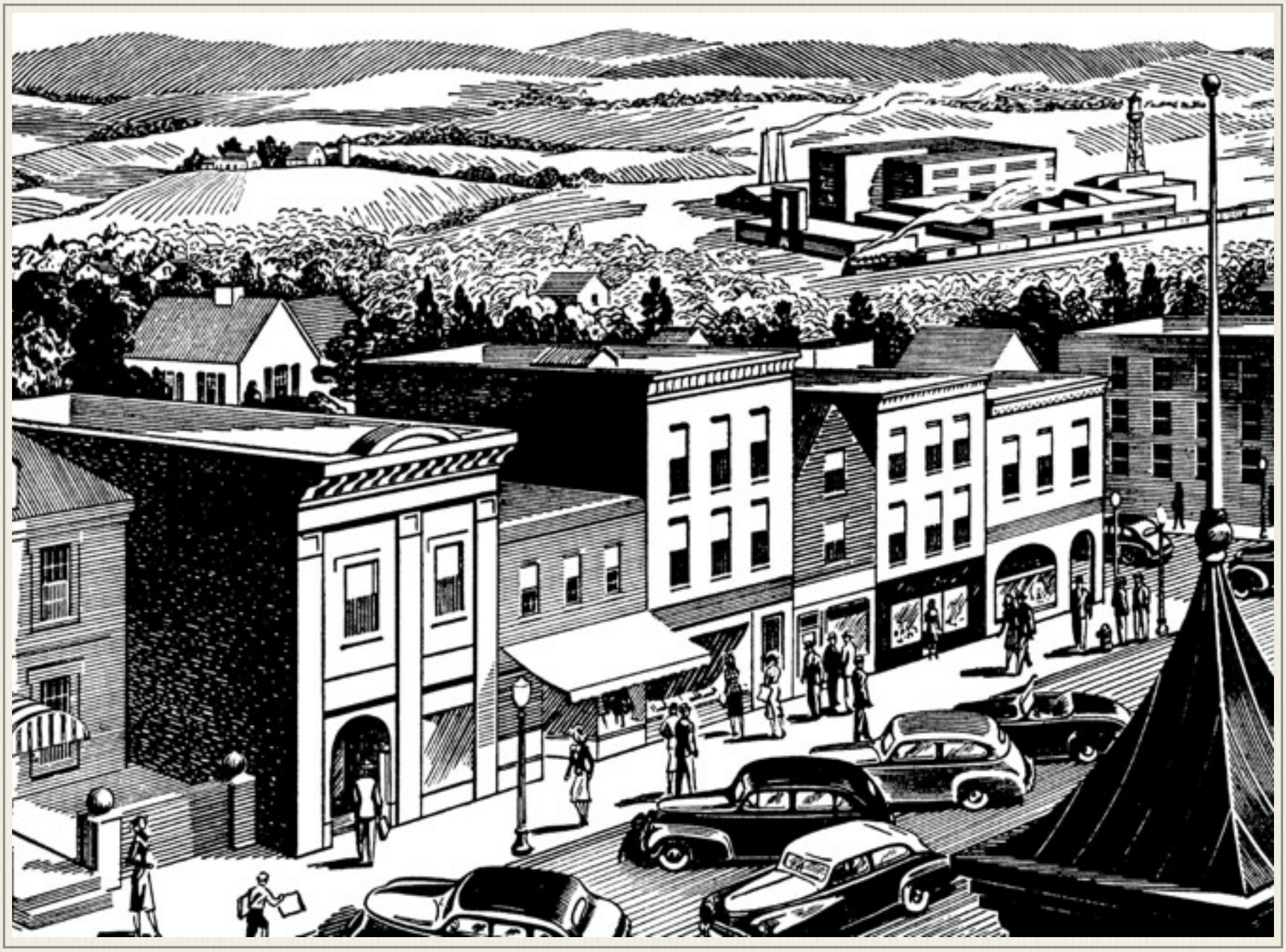
分布式系统导论

陈康 著



前言

本书阐述的是在分布式系统中经常使用的技术，方法和原理。随着分布式系统变得越来越重要，大数据以及云计算技术也逐渐发展起来，技术人员对此有非常的渴求以期望建立完善的知识体系。在国内还缺乏一本能够深入讲解分布式系统原理的教科书的背景下，本书正是为了满足这样的需求而撰写的。

本书可以作为信息类专业如计算机系，软件学院以及相关研究所的研究生教材使用，也可以作为有兴趣的本科生进行深入学习使用。本书的另外一类读者对象是各个技术公司，企事业单位的技术人员，特别是进行分布式系统底层开发的开发人员，希望能够对这些读者起到技术原理梳理的作用。阅读本书的预先知识不需要太多，读者只需要有基本的编程经验，并且对于操作系统以及网络数据传输有一定的了解即可。

分布式系统所采用的技术非常的广泛，为了解决一些技术难题，研究人员以及技术专家提出了许多聪明的办法来解决实际问题，在本书中不能一一阐述，希望本书能够起到抛砖引玉的作用。

本书实际上是课堂讲课过程的搜集整理，口语化比较明显，另外也由于作者的能力所限，成书难免存在各种各样的错误，希望读者谅解以及提出批评指正。

本书的完成需要感谢许多人。特别需要感谢的是谷歌公司的大力支持，感谢谷歌公司的朱爱民先生和胡菲然女士，没有他们的坚持与鼓励，本书无法完成。感谢参加我课堂的各位研究生与本科生，在课堂的讨论中也澄清了许多关于分布式系统的概念以及技术细节，对于成书起了很大的帮助。

另外，我也要感谢我的家人的陪伴。

陈康

2015年 春

清华园

第 1 章

分布式系统概论

分布式系统是一大类非常重要的计算机系统，在整个计算机技术中占有非常重要的作用。本章是本书的内容概述，通过本章的阅读读者可以大致了解分布式系统所要研究的内容以及各个部分的相互联系。本章的目的是在提供总体认识的基础上，使得读者在阅读其它章节的时候做到心中有数，理解每一个章节在分布式系统的全局中所处的位置。

本章的内容主要包括以下几个方面：

1）分布式系统的组织方式：分布式系统有多个方面的组织方式，这与实际的应用是密切相关的，并且与底层的组成结构有关。对于理解分布式系统的结构来说，不同的组织方式会有不同的侧重，了解系统的组织还是非常有必要的。这部分内容能够帮助用户有一个感性的认识。

2）分布式系统中的相关模型：模型往往是实际系统的抽象，在分布式系统中会涉及到不同的模型对象，例如通信的模型，节点失效的模型等，如果需要对分布式系统有一个深入的理解，了解每一个算法所需要的抽象模型也是非常重要的一环。

3）分布式系统的研究课题：分布式系统有其特殊的一些属性，因此在研究之前需要对这些属性进行一些讨论，这里面就包括系统的功能属性，系统的可扩展属性，系统的可靠性可用性属性等。

分布式系统初探

在本书中描述的分布式系统特指是计算机组成的分布式系统。在现实生活中有许多分布式系统的例子，这给大家一个初步的映像，即分布式系统是什么。随着互联网的不断发展，可以说分布式系统已经深入到社会、工作、生活的各个方面。下面是一些典型的分布式系统应用。

1）电子邮件系统：电子邮件系统是许多机构和单位的必备信息基础设施，通过分布式的邮件服务器使得用户不在线的情况下可以接收邮件，并且在连接网络的时候获取电子邮件。邮件系统是对传统的人工邮件系统的模仿。可以说，现在的大企业，政府机构，学校等如果没有电子邮件系统支持的话，会带来很大的不方便，沟通也成了一个很大的负担。

2）聊天软件：当初腾讯开始模仿ICQ系统推出小企鹅QQ聊天软件的时候，谁也不会想象得出一款简单的在线聊天软件能够建立一个如此庞大的商业帝国。聊天软件是一个非常典型的分布式系统，使得分布在不同位置的两位用户能够进行面对面地进行聊天。

3）搜索引擎：搜索引擎也是一个非常重要的分布式系统。不仅仅是用户使用方式是分布式的，从全球不同的位置都可以享受搜索引擎的快捷服务；同样构造搜索引擎本身的技术也是分布式的。为了支持搜索引擎如此大的数据量以及用户数目，使用单台机器是远远不够的。尽管随着技术的发展，半导体工业带来了非常快速的计算机例如大型机，但是在处理互联网数据量的时候单台计算机的能力显然是完全无法满足的。为了构造搜索引擎，谷歌公司提出了一整套的分布式系统解决方法来处理大数据的问题。在本书的后面讨论中，将会展开深入的研究，给读者详细分析在分布式计算后台所采用的技术原理。

4）办公系统：办公系统是当前所有企业的工作基础，通过办公自动化能够大大加快企业的工作流程。不同的企业会依据通用的办公系统组件为基础建立自己企业特殊的办公管理方式。最典型的软件例如微软的Office组件，这可以说是当前所有文档的通用标准。传统的Office套件是在单机上运行的，只能让一个用户使用。随着云计算的发展，越来越多的办公系统被移植到分布式环境下，使得用户通过浏览器就可以操作办公系统。这种方式就将传统的单机的程序扩展到分布式系统中。

5）网络游戏：网络游戏也是当前一个非常典型的分布式系统。传统的游戏通过单机运行，这样的话有一个限制，即游戏的对手是计算机，这对于游戏的可玩性有巨大的影响。由于人工智能计算上的限制，单机游戏的人工智能模式往往会很快被玩家所掌握。而通过网络游戏，玩家的对手是另外的玩家，因此可玩性会大大提高。网络游戏是一个非常典型的分布式系统与用户密切结合的例子。

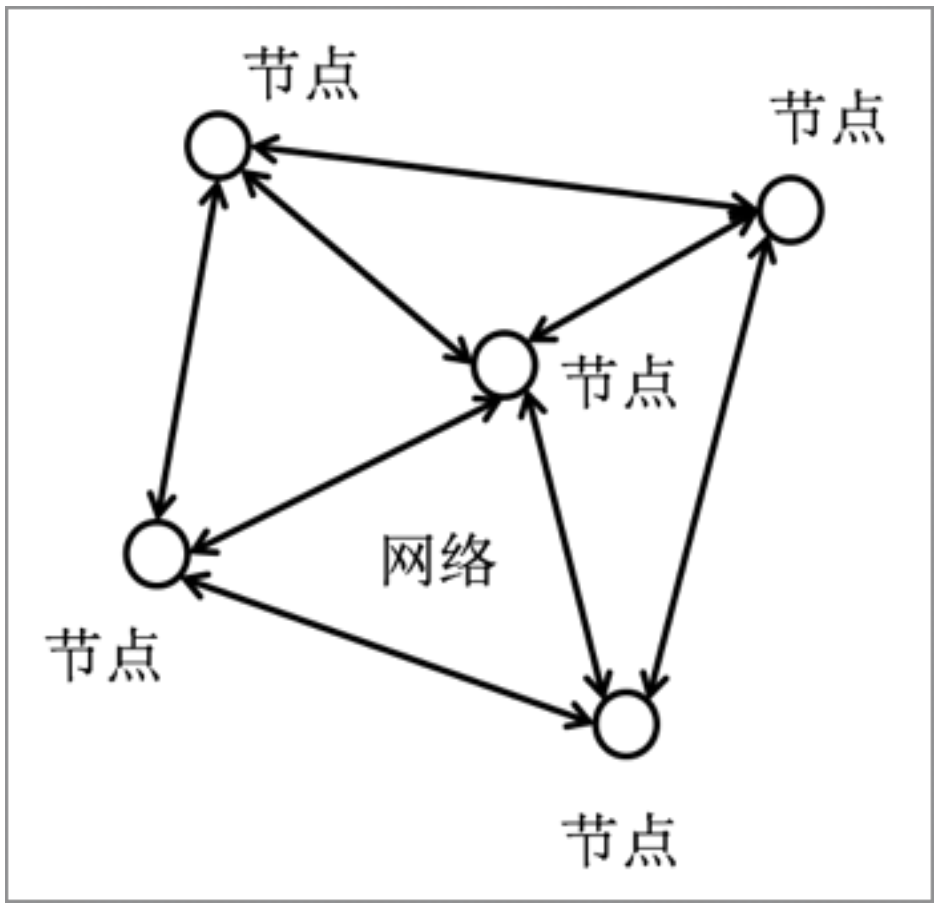
当然，实际的分布式系统还远远不止以上所描述的着一些应用，在平时的现实生活中也会遇见各种各样的分布式系统，并且许多分布式系统使用者还感觉不到。典型的如一辆汽车的控制系统也可以被认为是一个分布式系统。因此，分布式系统简单的说只要一个系统内部有超过一个以上的处理单元，彼此之间通过各种形式的网络进行通信就可以被称之为分布式系统。显然，分布式系统的定义是极为宽泛的，不仅包括了传统所理解的分布式系统的概念，即通过网络将多台计算机连接在一起，也包括了在一台多核的计算机内部通过总线形式将多个处理器的核连接在一起。

分布式系统这个概念中，有两个非常基本的比不可少的元素，即一个是具有多个处理单元，能够独立执行代码进行计算工作；另外一个是多个处理单元通过网络进行连接，这里的网络完成了数据交互的工作。

分布式系统概念中的这两个元素缺一不可。首先是多个处理单元，一个处理单元是构成不了分布式系统的，那是早期单核处理器时代的处理模型；

其次是网络连接也是一个必要因素，如果没有网络连接的话，多个处理器就不能够互相协作，也就只能被称为多个处理单元而不构成一个分布式系统。值得注意的是，处理单元在这里是一个抽象的描述，在实际系统中，它可以指代一台计算机，一个笔记本，一个移动的计算设备等。在多核的环境中，这个处理单元也可以表示为多个核心中的一个。在本书中，在不同的上下文环境中，处理单元可以被描述为服务器，计算节点，存储节点，节点，机器，进程，线程等。在概念上，处理单元就是一个能够执行代码的机构，只是在不同的上下文环境下用上面的某一个名词更加合适一点，读者可以从具体的上下文中分辨出来。

图1.1 分布式系统的概念图



因此，本书的讨论基础就是一个非常简单的模型，如图1.1所示。图中的圆点可以是物理的计算节点或者是逻辑的计算节点（因此，甚至是单机单核的系统中，多个处理的线程或者多进程也可以被认为是逻辑上的一个分布式系统，需要通过分布式系统的概念去理解）。

分布式系统具有许多不同的特征，例如处理节点可以处在不同的位置。有些分布式系统的处理节点距离非常近，例如在一个片上系统中或者多核处理系统。有些分布式系统的处理节点距离非常远如分布式在地球上的不同部分，例如电子邮件系统或者DNS域名解析系统。当然，也有非常多的分布式系统处理节点的距离会处在这两个距离之间，例如数据中心内部构造的集群系统，处理单元之间的距离会限制在数据中心的内部。

计算节点也可以作不同的层级组织。例如，可以定义这些计算节点的地位是相当的，典型如对等计算系统或者集群计算系统中的节点。或者，也可以对这些节点进行地位的划分，例如客户端/服务器的模型或者在对等模型中加入强节点作为较为可靠的服务中心。将在后文中对这些方面进行详细的分析。

为什么要使用分布式系统

在现实的生活中，使用分布式系统有着不可避免的理由。正是因为不是所有的与计算机相关的问题都是可以通过一台机器去解决的，因此必须要建立分布式系统来实现单台机器所不能够完成的功能。下面从不同的角度来看看建立分布式系统的理由。

系统功能角度

从功能上来说，有一些需要完成的功能通过单个计算机是无法完成的。例如电子邮件系统，不能指望在发送邮件的时候，邮件的接收方在线。在这样的情况下，可以通过邮件服务器做中转。在发送邮件的时候，实际上并没有直接投递到接收方的计算机中，而是投递到邮件服务器中。之后，接收方放可以从邮件服务器中获取数据。在这样的一个应用场景中，通过一台计算机是无法完成整个系统的功能的。

系统性能角度

单机系统遇到的另外一个显而易见的问题是性能问题。单机系统的性能无论如何强劲，但是面对更大的数据量的时候往往也就无能为力。通过分布式系统可以将多个计算节点的性能进行叠加，在计算能力不足的时候，就可以通过增加节点的方式来扩展性能。这种横向的扩展能力在机器的数目上不存在限制，因此在系统构建的时候必须要保证所有的服务器都能够同时进行数据处理，这就需要在软件上解决负载均衡的问题。

系统特性角度

单机系统在某些时候是无法提供一些必要的特性的，其中一个非常典型的特性就是系统的容错性（可靠性，可用性）。单机系统由于物理上的限制不能提供系统的容错性，这是因为一旦这台机器出现问题的时候，依赖于这台物理机器的之上的系统就不能继续工作，例如运行在物理机器上的操作系统和应用程序也就同时崩溃了。而分布式系统则可以用来解决这样的一个问题。由于分布式系统存在多台机器，那么在某一台机器失效的时候，其他的机器可以接替它的工作，使得整个系统仍然能够进行正常工作。

另外，还有其它的一些特性也需要通过分布式系统获得，例如系统的安全性可以通过分布式系统完成，通过多个机器互相协作可以帮助抵御诸如拒绝服务攻击这样的行为。在安全性方面的研究是当前计算机系统的研究的热点，这既是分布式系统能够带来对单机系统更好的安全性，而分布式系统本身也带来了特有的安全特性。

