点

- 仅关注消息传递；
- 有些功能通常情况下用不到；
- 违背了访问透明性；

底层通信

➢ 回顾：

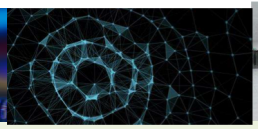
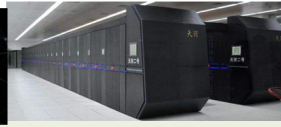
物理层： 包含发送数据的规约和实现，负责发送和接收端的传输；

数据链路层：将发送数据整理成帧，并且提供验错和流控的功能；

网络层：描述数据包如何在网络中的计算机之间如何路由；

➢ 观察：

对于很多分布式系统而言，最底层的接口其实是网络层；

传输层

- 重点：

对于大部分分布式系统，传输层提供实际的通信功能；

- 标准的网络协议：

- TCP：面向连接、可靠的、面向流的通信；
- UDP：不可靠的（尽力而为）数据报文通信；
- Quic?

QUIC 全称：Quick UDP Internet Connections，是一种基于 UDP 的传输层协议。由 Google 自研，2012 年部署上线，2013 年提交 IETF，2021 年 5 月，IETF 推出标准版 RFC9000。

中间件层

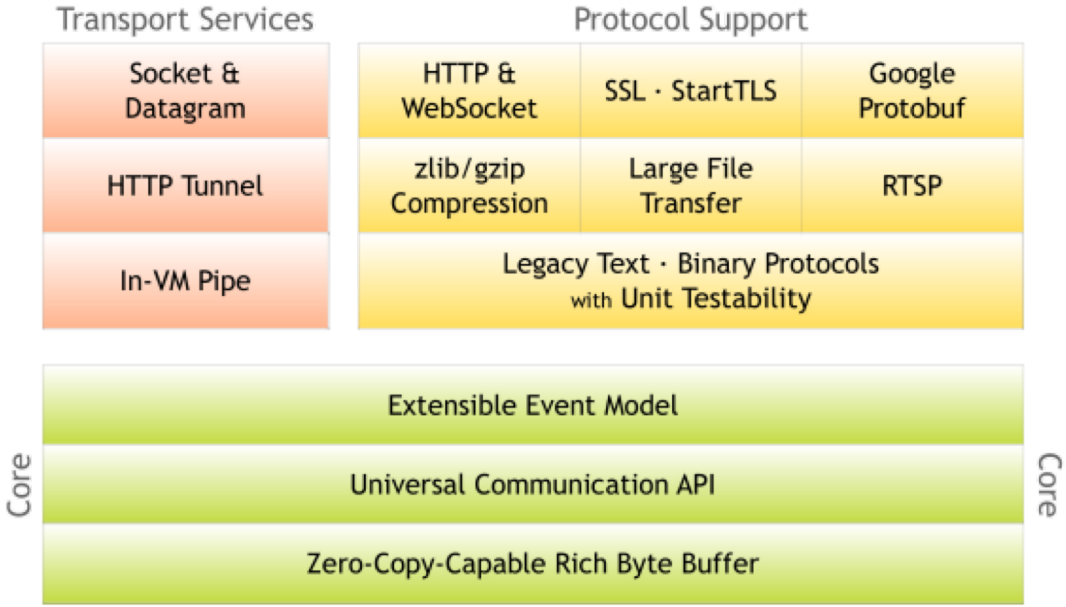
➢ 观察发现：

中间件提供通用的服务和协议，可用于支撑很多不同的应用。

- 包含丰富的通信协议；
- 包装/解包装数据，对于系统集成非常重要；
- 命名协议，允许资源的共享；
- 安全协议用于安全的通信；
- 扩展机制，例如复制和缓存；

中间件层

Netty 是一个基于NIO的客户、服务器端的编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户、服务端应用。Netty相当于简化和流线化了网络应用的编程开发过程，例如：基于TCP和UDP的socket服务开发。



改进后的网络模型

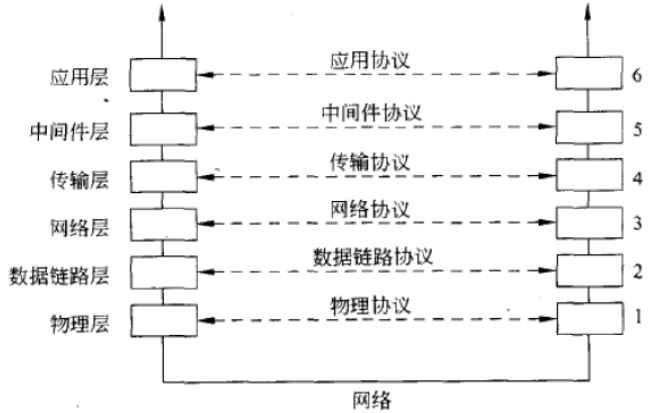
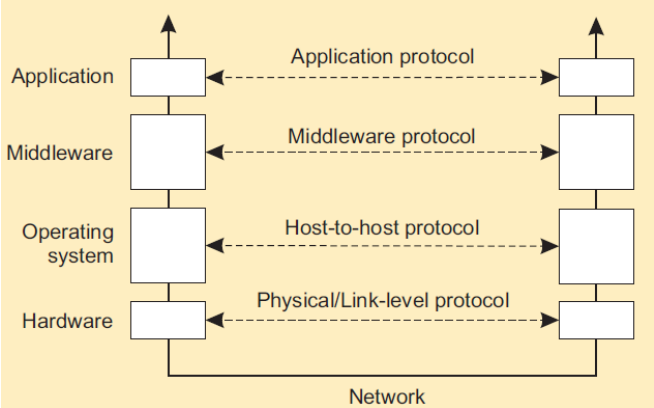
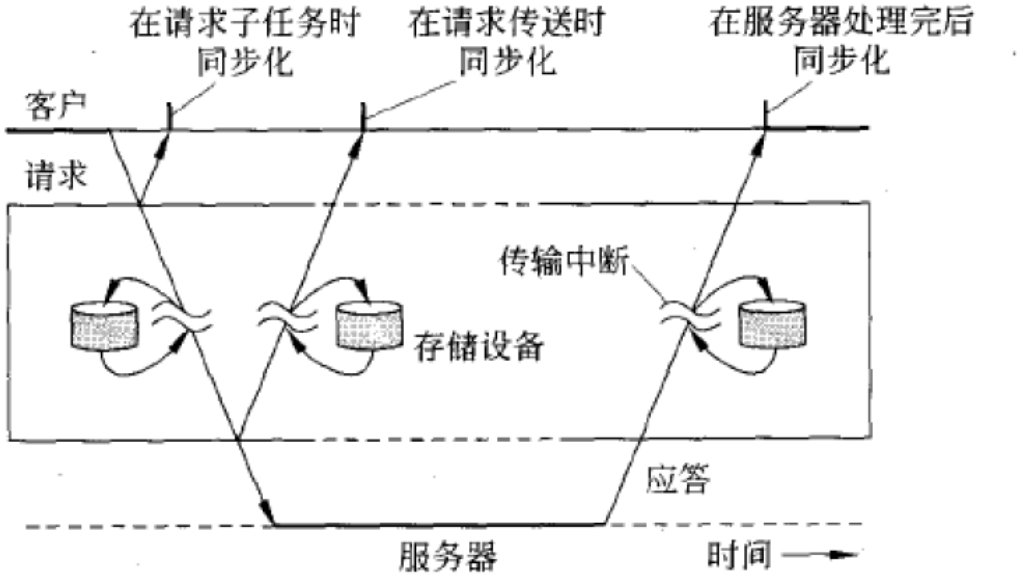


