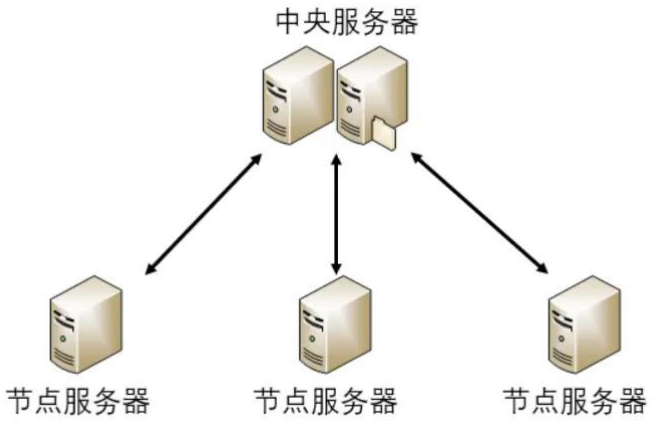
第六章：分布式程序设计

分布式系统指的是通过网络相互连接，共同完成某种特定计算任务的计算机系统。随着分布式系统以及大数据云计算等新兴计算范式的兴起，分布式编程的地位和重要性不断凸显。本章讨论分布式程序设计的基本概念和程序设计技术，本章先讨论分布式的编程模型，并指出其与单机编程模型的重要联系和区别；接着，本章讨论远程过程调用，这是分布式编程的重要基础；接下来，作为分布式程序设计的两个重要示例，本章分别讨论分布式计算和分布式存储。

6.1 分布式编程模型

6.1.1 编程模型

在开始讨论分布式系统的协议、原理与设计之前，首先给出在本文中研究的分布式系统的基本问题模型。主要的概念包括节点、通信、存储、失效。



节点。节点是指一个可以独立按照分布式协议完成特定功能的实体。在具体的实现中，一个节点往往是一个操作系统的进程，这些进程可以分布在同一个主机上，也可以分布在互相联网的不同主机上。本章中考虑的节点都是普通的计算机，而不是工作站或者超级计算机；并且，本章不考虑拜占庭问题，即认为节点都按照预期的分布式协议工作，不考虑节点存在恶意的情况。

通信。节点与节点之间是完全独立、相互隔离的，节点之间传递信息的唯一方式是通过网络进行通信。网络是不可靠的，即一个节点可以通过网络向其他节点发送消

息，但发送消息的节点无法确认消息是否被接收节点完整正确收到。

存储。节点可以在本地存储（如硬盘、SSD 等介质）存储数据，这类节点称为有状态节点；否则，不存储数据的节点称为无状态节点。由于我们假定每个节点都是普通的计算机，即其存储能力有限，一般数百兆到数百 G。

失效。分布式系统处理的最核心问题是各种失效（failure）问题。这些问题主要包括机器宕机和网络异常，下面我们分别进行讨论。机器宕机是最发生的失效。根据实践经验，在大型分布式系统中，每天宕机发生的概率是千分之一左右，处理每个宕机事件约需要 24 小时。当宕机发生时，我们认为节点处于不可用状态（unavailable），但宕机事件处理完成，机器状态恢复后，我们认为节点重新回到可用状态（available）。

分布式系统依靠网络进行通信，且网络会出现异常，这些异常可能包括但不限于网络不可达（也称为网络分化、network partition）、数据包丢失、消息乱序、不可靠的 TCP 连接、存储损坏、拥塞等。其中网络分化意味着网络被分成了互相不可达的若干子网，子网间的彼此完全不能通信；例如，加入分布式系统包括多个机房，如果机房间的光纤被损坏，则会发生网络分化。数据包丢失意味着数据包没能够从发送端到达接收端，这种情况在不可靠的传输协议（如 UDP）中非常常见。消息乱序意味着消息的发送顺序和接受顺序发生了混乱，乱序可能是由于节点引起的，也可能是由于网络原因引发的；例如，节点 A 先后发送两个信息给节点 B 和 C，但不能消息会先后到达 B 和 C，也不能假设 B 和 C 的回复会先后到达。数据存储损坏意味着在节点上的存储发生了故障，从而数据不可访问或丢失。拥塞意味着节点或者网络发生了性能的降低，从而使得计算或服务特别缓慢；例如，磁盘故障可能导致磁盘的读写缓慢；而网络故障可能导致网络访问延迟的增大等。