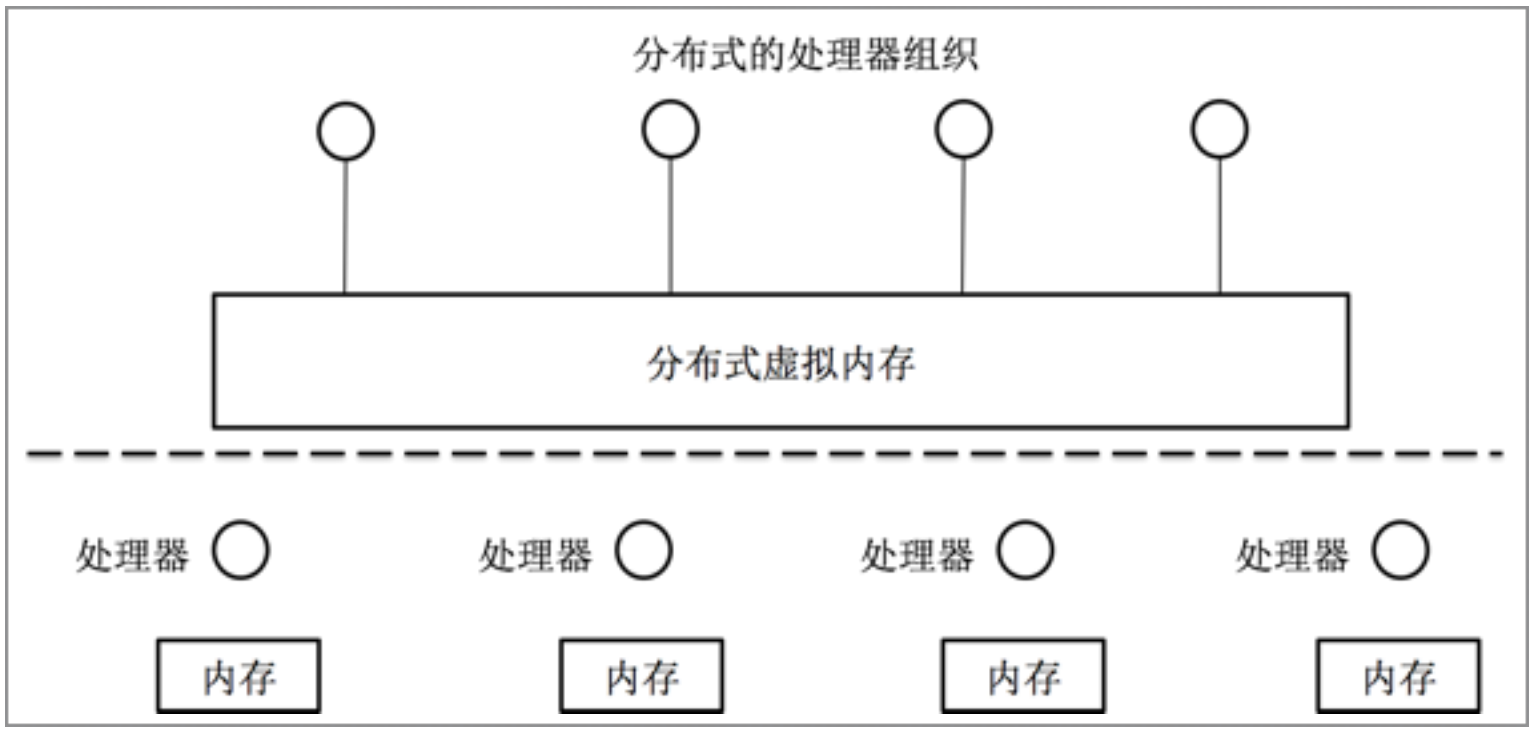
分布式系统的组织方式

在进行深入的讨论之前，下面讨论几个典型的分布式系统的组织方式。这里列举的分布式系统的组织方式与具体的应用无关，而是一类应用的组织方式，因此这里的一个组织方式可能对应着实际的一大类应用系统。

分布式共享内存模式 Distributed Shared Memory

分布式共享内存系统由于性能方面的问题，在当前的分布式系统中的应用不是很多，但是对于分布式共享内存系统的研究可以帮助理解分布式系统中的一些基本概念和原理，特别是对于分布式系统中的一致性的理解有帮助。将专门列出几章的篇幅讨论在分布式共享内存中的一致性问题。分布式

图1.2 分布式共享内存的组织



共享内存的概念非常简单。图1.2就是典型的一个分布式共享内存系统的基本构成。

共享内存方式是计算机系统模型中一个非常典型的模型，这个典型的模型被许多单机系统所采用，例如当前的多核处理计算机系统，基本上都是使用了共享内存的方式。共享内存的方式有一个优点就是对于程序员来说可能比较好理解，因为这是他们所熟悉的体系结构，并且平时使用的单机系统就是这样的结构。一个自然而然的想法就是建立一个分布式共享内存系统，使得用户在使用多台计算机的时候，能够看到同样的共享内存的体系结构。在这个分布式系统中的每一台计算机都提供了自己的处理单元，并且所有的计算机内存在逻辑上被合并为一个统一的内存地址空间。这样的话，不仅是程序员，对于应用程序来说也认为自己运行在一台大型的计算机之上。分布式共享内存的方式能够将多个计算机连接为一台大型的计算机，是一项与现有系统兼容的令人映像深刻的技术。但是，由于在分布式环境中所采用的网络技术无论是带宽还是延迟都无法与单个节点的各种总线系统相媲美，在性能上也就大打折扣，因此分布式共享内存在实际使用中非常少。

分布式共享内存系统的概念简单并且清晰，而构建起来又极其困难，这其中的原因会涉及到一个分布式系统的基本问题，即一致性的问题。因此，分布式共享系统一般被用来阐述分布式系统的一致性问题，帮助读者在一个具体的例子上理解这样一个概念。

客户端与服务器模式

除了分布式共享内存的方式之外，其它的分布式系统的组织方式都将每一台计算机作为一个独立的计算机节点，都被成为非共享内存方式。客户端与服务器的模式是一大类非常典型并且十分重要的模式。客户端与服务器的模式也是一种标准的网络编程方式。在网络程序的通信两端，一端使用了服务器的编程方式，等待另外一端的连接，而另外一端则使用客户端的编程方式，连接服务器并进行网络通信。随着这一模式的标准化与通用化，这一说法也被逐渐固定下来，指代了一种网络服务的工作方式。

图1.3 客户端与服务器模式

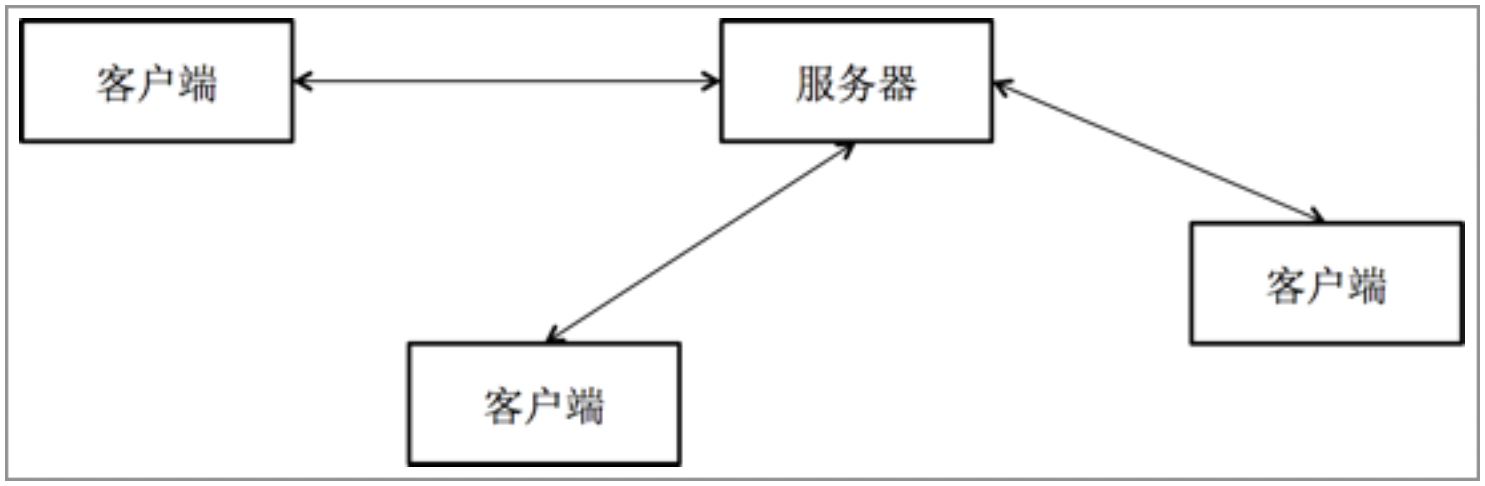


图1.3是客户端服务器的模式，客户端将请求发送给服务器端，而服务器端则将回答返回给客户端。这种方式的一个典型的应用是网页浏览器和网络服务器。这种模式的构建起来的互联网应用可以说是最重要的应用了。这一层的典型的应用层协议就是HTTP协议以及建立在这种协议周边的一系列协议。这些协议不仅起到了信息传播的作用，同时现在越来越成为网民的沟通工具。客户端服务器的模式在这种分布式系统中起到了重要的作用。

集群计算模式Cluster Computing

集群计算模式是当前大规模数据处理的基本方式，同时也是高性能计算的基本方式。集群计算模式指的是在一个局域网环境中，数量众多的计算机通过网络进行互联，并共同进行数据处理。集群计算是网格计算以及云计算的一个构建基础。集群计算一般局限在一个数据中心或者计算中心的内部，在物理位置上也相近。在集群的规模上，节点的数目可以从几十个节点开始一直到数千个节点，甚至在典型的高性能计算集群中会有上十万个节点构成了一个集群计算机。在数据中心进行大规模数据处理的集群计算机也会有数千个节点共同工作。集群的配置是极其灵活的，可以根据问题的规模以及资源的多少进行按需配置。另外，集群的规模可以进行动态的扩展，在性能不

够的时候通过后期投资，增加节点与网络，就可以提高整个集群的计算性能。

在集群系统中，多个服务器被安排在一个机架上，而多个机架通过机架间的网络共同构成一个集群计算机。集群计算机中的每一个节点都是独立的计算节点，包括了完整的操作系统以及应用软件。集群之间的内部互联网络可以是常见的商用以太网，或者是性能强劲的InifniBand网络。InfiniBand相对来说具有更短的延迟以及更高的网络带宽，可以用来提高计算的速度，一般用以高性能计算，解决大规模的科学计算问题。在高性能计算集群中，为了将速度提高到一个更高的高度，甚至也会通过专用网络的形式进一步提高网络带宽，降低延迟。

集群中的物理节点往往是同构的，采用相同配置与性能指标的计算机连接在一起。但是，这并不是说明集群中的所有节点会扮演相同的角色。为了方便集群的使用，图1.4是通常的集群中各个节点的角色分配与工作方式。

图1.4 集群计算模式中的节点角色分配

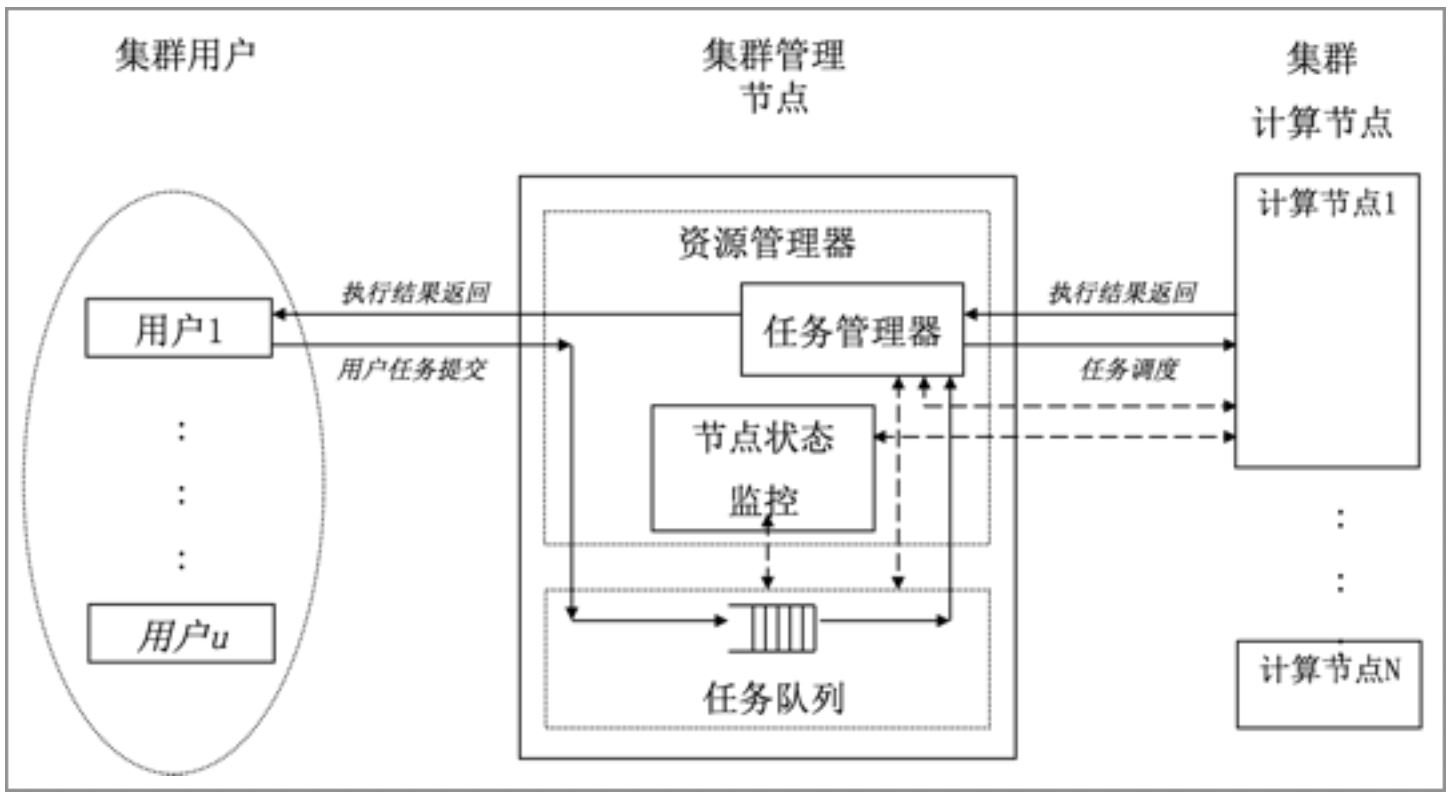


图1.4是集群计算机系统中的典型的角色分配方式。集群计算机由于性能强大，除非有特殊的建设目的，通用的计算机集群往往会给多个用户多个应

用程序提供服务。用户可以通过分时的方式进行使用，或者如果资源允许的话，多个应用程序也会共同运行在一个集群中。在这种方式下，就需要一种方式让用户可以提交自己的计算任务。在集群中，往往有其中的一个节点将担任这样的角色，这个节点往往被称之为头结点（Head Node），或者管理节点（Manager）。头结点中有一个任务队列，新的任务将被插入到这个任务队列中，然后按照某一种先后顺序被调度到后台的计算节点中。头结点除了有任务管理作用外，还起到了系统监控的作用。系统监控模块收集后台数目众多的计算节点的监控数据，观察是否可以调度新的任务到计算节点。

上述的机制是典型的集群的运作方式，无论是高性能计算集群或者是大数据处理的集群，都通过类似的方式运行。可以看到，在这里软件是整个集群正常运行的关键。除了每一个单机的操作系统以及普通的单机应用程序外，集群系统也需要运行支持集群正常运行的特定的软件。

1）集群系统的安装部署软件：集群系统的安装与部署与单机系统有很大的不同。集群系统由于数目十分庞大，不太可能通过光盘或者U盘进行系统安装，否则的话会费时费力。集群系统的安装需要一种自动化的方式安装数据中心中的所有节点。典型的方式是通过安装头节点，并通过类似Intel PXE等的技术进行操作系统的远程自动安装。在系统安装完成后，就可以通过操作系统的支持完成应用程序的部署。

2）支持网络执行的集群计算操作系统软件：集群中的操作系统必须要支持网络数据通信，除此之外与单机的操作系统并无本质上的区别。当前的集群计算机的操作系统大多数为Linux操作系统，原因是Linux操作系统具有一个很大的生态系统。由于是开源软件，可以依据特定的集群进行修改来提高性能。由于生态系统完善，也很容易得到社区的支持。Linux操作系统的各个发行版都可以用于集群操作系统中，当然在性能上都有差别，需要一定的调优。例如一个大数据处理的套件叫做Hadoop，其推荐的运行环境为RedHat，或者其开源编译的版本CentOS，或者为Oracle Linux。Windows，

BSD等都可以作为集群中的操作系统，但都远远不及Linux成熟以及各项软件支持完善。

3）集群系统的监控软件：监控软件是集群系统中必须具有的软件功能，如果没有监控系统，就完全无法了解集群的运行状态，从而失去对集群系统的控制。集群系统需要监控的数据指标是极其庞大的。监控服务器CPU、内存、网络、磁盘IO等系统参数进行全面的收集与存储，以全面了解集群运行的状况，并提供告警功能。监控虽然是一项简单的基础性功能，但是要对数目庞大的计算机节点进行监控也是一项技术挑战。这一基础软件往往是集群操作系统安装之后第一个必须要配备的软件，以全面掌握集群中各个节点的状态以及程序运行的状态。

4）集群系统的调度器软件：从前面的集群系统的使用模式可以看到，集群调度软件是使用一个集群的基础。调度器软件控制整个集群真正发挥作用的应用程序的执行。调度器软件通过搜集监控系统的反馈信息去决定系统的下一步需要执行的应用程序。调度器软件的接口包括面向最终用户与应用软件的接口以及对于后台的调度的接口。调度器软件面向用户与应用程序的接口将接收用户发送过来的任务，并放置到一个任务队列中。调度器软件有一个重要的模块是任务调度算法，任务调度算法可以依据不同的目标来优化整个集群的性能。简单的调度算法例如先进先出算法FIFO，或者例如优先队列算法Priority Queue。当然，依据集群形式的不同，任务调度算法还有可能具有非常复杂的形式。

5）集群系统的应用程序编程环境：以上各个模块都是系统的基础软件模块，这些模块的存在都是为了能够在集群环境中去有效地运行应用软件。集群的应用软件的编写与传统的单机应用程序的编写有很大的不同。其中一个最大的不同是分布式环境下应用软件执行的环境不是一个单机的环境，而是集群中的可能数千个节点。因此不太可能去为每一台节点去编写一个应用程序，然后通过网络通信的方式将它们连接在一起，如果是这样的话对于应用程序员来说负担太重了。应用程序的编写方法必须具有这样的能力，即应用

程序编写与系统的规模无关，使得相同的应用程序可以运行在不同数据的节点之上。完成的应用程序能够依据系统的规模的不同进行执行，去处理不同的数据量。因此，在集群中，往往需要一个运行时系统去完成这样的功能。针对不同的需要解决的问题，当前集群中的编程方式典型的有基于消息传递的编程模式MPI，主要用于高性能计算，是高性能计算领域编程的事实上的标准。另外的典型计算模式为进行大规模数据处理的MapReduce编程方式，屏蔽了底层系统的扩展性问题和可靠性问题，这在以后的讨论中会展开研究。

在集群的应用方面，传统的集群计算机通常用以处理大规模的科学计算问题，形成了一个专门的学科，被称为高性能计算或者说是并行计算。当然，随着数据量的扩大，集群计算现在有了一个更为重要的应用场景，即用以大数据的处理。这两个方向各自有自己独立的发展，但也出现了逐步融合的趋势。

对等计算模式

对等计算是与服务器和客户机计算相对应的。这里只是使用方式的对应，对于底层的网络技术来说，采用的是相同的网络数据传输方法以及编程方法。客户端服务器的计算方式在互联网中使用得非常广泛，用户往往将计算提交给服务器去完成。因此，从这个角度来说，计算的分配不是均衡的，在服务器端具有强劲的性能，而客户端的性能比较低。由于是网络中的服务计算机，需要的计算能力很高，因此很有可能通过大规模集群的方式完成。在这样的模式下，数据流动的方式往往是服务器将大量的数据传输给客户端，而客户端则将必要的少量的数据传输给服务器。例如网络下载，视频传输等，都是典型的这种数据传输的方式，这也是一种被称为非对称网络ADSL存在的基础。但是，在这样的模式下，客户端之间的网络带宽往往是被忽略的，将大量网络带宽的压力推送到服务器端。

对等计算的思考方法正好与上述的思考方法相反。当然，对等计算并不反对在客户端与服务器端进行数据的传输，因为这是不可避免的。对等计算希望通过客户端之间的相互帮助来提高系统的性能，充分利用数目众多的（数百万甚至上亿）客户端的带宽能力来提高数据传输效率。由于客户端之间并没有像服务器与客户端一样有明显的性能差别，客户端的地位是相当的，因此被称为对等计算，或者点对点计算Peer-to-Peer Computing。