图 4.3 调整后的网络通信参考模型



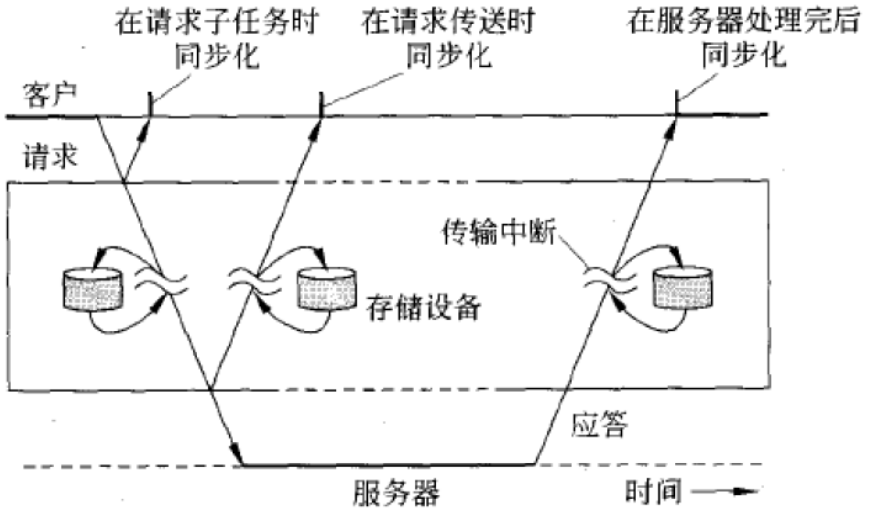
通信类型



邮件系统

- 瞬时通信 VS 持久通信；
- 异步通信 VS 同步通信；（二者之间的组合）

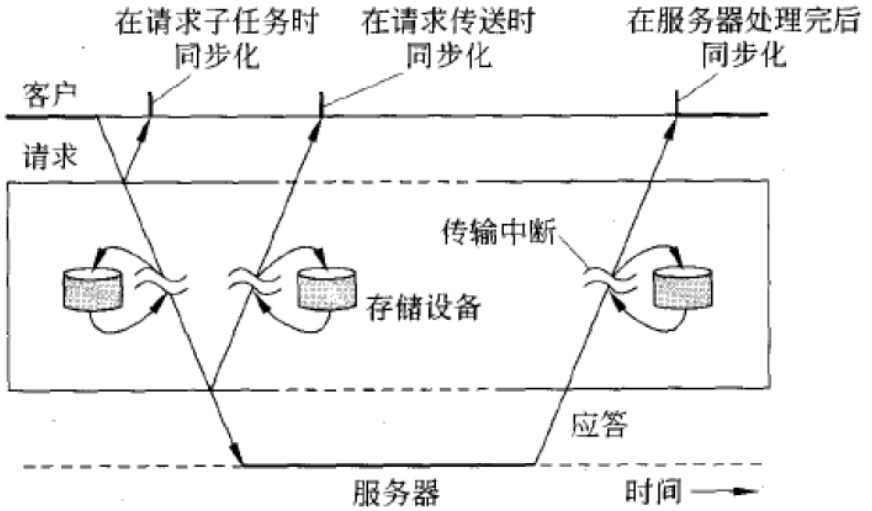
通信类型



- **瞬态 VS 持久通信**

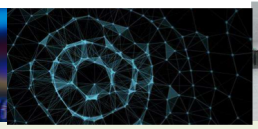
- 瞬态通信：当消息不能传递到另外一个服务器的时候，丢弃消息；
- 持久通信：消息存储在通信服务器直到消息被传递出去；

通信类型



➢ 同步发生时刻

- 请求提交时；
- 请求传输时；
- 请求处理完后；


客户端/服务器通信

➢ 观察发现：

客户/服务器计算系统一般是基于瞬态、同步的通信方法：

- 客户和服务器在通信时必须处于活跃的状态；
- 客户端发出请求后，被阻塞直到收到应答；
- 服务器只是等待到来的请求，然后处理这些请求；

➢ 同步通信的缺点：

- 客户端等待回应的时候不能做其他工作；
- 失效必须即刻处理；
- 对于一些应用，该模型不适用（mail、news）

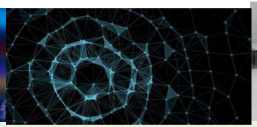
消息

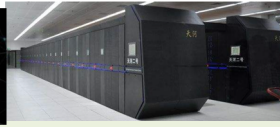
➢ 面向消息的中间件

目的在于进行高层次的持久化、异步通信：

- 进程之间相互发送消息，这些消息会被排队；
- 发送进程可以继续做其他事情，不需要等待及时回应；
- 中间件提供容错机制；

ActiveMQ　RabbitMQ by Pivotal　APACHE kafka® A distributed streaming platform





远程过程调用

基本的过程调用

➢ 观察发现

- 应用程序开发者熟悉见到的过程模型；
- 定义良好的过程操作一般是隔离的（黑盒）；
- 过程函数是可以分离执行的；

➢ 结论

- 调用者和被调用者之间的通信可以通过过程调用的机制隐藏掉；

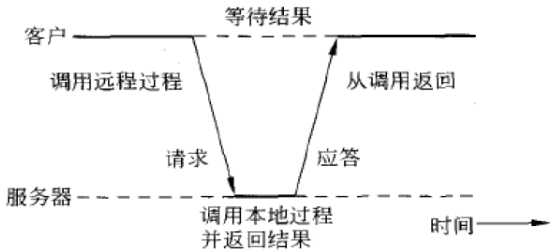
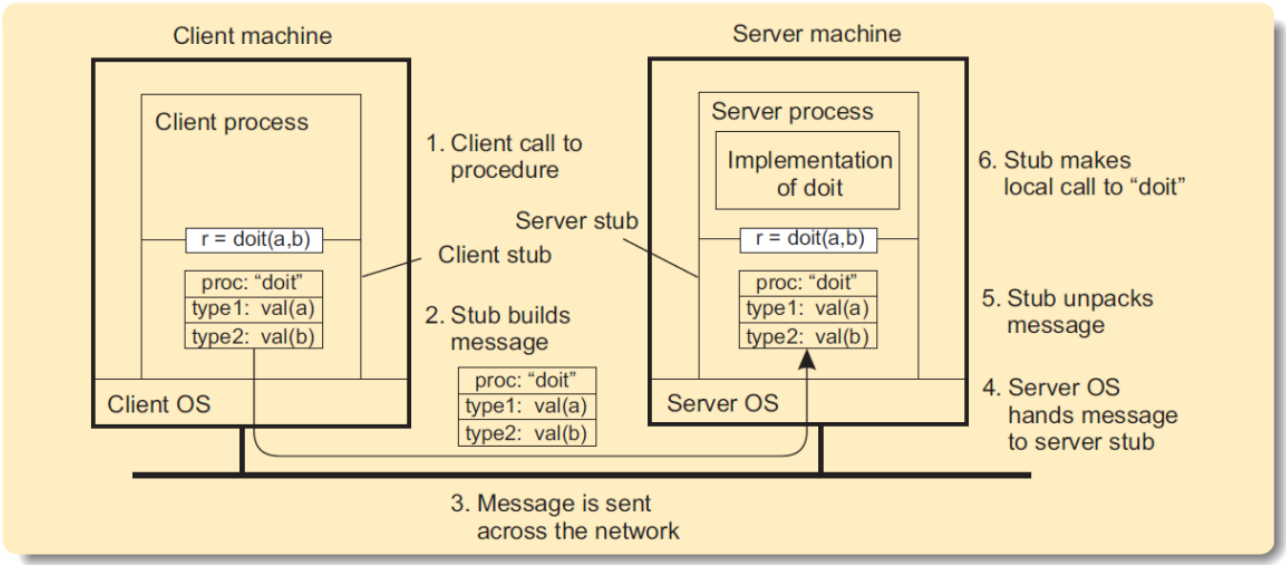


图 4.6 客户与服务器之间的 RPC 的原理

基本的过程调用



1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters; calls server.
6. Server does local call; returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client's OS.
9. Client's OS gives message to stub.
10. Client stub unpacks result; returns to client.

RPC参数传递

➢ 不仅仅是将参数封装在消息中：

- 客户端和服务器可能具有不同的数据表示：
- 封装参数意味着将一个值转换为一个字节序列；
- 客户端和服务器应具有一致的编码机制；
  - ✓ 基本的数据类型如何表示；
  - ✓ 复杂的数据类型如何表示；