接下来，我们重点讨论不可靠的 TCP 连接。尽管网络编程模型通常认为 TCP 传输是可靠的，但在分布式系统中，不能简单认为 TCP 协议是可靠的，这主要有三方面原因：首先，TCP 协议只提供了网络协议栈层面的可靠性，即只能保证数据包从发送端到达接收端，而分布式处理的应用一般位于应用层，TCP 协议无法保证应用层能够正确处理这些数据包，更无法保证能够给出正确的返回结果；其次，TCP 协议无法保证底层网络的可靠性，例如，当接收端收到发送端发来的信息并处理完毕后，网络发生

了分化（即网络断开），则返回的数据包无法到达发送端，则接收方会由于收不到数据包的确认信息而发送失败；最后，TCP 协议只能保证一个链接内的数据有序性，而不能保证多个 TCP 连接之间的数据有序性；而在分布式系统中，我们往往需要在两个节点间建立多个 TCP 连接。综上，在分布式系统中，我们不能简单认为 TCP 协议是可靠的。

面向分布式系统中可能产生的异常，基本的处理原则是一定要假设所有异常都可能发生，并且必须被处理。这里一个经常发生的错误认识是 “可以忽略小概率事件” 。这个认识的基本问题在于只关注了概率，而忽略了样本量的大小。例如，假定在某个分布式系统中，某种失效事件每天发生的概率是 10^-9，但系统每天处理的请求量是 10^8，则每天发生这种失效事件的概率是 1-(1-10^-9)^10^8=10%。

6.1.2 副本与一致性

副本（Replica）指的是分布式系统中为数据或计算提供的冗余，以应对可能出现的失效。副本可分为数据副本和计算副本。数据副本指的是不同的节点存储同一份数据的拷贝，这样，当一个节点失效时，可以从其他副本中访问该数据；例如，GFS 文件系统中的同一个 chunk 的数个副本就是典型的数据副本。而计算副本指的是不同节点进行同一个计算任务，这样，当一个节点失效时，其它节点仍然可以完成该计算任务；例如，MapReduce 系统中的 worker 节点就是典型的计算副本。

尽管副本提供了缓解失效的有效机制，但它也引发了数据的一致性（consistency）问题。例如，如果某个文件有 3 个副本 A、B 和 C，某次更新只更新了副本 A 和 B，但没有更加副本 C（一种典型的情况是更新操作发生时，副本 C 正好不可达），这样三个副本的数据之间出现了数据不一致。

分布式系统必须提供有效的一致性保证机制，解决数据不一致的问题。按照数据一致性机制能提供的保证的强弱程度，可以分为弱一致性（weak consistency）、强一致性（strong consistency）和最终一致性（eventual consistency）。弱一致性一般无法保证数据更新发生后，能够读取到其最新的值；强一致性保证数据更新后，总是能够读取到其最新的值；而最终一致性保证数据更新后，所有的副本最终都会更新为这个值，但需要的时间无法保证。

6.1.3 设计指标

一个分布式系统可以有不同角度的设计指标，其中最重要的是可扩展性（scalability）、可用性（availability）和一致性（consistency）。可扩展性指的是通过提高单个节点的系统性能、存储容量、计算能力等措施，提高分布式系统的性能。可用性指的是分布式系统能够应对各种失效情况，提供正常服务的能力。一致性指的是系统通过副本等机制，为用户提供一致数据或计算的能力；越强的一致性，对用户越易用。

但是，在分布式系统中，由于网络分化（P）的存在，可用性（A）和一致性（C）往往很难同时做到，往往只能做到分化、可用和一致三个指标中的最多两个，这称为 CAP 理论。例如，如果存在网络分化和保证系统可用，则无法保证系统的一致；类似的，如果要保证系统的可用和一致，则不能出现网络分化。

在实际的分布式系统中，要采用哪些设计指标，取决于具体的场景和需求，进行合理的设计和取舍，并没有统一的答案。例如，在社交平台系统中，我们可能重视网络分区的容错和可用性，进而舍弃一致性，则用户可能无法及时看到最新的发帖或评论；而在电商交易系统中，我们可能需要系统的分区容错和一致，进而舍弃可用性，则一旦出现网络故障，可能需要暂停服务等待系统恢复。