图1.5 客户端服务器计算模式与对等计算模式的对比

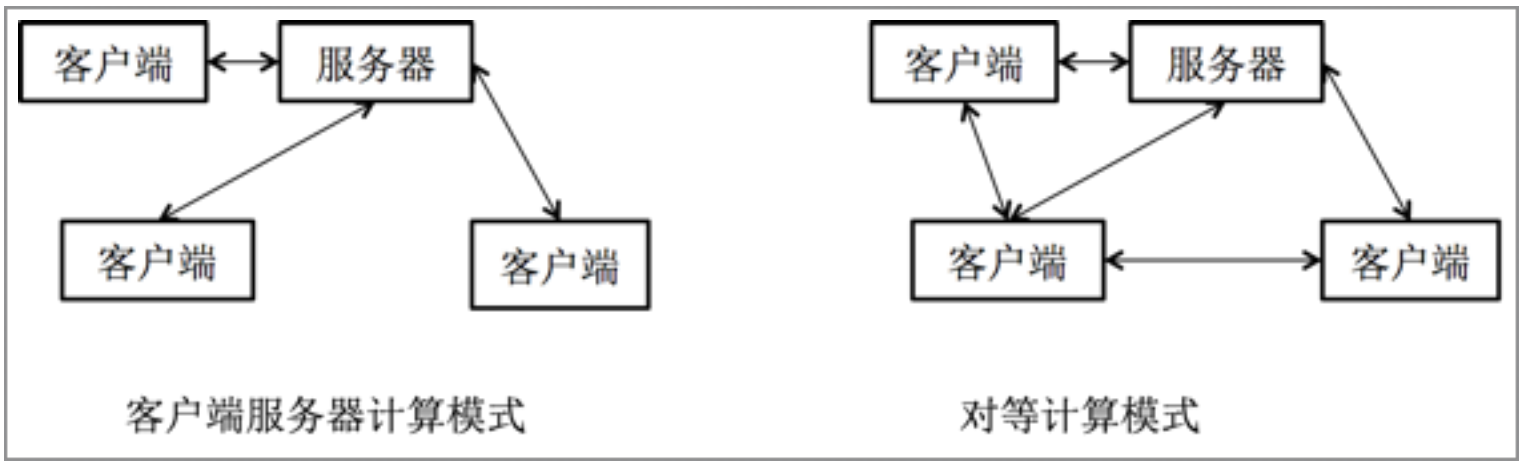


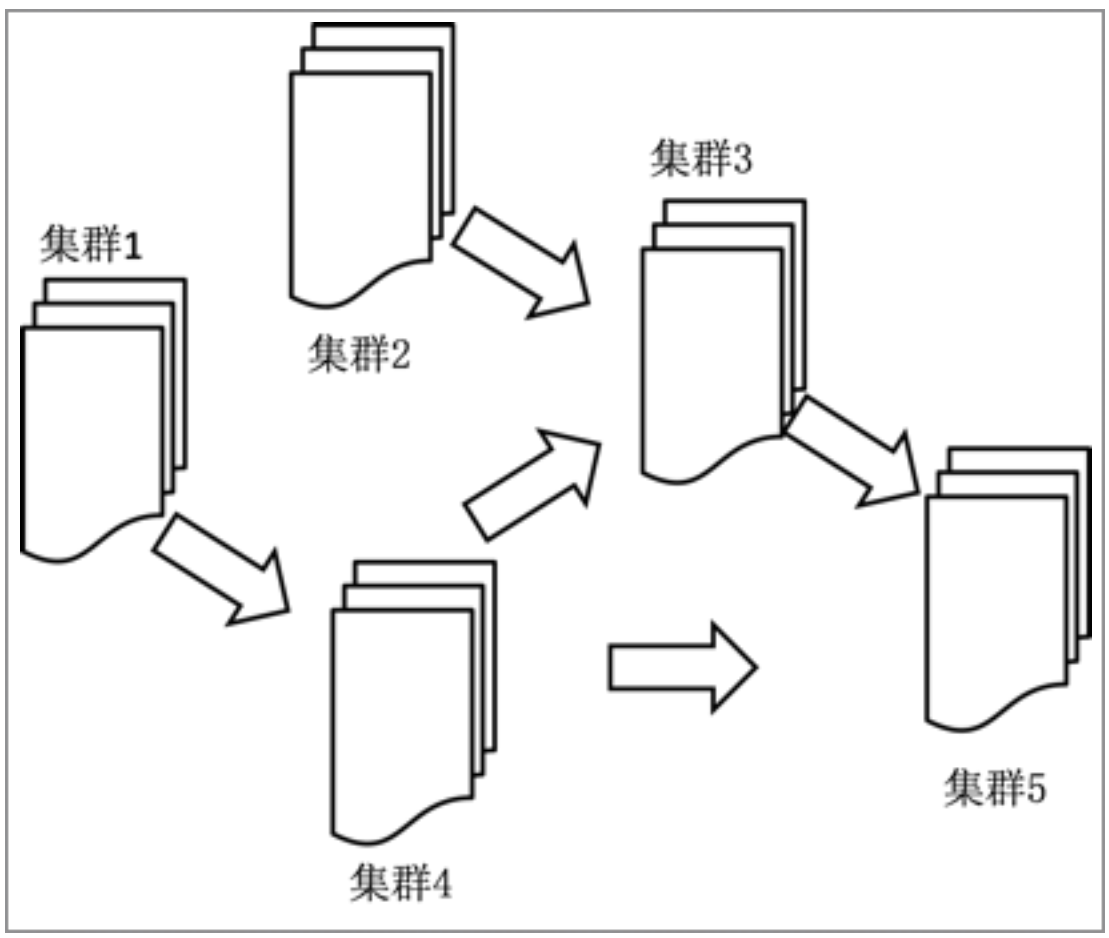
图1.5是客户端服务器计算域对等计算的对比。从这个图中可以看出，客户端服务器计算的网络流量发生在服务器与客户端之间，两个客户端之间并没有数据的交流。而对等计算的数据流量发生在两个或者多个客户端之间，虽然每个客户端所能提供的网络带宽是非常有限的，但是想象一下互联网规模的客户端所能够提供的聚合带宽就非常的惊人。因此，对等计算往往适合一大类需要大量数据传输的应用，例如数据下载，或者网络视频播放等。

对等计算的最基本问题是由于现在已经去除了中心的服务器，因此资源如何寻找就是一个关键的问题。最直接被采用的方法被称为广播的算法（或者被称为洪水算法flooding）。但是通过广播的方式就会造成许多不必要的流量，导致性能的下降。因此，随着研究的发展，对等计算逐步建立一个重要的概念，被称为覆盖网络，或者是应用层路由，这在本书的讨论中将逐步展开。

网格计算

网格计算是在集群计算出现之后出现的概念，这个概念更多的应用领域还是出现在高性能科学计算方面。网格计算与集群计算密不可分，用一句话说明网格的概念就是“集群的集群”。在真正使用得集群应用方面，往往每一个集群的构造都有一个特定的应用，并针对这样的一个应用进行充分优化。但是，在科学计算的过程中，往往需要解决多个不同的科学问题，然后综合各个部分才能够获得最终的结果。网格计算就是在这样的应用需求下产生，以提高各个科学计算集群之间的相互沟通的能力。

图1.6 网格计算模式的概念构成



以一个典型的地球系统模式科学计算为例。为了理解当前地球各个系统的交互模型以及对于未来的气象水文进行预测，现在的基本研究方法就是提出不同的模型进行计算机的模拟计算。从模型的角度来看，大气的模型，海洋的模型，陆地江河的模型等是各不相同的。每一个模型的计算量都非常大，都需要建立独立的集群计算机进行模拟计算。

图1.6是网格计算的模式。在这个模式下，各个计算的部分分别由不同的集群计算完成。但是，为了理解整个地球的运作模式，单个的模型往往还是不够的，因为大气和海洋，大气和陆地，海洋和陆地之间都会有相互影响的方面。这个时候需要一些新的就群去完成两个模式之间的交互，被称为耦合模式。网格计算就可以应用在这样的一个场景下。

云计算

云计算是近几年来兴起的一个重要的计算模式，其所包含的含义非常的广泛。云计算从互联网应用，集群计算，移动计算发展过来。云计算包括了各个层面的含义，包括底层的基础设施，中间的计算平台，上层的应用构建直至终端用户的体验。很难给云计算下一个确切的定义，但是云计算与提供服务是密切相关的，云计算中的功能都以服务的形式提供。云计算中有三个非常重要的概念，即基础设施即服务（Infrastructure as a Service），平台即服务（PaaS：Platform as a Service），软件即服务（SaaS：Software as a Service）。虚拟化在云计算中起到了非常重要的作用，包括网络的虚拟化，计算的虚拟化以及软件的虚拟化等。

分布式数据库

分布式数据库不是分布式系统的一种组织方式，而是一大类非常重要的应用。数据库在单机系统中已经是一个非常重要的组成模块，由数据库系统发展出来的一系列非常重要的概念在分布式系统中也处于一个非常重要的地位。

数据库是数据存储除了文件方式（通常被认为是存储非结构化数据的）之外的另外一种存储方式，其核心的概念是数据的结构模型以及在这个结构模型上的操作。数据库系统由数据存储的引擎，数据操作与查询语言以及对于事务的处理构成。数据库系统发展出来的一个非常重要的概念是事务

（Transaction）。通过使用事务的方式能够保证整个数据库系统的正确性。分布式数据库系统是一个非常大的课题，值得用另外一本书去阐述。在这里，需要特别指出数据库中的事务的特征，这些特征将在理解分布式系统的时候起到重要的作用。在分布式系统，或者分布式数据库中，事务被定义为一组操作，这一组操作被认为是逻辑上的一个完整的操作。数据库中的事务具有四个方面的特征，即：ACID：Atomic，Consistency，Isolation，Durability。这四个特征各有各的含义，列举在下面：

原子性，即Atomic：事务中的一组操作必须是全部完成，或者任何一个操作都没有发起，不能出现只执行部分操作的情况。如果这组操作中的某一个操作没有成功，那么所有操作就被必须要求都没有成功，使得数据库的状态完全没有任何改变。值得注意的是，这种原子性，包括以下的其它特性，必须要求数据库在所有状态下都保持，例如即使是硬件出错，停电，网络出错等任意情况下，都必须保持原子性。

一致性，即Consistency：一致性的要求是需要使得数据库能够从一个正确的状态转换的另外一个正确的状态。一致性说明了操作上的一些明确定义的不变性，而一致性的要求就必须保证在事务的发生前后，这些不变特性不能够受到破坏。例如，某一个事务是在银行的两个账户之间进行转账操作，这么一个不变性就是这两个账户的余额总数保持不变。事务需要保持所有的需要定义的不变特性。值得注意的是，一致性并不会去保证应用程序所需要的特性，这种特性需要应用程序自己保证。一致性会去保证底层的数据库所定义的一些规则。

隔离性，即Isolation：隔离性关系到两个或者两个以上的事务。隔离性的含义是一个事务不能观察到另外一个事务的中间状态。如果观察到另外一个事务的中间状态的话，就相当于隔离性被打破，那么事务的操作就会基于一个中间的状态，这往往是不正确的。隔离的特性使得事务的操作能够等价于顺序操作事务（按顺序一个一个去执行事务）。提供隔离性实际上是进行

并发控制的主要目标。并发控制的含义在分布式系统中也十分重要，使得在多个客户端发出请求的时候，让系统能够按照某一种顺序去完成操作。

持久性，即Durability：持久性的要求是一旦一个事务被提交（执行成功），那么事务产生的效果将永远存在。这样的话，持久性的要求通常需要一个永久的存储例如硬盘去完成。事务被提交之后，所有的结果将被存储到硬盘中。

虽然上述的几个特性都是数据库中的事务的特性，但是这几个特性在分布式数据库，以及一般的分布式系统中都非常重要。在本书的后面的一部分内容将在除了分布式数据库的其它的环境中讨论事务的组成以及特性。例如，在一个分布式文件系统中，对于一组文件的操作的话就有可能需要通过事务去完成，以保证所有的操作都是“完整的”操作，而不是一部分的操作。通过这种方式的话，使得整个系统都具有可恢复性，在机器重启进行恢复的时候，系统中不会存在执行到一半的情况。这种特性能够保证即使出错的情况下，文件系统的本身也不会遭到破坏。在讨论日志系统以及两阶段提交的时候将回到这个问题。

分布式系统中的概念与模型

前面已经展示了分布式系统的一些基本等概念，这包括两个重要的部分，一个是处理的单元，另外一个是连接这处理单元之间的网络。这两个部分缺一不可。但是，现在还没有深入地展开对于这两个部分内部模型的一些讨论。本节首先讨论消息在分布式系统中的网络传递模型有哪些类型；之后说明在分布式系统的模型划分；最后是分布式系统中的出错模型。这些都是一些基本的概念，在本书以后的讨论中都会需要应用到这里的概念。这些概念都十分简单，但是对于理解整个分布式系统来说却是十分重要的，读者一定要引起重视。

分布式系统的本地性原则

这是一条基本的原则，在分析系统的时候非常重要，特别是从一个单机系统转换到分布式系统环境下，经常对这一条原则感到非常不适应。但是实际上这条原则是非常简单也便于理解的。

本地性原则说的意思是：在分布式系统中的每一台服务器都不能够知道系统的全局信息，每一台服务器只能依据自己本地的状态（本地内存或者本地硬盘，依据不同的系统环境而定），以及其它系统发送给自己的信息做出判断。这是一个显而易见的原则，但是对于理解分布式系统以及相关的算法却是极其重要的。分布式系统中的一个节点如果需要去了解整个分布式系统的状态的话，唯一可以做的事情就是发送请求消息给系统中的所有节点，等待所有的节点返回信息给自己。即使是在这样的情况下，当消息传递到搜集进程的时候，整个系统的状态非常有可能已经发生了改变。因此确切获知系统某一个时刻的状态是不可能的。特别的，分布式系统的每一个组成部分的

时间都是本地的，而不是全局的。每一个处理单元都可以访问本地的时钟，但是访问其他处理单元的时钟的时候会引入一个无法避免的延迟，因此各个处理单元的时钟之间是有漂移的。整个分布式系统不存在一个绝对的全局时钟。

本地性原则是设计整个分布式系统的基本原则，算法的每一个步骤都需要依据这一个原则进行，因为只能从这么多信息中继续下一步的工作。另外一个值得注意的地方是本地性原则是运行时的状态，而算法或者协议可以是全局可知的。例如，一个浏览器服务器的系统中，可以合理的假设通信的双方都遵循着TCP/IP协议的每一个步骤。这样的话，无论是客户端还是服务器端都可以依据当前的状态知道下一步该做什么样的工作，而不至于使得整个系统无法继续运行。

分布式系统中的操作顺序

这部分内容将在分布式系统的时间这一章节中详细讨论这个问题，在这里给读者提出一些最基本的概念。读者应当都知道这些事实，无非就是需要澄清一下。首先的一个概念是从分布式系统中的每一个处理单元自身来说，对于操作的先后顺序定义是极其方便的。即，一个单线程程序的执行流程本身就定义了每一条指令（操作）的先后顺序，先执行的指令先于后执行的指令。但是，在分布式系统中两个处理单元之间的操作的先后顺序往往是无法定义的。由于缺乏全局的时钟，往往不能够判断在一个进程A内部的一个操作a和在一个进程B内部的另外一个操作b的操作顺序。首先是定义这个顺序可能没有意义，其次通过本地时间戳的方式（没有全局时钟）无法比较a和b谁先谁后。这实际上是分布式系统的一个本质的问题。只有在某些特定的条件下能够定义a和b的先后顺序，例如a是发送消息的操作，而b是对应的消息接收的操作，那么a必然是在b之前。分布式系统中的操作顺序往往也是理解系统是否正确的一个关键所在。

消息在网络中的传输模型

消息在网络中的传输看起来是一个非常简单的概念，但是在这样一个概念同样需要分不同的情况进行讨论。依据不同的情况，可以比较完善地把消息在网络上的传递分为以下的各种类型。这些情况在实际系统中是非常常见的，读者往往需要在理解具体系统的时候，参照当前系统所需要的消息传输模型。

消息正常传输：这种情况是最为常见的情况，如果不是这样的话，当前的许多网络应用程序基本无法运行。消息正常传输包括两层含义，一个是消息被正确完整的传递到目的机器；另外一个是消息传递是流畅的，在一个合理的时间范围内被正常传递到目的机器。在理解分布式系统的时候，往往就不能够在消息正常传输的情况下去理解系统，这就产生了以下的不同的情况。

消息传输有了一定的延时，通过超时处理：这种情况下消息有可能完整正确的传递到目的机器。由于网络情况比较复杂，消息传递的时候往往产生一定的延迟。在这种情况下，目标机器有两种选择去处理，如果它理解的网络情况比较好的情况下，那么目标机器可以采用超时的机制去处理延迟的情况，即出现超时的话就立即认为消息已经丢失，即使在不久的将来数据能够传输到，那么也认为这部分数据已经丢失。这其实是一种将异步的网络作为同步的网络进行处理的方法。目标机器当然也有另外一个处理手段，即还是继续等待，这就是下面的模型。

消息传输有延时，不通过超时进行处理，而是继续等待：同样的，在这种情况下，消息也是有可能完整正确的传递到目的服务器。如果服务器采用的方法不是去使用超时，而是继续等待的话，那么就会是另外一个模型。在这样的条件下，服务器可能永远也不会收到消息，因为消息传递在网络上的延迟从理论分析意义上是没有上界的。这样的一个特性会导致分布式系统中

的一个重要的问题无法解决。这种模型在理论分析上具有比较重要的意义，而在实际的系统中，往往需要进行简化处理，通过简单丢弃就可以让系统进行下一步的操作。另外，在这个模型下面，消息丢失和消息延迟，甚至远端服务器的出错都是不可区分的，即如果一个服务器不能够等待到一条消息，那么实际上它是无法区分这些情况的（本地性原则）：消息丢失，消息延迟（很长一段时间），发送消息的机器失败。消息丢失可以在理论上认为是消息延迟无穷长，因此也被归为本模型。

消息被篡改：这是与上述所有消息传递的模型所不同的模型。在这个模型中，消息在网络中可能会被篡改。一条消息包含了消息的发送者，消息的接受者，以及消息内容的本身。消息的发送者可能被篡改，使得接收者以为从另外一个消息源中获得消息。消息的内容可能会被篡改，使得接收者者接收了错误的消息。消息的接收者也可能被篡改，使得一个不应当收到消息的接收者接收了消息。这些消息的篡改方式在出现恶意攻击的网络中都有可能出现。因此，这个消息的传递的模型与前面所说的模型完全不同，在某些条件下需要考虑这个问题。当然，消息被篡改的基本处理方式是对消息进行签名和校验，使得消息的各个部分维持其完整性，这样的话消息的接收方就获知消息在通路中是否发生了修改。因此，为了理解这方面的内容，读者最好具有计算机密码学的相关知识。注意，这里的篡改模型往往也被认为是属于拜占庭问题的一类，将在讨论系统的出错模型的时候回到这个问题。