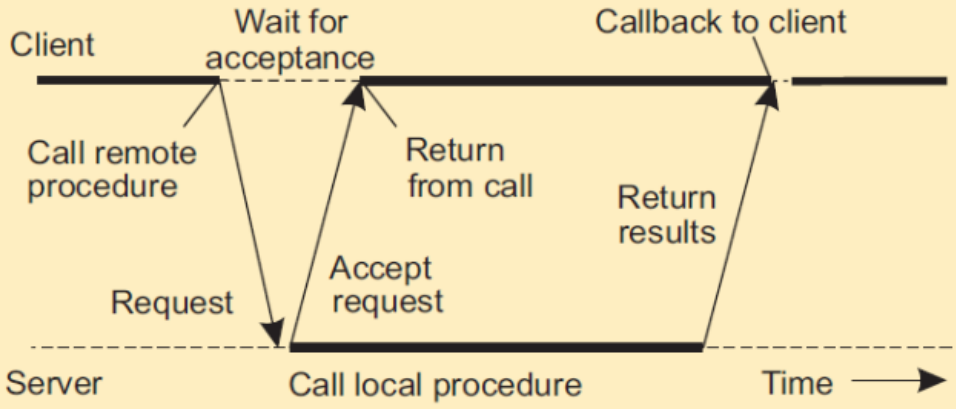
➢ 结论：

客户端和服务器需要正确地解释消息，并将其转换成与机器无关的表达形式。

异步RPC

- **本质：**

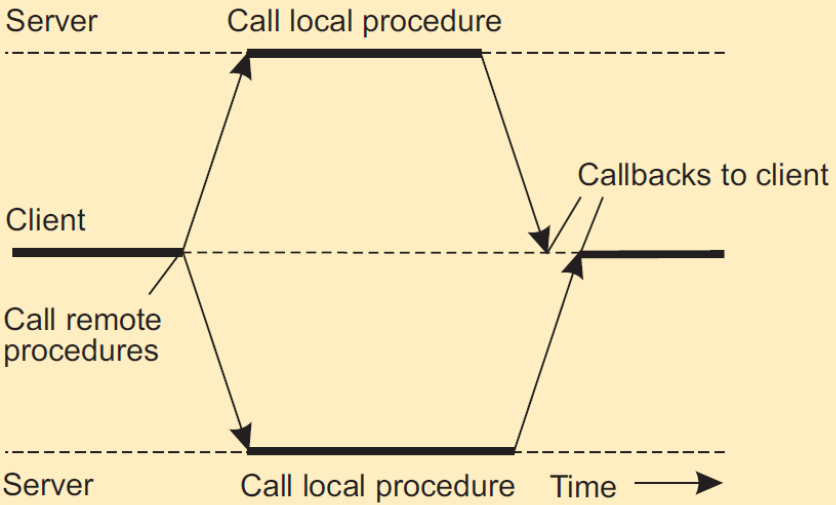
摒弃严格的请求-响应行为，但是让客户端连续运行而不需要等待服务器返回信息；



多RPC调用

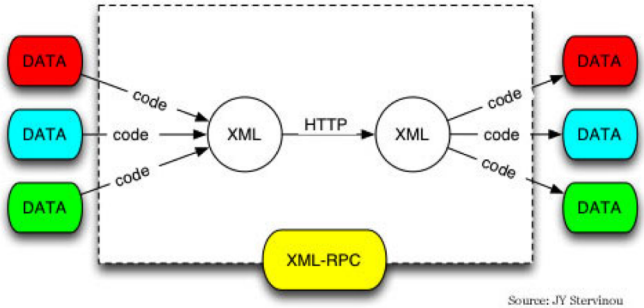
- 本质

  发送RPC请求到一组服务器上

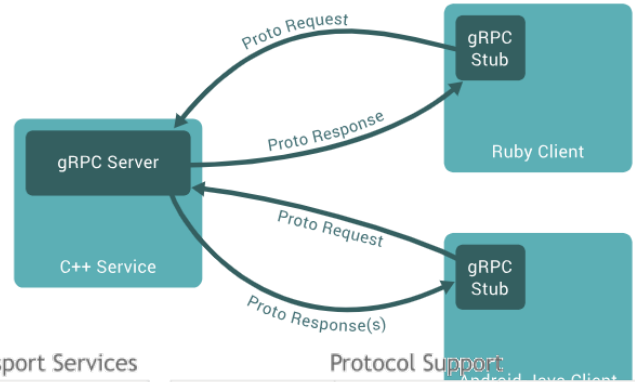


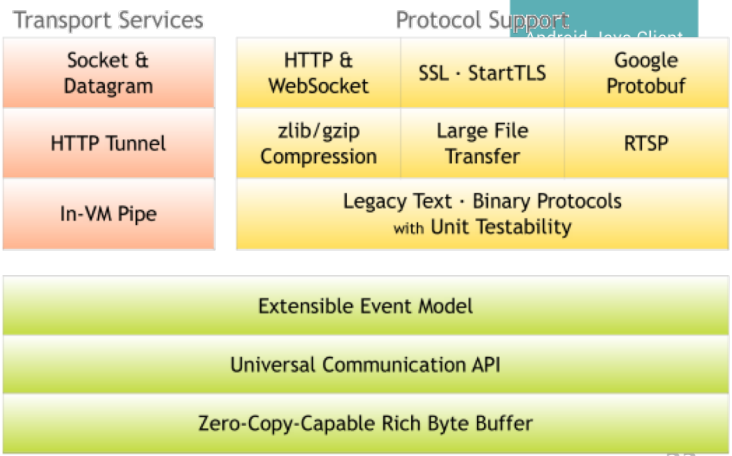
RPC的实际应用

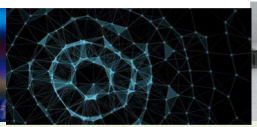
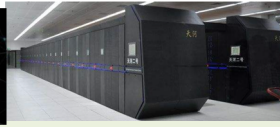
- 关键点
  - 对象序列化协议；
  - 调用控制协议；



Source: JY Stervinou





DCE RPC

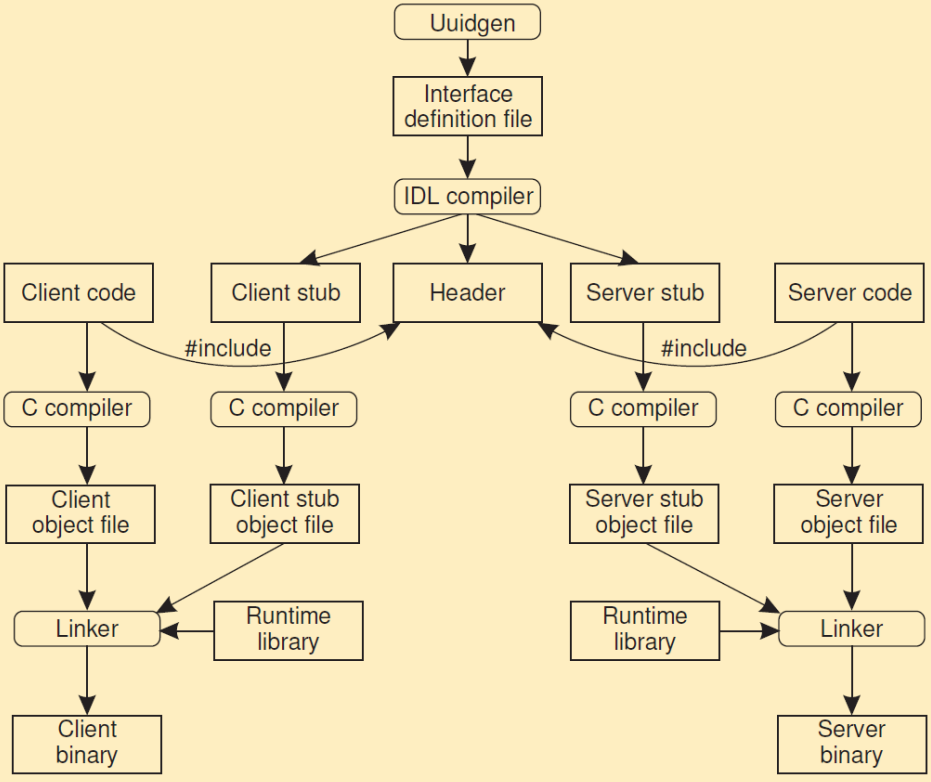
https://en.wikipedia.org/wiki/Distributed_Computing_Environment

- **DCE（分布式计算环境）**

  OSF（开放软件基金会）下的一个软件系统框架（慢慢消亡）

- **DCE的构成**
  - 远程过程调用 DCE/RPC；
  - 命名/目录服务；
  - 时间服务；
  - 安全服务；
  - 分布式文件系统 DCE/DFS；

DCE RPC


Uuidgen
Interface definition file
IDL compiler
Client code
Client stub
Header
Server stub
Server code
#include
#include
C compiler
C compiler
C compiler
C compiler
Client object file
Client stub object file
Server stub object file
Server object file
Linker
Runtime library
Runtime library
Linker
Client binary
Server binary

RPC实现过程

- **Linux的RPC编译工具：rpcgen**
- **先编写接口文件：rpc_test.x**

```
program TESTPROG {
version VERSION {
    string TEST(string) = 1;
} = 1;
} = 1;
```

版本编号

RPC程序编号

RPC函数

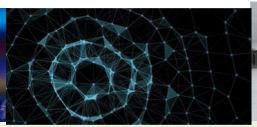
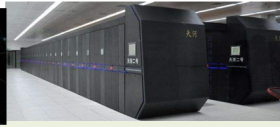
- **编译接口文件：rpcgen rpc_test.x**