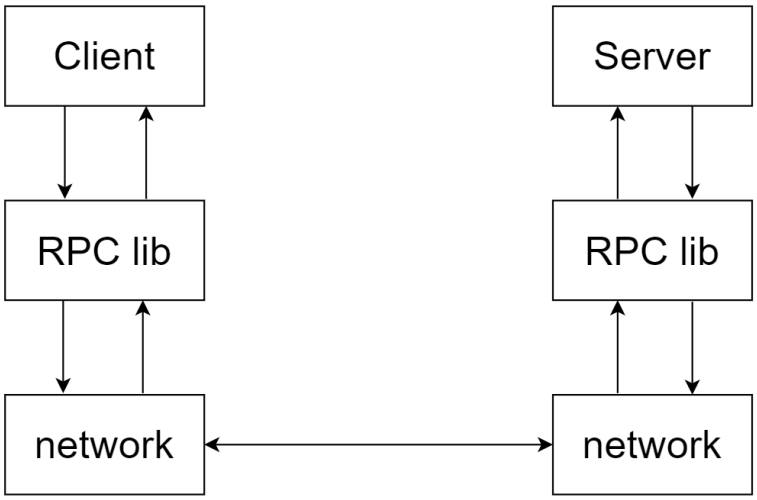
6.2 远程过程调用

6.2.1 基本概念

远程过程调用（Remote procedure call，RPC）以一种应用层的网络协议，它允许一个节点（进程）内的函数调用另一个节点（进程）内的函数，而不用关心底层的网络通信细节。以下，为简化起见，我们也经常将远程过程调用称为远程函数调用，以强调其一般是有返回值的，并简称为 RPC。

下图给出了远程过程调用的典型流程：首先，客户端进程需要识别远端函数（如利用主机名和函数名组合），并对该过程进行调用，这个远程调用的语法一般和本地函数调用的语法类似；接着，客户端的 RPC 库负责将远程调用的函数名以及函数参数封装为网络通信的包，这个过程中，一般涉及到网络数据报格式的选择、以及本地数据

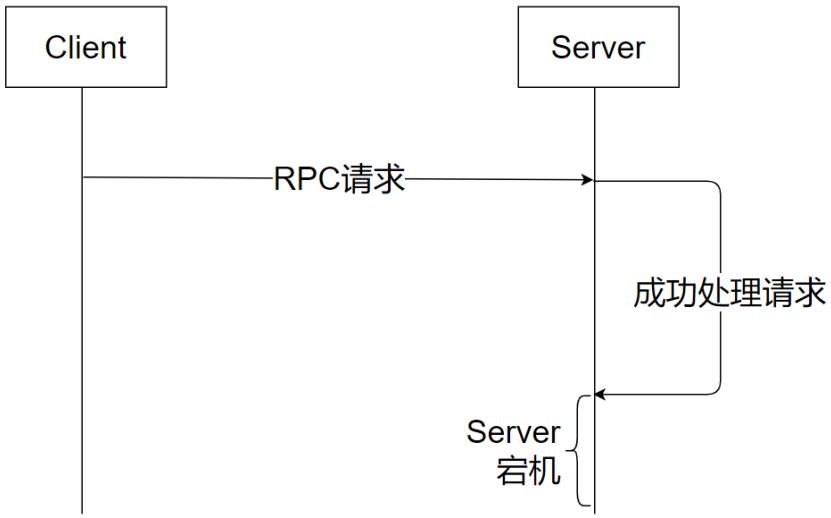
到网络数据报格式的转换；接下来，数据报通过网络协议发送到服务器端，在这个过程中，我们可以选择任意合适的传输协议，但一般的，为了传输可靠性，一般可以通过 TCP 连接进行数据报的传输；接着，数据报到达服务器端后，RPC 库负责将函数名和参数等信息从数据报中读取出来，并调用相应的服务器端的函数，完成预期的操作；接下来，函数调用的结果从服务器端返回给客户端，其过程是上述过程的逆过程，不再赘述。最后，客户端得到了远程函数调用的结果；从执行结果角度分析，远程过程调用和本地过程调用的结果相同。