以上就是消息在分布式系统中进行网络传输的模型。可以看到，这些模型的分析都是简单且容易理解的，可以通过仔细思考进行归类。恰恰是这种对模型的归类，使得可以依据不同的场景下依据不同的模型展开思考，可以得出不同的结论。

分布式系统的模型假设

从上面对于消息传输的模型讨论中可以看到，将来在设计和思考分布式系统的时候，其实都是需要一定的模型假设的。这里会基于上述的消息传输的模型来建立分布式系统自身的模型，这些模型都是基于对时间处理的假设。分布式系统的模型可以被分为同步模型，异步模型以及半异步模型三个类别。这三个类别对于理解分布式系统的运行也是十分重要的。这种划分的主要依据是进程对于一件任务的执行时间，消息传递的时间以及时钟漂移的情况是否有时间上的限制来进行的。需要注意的是，这里的模型实际上是在理论讨论中的一些假设前提，在实际的系统中可能会采用其中的一种模型进行讨论。至于采用哪一种模型，则由需要构建的分布式系统的应用来决定。

同步模型：在这个模型下面：1）分布式系统中的每一个处理单元在进行处理的时候，每一个步骤都可以设定一个时间的上限内完成；2）分布式系统中的消息传递能够在一个已知的上界时间中完成；3）由于时间在不同的处理单元是有漂移的，而这个漂移也在一定的已知范围内。总之，所有的操作都可以在一个时间的界限之内完成，如果不是这样的话，就可以认为对应的操作已经失效。这个同步模型情况下，简单地说可以认为分布式系统能够按照一轮一轮的方式进行处理，即在一轮操作中，所有的节点知道其它节点是否完成了相应的操作。从这个意义上来说，同步的分布式模型有了一个全局可以参考的时钟。这样的话，同步模型下的分布式系统在时间上具有可以预测的特性，即过了一段已知的时间之后，可以预先的知道系统中的操作情况和状态如何。与前面所讨论的消息传递模型类似，在同步模型下，可以通过超时的方式去处理消息的传输的过程，超过一定的时间，就可以认为消息已经丢失，而不用再去理会是否消息具有额外延迟的情况。同步模型只能说是实际分布式系统的一个理论的近似，它具有比较好的性质，但是实现起来却是比较困难的，因为实际的硬件系统经常不遵循同步模型的规定。例如，对于消息延迟来说，其上下界限会有可能会非常宽泛。如果系统的超时设定为最短的消息延迟，则会造成大量的无谓的消息丢失。如果系统的超时设定为最长的消息延迟，则会造成整个系统的性能急剧下降。因此，实际的系统

往往并不是同步的模型，生硬的套用同步模型去构建分布式系统是不合适的。

异步模型：异步模型是分布式系统中的最为基础的模型，它对所有的操作时间并没有设定一个上界。这样的一个模型当然适合任意的情况。实际的分布式系统当然也属于异步的模型。在这个异步模型的假设前提下，进程进行工作的时候操作没有一个时间的上限，这个条件包括了进程失效的情况，即操作时间变成了无穷大。消息传递的时间也没有一个上限，可以以任意长的时间去传递消息，这个条件也包括了消息丢失的情况，即消息的传递时间变成了无穷大。这样的话，异步的模型下面没有全局的时钟，使得两个本地时钟之间完全没有办法进行比较，因为本地时钟的漂移也是没有上限的。在异步模型下，操作之间的顺序没法通过时间戳的方式获得，因此需要其它方法去定义操作的顺序。这部分内容将在涉及到分布式系统中的时间那一章中去详细展开这方面的讨论。由于异步模型中关于操作的时间是不可预测的，因此不能够通过超时的方式去处理消息未到达或者丢失的问题。实际的系统当然是异步模型的，但是在纯异步的模型下对于应用的处理是极其困难甚至是不可能的。在分布式系统中有一个基本的问题，即如何在分布式的环境中使得进程可以达成一致的意见，例如共同去决定下一步的操作是删除文件，还是对文件进行重命名。这是一个有关于一致性协商的协议（consensus）。显然，有很多问题都会被归结为一致性协商。在分布式算法中有一个基本的结论，即FLP理论，说的是在异步模型条件下，即使只有一个进程出现错误，一致性协商也是不可能的。这是一个令人沮丧的结论，因为它基本上指出不能够指望分布式系统能够干任何事情。但是好在实际的分布式系统并不是纯粹的异步方式的。例如在一个数据中心内部的集群系统，绝大多数的时间内绝大多数的数据传输都可以在一定的时间上界内正确完成。这样的话就导致了下面的一个半异步的模型。

半异步模型：半异步模型是介于同步模型以及异步模型之间的模型。绝大多数的实际系统是符合半异步模型的。由于处在上述两个模型之间，因此

实际上半异步模型可以有很多种类型，对半异步模型的建模在理论上是很有意义的。在实际的系统中，也可以应用系统的需要给出建议的半异步模型。有一个半异步模型是非常重要的，即整个模型在大部分时间中是处在异步模型的控制之下，但是有一段时间在这个系统中的大多数的节点以及这些节点之间的网络是良好的。这些节点的所有操作都可以在一个时间范围内正确完成，所有的消息传递过程同样在一个时间范围之内正确传输完毕。例如，对于实际的集群系统来说，也恰恰符合这样一个假设，即系统在大多数情况下都工作良好，但在某一些情况下出现消息延迟，节点工作出现问题。在这里，读者会发现实际与理论的假设正好相反，但是理论的条件更加严酷一点，自然如果一个算法能够在上述半异步的理论模型下能够工作，自然在实际条件下也能够工作。这里的一个著名算法就是一致性协商中的Paxos算法，在异步的模型下可以帮助系统中的节点商量出来一个一致的结论，这个算法在实际中获得了广泛的应用。

从上面的模型假设讨论中大家可以看出，模型假设是对实际系统的抽象，这样的话就可以便于对算法的分析。某一个算法可能只是在某一种模型下面才能工作，因此在实际系统应用算法的时候，必须观察算法所运行的环境，从而去确定算法是否能够在实际系统中工作。

分布式系统中的错误模型

分布式系统针对单机系统一个重要改进就是需要对系统中的错误进行处理。诚然在单机系统中也需要进行一定的错误处理，但是单机系统毕竟不能处理整个系统的出错。而分布式系统则可以帮助处理在系统中的某一些单机整体出现错误的情况。从这一个角度来说，实际上也需要对分布式系统中可能出现的单机错误的情况进行分类，从而获得出错的一些模型假设。这些出错模型能够帮助理解一些分布式算法在哪些条件下能够处理哪一些类型的错误。同样的，这些出错的模型也是非常好理解的内容，但是对于理解算法或者实际系统的工作过程会非常有帮助。另外需要注意的是，分布式系统中的

错误模型既有可能发生在处理单元，即处理进程中，也有可能发生在消息传递的信道（channel）中。在这里，信道就是在处理单元之间进行传输数据的网络，在本书中信道和网络连接是同义词。

失效-停止模型 **fail-stop model**：在这个模型下，系统中的某一个模块会停止工作，并停止工作之后就再也不会继续工作。这种失效-停止的模型使得分析分布式的算法能够得到简化，因为不必担心出现问题的模块会继续工作，从而引入系统的复杂性。某一些分布式系统中的模块也具有失效-停止模型的特性。

失效-重启模型 **fail-restart model**：这个模型是对上一个模型的修正，即系统的某一个模块会停止中作，然后过一段时间可以恢复，恢复完成之后继续工作。在这个模型下，算法可以将一部分的信息记录到磁盘中，这样的话在恢复的时候可以从磁盘中获取数据进行下一步的计算。恢复的过程不一定会从失败的下一条指令继续执行，因为这几乎是不可能的。恢复过程会依据现场的情况决定下一步的工作。因此，算法在设计的时候需要考虑如何在恢复的时候能够正常工作，并且与系统中的其它模块进行正确的交互。

任意错误模型**arbitrary fail model**：任意错误模型是一大类特定的错误模型，在这里的错误情况没有进行任何限定，因此可以出现可以任意错误。在实际上往往是指系统中出现了恶意的节点，试图去破坏系统的运作。由于这个问题与一个被称为拜占庭将军问题的一致性协商协议紧密相连，因此，这种错误模型也被称为拜占庭错误。

时间错误模型**time failure model**：时间错误模型发生在同步系统中，如果系统中的某一个模块在一定时间内不能够提供一个特定的结果，不能指出已经完成操作，那么就认为发生了时间错误。

从上面的错误模型中可以看出，这些错误模型实际上也是非常简单，便于理解的。但是，恰恰是这样的分析是进行分布式容错分析的本质。在不同的出错模型下会有不同的处理的手段。在实际系统中经常出现的是失效停止

模型或者是失效重启模型，这个模型在处理数据中心网络是一个合理的错误假设模型。不过拜占庭模型可能并不是像读者想象的那样可以真正用来处理任意类型的错误。一个原因是因为解决拜占庭问题的方法会带来系统的复杂性以及性能的降低，另外一个原因是在实际系统中有可能很难达到拜占庭问题的假设。这在网络安全形势日益严峻的当前是一个非常有可能出现的现象，可以说实际系统出现的可能情况会更加的糟糕，比拜占庭军将军问题的假设还要糟糕。关于这一个课题的讨论在之后的有关章节展开详细的研究。

分布式系统的研究课题

这一小节将主要介绍分布式系统中特有的需要研究的问题。当然，单机的问题是整个分布式系统的基础，为了构造一个完整的正常运行的分布式系统，读者需要多个方面的深入得知识。这方面的知识储备包括操作系统，计算机网络以及相关的算法与数据结构等。

下面，以分布式文件文件系统为例来阐述分布式系统中所需要展开讨论的研究课题。分布式文件系统可以说是单机文件系统的扩展，将文件系统架构在多个节点之上，而不是一台机器的内部。

建立分布式系统，满足功能需求

建立分布式系统的首要目的当然是需要完成一定的功能，提供一定的服务，满足一定的需求，然后才能够再谈其他方面的性质问题。

以分布式系统为例，分布式系统首先需要做的自然是如何有效构造一个文件系统，使得能够满足用户或者应用程序对于文件系统的需求。文件系统是在上层应用和底层磁盘中间的一层软件，用以将上层的应用以及磁盘块组织之间建立连接。文件系统的最本质的作用是用于地址定位，即将文件的数据定位到磁盘的对应的数据块的地址。

分布式文件系统实现的功能也是一个文件系统，因此也需要提供正常文件系统的大部分的功能。这包括两个部分，一个是对于上层应用或者用户的目录树的结构，然后是通过本地文件系统以及网络将数据保存到多个服务器中。分布式文件系统同样需要进行地址的定位，将具体的数据块通过名字空间定位到某一个节点的某一个文件。由于已经有了本地文件系统，定位到某一台机器之后，具体的磁盘位置可以交由本地文件系统负责。因此，分布式

文件系统中，包括了目录树名字空间的管理，数据空间的管理，以及提供用户访问的客户端等的3个模块。

分布式系统的扩展性问题

建立了分布式系统的功能是整个分布式系统的第一步，这一步是对单机系统的扩展。例如分布式文件系统就是对于单机文件系统的扩展。但是，分布式系统有其固有的特点，需要研究一些针对分布式系统的特有的问题，在建立功能的基础之上去解决这些问题。首先遇到的一个问题就是分布式系统中的性能问题。

还是以分布式文件系统为例，在提供文件读写的基础之上，现在需要考虑到整个文件系统运行在一个分布式的环境下面。这个环境中的节点的数目可能达到了数千个，这就有了一个最基本的性能要求，即所有的服务器都需要参与到数据存储以及数据服务的功能。简单的说，就是所有服务器都要承担起自己的责任，而不能让少数服务器承担责任，而其它服务器却空闲不工作。这样的特性通常被称为负载均衡，即所有的服务器上负载相类似，不存在某一部分的负载特别高，否则的话整个分布式文件系统的性能会急剧下降。针对这里的分布式文件系统的负载均衡，简化来说就有两个方面的要求，一个是数据存储的负载均衡，即所有服务器存储的数据量相当，避免某一部分服务器的存储空间用尽的时候另外一部分服务器的存储空间还十分空闲；另外一个是数据访问的负载均衡，因为不同数据块的访问热点情况是不同的，不能将所有的热点数据只集中在少数服务器中。这两点都需要在设计和实现分布式文件系统的过程中考虑到。

分布式系统的扩展性问题就是特别针对性能问题提出的，在分布式系统中性能随着系统规模的增大而增大。这其中最为重要的考虑就是上面所描述的系统负载均衡，在系统扩展的时候，仍然可以让所有的节点都负担类似的性能负载。

分布式系统的可靠性问题

分布式系统的另外一个特性是关于可靠性的讨论。从前面分布式系统的特征讨论中可以看出，分布式系统需要提供一定的可靠性，能够处理一整个完整的机器，一个机柜，甚至一个数据中心出现错误的情况。分布式系统的可靠性能够保证一定程度的错误容忍，并且为系统的可用性提供基础。

还是以分布式文件系统为例，在一个运行在数千个节点上的分布式文件系统不应当只是因为其中的一个或者两个节点出现错误，整个系统就不能继续工作了。因此，为了提供系统的可靠性，需要做大量的工作。首要的一个工作就是探测系统的模块出现错误的情况。这需要分布式的系统状态监测模块，能够在模块出现错误的时候，在一定的时间范围内通知相应的模块做出下一步的处理。然后是进行错误处理的工作，例如在文件系统中就需要处理名字空间出现丢失，数据块出现丢失的情况，使得用户在这个时候仍然可以继续对文件系统的内容进行读写。

处理可靠性问题的最基本方法是进行计算复制以及数据复制。例如在分布式文件系统中就可以通过数据副本复制的方式将数据的多个副本存放在不同的服务器中，甚至存放在不同的数据中心的位置。这样的话，在出现某一些错误的时候就有可能从其它的服务器中获得数据，继续提供服务。这种通过计算副本或者数据副本的方式是提供分布式可靠性的基础性手段，在许多分布式系统中都有所采用。在后面的章节中会再次详细研究这样的一个问题。但是，副本的方式本身会带来分布式系统的另一个特有的问题：系统的一致性问题。

分布式系统的一致性问题

分布式系统中的一致性问题会涉及到所涉及的分布式系统是否是正确的，或者在哪一些程度上被理解为正确的。例如，前面所说的分布式文件系

统会使用数据副本的方式建立系统的可靠性。这样的话，每一个数据块都会在这个分布式文件系统中有3个或者3个以上的副本（Google文件系统的设计）。对于这些数据副本来说，有一个最基本的要求就是这几个数据副本应当完全一致。否则的话，如果不同的客户端去读取同一个文件的相同偏移地址位置的数据的时候，获得的结果是不一样的。这样的话，从客户端观察的角度来说就会对整个系统产生困惑，不知道哪一个结果是正确可用的。

维持数据副本的完全一样是分布式系统的一致性的一个体现。不仅仅是在分布式系统中，任何其他的例如多核系统涉及到多个处理单元的情况，或者涉及到多个数据副本的情况都会带来系统的一致性问题。

一致性其实是涉及到对于系统的正确性的理解，甚至正确性的本身也是可以商榷的。例如维持副本的一致性的角度，一般是希望能够让副本完全一致，但是这会带来许多额外的开销，特别是在网络或者模块出现错误的时候是十分困难的。有一些应用系统可以维持副本在一段时间内的不一致，这个不一致可以返回给客户端，因为在某些情况下有一个回答要比没有回答好得多。前者可以被称为是一个比较严格的一致性，而后者可以被称为是一个放松的一致性。严格的一致性便于客户端理解整个系统的行为，但是维护起来非常麻烦，效率低。放松的一致性使得客户端可能出现困惑，但是维护负担低，能够提高系统的可用性。因此，从这个角度来看，一致性没有一个对错之分，也没有一个好坏之分，只能说依据所需要构建的系统来选取自己的一致性的模型。

系统的安全性问题

本书很少涉及分布式系统中的安全性问题，这是一个很大的课题，完全能够独立成书展开研究。本书关于安全性的部分的内容是分布式系统中的安全性的入门，关于这些方面的深入的知识读者可以参考其它的资料。关于安全的课题总结来说有三个方面的问题需要解决。安全性需要满足正确的人能

够被授权看到被允许看到的信息，其它未被授权的人不能够看到不该看到的信息。从分布式系统的两个组成模块来看，一个是节点内部的安全性，包括数据的访问控制以及数据的安全存储；另外一个是数据在网络中的安全传输。因此，安全性会包括三个方面的内容：1）访问控制，只有有权限的用户可以看到被授权的信息；2）密码学算法，这是数据传输以及数据存储的基础；3）系统的安全性，不能够被攻击者所控制，需要避免软件的漏洞，防止信息泄露。下面结合分布式文件系统的特征，对这些问题做一个简要的介绍。

1）访问控制问题：在分布式系统中，访问控制问题与单机的目录权限访问控制类似。在类似UNIX系统中，目录和文件都有一个简单的访问控制模型。在这个模型中，每一个文件或者目录具有三个权限位，分别是可读，可写和可执行（对于目录来说是列目录）。而每一个权限位都对应着具体的用户，或者组，或者其它用户的访问。在分布式文件系统中也同样需要提供类似的权限访问机制。对于访问控制的模型的话有基于角色的访问控制的模型（RBAC，Role Based Access Control）可以参考。

2）数据的加密解密：密码学算法方面的内容主要是用于数据加密以及身份认证。密码学算法与数学密切相关，对于现代分布式系统，特别是对具有关键作用的分布式系统例如银行系统等起到了极其重要的作用。因此，学一些密码学的知识对于理解分布式系统是极其重要的。现代密码学的发展逐步建立了一整套理论体系，对于分布式系统来说，必须要理解两类密码学算法的特性以及相关的应用。

密码学算法的其中之一被称为是对称密钥算法，在这类算法中，明文T通过密钥K以及加密算法A进行加密，获得了加密的密文C=AK(T)。在解密的时候，也是通过同样的解密算法获得，即T= AK(C)。由于加密密钥以及解密密钥是同一个密钥，因此这类算法被认为是对称加密算法。这类对称密钥的算法能够以很快的速度进行加密以及解密，通常用于数据存储的加密以及

数据在网络上的数据传输加密，并且不会对数据存储以及数据传输的速度带来过多的影响。

密码学算法中的另外一类算法被称为是非对称密钥算法。在这一类算法中，加密密钥与解密密钥是不同的，分别被称为是私钥（Ks）和公钥（Kp）。公钥和私钥被称为一个公私钥对，是与一个用户U紧密联系在一起的。用户U需要在一个非常安全的位置保存自己的私钥，而公钥则可以广泛的发送给需要进行通信的另一方。所有其它用户如果试图与用户U进行通信，那么首先需要取得用户的公钥，并通过公钥Kp以及非对称密钥算法R进行加密。C=RKp(T)。然后密文信息C就可以通过网络传输给用户U。由于只有用户U才拥有私钥Ks，密文信息C传输给任何其它的人是无法对此进行解密的。用户U获得信息C之后，可以通过自己的私钥进行解密T=RKs(C)，获得信息的明文。