  产生 rpc_test_clnt.c、 rpc_test_svc.c、rpc_test.h

RPC实现过程

- **rpc_test_clnt.c的内容**

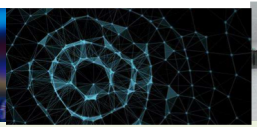
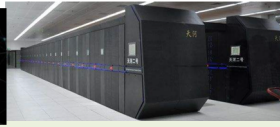
```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "rpc_test.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

char **
test_1(char **argp, CLIENT *clnt)
{
        static char *clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, TEST,
                (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
                (xdrproc_t) xdr_wrapstring, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
~
~
```

RPC实现过程

rpc_test.h

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _RPC_TEST_H_RPCGEN
#define _RPC_TEST_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif


#define TESTPROG 87654321
#define VERSION 1

#if defined(__STDC__) || defined(__cplusplus)
#define TEST 1
extern  char ** test_1(char **, CLIENT *);
extern  char ** test_1_svc(char **, struct svc_req *);
extern int testprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define TEST 1
extern  char ** test_1();
extern  char ** test_1_svc();
extern int testprog_1_freeresult ();
#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_RPC_TEST_H_RPCGEN */
```

RPC实现过程

rpc_test_svc.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "rpc_test.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
testprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        union {
                char *test_1_arg;
        } argument;
        char *result;
        xdrproc_t _xdr_argument, _xdr_result;
        char *(*local)(char *, struct svc_req *);

        switch (rqstp->rq_proc) {
        case NULLPROC:
                (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
                return;

        case TEST:
                _xdr_argument = (xdrproc_t) xdr_wrapstring;
                _xdr_result = (xdrproc_t) xdr_wrapstring;
                local = (char *(*)(char *, struct svc_req *)) test_1_svc;
                break;
```

RPC实现过程

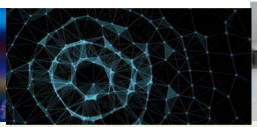
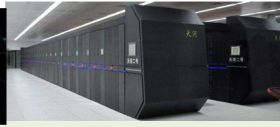
- 生成**client stub**代码：**rpcgen –Sc –o rpc_client.c rpc_test.x**

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "rpc_test.h"


void
testprog_1(char *host)
{
        CLIENT *clnt;
        char * *result_1;
        char * test_1_arg;

#ifndef DEBUG
        clnt = clnt_create (host, TESTPROG, VERSION, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
#endif  /* DEBUG */

        result_1 = test_1(&test_1_arg, clnt);
        if (result_1 == (char **) NULL) {
                clnt_perror (clnt, "call failed");
        }
#ifndef DEBUG
        clnt_destroy (clnt);
#endif   /* DEBUG */
}


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

        if (argc < 2) {
                printf ("usage: %s server_host\n", argv[0]);
                exit (1);
```

RPC实现过程

- 生成server stub代码：**rpcgen –Ss –o rpc_server.c rpc_test.x**

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "rpc_test.h"

char **
test_1_svc(char **argp, struct svc_req *rqstp)
{
        static char * result;

        /*
         * insert server code here
         */

        return &result;
}
~
```

RPC实现过程

```
#include <time.h>
#include "test.h"

char ** test_1_svc(char **argp, struct svc_req *rqstp)
{
static char * result;
static char tmp_char[128];
time_t rawtime;

/*
* insert server code here
*/
if( time(&rawtime) == ((time_t)-1) ) {
strcpy(tmp_char, "Error");
result = tmp_char;
return &result;
}
sprintf(tmp_char, "服务器当前时间是 :%s", ctime(&rawtime));
result = tmp_char;

return &result;
}
```

1. **gcc -Wall -o rpc_client rpc_client.c rpc_test_clnt.c**
2. **gcc -Wall -o rpc_server rpc_server.c rpc_test_svc.c**

服务器端代码

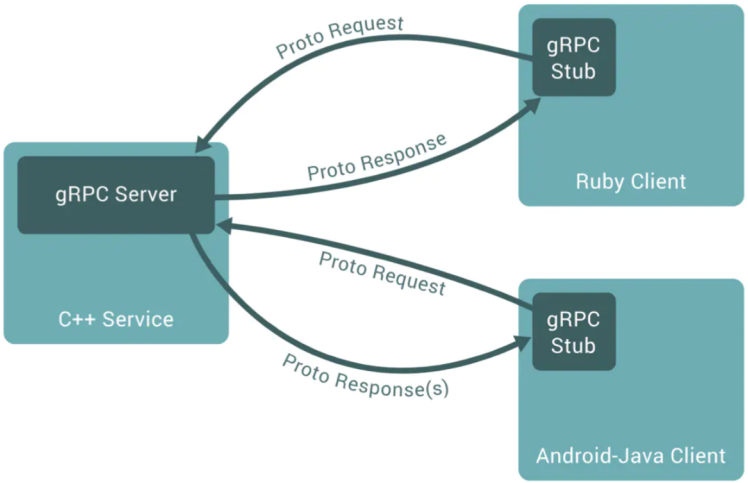
Protobuf & gRPC

- **语言无关、平台无关:** 即 ProtoBuf 支持 Java、C++、Python 等多种语言，支持多个平台
- **高效:** 即比 XML 更小（3 ~ 10倍）、更快（20 ~ 100倍）、更为简单;
- **扩展性、兼容性好:** 你可以更新数据结构，而不影响和破坏原有的旧程序;

```
message Example1 {
    optional string stringVal = 1;
    optional bytes bytesVal = 2;
    message EmbeddedMessage {
        int32 int32Val = 1;
        string stringVal = 2;
    }
    optional EmbeddedMessage embeddedExample1 = 3;
    repeated int32 repeatedInt32Val = 4;
    repeated string repeatedStringVal = 5;
}
```

Protobuf & gRPC

gRPC和restful API都提供了一套通信机制，用于server/client模型通信，而且它们都使用http/https作为底层的传输协议。



- gRPC可以通过protobuf来定义接口，从而可以有更加严格的接口约束条件；
- 通过protobuf可以将数据序列化为二进制编码，这会大幅减少需要传输的数据量；
- gRPC可以方便地支持流式通信；

客户-服务器绑定

- 绑定步骤

  定位服务器所在的机器；

  定位该机器上的服务器（即相应的进程）

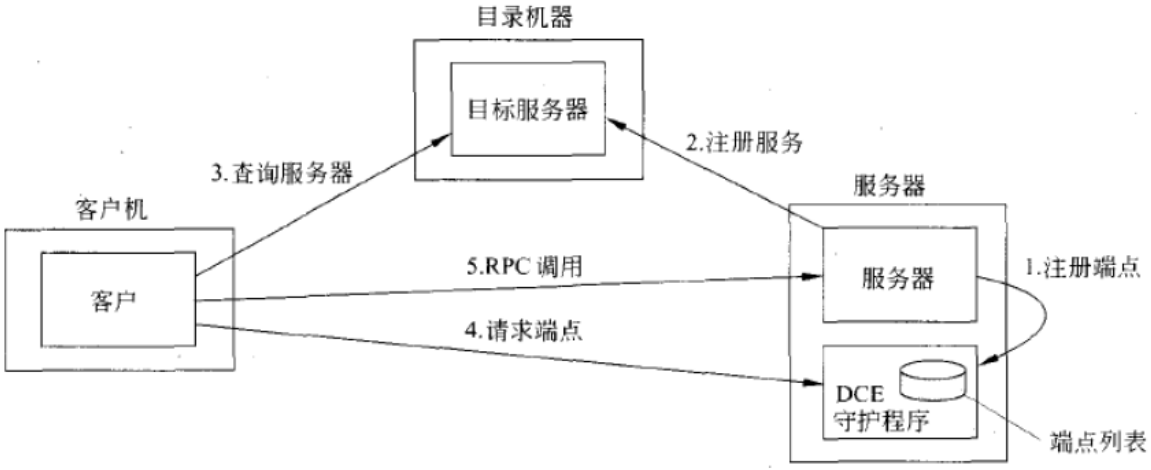


图 4.13 DCE 中的客户-服务器绑定过程

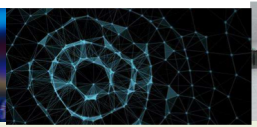
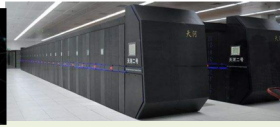
面向消息的通信

远程过程调用的缺点？

- 服务器不一定运行
- 同步阻塞

面向消息的通信

- 面向消息的瞬时通信
- 面向消息的持久通信

面向消息的瞬时通信

➢ Berkeley套接字

原语	含义
socket	创建新的通信端点
bind	将本地地址附加到套接字上
listen	宣布已准备好接受连接
accept	在收到连接请求之前阻塞调用方
connect	主动尝试确立连接
send	通过连接发送数据
receive	通过连接接收数据
close	释放连接