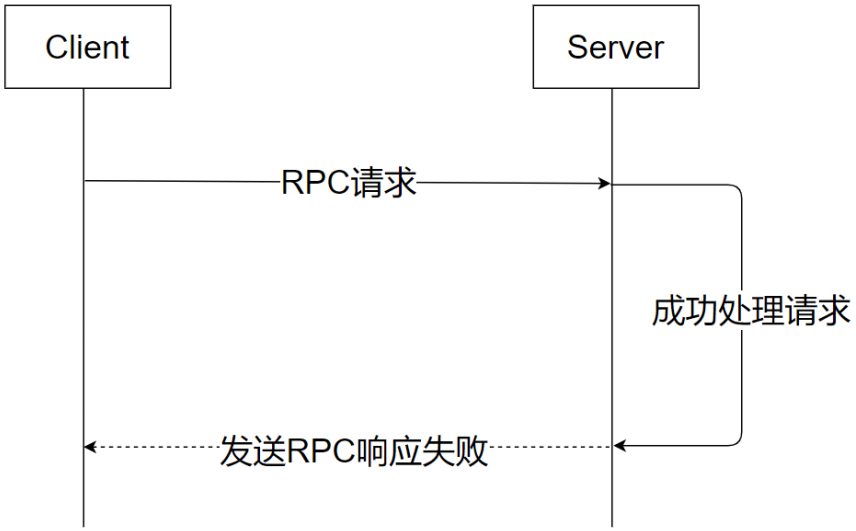
和本地函数调用机制相比，远程函数调用有三个明显的特征和区别：首先，远程函数调用的调用者（客户端）和被调用者（服务器）一般位于不同的进程中，甚至是不同的主机中，其调用过程往往需要底层网络协议栈的支持，因此，RPC 本质上属于进程间通信的机制。其次，由于调用者和被调用者间被网络隔离，因此 RPC 提供了“又一个中间层”，允许我们在遵守 RPC 调用协议的基础上，分别给出客户端和服务器的实现，提供了更大的灵活性，例如，我们可以用不同的编程语言和框架来分别实现客户端和服务器。最后，由于分布式系统中的失效现象，RPC 的执行结果比本地函数调用的执行结果更加复杂，一般的，RPC 的执行结果分为“成功”、“失败”和“超时”三种，也称为分布式系统的三态；其中“成功”和“失败”指的是客户端已经得到服务器的确定响应，而“超时”则指客户端在等待了特定的超时时间后，没有得到服务器的响应。造成 RPC 超时的原因可能是多方面的，例如，可能是客户端的调用请求根本就没有到达服务器、可能是服务器已经宕机、可能是服务器端调用结束后没来得及返回结果、也可能结果在返回客户端的过程中丢失，等等。

客户端的调用请求未到达服务器：当客户端尝试进行远程过程调用时，其请求有时可能无法成功传达给服务器。这种情况可能由多种原因导致，包括但不限于网络中断、路由器或交换机故障、或其他核心网络设备的问题。除此之外，网络安全策略，如防火墙或安全组设置，也可能拦截或过滤掉这些请求。在某些高流量或拥挤的网络环境中，数据包也可能会被丢弃，从而导致请求未能到达服务器。

服务器宕机：服务器的宕机或失效也是导致 RPC 超时的常见原因。服务器可能会因为物理硬件故障、如内存、CPU 或磁盘损坏，而突然停机。软件错误或资源限制，例如，由于代码缺陷导致的应用程序崩溃或内存、CPU、磁盘等资源的极度消耗，都可能导致服务器失去响应。



结果在返回客户端时丢失：即使服务器正确地处理了客户端的请求并发送了响应，该响应在传输过程中也可能丢失或被延迟，导致客户端未能接收。这种情况可能是由网络问题，例如网络中断、不稳定的路由或数据包丢失所引起的。



针对 RPC 存在的调用结果三态问题，我们需要仔细设计所需要的 RPC 调用的语义。具体的，我们可以至少提供以下三种不同的调用语义：最多一次（at most once）、最少一次（at least once）和正好一次（exactly once）。在最多一次的调用语义中，客户端只进行一次的 RPC 请求，而不管是否能够收到服务器的返回结果；这种调用语义无法确认被调用函数是否在服务器端被执行过、也无法确认执行是否已经正常结束；很多 RPC 框架默认采用的最多一次的调用语义。

最少一次调用的语义指的是客户端反复重试执行同一个 RPC 请求，直到能够收到并确认执行结果为止（注意执行结果有可能是成功或者失败）；这种调用语义一方面可能导致 RPC 反复执行不终止（亦即一直没有收到服务器端的响应），另一方面还可能导致服务器的函数被多次执行，在远程函数会产生副作用的情况下，会产生错误的执行结果。

正好一次的执行语义将最多一次和至少一次结合起来，即客户端反复重试 RPC 执行，直到能够确认执行结果，但为了避免服务器端的远程函数被多次执行，需要引入识别重试调用的机制，当服务器端识别到重试调用时，不进行函数调用，而是直接返回恰当的结果。例如，我们可以给每个 RPC 调用加入唯一的令牌（token），即对不同的调用，其令牌值不同，而对于同一个 RPC 调用的重试，我们令其总是携带相同的令牌，其伪代码如下：