需要注意的是，非对称密钥的加解密算法的复杂度要远远高于对称密码算法，但是普遍认为保密性能要更高一些。因此，非对称加解密算法并不会用来对大量数据进行加密保存或者加密传输，而是用来做一些小数据的加解密操作。

使用非对称密钥加解密算法能够很自然做一些特殊的操作，例如数字签名。数字签名的含义是让用户通过电子签名的方法将数据做上签名标记，就类似在纸质内容上进行签名。用户可以通过自己的私钥对内容的哈希值进行加密，加密过后的信息就作为这部分信息的签名。注意哈希算法以及内容都是所有人可见的，可以算出这个内容的哈希值。因为所有人都可以通过这个用户的公钥解密签名过的哈希值，通过比较之后就获知内容是否有一个有效的签名。而签名的用户也不能够进行抵赖，因为只有这个用户才拥有对应的私钥，才能够进行合法的签名。通过公私钥对能够建立很多很有意思的应用。

由于公私密钥加解密算法具有的良好特性，它被广泛应用于关键的一些应用，例如网上银行，网上购物，数字身份的签名和认证等。这些应用基本上都是分布式的应用，因此，理解计算机密码学对于理解实际的分布式系统是非常有意义的，建议读者可以阅读一些应用密码学的基础知识，以加深对本小节内容的讨论。

3）系统的安全性：对于分布式系统来说，安全性的另外一块非常大的内容是系统本身实现的安全性。前述的访问控制算法以及密码学上的算法都具有理论的指导意义，但是实际应用的时候必须要落实到代码上才能够真正工作。理论上的完善不一定带来实际工作系统的完善。这一点从计算机程序产生开始的时候就一直存在。由于现在的计算机系统包括分布式系统越来越复杂，各个组成模块也越来越多，系统中的模块难免会出现实现上的问题。例如，在C语言中，使用标准的C函数库可以导致缓冲区溢出的问题。特别是许多字符串操作函数都会有危险性，这就包括常见的strcpy，strcat，gets等的函数。这些函数由于本身函数定义的限制，没有对内部的缓冲区进行检查，因此会出现安全漏洞。许多程序员以及程序会依赖这些函数来完成一定的功能，因为这些函数使用起来也非常简单方便。例如在下面的程序中，使用了gets操作。

```
void main(){
  char buf[30];
  gets(buf);
}
```

这里的buf数组会存放在程序的栈中。不幸的是应用的程序的栈中还保存着函数运行的上下文，包括局部变量，函数返回地址，函数参数等。通过gets这部分的内容会非常有可能破坏栈上的内容，其中一个最重要的内容是函数的返回地址。如果返回地址被修改为攻击者注入的代码，通过ret指令就会直接执行注入的代码。如果此时的程序处在最高的管理员优先级执行，

那么攻击者就会获得管理员的权限，从而获得整个系统的控制。因此，这里的关键点在于程序员自己在写程序的时候不小心，选择函数不当，导致了应用程序出现了被攻击的可能。

图1.7一个典型的栈溢出攻击

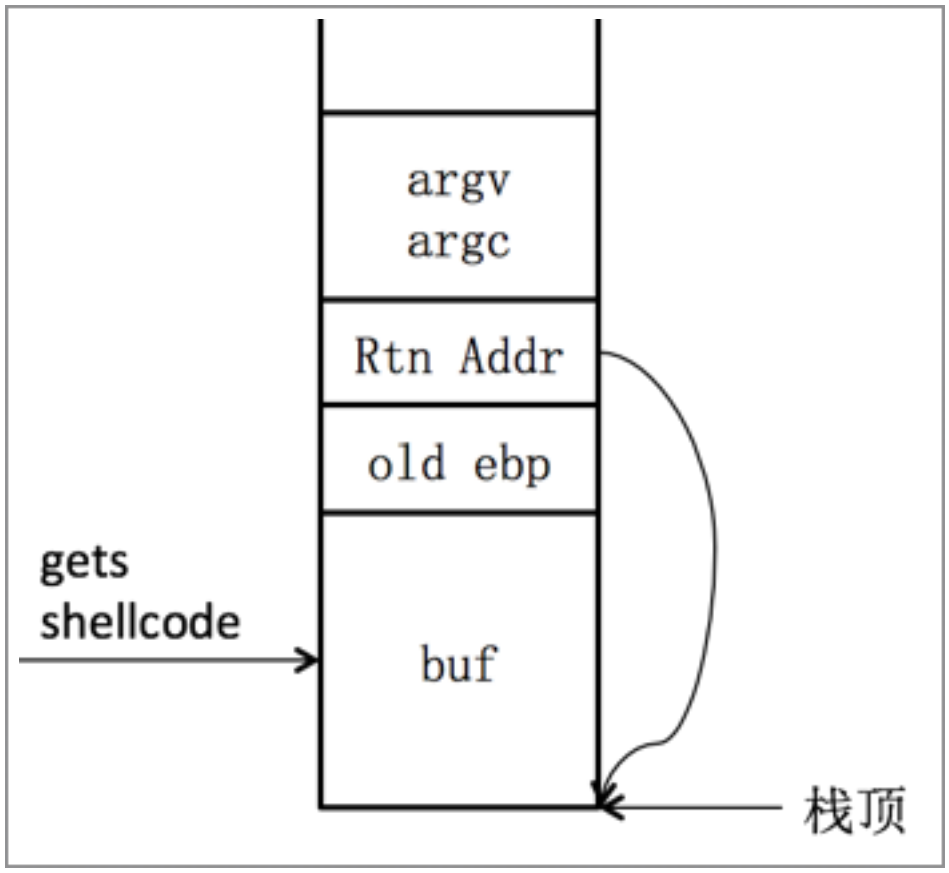


图1.7绘制了程序的堆栈以及注入式攻击的方法。可以看到，由于缓存区溢出的存在，攻击者可以在缓冲区中注入代码，并修改堆栈的返回地址，使得函数返回的时候去执行注入的代码。

上述的栈缓冲区溢出是最为简单的一种缓冲区溢出攻击的方式。随着标准库的改进以及旧的标准函数的废除，这种缓冲区溢出的攻击方式已经得到了极大的改善。但是，随着互联网络的发展，不断地会涌现出新的程序的漏洞，也为攻击提供了新的手段，例如堆缓冲区的溢出，脚本注入攻击，跨站点攻击等。分布式系统的安全特性与操作系统，网络等相关内容紧密相关。

由于系统安全性的分析非常复杂，在本书中将选择与分布式系统原理紧密相关的几个实例展现其中的几个问题，帮助读者理解系统安全所需要考虑

的一部分内容。要全面理解分布式系统的安全性以及相关的防护方法需要掌握大量的其它背景知识，在本书中不能够全面覆盖。

以上是分布式系统中的相关的研究课题。本书的重点是分布式系统的原理，因此会从这几个课题中选取典型的分布式系统进行剖析。本书的讨论重点是分布式系统的扩展性，可靠性以及一致性，这些对于理解一个分布式系统来说是关键性的内容。

本书内容概要

在介绍本书的内容组织之前，再次说明一个术语的用法。在本书中，一个处理单元可经常是一个服务器或者普通计算机，而服务器往往也被称为节点，因此在本书中，除非有特殊的说明，否则服务器，计算节点，节点，计算机，机器等词语是可以互换的名词，在不同的上下文环境中选择使用不同的名词。

本书的第一部分内容将以几个实际的分布式系统为例子，描述分布式系统在构造的时候的基本原理与方法。这部分内容包括如何构造分布式的存储系统，一个是分布式文件系统，一个是分布式的键值对存储以及分布式数据库。在这个基础之上是如何构造一个分布式的计算系统，包括消息传递模型，MapReduce模型等。之后也会讨论如何去构造在这些基础上的应用程序，一个典型的应用是如何建立迭代的运算PageRank，一个是如何支持数据库的查询。这一部分将研究的重点放在系统的可扩展性。

本书的第二部分描述的是分布式系统中的一个比较困难的概念，即系统的一致性。一致性模型在不同的场景下具有不同的表现形式，例如在多核处理器内部表现出来的是内存模型（ memory model ）即处理指令到达内存单元的顺序以及被各个处理器观察到的情况，在分布式共享内存中表现出来的是不同的节点上观察不同的内存单元的内容变化顺序的不同，而在分布式系统中则表现在对于不同的数据对象的操作顺序的不同。可以看到，一致性与操作顺序有密切的关系，对于多个操作来说，不同处理单元看到的情况也是不一样的。一个一致性模型允许出现的操作顺序越多，那么对应的一致性越弱，反之允许的可能性越少，一致性条件越强。这一部分的最初内容将讨论分布式系统的时间以及分布式系统的状态，这对于理解一致性以及之后的内容都是很有帮助的。

下面的一部分内容考虑系统的容错方法。在分布式系统中，随着系统模块地增多，越有可能出现模块失效的情况，这个时候就很有必要进行错误的处理。在分布式系统中，往往会有很多的处理单元，仅仅因为少数处理单元失效，就造成整个系统不可使用是不可接受的。这部分将先讨论一个重要的概念，即原子性操作。原子性操作对于系统的可恢复性具有比较重要的作用，将操作组织成原子性的操作的话就可以保证在系统中不会出现中间状态，使得系统可以方便恢复。讨论在单机环境下通过日志达到原子性操作，在多机环境下通过两阶段提交达到原子性操作。之后将通过具体的实例研究容错的方法，包括主副本备份，以及分布式环境下的一致协商协议（Consensus）。

分布式系统的安全性问题一直是一个重要的问题，在本书的最后一部分将集中讨论这一部分的问题。首先是延续上一部分的分布式环境下的一致性协商协议在出现恶意错误的情况下的处理办法，这就是著名的拜占庭问题。之后，为了隐藏在网络中的操作，需要分布式环境下的匿名特性，将研究如何在能够正常操作的同时隐藏自己的身份。

本书的绝大多数章节都通过一个具体的例子来说明相应的概念。大部分章节首先提出一个需要解决的问题，然后剖析这个问题的各个方面，最终通过构造方法来完成问题的解决。各个章节基本独立成章，因此读者可以依据自己的选择去阅读相应的章节。

分布式系统是一个很大的课题，不太容易在一本教材中就全面阐述所有的问题，本书将揭示其中一些比较重要的概念，算法以及应用实例，为读者能够继续深入打下基础。

思考题

这里是第一章，本章的思考题有许多内容可能需要读者了解相应的章节之后才能够有进一步深入的了解。这里的思考题（包括本书的其它章节的思考题）都不一定具有一个唯一的答案，目的是开拓读者的思维，加深对于分布式系统的理解。

1）各种性质举一个例子说明

参考文献

分布式算法，书

云计算论文

计算机网络

实用密码学

第二章

分布式系统的组成与编程模式

本章是本书中唯一具体涉及到具体编程细节的章节，本章的主要目的是帮助读者回顾一些与分布式系统相关的技术，包括计算机网络的基础，操作系统对于分布式系统的支持，分布式系统的编程方式等相关的内容。在本章中主要对这些的知识进行原理性的介绍，对相关的技术进行梳理。

分布式编程中程序员会涉及到很多的概念和方法，重要的两点内容实际上就是并行的编程的基本概念以及有关于网络编程的基本技术。这两方面的内容实际上都是非常的丰富，每一个方面的内容都完全可以写成一本教科书。本章无法对这两方面的内容展开详细深入的讨论，只能从原理的角度，对相应的知识点做一个解释与分析。这些相关方面的编程概念与方法在进行分布式编程的时候非常重要，在分布式的编程方面，对应的一些概念也是相通的。

本章将对一些具体的技术进行讲解，并通过一些具体的例子来进行进一步的说明。有些读者可能已经非常熟悉相关的内容，这部分读者可以将本章作为一个知识点的总结。对于一些不太熟悉相关内容的读者来说，本章的内容可以作为进一步深入相关内容的出发点。本章的内容扩展之后基本可以独立成书，对于实现实际的系统是必须的，因此建议不熟悉的读者可以参考相关的文献来深入学习这部分内容。

分布式系统的架构模块

每一个分布式系统的构成都离不开单机系统模块的支持。依据上一章的讨论分布式系统具有两个必不可少的模块，一个是单机的处理单元，一个是用以联系的网络通信。因此，分布式系统的架构模块也是由这两个部分组成。

在单机处理单元的角度来看，最重要的支持莫过于对于应用程序的支持，这一部分的平台软件就是操作系统。网络通信来说，则需要在多台机器之间架构硬件的互联网络，并通过软件进行数据通信。

图2.1 分布式应用程序的基本构成模块

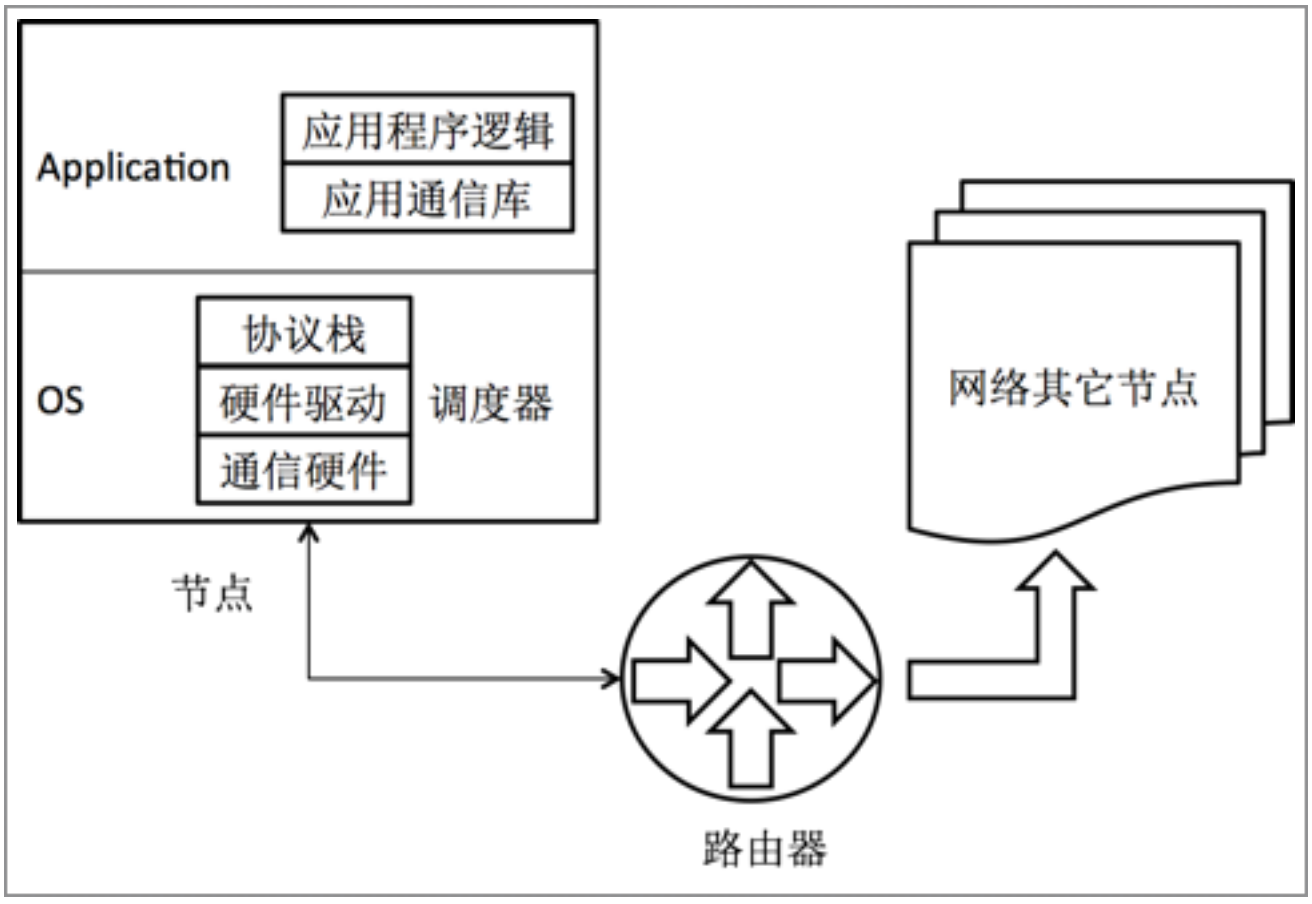


图2.1是分布式应用程序的构成模块。图中可以看出，在单机上的运行环境主要由操作系统提供支持，使用操作系统的调度器来调度程序的执行。单机的环境包括操作系统所提供的处理器分配，内存资源，存储资源，以及运

行时提供的函数执行环境，例如C语言运行时，Java语言虚拟机运行时等。这些实际上都是系统软件，共同构成程序在单机中的执行环境。

而对于网络模块来说，从单机的角度来看，最重要的事情是将网络数据包按照网络协议的形式包装，然后通过网卡模块发送到网络上。一旦数据包发送到网络上，那么此时网络系统将接管这一部分的数据包。分布式系统中的网络模块最重要的作用就是进行数据包的路由，依据数据包中标定的目标地址，将数据包正确传递到目的服务器。在本章的余下的部分内容将依据这一个分布式系统的架构模块展开讨论。

首先是有关计算机网络的基础知识，并指出其中对于理解分布式系统来说最重要的内容。这部分内容有可能在进行分布式编程的时候不会直接使用到，但是理解这部分内容对于理解整个分布式系统是如何运行的将起到比较关键的作用。对于网络知识的深入理解也是对于实际分布式构建起到关键的作用。

之后简要回顾操作系统中的进程与线程的概念，这是分布式程序的执行环境。这一部分将着重讲解有关于并行编程的基本概念，包括多进程编程，多线程编程，一些同步的技术等。这部分内容无论是针对单机多核的编程，还是在分布式下的编程都是一些关键性的概念，特别是对于多线程的同步处理是读者需要重点关注的。

本章还将对操作系统所提供的网络编程接口进行分析，帮助读者了解进行编程通信的时候需要使用的基础知识。随着现代操作系统的发展，为了能够尽量提高操作系统对于网络应用程序特别是服务器端的处理能力，操作系统内核也添加了与传统方式不同的编程接口，将对此做一个简要的介绍。

底层的网络编程接口能够带来比较高的通信效率，同时也带来了编程的复杂性，因此在实际系统中往往都是对这些底层的编程接口提供包装，方便程序员使用。这其中远过程调用是一项比较重要的方法，也是构造跨节点的