面向消息的瞬时通信

- **Berkeley套接字**

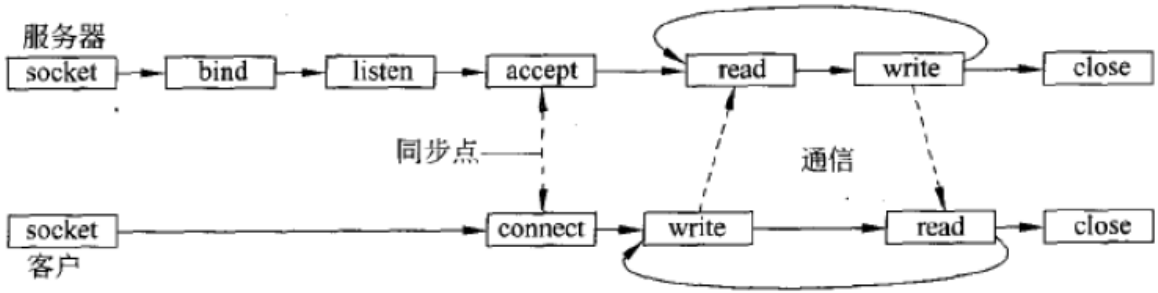
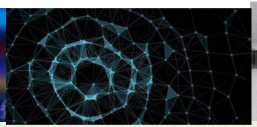
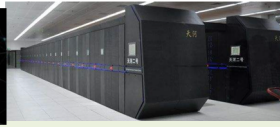


图 4.15 使用套接字的面向连接通信模式

面向消息的瞬时通信

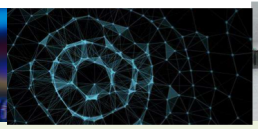
➢ Socket Python Code

Server

```
1  from socket  import *
2  s = socket(AF_INET, SOCK_STREAM)
3  s.bind((HOST, PORT))
4  s.listen(1)
5  (conn, addr) = s.accept()  # returns new socket and addr. client
6  while True:                # forever
7    data = conn.recv(1024)   # receive data from client
8    if not data: break       # stop if client stopped
9    conn.send(str(data)+"*") # return sent data plus an "*"
10 conn.close()               # close the connection
```

Client

```
1  from socket  import *
2  s = socket(AF_INET, SOCK_STREAM)
3  s.connect((HOST, PORT)) # connect to server (block until accepted)
4  s.send('Hello, world')  # send same data
5  data = s.recv(1024)     # receive the response
6  print data              # print the result
7  s.close()               # close the connection
```


简化Sockets使用

➢ 观察发现

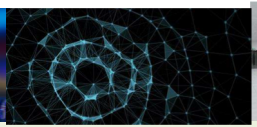
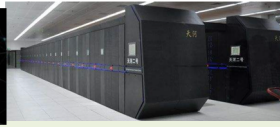
套接字属于底层编程，容易出错。然而，这些套接字的使用模式都非常类似 (模板化)。

➢ ZeroMQ

利用配对sockets提供高层次的表达，即一个用于在进程P发送消息，另外一个在进程Q接收消息。所有的通信都是异步的。

➢ 三种不同的模式

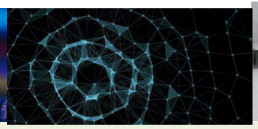
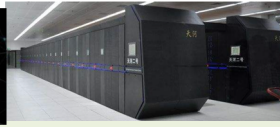
- Request- Reply;
- Publish-subscribe;
- Pipeline;

Request-reply

Server

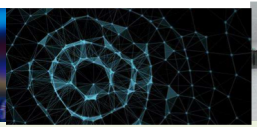
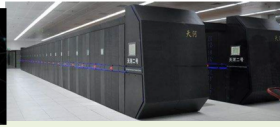
```
1  import zmq
2  context = zmq.Context()
3
4  p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
5  p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect
6  s  = context.socket(zmq.REP)    # create reply socket
7
8  s.bind(p1)                      # bind socket to address
9  s.bind(p2)                      # bind socket to address
10 while True:
11   message = s.recv()            # wait for incoming message
12   if not "STOP" in message:     # if not to stop...
13     s.send(message + "*")       # append "*" to message
14   else:                         # else...
15     break                       # break out of loop and end
```

Request-reply

Client

```
1  import zmq
2  context = zmq.Context()
3
4  php = "tcp://"+ HOST +":"+ PORT # how and where to connect
5  s   = context.socket(zmq.REQ)   # create socket
6
7  s.connect(php)                  # block until connected
8  s.send("Hello World")           # send message
9  message = s.recv()              # block until response
10 s.send("STOP")                  # tell server to stop
11 print message                   # print result
```

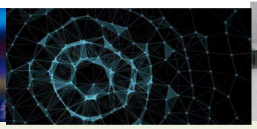
Publish-subscribe

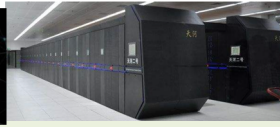
Server

```
1  import zmq, time
2
3  context = zmq.Context()
4  s = context.socket(zmq.PUB)          # create a publisher socket
5  p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
6  s.bind(p)                            # bind socket to the address
7  while True:
8    time.sleep(5)                      # wait every 5 seconds
9    s.send("TIME " + time.asctime()) # publish the current time
```

Client

```
1  import zmq
2
3  context = zmq.Context()
4  s = context.socket(zmq.SUB)            # create a subscriber socket
5  p = "tcp://"+ HOST +":"+ PORT          # how and where to communicate
6  s.connect(p)                           # connect to the server
7  s.setsockopt(zmq.SUBSCRIBE, "TIME")    # subscribe to TIME messages
8
9  for i in range(5):   # Five iterations
10   time = s.recv()    # receive a message
11   print time
```

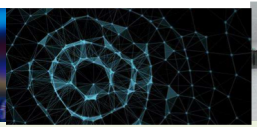


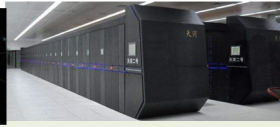


Pipeline

Source

```
1  import zmq, time, pickle, sys, random
2
3  context = zmq.Context()
4  me  = str(sys.argv[1])
5  s   = context.socket(zmq.PUSH)          # create a push socket
6  src = SRC1  if me == '1' else SRC2      # check task source host
7  prt = PORT1 if me == '1' else PORT2     # check task source port
8  p   = "tcp://"+ src +":"+ prt           # how and where to connect
9  s.bind(p)                               # bind socket to address
10
11 for i in range(100):                    # generate 100 workloads
12   workload = random.randint(1, 100)     # compute workload
13   s.send(pickle.dumps((me,workload)))   # send workload to worker
```





Worker

```
1  import zmq, time, pickle, sys
2
3  context = zmq.Context()
4  me = str(sys.argv[1])
5  r  = context.socket(zmq.PULL)       # create a pull socket
6  p1 = "tcp://"+ SRC1 +":"+ PORT1     # address first task source
7  p2 = "tcp://"+ SRC2 +":"+ PORT2     # address second task source
8  r.connect(p1)                       # connect to task source 1
9  r.connect(p2)                       # connect to task source 2
10
11 while True:
12   work = pickle.loads(r.recv())     # receive work from a source
13   time.sleep(work[1]*0.01)          # pretend to work
```

MPI （消息传递接口）

➢ 套接字的缺点

- 套接字的抽象层不佳，只提供简单的Send和Receive操作；
- 套接字使用通用的协议栈（TCP/IP）进行网络通信；不适用于专用协议；
- 灵活性比较差，提供的功能简单；

MPI （消息传递接口）

➢ 消息原语

Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until transmission starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

面向消息的持久通信

➢ 消息队列系统（面向消息的中间件）

- 通过中间件层的队列支持实现异步持久的通信。队列相当于通信服务器的缓冲区；

➢ 队列操作原语

Operation	Description
put	Append a message to a specified queue
get	Block until the specified queue is nonempty, and remove the first message
poll	Check a specified queue for messages, and remove the first. Never block
notify	Install a handler to be called when a message is put into the specified queue