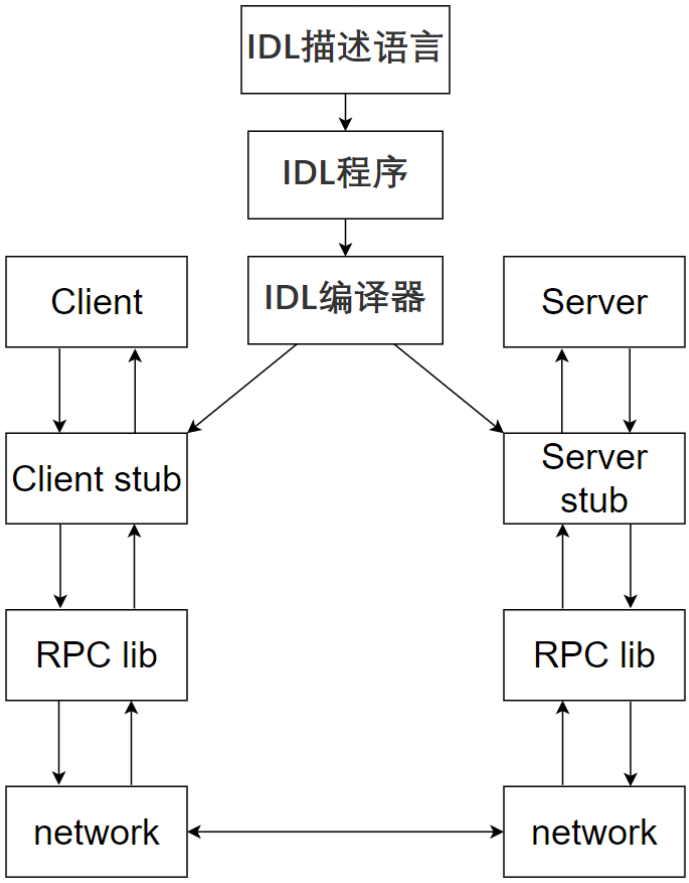
```
void client(){ // call an RPC function "f(args)"
  while(1){
    res = rpc_call("f", token, args...);
    if(res != TIMEOUT)
      break;
  }
}
void server(f, token, args...){
  if(in_cache(token, cache))  // the RPC has been processed
    return cache(args); // thus we return the result directly
  return f(args); // otherwise, the RPC is dispatched
}
```

客户端程序 client 在调用函数时，总是携带标识该次 RPC 的令牌（第 3 行）；这样，服务器端可以通过识别该令牌（第 9 行），来判定该调用是否已经被执行过（第 9 行），从而直接返回缓存的结果（第 10 行），还是指派到目标函数执行（第 11 行）。

6.2.2 RPC 框架

尽管 RPC 从概念上来讲并不复杂，但从零开始设计实现一个 RPC 系统比较枯燥和易错，其中重要的原因是 RPC 系统既涉及到高层 RPC 调用接口的设计，还需要处理底层网络通讯协议的细节。因此，为了降低 RPC 设计和实现的复杂性，研究者提出了 RPC 框架，这些框架允许开发者聚焦在高层 RPC 接口的设计上，把底层网络通信的部分进行了封装和自动化。

下图给出了 RPC 框架的一般架构，核心是接口描述语言（Interface description language, IDL）及其关联的接口编译器。接口描述语言是一个领域专用语言，用来描述要实现的 RPC 服务器的接口，这样，客户端程序可以对这些接口进行调用。从这个角度来说，接口非常类似于 C 语言中的头文件；RPC 中的接口编译器接受接口描述语言构建的接口描述程序作为输入，自动生成服务器和客户端的桩代码；用户可以基于编译器自动生成的桩代码，进一步构建特定的 RPC 程序。