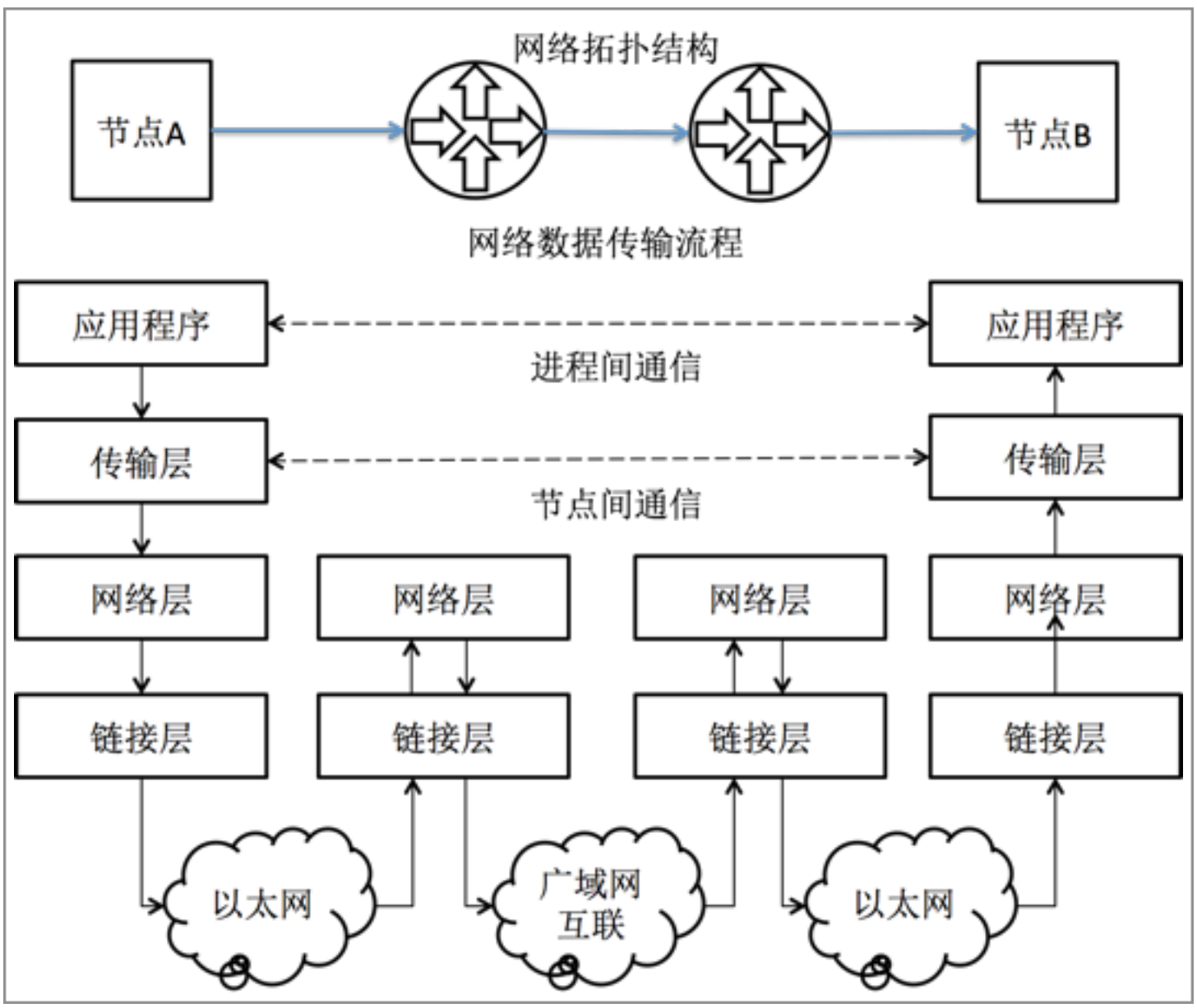
进程间通信的方法。远过程调用具有不同的实现方式，可以展开讨论看看如何实现远过程调用。

本章的最后将讨论分布式系统中的编程模型。在分布式系统中，两个主要的编程模型分别为共享内存编程模型以及消息传递模型。共享内存编程模型是当前单机编程的主要模式，是程序员熟悉的概念，对这个模型的理解能够构造有效地多线程程序。而消息传递模型是高性能计算程序的事实上的标准。这两个模型都是分布式系统中底层的模型，也需要对它们有所了解。

计算机网络回顾

本小节回顾一下计算机网络的基本组成与体系架构。读者可能都知道网络中的协议分层的情况，首先回顾一下协议的分层情况。图2.2是实际工作中的典型网络协议的分层情况，例如在以太网的基础上建立的网络传输结构就是这种结构。

图2.2 典型的网络协议分层结构



最底层当然是网络硬件层，负责将数据传输给下一个与之直接连接的另外一个网络端口。硬件之上是网络层的驱动，即媒体访问层（MAC），是

硬件与软件的接口。媒体访问层的主要工作是依据不同的物理硬件提供的访问接口去操作具体的网络硬件。在以太网中，媒体访问层还负责在一个局域网络中进行数据路由传输。在以太网中，每一个网络端口是有地址的，是一个5个字节的地址。因此，在一个以太网的数据包（几乎所有的网络都以数据包的形式进行数据传输）会填写源地址与目标地址，在以太网中的设备包括网桥和交换机就可以依据这个信息进行数据传输。从本质上来说，以太网仍然是一个广播性的网络，因此，每一个网口只需要依据自己的地址将数据接收过来即可。这种广播的方式使得以太网只能在一个局域网的环境中进行数据传输。

在媒体访问层之上是IP网协议层，即互联网络协议。这一层的主要工作是为了解决以太网层不能解决的问题，即如何在跨多个局域网之间传输数据。因此，这一层在以太网中封装了另外的数据包，即IP数据包。IP数据包仍然是数据包的形式，但是提供了一个全球统一的编址命名空间即IP地址空间。IP地址为4个字节。由于需要进行全球的网络传输，这一层的最重要的工作是进行网络路由，因此这一层最重要的设备是路由器。通过路由器，以及目标IP地址所提供的信息，网络层就能够把数据包路由到目标地址。实际上路由器工作在局域网的边缘，在数据包需要跨局域网的时候，就会去除MAC层地址，然后依据目标网络的访问情况，加入目标网络的媒体访问层数据包。

需要注意的是，无论媒体访问层以及网络层，都是按照数据包的形式进行数据传输的。但是实际上在网络应用中，大量的数据是依据一种流的形式进行数据传输的。一个典型的应用就是通过网络进行视频的播放。视频的播放可能数据量很大，并且以流媒体的形式存在。这个时候，就需要底层网络进行支持。另外一点是网络层的主要目的是将数据包进行路由，但是不会对数据包是否能够真正到达对方节点进行保证。因此也需要一种高层的协议去确认网络包的到达性，并尽可能的将数据包送达对方。这两个功能的保证就需要传输控制层TCP协议去完成。在TCP中的最重要的工作是将数据包的形

式变为数据流的形式，并保证可靠的数据传输。TCP依靠IP层提供不同端口（port）上的数据流，并完成数据保序，数据重传，流量控制等工作。端口的含义是可以在TCP层上区别不同的数据流，因为在两个服务器之间，不同的程序可以通过TCP建立连接，因此可以产生不同的数据流。为了区分这种不同的数据流，可以通过不同的端口进行区分。端口可以被认为是在进程内部进行网络数据的定位，占据2个字节的地址空间，因此最多可以提供65535个端口。

TCP/IP可以说是当前网络的基础，如果没有TCP/IP协议族，那么整个互联网是不可想象的。在TCP可靠传输之上，就可以建立不同的网络应用。例如HTTP，FTP等都是在TCP协议之上的应用层协议。

图2.3 HTTP协议的工作流程

以HTTP协议为例子，将网络数据传输以及应用层协议构成进行一个简要的总结。这里使用的工具是telnet客户端连接程序，这实际是一个最基本的TCP层客户端，几乎不会对数据进行任何上层的封装，因此可以通过telnet工具可以连接任何一个远端的服务器。

在图2.3中，在Linux下通过telnet工具与服务器 www.tsinghua.edu.cn 的80号端口进行连接。第一步的工作将进行域名解析即DNS（Domain Name Service）解析。这一部分的工作将 www.tsinghua.edu.cn 这样的域名翻译为166.111.4.100这样的IP地址。之后，网络数据包传输的时候就可以到达对方的服务器，即166.111.4.100这台服务器。在这台服务器上有一个网络服务器程序，开放的端口是80端口，由于采用的是HTTP协议，默认的端口号即为80。连接需要通过三次握手，完成之后就会显示 connected to www.d.tsinghua.edu.cn 。当显示这样的信息的时候，就表明TCP的连接建立完成。此时，服务器端会等待客户端输入的命令。在telnet客户端的命令行中键入HTTP协议的最基本的命令：GET / HTTP/1.0\n\n。注意，HTTP协议要求键入两个回车（\n）来完成一条命令，即需要一个空行。这个时候，TCP会将此命令传输给服务器，服务器进行解析之后，会知道客户端希望获得 / 目录下的一个默认的文件（可配置，例如index.html）。之后的内容是服务器传输给客户端的文本，这个文本分为两个部分，分别为HTTP的响应头部head以及响应体body，用一个空行隔开。这也是HTTP协议所规定的内容。而响应体中的内容将最终显示在客户的浏览器中（一般用浏览器去访问网络服务器中的内容），浏览器也会依据这里面的HTML文本做出相应的动作。这就是一次TCP操作的全过程。

上述内容是对整个计算机网络知识的回顾，当然是以一种非常简要的形式进行回顾。这部分内容对于构建以及理解实际的分布式系统是非常重要的。之后将回到这个主题，讨论在操作系统中如何对网络程序的构建进行支持。

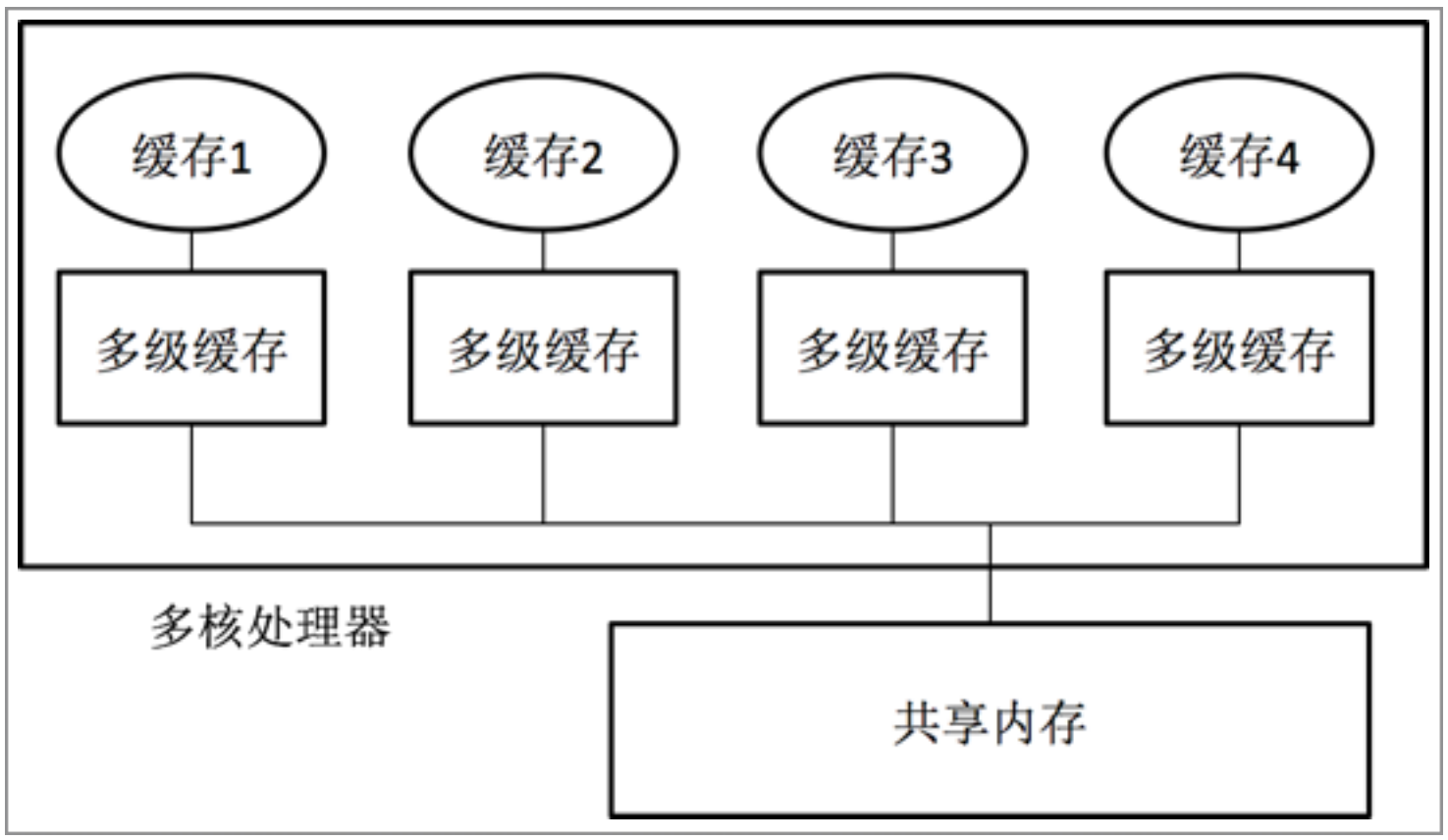
操作系统与多线程程序

操作系统对于编程重要的支持就是进程与线程，以及多进程多线程之间的通信方式。在本小节将对这些概念进行讨论。在读者的知识结构中，必须要建立进程，线程以及程序运行在操作系统环境中的一个清晰的概念。当然本书的主要目的不是进行编程讲解，不会涉及太多的具体的编程的技术，有需要的读者可以参考对应的编程手册。

多核处理器体系结构

首先看一下一个普遍的多核体系结构。并行编程的重要性现在无需多言，随着多核的发展，现在在处理器上集成了越来越多的处理器核心。这样

图2.4 共享内存的多核处理器体系结构



的话，实际上在一个单机系统当中也构成了一个分布式系统。在多核处理器上，并行执行的程序在物理机制上就能够保证多个执行线索可以真正并行执行，而不用跟传统的单核处理器一样，通过分时复用的方式使得多个线程能够轮流使用处理器。这种通过时间片方式实现的多个线程的并行执行实际上在真正物理处理器上是串行执行的，与并行执行有着本质的区别。当前的多核处理器结构可以用图2.4的方式进行说明，这是对称多核处理器结构的基本组成图。

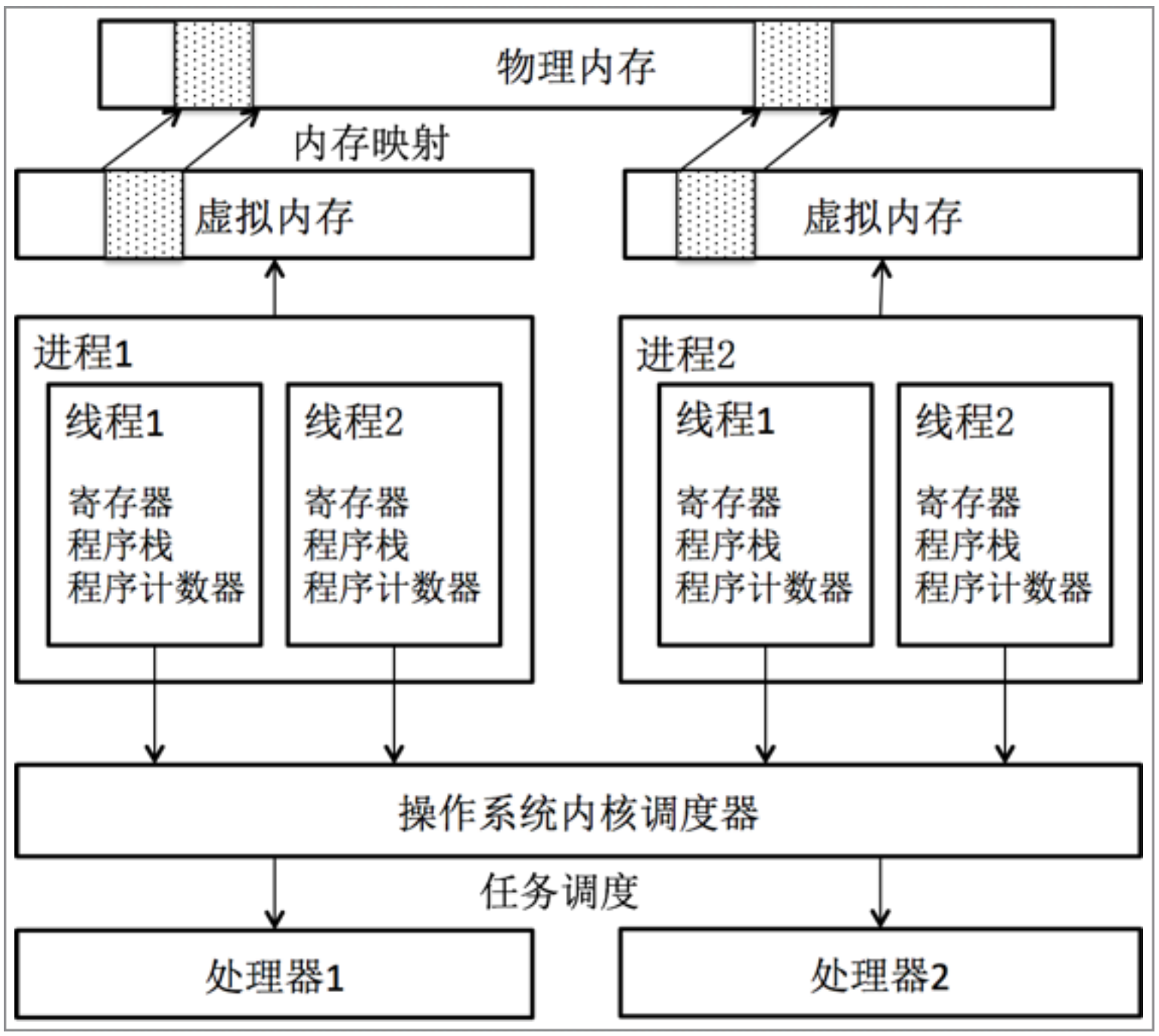
图2.4显示了一种共享内存的多核处理器体系结构。当然多核处理器的体系结构有多种形式，图中显示的是其中最基本的一种，也能够帮助读者去理解其它的多核处理器体系结构。在这个处理器中，多个处理器内核分别有自己的缓存结构，这个缓存结构有可能分为不同的层次结构，被称为多层次的缓存，现在普遍采用了L1，L2，L3三级缓存。缓存能够用来保存内存中常用的数据到处理器内部。然后多个处理器的内核通过前端总线连接并与内存连接。前端总线可以实现多核处理器多个核之间的互联，当然最重要的作用是访问内存，即从内存中获得数据或者将数据保存到内存中。

上图不仅仅是一个机器的模型，同时也是应用程序所看到的机器模型，也可以被称为是编程模型。从应用程序的角度来看，现在可以运行在多个核心的处理器上，并且也能够访问到相同的，共享的内存资源。这也是当前并行程序执行的基本模型。因此在开发应用的时候，就需要依据这样的一个机器模型完成。首先是需要充分利用多个处理器的计算资源，不能将计算只分配到一个处理器中，而不是多个处理器中。在操作系统的支持下，这需要应用程序通过多进程或者多线程的方式去实现多核之间的并行。进程之间的虚拟内存是互相隔离的，需要进程间通信机制，而线程则可以通过共享内存进行通信，这是这两种多核并行化方式的最大的区别。从这里的多核处理器的体系结构可以看出，这种结构是完全可以支持多个线程通过共享内存进行通信的，可以直接映射到硬件中完成。