消息队列模型

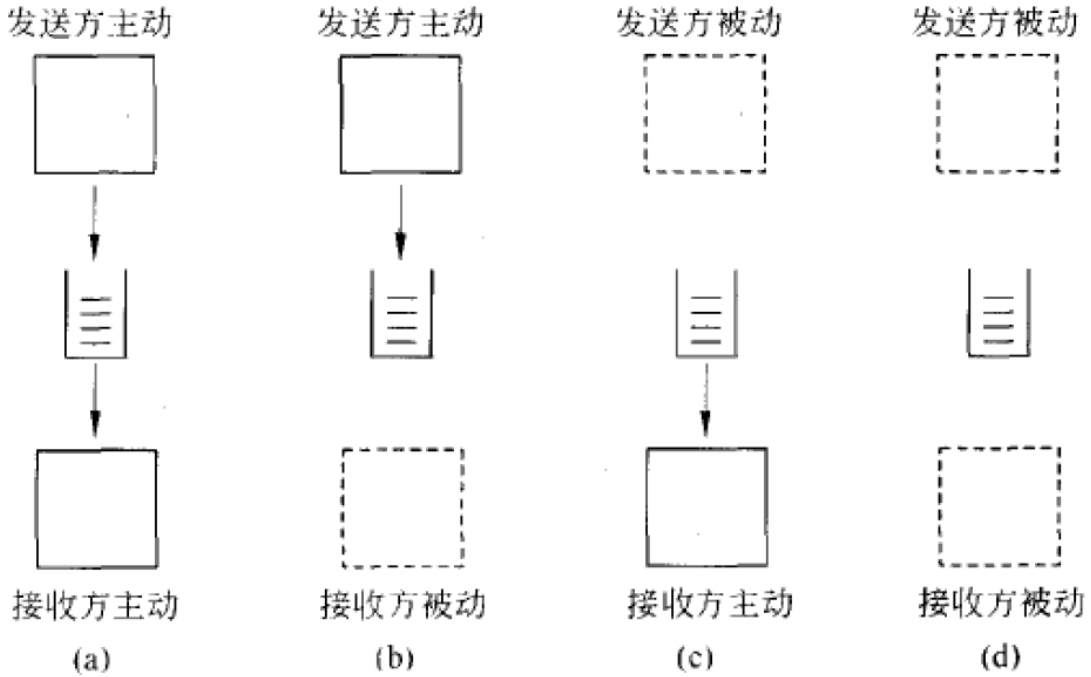


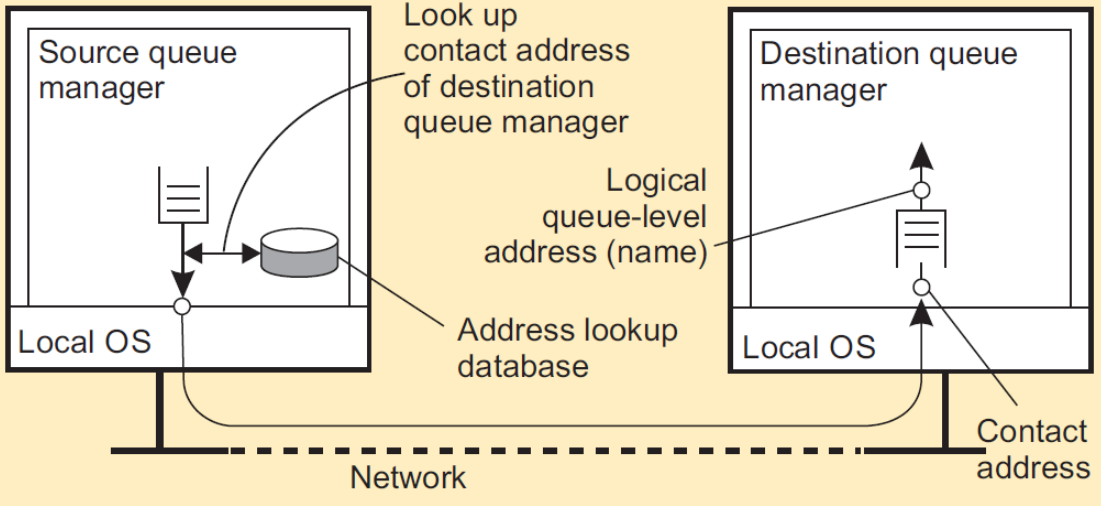
图 4.17　使用队列的松散耦合通信的 4 种组合方式

消息队列系统的一般体系结构

➢ 队列管理器

队列是由队列管理者来管理的。 应用程序仅将消息放在本地队列中，然后由队列管理者将消息路由到其他地方。

➢ 消息路由



消息队列系统的一般体系结构

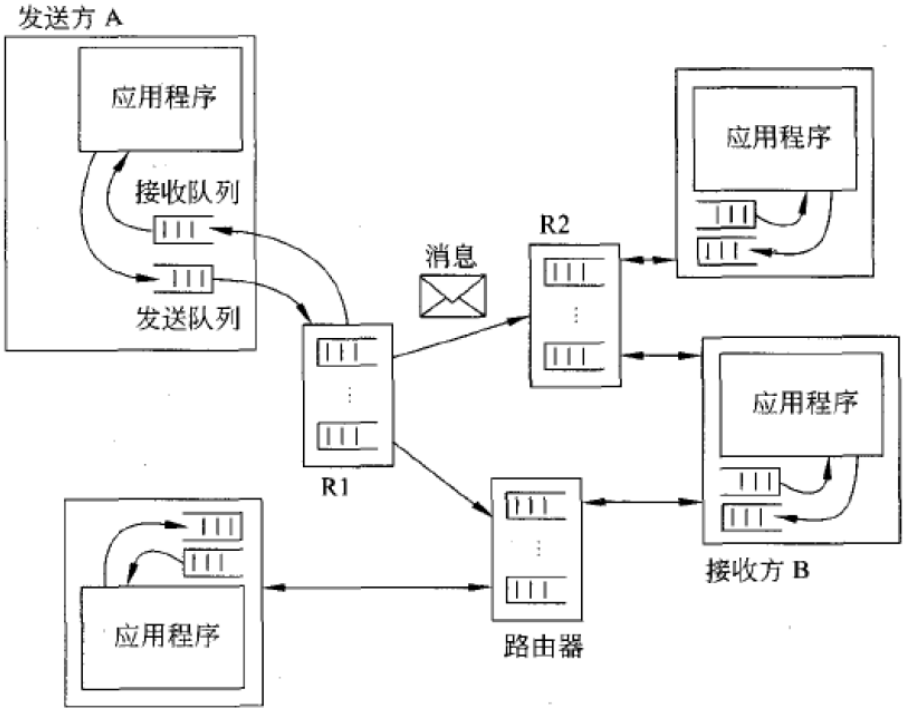


图 4.20　含有路由器的消息队列系统的一般组织结构

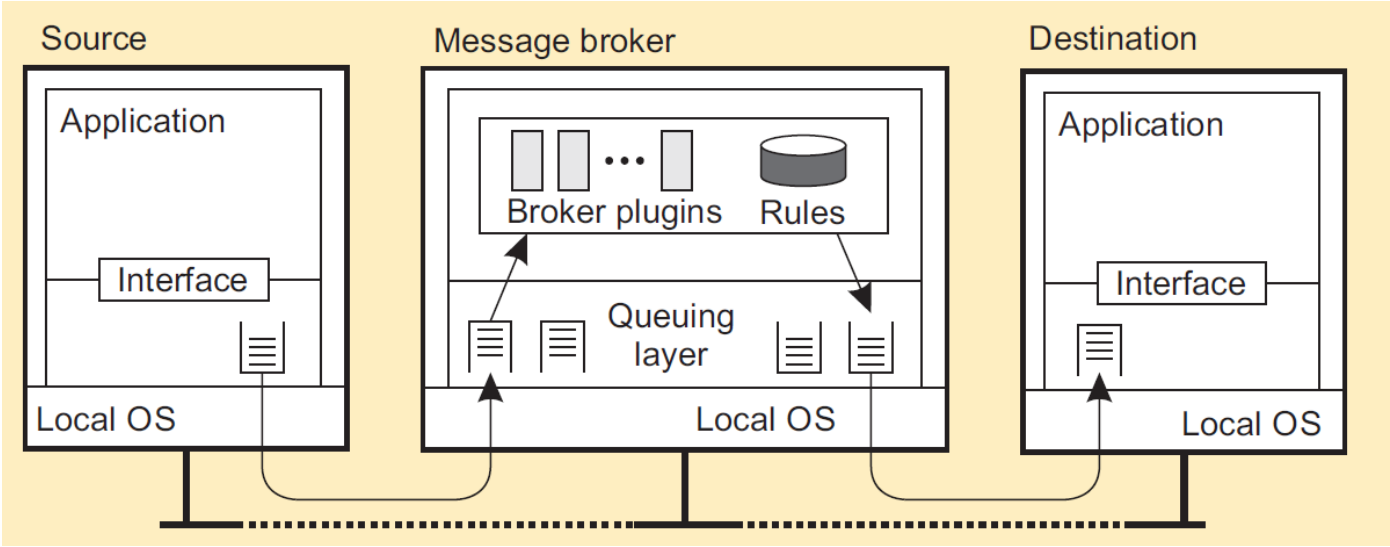
消息转换器(Message Broker)