6.2.3 TI-RPC

接下来，作为示例，我们讨论一个具体的 RPC 系统 TI-RPC（Transportation Independent RPC）。TI-RPC 是一个跨平台的 RPC 实现，在 Linux 系统一般以共享库的形式默认安装。用 TI-RPC 实现一个分布式应用，一般包括以下四个步骤：第一，准备分布式服务的描述，TI-RPC 用一种称为外部数据表示（External Data Representation，XDR）的数据格式来描述分布式服务；第二，用 RPC 编译器工具（称为 rpcgen），自动根据数据格式描述，生成服务器及客户端的桩代码；第三，用户根据业务需要，实现特定的程序逻辑，并把它与桩代码链接在一起。这通常涉及为服务器定义真正的操作逻辑和为客户端定义调用逻辑。第四，分别在服务器和客户端上编译和运行这些程序。服务器开始监听指定的端口，等待客户端的请求，而客户端则连接到服务器，并发出 RPC 调用。

总的来说，使用 TI-RPC 开发分布式应用的过程涉及到描述服务、生成桩代码、实现具体逻辑和运行应用这四个主要步骤。这种方式使得开发人员可以专注于业务逻辑

的实现，而不必深入了解底层的通信细节。

下面，我们先用 TI-RPC 实现一个简单的远程函数调用服务器，其中包括一个远程平方函数

```
int square(int x);
```

该函数接受整型参数 x，返回 x 的平方。

第一步，我们首先给出如下文件 square.x：

```
/* square.x */
/* the RPC protocol for calculating squares, among others */
program SQUARE_PROTO{
    version SQUARE_VERSION{
        int SQUARE(int) = 1;
    } = 1;
} = 0x20000001;
```

这段代码定义了一个 RPC 接口描述，命名为 SQUARE_PROTO，专门用于远程计算整数的平方。在这个接口中，它指定了一个版本 SQUARE_VERSION，并在该版本下定义了一个名为 SQUARE 的远程方法，该方法接受一个整数并返回其平方值。为了确保唯一性，给这个 RPC 程序分配了一个标识符 0x20000001。

第二步，使用 rpcgen 工具对 square.x 文件进行处理，会产生一系列源代码文件。这些文件可以被用来构建客户端和服务器应用程序。

```
rpcgen square.x
```

这条命令会产生以下文件：

square.h: 这是一个头文件，包含程序定义、版本号以及为所描述的 RPC 服务所需的所有数据结构的定义。当编写客户端和服务器的实现代码时，需要包含这个头文件。

square_clnt.c: 这是客户端存根的源文件。存根是一个为特定 RPC 程序生成的一组过程，用于发送和接收消息。客户端存根提供了与远程过程调用相对应的本地过程，客户端程序可以直接调用这些本地过程。

square_svc.c: 这是服务器存根的源文件。服务器存根读取请求消息，调用相应的服务器过程，然后发送回应消息。

第三步，编写服务器和客户端代码。首先，我们在一个名为 square_server.c 的文件中实现平方函数。

```
#include <stdio.h>
#include "square.h"
int main(int argc, char *argv[]){
    CLIENT *cl;
    int  *result;
    int  square_arg;
    if (argc != 3) {
        fprintf(stderr, "usage: %s server_host value\n",
argv[0]);
        exit(1);
    }
    cl = clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "udp");
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
    square_arg = atoi(argv[2]);
    result = square_proc_1(&square_arg, cl);
    if (result == NULL) {
        clnt_perror(cl, argv[1]);
        exit(1);
    }
    printf("%d^2 = %d\n", square_arg, *result);
    clnt_destroy(cl);
    return 0;
}
```

最后，编译并运行。

```
$ gcc -o square_server square_server.c square_svc.c -lrpcsvc
$ gcc -o square_client square_client.c square_clnt.c -lrpcsvc
$ ./square_server
$ ./square_client localhost 4  # This should print "4^2 = 16"
```

6.2.4 gRPC

gRPC 是由谷歌开发并开源的一款高性能、跨平台、多语言支持的 RPC 框架。该框架底层采用 HTTP2 作为传输协议，采用 protocol buffer 作为服务接口描述语言。由于 gRPC 框架的现代性、通用性和高性能，它已在分布式系统、微服务、物联网等领域中得到了广泛应用。

本节主要以 gRPC 为示例，讨论远程过程调用的另一个主流系统的主要技术特点和应用。用 gRPC 实现分布式系统，仍然包括上述四个主要步骤，下面我们再次考虑前面的平方服务器的例子。

首先，我们给出基于 Protocol Buffer 的服务接口的实现：

首先，定义服务器与客户端交互的数据结构。

```
// rpcs.go
package main
type SquareArgs struct {
    Number int
}
type SquareReply struct {
    Result int
}
```