进程与线程

通过上述的模型可以看到针对程序执行的一个最基本的概念就是进程和线程的概念。这两个概念在当代的任何一本操作系统的教科书中都可以得到详尽的描述以及相关的实现方法。对于从应用程序员的角度来看待进程和线程，必须要明白一个最基本的概念，即进程本身并不能够执行，它只是一个资源的容器。进程作为一个资源的概念实际上是线程被发展出来之后的事情，在操作系统中没有线程概念的时候，进程是作为基本的执行单元在操作系统中存在的。

图2.5 进程，线程，虚拟内存映射以及任务调度



资源容器的含义是在这个进程当中，所有计算资源，存储资源，以及网络通信资源等的集合。进程的典型重要资源包括创建的线程，这实际上是处理器的虚拟化抽象；包括虚拟内存的地址空间，这实际上是物理内存的虚拟化抽象；打开的文件系统中的文件以及通讯的描述符还有一些相关的外围设备的虚拟化抽象。值得注意的是，进程本身作为容器现在不再是执行的单元，操作系统所谓的进程调度也不再是调度进程而是去调度线程。线程是进程之内真正执行的部件，有一些原先针对进程的资源现在自然而然转到了线程中，其中最重要的就是内存中的执行栈。执行栈用来保存函数内部的临时数据以及函数之间的动态调用关系，因此需要针对每一个线程分别设置自己的执行栈。由于进程是一个资源的容器，而线程则是在进程所划定的范围内执行，因此线程之间可以共享进程的一些资源。进程与线程的关系可以从图2.5可以看到。

从图2.5可以看到，进程和线程共享同样的地址空间，能够访问到同样的内存以及相关的外设抽象资源。线程调度器则将线程调度到处理器上进行执行。不同的进程之间有不同的内存地址空间，它们之间是相互隔离的。随着前面所描述的多核处理器的发展，现在在一个中央处理器CPU的内部集成了越来越多的执行核。在这种情况下，即使是同一个进程内部的多个处理器之间现在也可以真正存在着并行性。

这样的话，即使是在单机的情况下，也可以存在不同形式的并行，也就存在着针对不同情况的并行编程技术。这些并行编程就是以进程以及线程的概念为基础，衍生出了多进程并行以及多线程并行的程序编程方法以及运行方式。在多进程并行上，基本方法就是通过多进程的方式去共同完成一件任务。由于进程概念在操作系统中实际上起到了隔离的作用，因此为了合作去共同完成同一件任务，多进程并行的编程需要更多的是进行进程间的通讯。而线程则不同，由于线程能够访问同样的地址空间，因此线程间通讯完全可以通过在同一个地址空间中的共享变量来完成，但是对共享变量的访问应该

有所限制，否则的话就可能产生冲突。因此，在线程上的并行编程更多考虑的情况是线程之间的同步与互斥。

并行编程在实际操作过程中一个重要的概念是同步。这是由于两个原因会产生在并行编程环境中的同步，一个是通过多个处理单元进行处理的时候，会发生去访问共享资源的情况；另外一个是并行编程打消了原有程序之间的先后顺序关系，为了恢复这种关系需要一定形式的同步方法。因此，从本质上来看，同步方式有两种，一种是对于共享资源的保护，被称为互斥访问；另外一种是在执行的过程中维持一定的顺序，通常被称为信号事件编程。在对共享资源的保护方面，最重要的内容实际就是对于资源如何通过保护的方式，使得在多个线程或者多个进程访问同一个资源的时候能够起到协调的作用。如果对共享的资源进行不同步的访问，则多个线程的访问会带来资源状态的不一致，会导致程序员无法预知的后果。而顺序性方面则主要是在多个处理单元之间，如果某一个处理单元内部发生了一件事情，则另外一个处理单元需要做相应的动作。所有的并行编程处理接口同步单元都围绕这两个问题构建。

由于多线程程序是在一个进程内部划分多个线程来实现并行性，并通过共享内存的方式进行通信，因此下面首先回顾一下多线程程序的编程基本概念，并解析同步中的技术。之后再讨论多进程之间的进程间通信方法。

多线程并行编程与线程间的同步

几乎所有的当代操作系统都支持通过多线程的方式实现单机处理的并行化。多线程并行因为需要切换较少的上下文，一般被认为比多进程并行化能够达到更高的效率。多线程程序在不同的平台上具有不同的表现形式，例如Linux下的pthread，Windows下的WinThread，Java中的Thread，以及C语言编程库OpenMP等。虽然各个编程库的形式不同，但都使用了类似的概念。

多线程程序在资源上是共享同一个进程的内存地址空间，但是有着不同的执行堆栈。多线程程序在共享信息方面是非常方便的，同前面所说的方式一样，直接在共享的内存地址空间中放置共享变量即可。因此，在多线程程序编写中重点内容是多个线程之间的互斥与同步，这就形成了多种互斥与同步的方式。可以统一称这些所有方法为多线程环境下的线程同步模式。总体来说，线程同步方式有两个大类：1）处于对共享资源的保护，需要通过互斥的方式，使得每次访问共享资源的线程只有一个，避免对资源一致性的破坏；2）多线程程序本质是无法定义顺序的，处于恢复顺序的考虑，需要使用同步方法来在一定程度上恢复顺序。

在针对共享资源的访问上，主要是需要对资源继续互斥访问，使得对于共享的资源进行访问的时候不会出现冲突，程序员就可以对资源的状态变化有一个合理的可控的预期。如果对共享资源不进行互斥，则共享的内容开放给多个线程进行并行访问，则程序员就很难对访问进行控制，从而在推断共享内容的值发生的变化方面非常困难。

使用互斥的方式能够保证共享资源在访问的时候最多有一个线程能够进行访问。这样的话，每个线程起码在访问对应资源的时候可以假设当前访问只有自己一个线程，不会有其它线程进入打扰。有这个保证后，线程能够以一种一致的方式访问共享资源，共享资源的修改是原子性的，不会出现中间结果等不一致的情况。当然，如果每次访问的时候都进行互斥，实际的效果就是针对一定的资源所有访问的过程是串行的，减少了并行性，这对于提高系统的性能是不利的。因此，在多线程编程的时候，需要仔细考虑共享资源的粒度。粒度太大，则每次都有可能去访问共享资源，都需要进行互斥操作，降低了并行性；而如果粒度太小，则带来编程上的困难，如果出现不当的互斥，还容易造成死锁，对于性能也不利。因此，互斥粒度的选择是一个需要值得仔细考虑的问题。另外，一个可以优化的地方是对共享资源的操作进行细分，这个可以在系统软件的层面就可以解决。一个可行的方案就是将读写进行区分，因为多个线程在进行并行读的时候，是可以真正同时进行

的，不会产生不一致的情况。在多线程的一些实现中，提供了读写锁的机制，能够在并行读的时候可以使用读锁来支持并行的读操作，避免了不必要的互斥操作，从而能够提高读的速度。

在多线程的程序中，另外一种同步的工具就是去定义两个线程之间的执行顺序。在这种情况下，某一个线程执行操作之后会通知其它线程进行下一步的操作，另外的线程则是等待某个线程的通知。值得注意的是，前面所说的互斥资源的方法并不能达到这个目的。互斥方法能够在共享资源上保证每次只有一个访问者，但是并不会对哪个线程先访问，哪个线程后访问作出限制。以Linux为例，同步的工具在线程中通过使用条件变量获取。条件变量可以使得某个线程等待条件变量，而另外一个线程可以通过条件变量发出通知。这样就在线程之间建立了顺序关系，是一种非常重要的同步工具。另外，一些进程间通信的方式也可以用来同步线程。例如，通过使用消息队列的方式可以在两个线程之间建立联系，一个线程A等待另外一个线程B的通知，而线程B可以通过消息队列发出消息，驱动线程A的执行。

下面就通过Linux下标准的pthread线程库的编程方法来说明上述两种线程直接的互斥与同步方法在实际编程代码中的使用。当然，进程间通信的方式也可以用来实现线程的互斥与同步，后面会举例说明进程间通信的方式。同样的，这里是一个简要的介绍，帮助读者回顾相关的编程技术，如果读者没有这方面的知识储备的话，可以参考其它的资料以及样例学习多线程的编程技术。

线程的创建与结束

线程的启动可以通过标准函数库的方式建立。需要注意的是，创建进程的时候会启动一个默认的执行线程，从C语言的角度来看，就是main函数开始执行的线程。这个线程往往被称为是主线程。其它的线程一般都是通过函数库来完成（OpenMP可以通过默认的环境变量建立）。

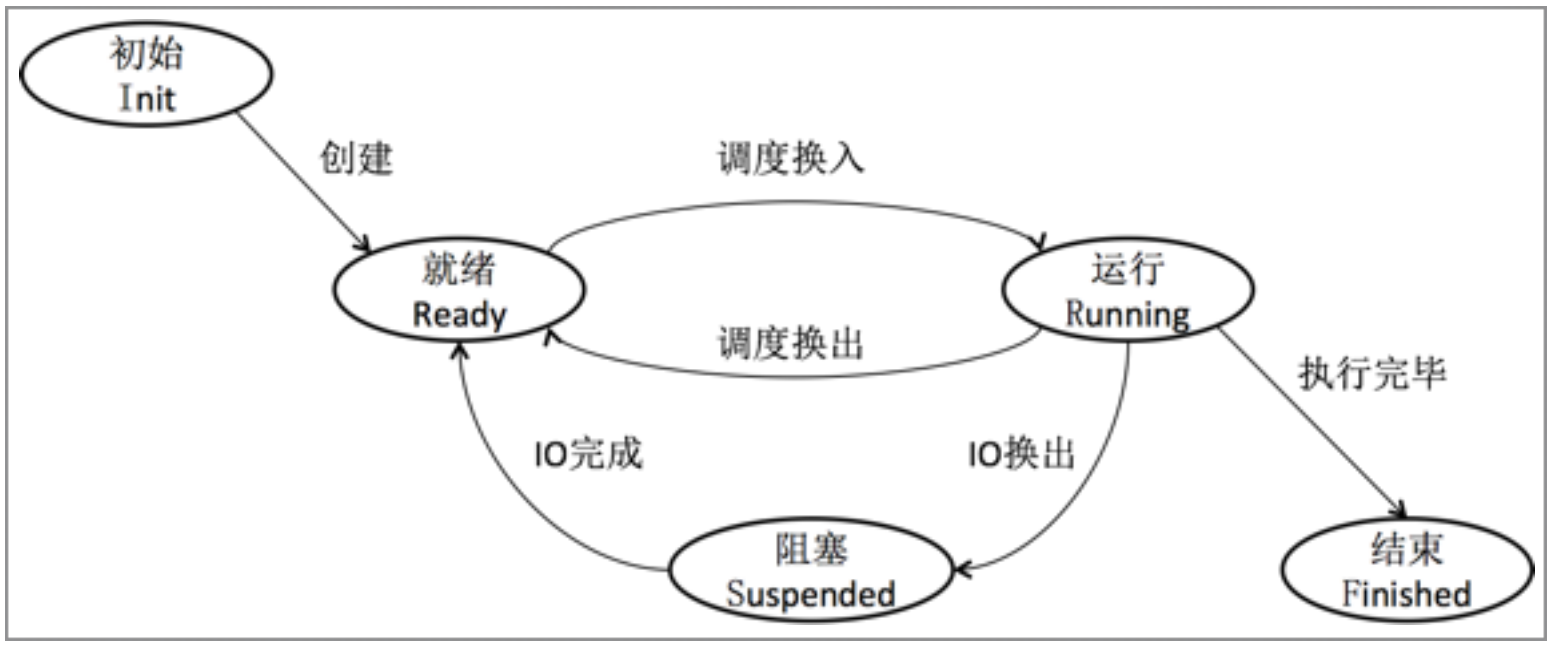
在pthread中的线程创建函数为

```
int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
```

可以看到在这个函数中，最重要的一个参数为void *(*start_routine)(void*)。这个参数实际上是一个函数指针。这个函数就是线程创建之后需要执行的入口函数。之后线程的执行都通过这个函数去调用其它函数去完成。

线程从创建的时候开始其生命周期，在结束之前会经历就绪（Ready），运行（Running），阻塞（Suspended）以及终止（Finished）等几个状态，可以从图2.6中观察到这样的一个执行流程。进程被创建完成后被放入到就

图2.6 线程的运行状态



绪队列中，就绪队列中的所有线程都是可以被调度执行的，至于何时被调度，这将依赖于调度器的算法实现。调度器会将线程调度到物理处理器中进行执行，使得线程处于运行状态。在执行的过程中，会不可避免出现两种情况从运行状态退出，一种是时间片用完，进程将被调度换出到就绪队列，等

待下一次被调度；另外一种是遇到IO操作，线程无法继续执行，将被IO换出，线程处于阻塞状态，等待条件满足后被放置到就绪队列。

在线程的入口函数执行完毕的时候，线程就进入到终止状态，作为线程最终的状态，不会再切换到其它的状态。线程也可以自己通过执行pthread_exit()来退出线程的执行，进入到终止状态。另外还有一个函数使得线程会进入到终止状态，即pthread_cancel()。这个函数可以将另外一个线程设置为终止状态。这两种方法都可以使得线程退出到终止状态，但是在编程的时候需要仔细设置，避免资源泄露。特别是pthread_cancel的方法是由一个线程退出另外一个线程，需要特别小心。因为这个函数只是将线程放置倒终止状态，操作系统可以回收诸如堆栈之类的内存资源，但是像打开的文件描述符之类的资源是不会被回收的。在进程结束的时候，这类资源会被回收，但是，如果这个线程还打开了一些系统的资源，那么这些资源对于操作系统来说无法判断是否还有进程需要使用，就无法进行回收。总而言之，操作系统在不同的上下文环境下对一些资源不能进行回收，造成了资源的泄露与浪费。

线程进入终止状态的时候，可以使用pthread_join获得线程的终止状态与返回值，其作用类似于进程之间的wait函数。

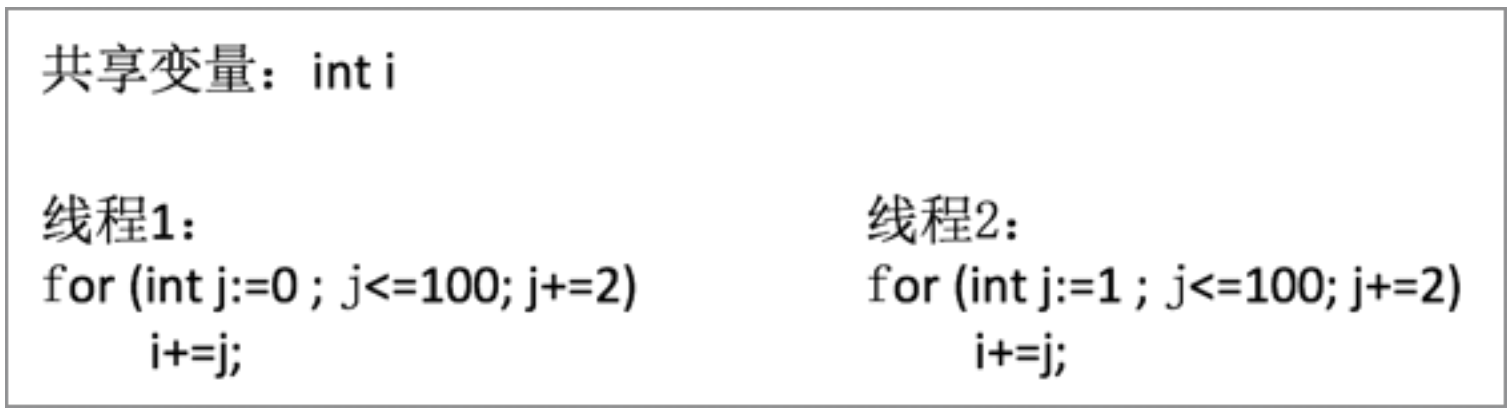
线程的互斥锁

前面已经讨论了线程同步的必要性，线程同步的第一种形式为线程的互斥锁，用以保护共享资源。通过互斥锁，使得在同一个时刻只有一个处理单元能够访问资源，从而避免并行访问带来的不一致问题。如果不使用互斥锁对资源进行互斥访问，那么程序的行为是不可预测的，并且可能不反映程序员真正的意图。

图2.7是非互斥访问的多线程程序的例子。从程序员的角度来看，这个多线程程序的本来的意图非常有可能是计算0+1+2+3+……+100的总和。最后

的总和会加入到global_count值中。虽然在C语言中，这里指使用了+号来表示一个加法操作，但是实际的程序中非常有可能通过多条指令完成，包括从：1）从内存中读取global_count的数值到一个寄存器；2）将一个i的值加入到这个寄存器中；3）将寄存器的值写入到内存中。两个线程在进行调度的时候，很有可能会在这三条指令的中间打断，这就会导致数据在计算的时候，不同的i值会加入到同样的一个global_count值中，然后在写入的时候其中一个结果会被忽略掉。在这样的情况下，某一个i值就会被丢失，使得最后的计算结果变小。并且，计算的结果在多次执行的过程中还会发生变化，依据调度的结果的不同而不同。

图2.7 非互斥的共享变量访问



为了避免上述的问题，通用的做法是对全局的共享变量global_count进行互斥加锁。在pthread编程中，这部分的编程接口支持为pthread中的互斥数据结构：pthread_mutex_t，对应的有对于这个数据结构的加锁与解锁操作：

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

因此，在编写程序的时候，需要为不同的共享资源设立不同的互斥锁，并在访问对应互斥锁的时候首先进行加锁，使用完毕的时候需要增加一个解锁的操作。所有的这些操作的基本形式如图2.8所示。

图2.8 共享资源的互斥访问

```
共享变量：int i

线程1：
for (int j:=0 ; j<=100; j+=2){
      pthread_mutex_lock(m);
      i+=j;
      pthread_mutex_unlock(m)
}

线程2：
for (int j:=1 ; j<=100; j+=2){
      pthread_mutex_lock(m);
      i+=j;
      pthread_mutex_unlock(m);
}
```

同步的死锁与活锁

多线程程序带来了数据的并行性，资源的互斥访问带来的是对资源的安全访问。前面已经说过，这两个方面在某种程度上是矛盾的，设置锁的粒度需要仔细考虑。另外，多线程程序在使用锁的时候必须要注意死锁的问题。

图2.9 多线程的死锁

```
线程1:                  线程2:
lock(A)                 lock(B)
lock(B)                 lock(A)
do_something()          do_something_else()
unlock(B)               unlock(A)
unlock(A)               unlock(B)
```

图2.9中可以看到，两个线程分别成功执行第一条语句后，分别获得了A和B两把锁。这个时候，每一个线程都希望等待对方的所，但是这是永远不可能满足的，因此两个线程都将不能继续运行，产生了死锁。为了避免死锁的问题，有一个通用的编程规则，即永远以相同的顺序执行加锁操作，并且在使用完成一个资源的时候需要立即释放对应的锁。当然，在加锁完成之后