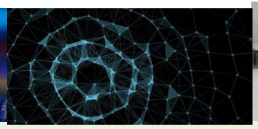
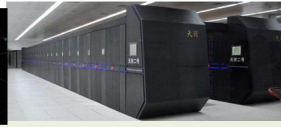
- **观察发现**

消息队列系统假定存在一个共同的消息协议；应用程序会在消息的格式上达成一致（即消息的结构和数据表示）；

- **转换器在消息队列系统中处理应用的异构性**
  - 将输入消息转换成目的格式；
  - 起应用层网关的作用；
  - 提供基于主题的路由功能（pub-sub）；

消息转换器的通用架构


Source
Application
Interface
Local OS
Message broker
Broker plugins
Rules
Queuing layer
Local OS
Destination
Application
Interface
Local OS

IBM WebSphere MQ

- 基本概念
  - 与应用相关的消息被放进或者移出队列；
  - 消息队列由队列管理器控制；
  - 进程可将消息放在本地队列中，或者通过 RPC 机制发到远端；
- 消息传输
  - 消息在不同的队列之间进行传输；
  - 消息在不同进程的队列之间传输时需要通信信道（Channel）；
  - 消息通道的两端是消息通道代理（Message Channel Agent）；

IBM WebSphere MQ

- **MCA的主要作用**
  - 利用底层的网络通信协议如TCP/IP等建立通信信道；
  - 从输出（输入）的网络传输包中封装（解封装）消息；
  - 发送和接收传输包；

IBM WebSphere MQ

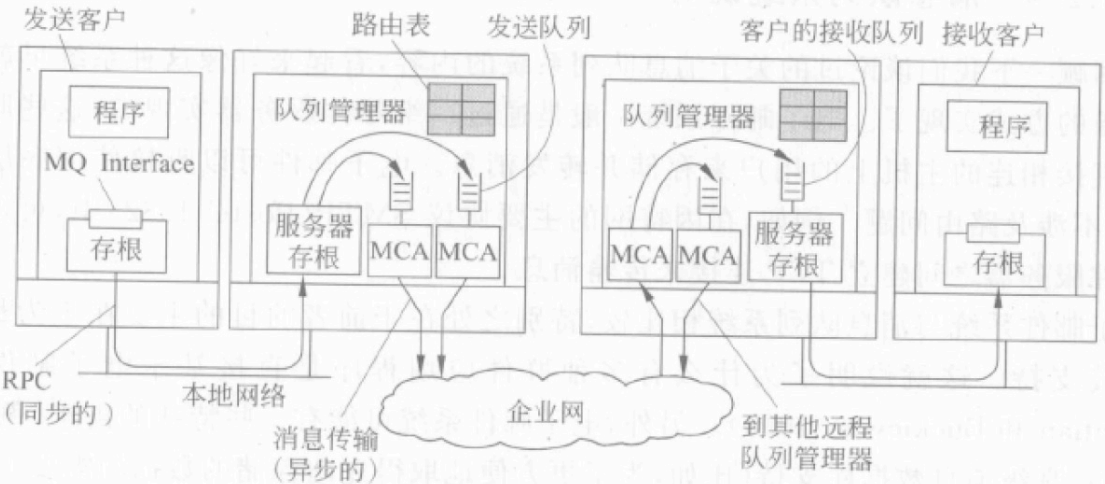


图 4.22 IBM 消息队列系统的一般组织结构

- 通道是单向的；
- 当消息到达的时候自动启动**MCA**；
- 任何队列管理器都可以创建通道；
- 路由是手动设置的；

MCA 属性

每一个MCA都有一组相关的属性，这些属性决定了通道的全部特性。

属性	描述
transport type(传输类型)	决定要采用的传输协议
FIFO delivery(先进先出传输)	表明消息将按照发送的次序到达
message length(消息长度)	单个消息的最大长度
setup retry count(建立连接的重试次数)	指定启动远程 MCA 的最大重试次数
delivery retries(消息交付重试次数)	MCA 将收到的消息放入队列的最大重试次数

图 4.23 与消息通道代理相关的一些属性

IBM WebSphere MQ消息传输

- 路由

路由被显式地存储在队列管理器中的路由表中。路由表的条目为（destQM, sendQ）对。路由表中的条目称为别名。

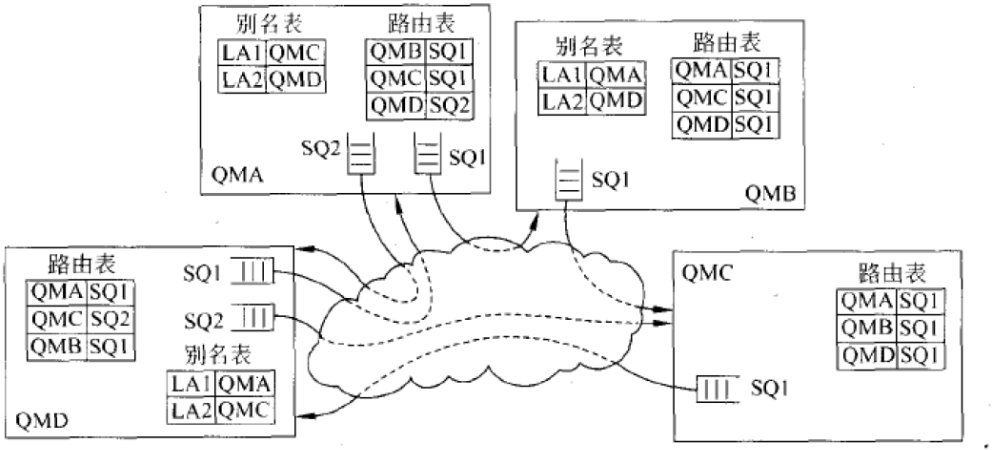


图 4.24 使用了路由表和别名表的 MQ 队列网络的常见组织结构

IBM WebSphere MQ 编程接口

原语	描述
MQopen	开启一个队列（可能位于远程）
MQclose	关闭一个队列
MQput	将消息放入开启的队列中
MQget	从队列（队列可以在本地）中获取消息

图 4.25　消息队列接口中的原语

面向流的通信

➢ 数据流

数据流是数据单元的序列，可以应用于离散的媒体即离散数据流，也可以应用于连续媒体即连续数据流；

➢ 数据流传输模式

- 异步传输；
- 同步传输；
- 等时传输；

流通信QoS

- QoS主要关注：时间、容量以及可靠性；

(1) 数据传输所要求的比特率。
(2) 创建会话的最大延时(例如，应用程序何时可以开始发送数据)。
(3) 端到端的最大延时(例如，数据单元到达接收端花费多少时间)。
(4) 最大延时抖动。
(5) 最大往返延时。