服务端存在一个名为 SquareService 的服务，它提供一个方法 Square，该方法接收 SquareArgs 作为参数，并将结果返回到 SquareReply 中。该方法核心逻辑是计算参数中给定数字的平方。在 main 函数中，服务器首先初始化并注册 SquareService，然后开始在 TCP 端口 1234 上监听客户端的连接请求。一旦有新的连接请求，服务器会异步处理它，确保并发处理多个请求。

```
// server.go
package main
import (
    "fmt"
    "net"
    "net/rpc"
)
type SquareService struct{}
func (s *SquareService) Square(args *SquareArgs, reply
*SquareReply) error {
    reply.Result = args.Number * args.Number
    return nil
}
func main() {
    squareService := new(SquareService)
    rpc.Register(squareService)
    listener, err := net.Listen("tcp", ":1234")
    if err != nil {
        fmt.Println("Error starting server:", err)
        return
    }
    defer listener.Close()
    fmt.Println("Server started at :1234")
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Connection error:", err)
            continue
        }
        go rpc.ServeConn(conn)
    }
}
```

客户端的主要目的是调用远程服务器上的 SquareService.Square 方法。首先与运行在 localhost:1234 的服务器建立连接。然后，它构建了一个 SquareArgs 实例，其中包含了一个数字 10，并请求服务器计算这个数字的平方。服务器返回的结果保存在 SquareReply 结构中。最后，客户端将计算得到的平方值输出到控制台。

```
// client.go
package main
import (
    "fmt"
    "net/rpc"
)
func main() {
    client, err := rpc.Dial("tcp", "localhost:1234")
    if err != nil {
        fmt.Println("Dial error:", err)
        return
    }
    args := &SquareArgs{Number: 10}
    var reply SquareReply
    err = client.Call("SquareService.Square", args, &reply)
    if err != nil {
        fmt.Println("RPC error:", err)
        return
    }
    fmt.Printf("The square of %d is %d\n", args.Number,
reply.Result)
}
```

服务器程序启动命令：

```
csslab@tiger:~$ go run server.go rpcs.go
Server started at :1234
```

客户端程序启动命令：

```
csslab@tiger:~$ go run client.go rpcs.go
The square of 10 is 100
```

6.3 分布式处理

6.3.1 基本概念

分布式处理是一种计算范式，其中任务被分发到多个计算资源（如多台计算机或服务器），以实现高性能、容错性和可扩展性。这些分布式系统中的节点相互协作，通过网络通信完成任务，通常涉及任务分发、数据一致性、负载均衡、容错机制和安全性等关键概念，被广泛应用于大数据处理、云计算、分布式存储等领域。

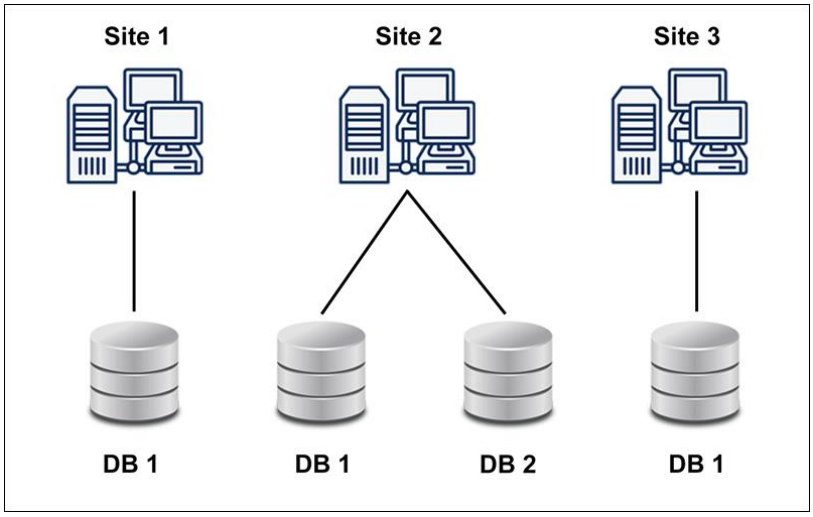
6.3.2 MapReduce

MapReduce 是一种经典的分布式计算模型和编程框架，最初由 Google 开发用于处理大规模数据集。它的核心思想是将复杂的数据处理任务分解为两个简单的步骤：Map（映射）和 Reduce（归约）。在 Map 阶段，数据被分割成小的数据块，每个数据块被一个或多个映射任务处理，生成一组 key-value 对。然后在 Reduce 阶段，这些中间结果被分组、排序，然后经过归约操作生成最终的结果。

6.4 分布式存储

6.4.1 基本概念

分布式存储（Distributed Storage）是一种数据存储方式，其中数据被分散存储在多个服务器或节点上。分布式架构旨在提供高可用性、可扩展性和容错性，以满足现代应用程序的需求，特别是那些需要处理大规模数据和大量用户请求的应用程序。



6.4.2 分布式 KV 存储

在分布式 KV 存储系统中，数据被存储为键值对（Key-Value Pairs），其中键是唯一标识数据的标识符，值是与该键相关联的数据。这些键值对可以在多个分片（shard）上分布存储，以满足大规模数据存储和访问的需求。客户端需要维护一个配置信息，指示它应该与哪个分片 KV 服务器通信以访问特定键的值。配置通常包括每个分片服务器的地址以及键的哈希范围。客户端使用配置来确定要访问的服务器。

分布式 KV 存储支持一组标准操作，包括获取（Get）、插入（Put）、附加（Append）和删除（Delete）。这些操作允许客户端检索与键相关联的值、更改键的值、追加值或删除键及其关联的值。

在客户端看来，在分布式 KV 存储系统中进行数据访问大致分为三步。首先，在客户端进行请求时，根据键（用户名）计算其哈希值。然后，根据配置信息和计算的哈希值，客户端确定应该与哪个分片服务器通信。这决定了数据的存储位置或访问位置。最后，根据操作类型执行相应的操作，客户端与目标服务器进行通信，并处理数

据的存储、检索、附加或删除。

服务器在接收到客户端的链接时，会调用相应地方法来进行处理，下面的代码完成了服务器端的 put 功能：

```
typedef struct MapNode {
    char *key;
    char *value;
    struct MapNode *next;
} MapNode;
MapNode *hashTable[TABLE_SIZE];
// hash
/* D. J. Bernstein hash function */
static unsigned int djb_hash(const char *cp) {
    unsigned hash = 5381;
    while (*cp)
        hash = 33 * hash ^ (unsigned char)*cp++;
    return hash % TABLE_SIZE;
}
void put(char *key, char *value) {
    unsigned int index = djb_hash(key);
    MapNode *newNode = malloc(sizeof(MapNode));
    newNode->key = strdup(key);
    newNode->value = strdup(value);
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
```

其中，服务器首先根据 key 计算客户端的索引（第 16 行），然后为当前客户端分配一个 MapNode 结构体（第 17 行），设置键和值，最后将结构体加入到 hash 表中，完成本次 put 操作。

为了更好的理解分布式存储的工作场景，我们假设正在构建一个社交媒体应用程序，其中用户可以发布帖子，并帖子以键值对的形式存储在分布式 KV 存储系统中。每个用户的用户名将是键，而用户发布的所有帖子将是与该键相关联的值。分布式 KV 存储系统将用于存储这些用户帖子。现在，我们可以考虑一个简单的分布式 KV 存储系统，其中有三个分片服务器：Server A、Server B 和 Server C。配置信息如下：

```
Config: [(Server A, [0, 99]),
        (Server B, [100, 199]),
        (Server C, [200, 299])]
```

当用户 A 发布了一篇新帖子，例如"Hello, World!"，将执行 Put 操作：首先客户端计算用户 A 名字的哈希值，例如 hash("A") % 300 得到哈希值 100。然后，根据配置信息，查询到哈希值 100 的帖子应该存储在 Server B 上。最后，客户端将用户 A 和帖子"Hello, World!"存储在 Server B 上，完成整个发帖过程。其他请求如 Get、Append、Delete 操作流程与上述流程相似。

此外，在分布式 KV 存储中，服务器可能会进行动态扩展或缩减。为了解决这个问题，通常存在一个分片主服务器。分片主服务器用于管理所有分片服务器的配置，负责添加新服务器、删除现有服务器并更新配置信息。当分片服务器的配置更改时，分片主服务器会更改配置，然后键值对会在不同服务器之间移动，以使数据保持一致。

6.5 本章小结

本章主要讨论分布式计算中的基本程序技巧。首先，我们讨论了分布式编程模型；接着，我们讨论了远程过程调用，并讨论了典型 RPC 框架；最后，我们分别讨论了两种重要的分布式范式：分布式计算和分布式存储。

6.6 深入阅读

除了本章中给出的论文外，读者还可以阅读分布式计算的相关教材和论文，如 Distributed Systems: Principles and Paradigms"（分布式系统：原理与范型）。