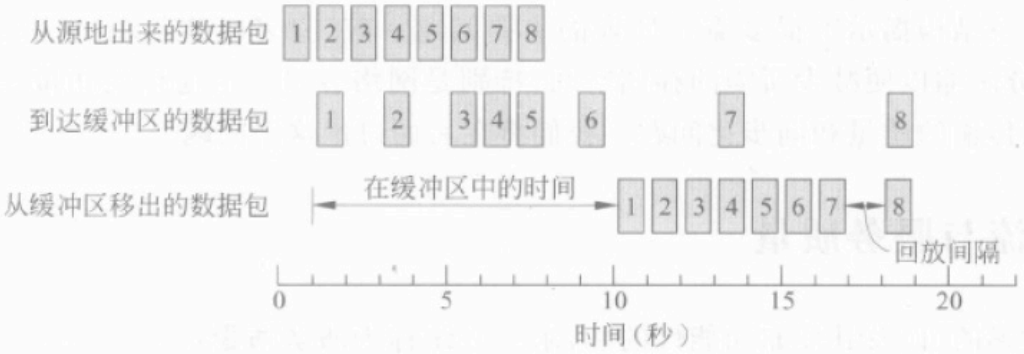


图 4.27 使用缓冲区来减少抖动

多播通信

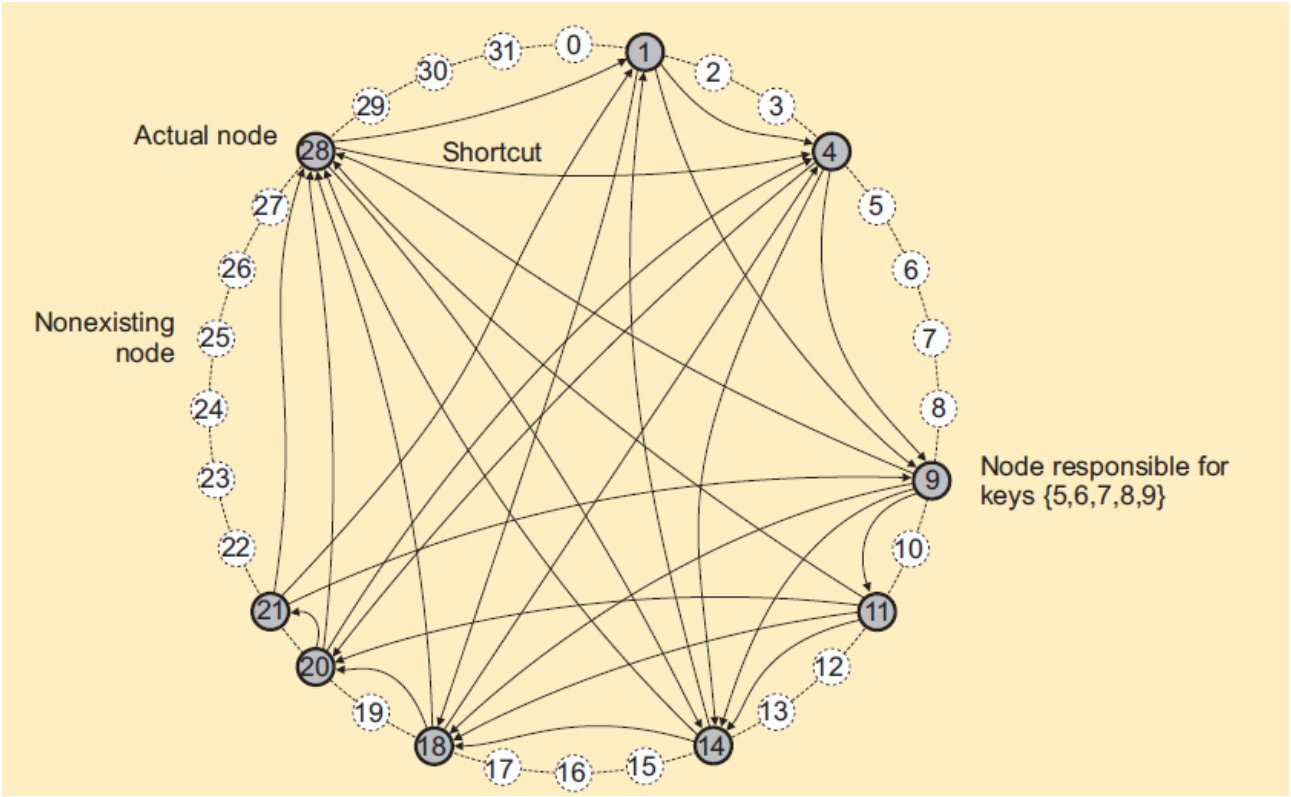
➢ 应用层多播

- 其本质是将分布式系统组织成一个覆盖网络，然后利用这个网络分发数据；

覆盖网络的构建方法：

- 组织成树，导致每对节点之间只有唯一的路径；
- 组织成网络结构，每个节点都有多个邻居节点 （**健壮性高**）；

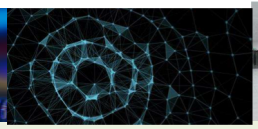
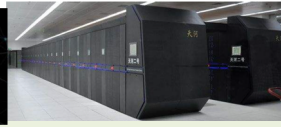
Chord结构中的应用层多播



Chord结构中的应用层多播（多播树构建）

Basic approach

1. Initiator generates a multicast identifier *mid*.
2. Lookup *succ*(*mid*), the node responsible for *mid*.
3. Request is routed to *succ*(*mid*), which will become the root.
4. If *P* wants to join, it sends a join request to the root.
5. When request arrives at *Q*:
   - *Q* has not seen a join request before $\Rightarrow$ it becomes forwarder; *P* becomes child of *Q*. Join request continues to be forwarded.
   - *Q* knows about tree $\Rightarrow$ *P* becomes child of *Q*. No need to forward join request anymore.

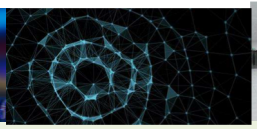
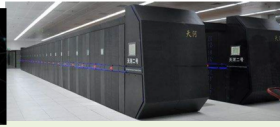




Gossip数据通信

- **感染协议（epidemic protocol）**

  起源于流行病理论，包含“已感染的”、“易受感染的”、“已隔离的”；

- **假设信息传播过程中不存在写-写冲突**
  - 更新由单个节点发起的；
  - 副本仅向有限的几个邻居传播；
  - 更新传播是滞后的，并不是立即进行；
  - 最终，每一次更新都会到达所有副本；
- **传播模型**
  - **anti-entropy （反熵）模型;**
  - **rumor spreading （流言传播）模型；**

Anti-entropy （反/逆熵）模型

Principle operations

- A node P selects another node Q from the system at random.
- Pull: P only pulls in new updates from Q
- Push: P only pushes its own updates to Q
- Push-pull: P and Q send updates to each other

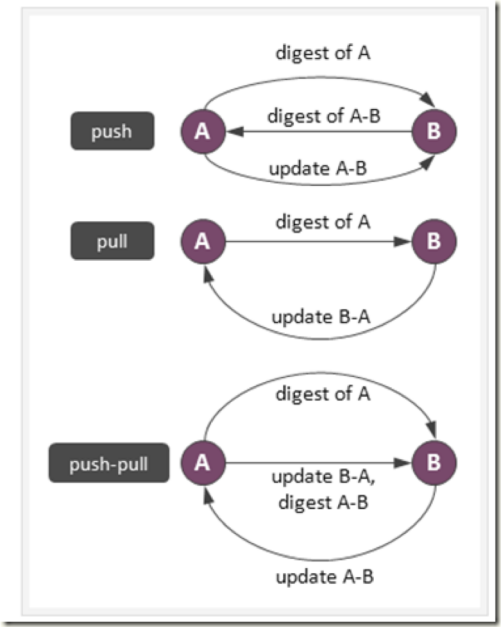
Observation

For push-pull it takes $\mathcal{O}(log(N))$ rounds to disseminate updates to all N nodes (round = when every node has taken the initiative to start an exchange).

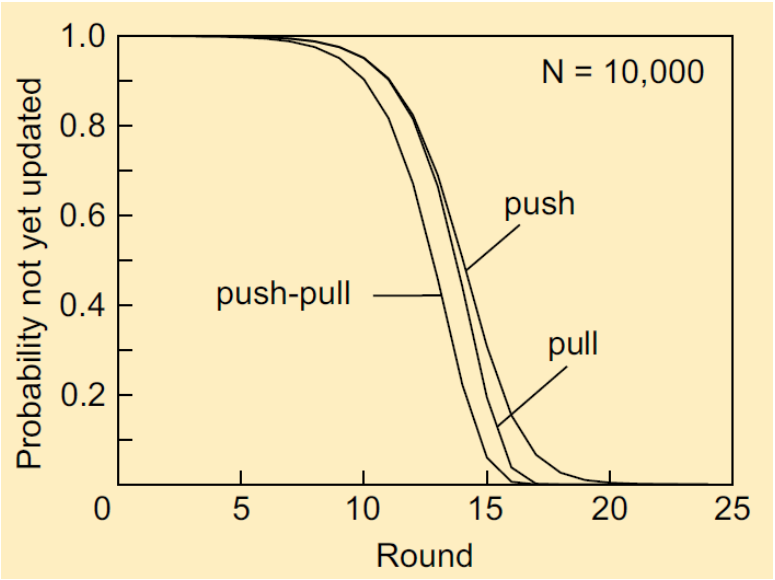
不同操作的性能表现如何？

Anti-entropy （反/逆熵）模型

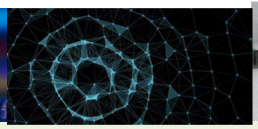
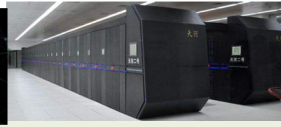
- **最终一致性方法**
  经过一番杂乱无章的通信，最终所有节点的状态都会达成一致；



Cassandra 中的anti-entropy协议



不同操作的性能差异

流言传播模型

Basic model

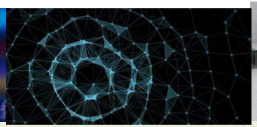
A server S having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, S stops contacting other servers with probability p_{stop}.

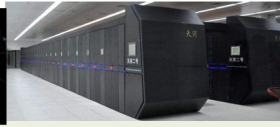
Observation

If s is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers

$$s = e^{-(1/p_{stop}+1)(1-s)}$$

具有良好的扩展性！





谢谢！