执行代码块的时候，也需要尽量让对应的代码块执行时间短小，不能长时间拥有互斥锁，否则的话会阻止其它线程的执行。

相对于死锁的操作来说，一个更为微妙的情况是活锁的状态。图2.10是一个活锁的状态。

两个线程在进行加锁操作的时候，程序员有了一个基本的死锁的概念。

图2.10 互斥产生活锁的状态

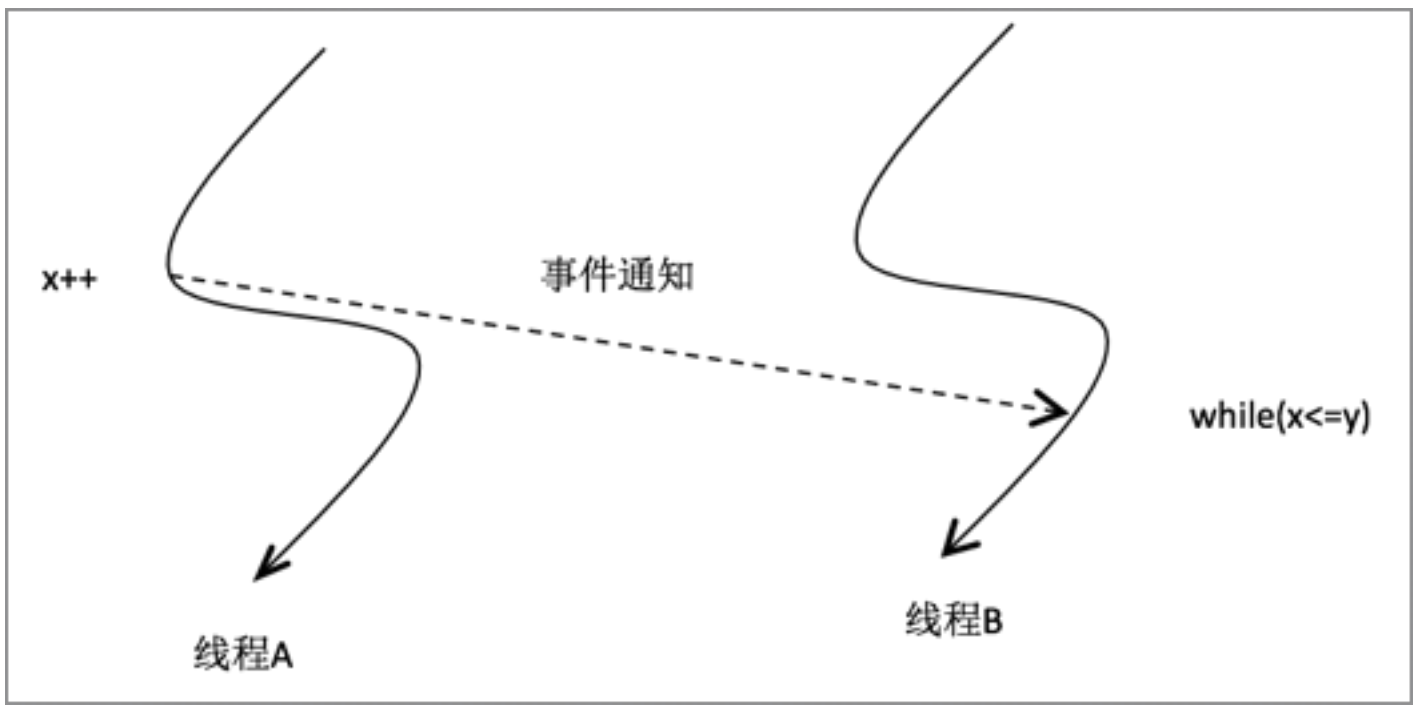
```
线程1：
try1:
        success=try_lock(A);
        if ! success then sleep(1); goto try1;
        success=try_lock(B);
        if ! success then unlock(A); sleep(1); goto try1;
do_something()
unlock(B)
unlock(A)

线程2：
try2:
        success=try_lock(B);
        if ! success then sleep(1); goto try2;
        success=try_lock(A);
        if ! success then unlock(B); sleep(1); goto try2;
do_something()
unlock(B)
unlock(A)
```

程序员首先去获得一把锁，并且在不能获得另外一个锁的时候去释放第一个锁。这两个线程都尽量避免去因为自己的原因去产生死锁。可是，不幸的是在一些特定的条件下，两个线程同时执行加锁，同时获取第二把锁失败，同时执行释放已经获得的锁，然后继续不断执行这样的流程，产生死循环。这种两个线程都继续执行，但是执行的都是无效的工作的状态被称为活锁。活

锁的状态比死锁的状态更加的微妙，因为在系统执行的过程中可能在绝大多数情况下都良好执行，只有在特定的顺序中会出现活锁，并且不同的执行需要的执行时间也不一样，非常难于调试。活锁状态也是进行多线程编程的过程中需要避免的状态。同样的，基于锁产生的活锁状态可以通过前面的加锁原则进行，但是如果是其它的资源访问工作产生的活锁则需要仔细分析去避免活锁的状态。

在分布式系统中，同样会存在共享的资源，包括共享的存储资源以及计算资源，同样需要进行互斥的操作。在分布式系统中，通常也会通过一台互斥锁服务器来完成对资源的互斥操作，其本质的操作与上述的多线程类似，因此也需要避免死锁以及活锁的操作。

图2.11 条件变量与线程之间的通知机制



条件变量

同步的第二种形式是对被打破的顺序特性重新的获得，是在并行执行中确定事件处理的顺序。并行本质上是无法确定时序的，通过同步的机制能够规定多个线程的执行顺序，满足程序中对于事件顺序的语义。这种定义事件

同步顺序的一般表现形式是当一个线程中出现某个事件的时候，会触发另外一个线程的某些工作。在具体的实现上，操作系统或者运行时库会选择不同的表现形式，例如在pthread中体现为条件变量，而在Windows操作系统中，给出的编程接口为。

通过条件变量在线程之间实现通知机制可以参见图2.11。在图2.11中，线程A不断增加x的值，并通知线程B，线程B得到通知后，检查x是否增加到足够的值满足x>y，才能够进行下一步的工作。在这里，x>y可以被称之为一个条件。使用pthread中的条件变量实现上述通知机制的程序图2.12中所示。

图2.12 条件变量在pthread中的使用

```
线程A:
lock(x_lock);
x++
pthread_cond_signal(&cnd);
unlock(x_lock);

线程B:
lock(x_lock);
while(x<=y)
      pthread_cond_wait(&cnd,&x_lock);
unlock(x_lock);
```

从图2.12中可以看出，由于x仍然为共享变量，因此需要一个互斥锁mutex进行保护。在线程A中会不断提升x的值，并行出信号pthread_cond_signal。在线程B中需要进行条件判断x<=y，并且在这个条件下进行等待pthread_cond_wait。值得注意的是，pthread_cond_wait是在获得锁的时候进行的函数调用，因此在这条语句的位置实际上线程已经获得了锁，其它的线程在原则上是不能获得这个锁的。因此，pthread_cond_wait的函数调用的方式是在调用的时候释放这把锁，并且在函数调用返回之前重新获得这把锁。

以上是进行多线程程序开发的所必备的基础知识要点，当然在实际的工作中还有其它的同步工具可以使用，这就包括了临界区（critical section），信号量（semaphore），管程（monitor）以及同步屏障（barrier）等各种方式。这些方式在实质上还是离不开线程同步的两个基本目的，即资源的互斥保护以及操作的顺序性，这也是多线程并行带来的与单线程程序本质的不同。这些同步的手段会应用到不同的场景中，以到达对应场景的不同的目的。在分布式系统中，由于有了多机环境，因此进行资源的同步也会更加困难一点。同步特别是保持顺序性的同步方式可以通过进程间通信的方式获得，因为进程间通信的方式可以用来传递数据，从而也定义了数据的传输方向，也定义了在这个方向上的两个进程的操作顺序。例如套接字 socket, UNIX域下的套接字，以及FIFO队列, 管道pipe，在进行数据传输的同时都可以用以定义操作的顺序。下面是对于进程间通信的讨论。

进程间通信

前面说过，基于进程的并行编程方式是传统的利用多个处理器进行并行处理的运行方式。在基于进程的并行编程中，程序员需要将任务分配给多个进程，并通过进程间的通讯来共同完成一个计算任务。前面已经说过，进程在操作系统之间是天然隔离的，因此在分配任务的时候，各个进程可以自己完成自己部分的工作，不会对其它进程造成干扰（至少在操作上是这样的，实际中性能上则完全有可能相互影响）。进程并行中需要考虑的因素是进程之间如何进行合作，实际上就是在各个进程之间进行通信的问题。进程间通信的主要目的是建立在多个进程之间的关系，包括数据传递以及事件的通知等。因此，在传统的操作系统编程方面，进程间通信是一项重要的内容。另外，由于进程之间可能存在的对于操作系统共享资源的访问，进程间通信也包含了互斥的操作。

在Linux环境下进程间通信的方法主要包含管道，信号，消息队列，共享内存，信号量以及套接口。实际上套接口已经属于网络编程的相关内容，但

这也是进程间通信的重要方式，将在后面的内容中介绍。在上述的几种进程间通信的方式中，信号以及信号量是属于同步的手段，并没有对应的数据在进程之间进行传递。其它的各种方式都能够在多个进程之间进行数据的传递。在Windows操作系统中，进程间通信的形式则更为复杂。下面就以Linux为例子，说明各种进程间通信的主要方式。在各种开源软件中都可以找到通过进程间通信进行工作的方式，了解其中基本的进程间通信是非常有必要的。

管道

管道一般用以在多个进程之间创建传输的机制，管道用以传输数据即按照先进先出的顺序使得数据在管道中进行流动。在Linux中能够通过pipe系统调用的方式在一个进程，或者是在两个有父子关系的进程之间建立数据传输的通道。管道有两种方式，一种是无名管道，一种是命名管道。命名管道除了在文件系统中占用一个文件资源之外，其使用方法与无名管道类似。下面给出一个使用无名管道的例子，而命名管道只需要使用mkfifo的系统调用，并给出一个文件名字即可。（示例代码，只留下主要部分，未进行出错处理。）

```
int mypipe[2];
pipe(mypipe);//获得一对管道描述符
if(fork()==0) {
  close(mypipe[0]);//子进程关闭管道的一个出入口
  read(mypipe[1], buf, 10);//读出父进程写入的信息
}
else{
  close(mypipe[1]);//父进程关闭管道的一端
```

```
    write(mypipe[0],buf,10);//写入另外一端
}
```

需要注意的是，与全双工工作的套接字不同，管道是单向的，即只能从mypipe[1]输出，从mypipe[0]输入，上面的程序就是这么工作的。在程序中，将一个缓冲区的字符串写入到管道的一端mypipe[0]，并且从另外一端mypipe[0]读出数据并打印。这里的例子使用了fork的系统调用，形成了两个进程，并在两个进程之间完成进程间通信。

管道在Linux系统中有非常重要的作用，Linux的shell程序使用管道作为将多个进程之间的输入输出进行串联的工具。shell将为多个通过管道进行通信的程序安排合适的管道，然后生成对应的进行去执行相应的程序，实现在多个进程之间的通信。shell使用的管道系统调用形式基本上与上面的程序类似。

管道需要建立在两个互相之间有联系的进程之间，因此不能够直接在分布式环境下使用管道。但是与管道类似方式的进行数据传输则非常普遍，例如网络套接字socket，其本质上的传输方式就是管道。从一个套接字写入的数据可以从另外一个节点的对应的套接字中读取。

信号

信号是整个类unix系统非常重要的进程间通信的工具，在整个操作系统的各个部分都使用信号来控制各个进程的工作。可以说信号在整个系统中无处不在。一个非常普遍的例子就是用户在控制台上可以使用crtl+c给前台进程发送终止执行的信号，接收到这个信号的进程在通常情况下会由操作系统退出执行。另外一个普遍的例子就是在每个进程退出的时候，都会给自己的父进程发送子进程退出的信号，这样的话，父进程会得到通知，知道子进程已经执行完毕，可以做一些后续的处理。这些都是在操作系统中经常使用的信号进行进程控制的方法。信号用来在不同的进程之间进行执行控制，本身不

适合于进行数据传递，因为除了信号的本身，在函数调用的接口中并没有附带数据的参数。

最基本的信号机制在两个进程直接传递信号，一个进程发出信号，另外一个进程可以接收信号。在操作系统中，信号不是随意的，操作系统已经规定了一些预先定义的操作系统信号，当然也留下了用户自定义信号的接口。信号的编程涉及到两个进程，发送进程调用kill函数向接收进程发送信号，而接收进程需要编写信号处理函数，在接收到信号的时候产生调用。针对每一个信号类型，如果进程没有定义信号处理函数，操作系统会自动调用默认的信号处理函数，一般就是直接结束进程。下面是一个进程对于control+c信号即SIGINT信号进行处理的方式（注意，未进行出错处理）。

```
void sig_handler(int signo){
  if(signo==SIGINT)
        printf(“control+c received.\n”);
}

int main(int argc, char** argv){
  signal(SIGINT, sig_handler);
  sleep(100);
  return 0;
}
```

在这个例子中，进程首先安装一个信号处理函数，即打印信息，然后就进入到睡眠状态。如果在这个时候用户按下control+c的话，就会触发信号，即向进程发送SIGINT信号。当进程捕获这个信号后，就不再使用默认的信号处理函数退出进程，而是打印出一条消息。

值得注意的是，用户进程一般不会主动调用信号处理函数。从这个意义上说，信号处理函数是进程提供给操作系统的回调函数，操作系统在一定条件下会主动调用这个函数。操作系统调用信号处理函数的时机也是一个值得注意的问题。进程在正常执行期间是不会被打断的，因为这个时候是用户层的代码在处理器中执行，并没有去检查发给自己的信号是否存在。只有在操作系统进行调度的时候，操作系统在唤起某一个进程之前，会检查信号是否存在。如果信号存在，操作系统会将信号处理函数的入口地址作为跳回到用户层的代码的入口；如果没有信号，则返回调度之前的现场。从上面的讨论可以看出，对于信号处理函数的调用时机是不确定的，依赖于调度器以及当前系统的状态，程序员不能对调度做出不合理的预期假设，即调度依赖于调度器的算法，不能预期信号接收程序在什么时间点能够进行信号处理函数的执行。

在分布式环境中，信号一般不会被直接使用，因为信号是操作系统内部的提供的机制，不会被提供为跨节点的调用。

信号量

信号量是unix系统中比较重要的进程间通信机制，能够用来同步一个资源有多个副本可用的情况。信号量是操作系统中一个重要的概念，针对资源访问的情况而定，特别适合于对于多个资源副本进行访问的情况。信号量可以被认为是一个整数S，对这个整数的操作是具有原子性的，在线程中实际可以通过互斥锁和条件变量实现。信号量具有两个操作原语，分别为P和V，其中V负责将某一个数值加入到这个整数中，而P则是从这个数值中扣除一个整数，当然这个扣除是有条件的，即此时数值不应当小于0。

function V(semaphore S, integer I):

[S ← S + I]

```
function P(semaphore S, integer I):
    repeat:
        [if S >= 0:
            S ← S - I
            break]
```

上述的定义摘自wiki。当然信号量还有其它的变形，其中一个变形就是将整数I置为常数1，即表示每次产生一个资源或者消耗一个资源。并且将条件修改为S>0，表示只有存在资源的时候才能够消耗资源。

信号量的一个典型的应用例子是生产者与消费者的例子。生产者产生数据，而消费者消耗数据，数据队列则是他们之间的共享资源。在这里，如果

图2.13 通过信号量的方式完成生产者与消费者问题

```
semaphore lock=1;
semaphore data_slots=0;
semaphore empty_slots = TOTAL_SIZE;
data data_buffer[TOTAL_SIZE];

void producer(){
    while(1){
        data = produce_data();
        down(empty_slots);
        down(lock);
        put_data(data);
        up(lock);
        up(data_slots);
    }
}
```

```
void consumer(){
    while(1){
        down(data_slots);
        down(lock);
        data=get_data();
        up(lock);
        up(empty_slots);
        consume(data);
    }
}
```

队列中的数据容量已经满了，则不能再往队列中插入数据，生产者必须等待。当然，如果队列中的数据为空，那么消费者也不能继续消耗数据，也必

须等待。生产者和消费者的代码如图2.13所示。在这里，常数被设置为1，代表了每次可以产生一个数据对象或者消耗一个数据对象。

图2.13是通过信号量解决生产者和消费者问题的一个示例，在这里lock被用户保护一个共享的数据队列data_buffer。信号量empty_slots用以指示数据队列中空的数据项数目，而data_slots用以指示在数据队列中已经有的数据项数目。信号量实际是互斥锁的一种扩展方式，这在上述的例子中也有体现（即lock）。在互斥锁中，信号量的资源数目设为1，而一般信号量的资源数目则可以超过1。信号量在分布式环境中没有对应的编程模型，一般是通过分布式环境下的互斥锁去管理共享资源。

消息队列

消息队列的概念非常简单，即进程将消息发送到操作系统所提供的一个队列中，另外一个进程则可以通过这个消息队列去获得里面的消息。消息队列建立了一个FIFO的管道，但与pipe不同，消息队列不需要进程之间先有自己的联系，消息队列是完全独立于进程存在的。许多的进程间通信方式都是独立于进程存在的，例如信号量，消息队列以及共享内存等。这些进程间通信的方式在文件系统的名字空间占有一个名字，使得进程可以通过某种方式找到这些进程间通信的对象。独立于进程的含义是即使需要消息队列的进程不存在，消息队列仍然可以用来保存消息，并等待消息的接受者在将来的某一个时间点启动去接收消息。

在Linux环境中使用消息队列是非常容易的事情，可以看一下消息队列的一些接口函数，就很容易明白在实际程序中如何使用。当然，如果需要具体编程的话需要查看消息队列的编程手册。

消息队列的创建，打开和关闭相关的函数，使用方法与文件类似：

mqd_t mq_open(const char *name, int flags, ... [mode_t mode, struct mq_attr *mq_attr])

int mq_close(mdq_t mqdes)

int mq_unlink(const char *name)

可以看到，这里mdq_t实际上就代表了一个消息队列。下面的函数向消息队列中发送一条消息：

int mq_send(mqd_t mqdes, const char *msgbuf, size_t len, unsigned int prio)

发送消息的缓存以及大小分别为msgbuf以及len，并具备一个优先级prio，可以看到，消息队列实际是一个优先级队列。下面的函数则从消息队列中获取一条消息：

ssize_t mq_receive(mqd_t mqdes, char *buf, size_t len, unsigned *prio)。

这个函数将从消息队列中最高的优先级中获得一条最老的消息，即每一个优先级队列实际上是一个先进先出FIFO的队列。

消息队列也是分布式环境中的进程间通信的重要方式。对于套接字来说，通信的进程双方必须都存在并且正常执行。而消息队列则无需这样的需求，因此可以建立独立的消息队列管理程序，接收消息，并异步地为消息的目的进程提供消息。有许多分布式系统的基础构建中都有消息队列的模块。分布式环境下的消息队列机制可以提供类似的编程接口。

共享内存

共享内存是标准的进程间通信进行数据传递的最快的方式。共享内存的方法就是在两个进程直接建立内存联系，这样的话，共享内存的两个进程可以直接通过内存传递数据，免除了通过内核的来回数据复制的工作。

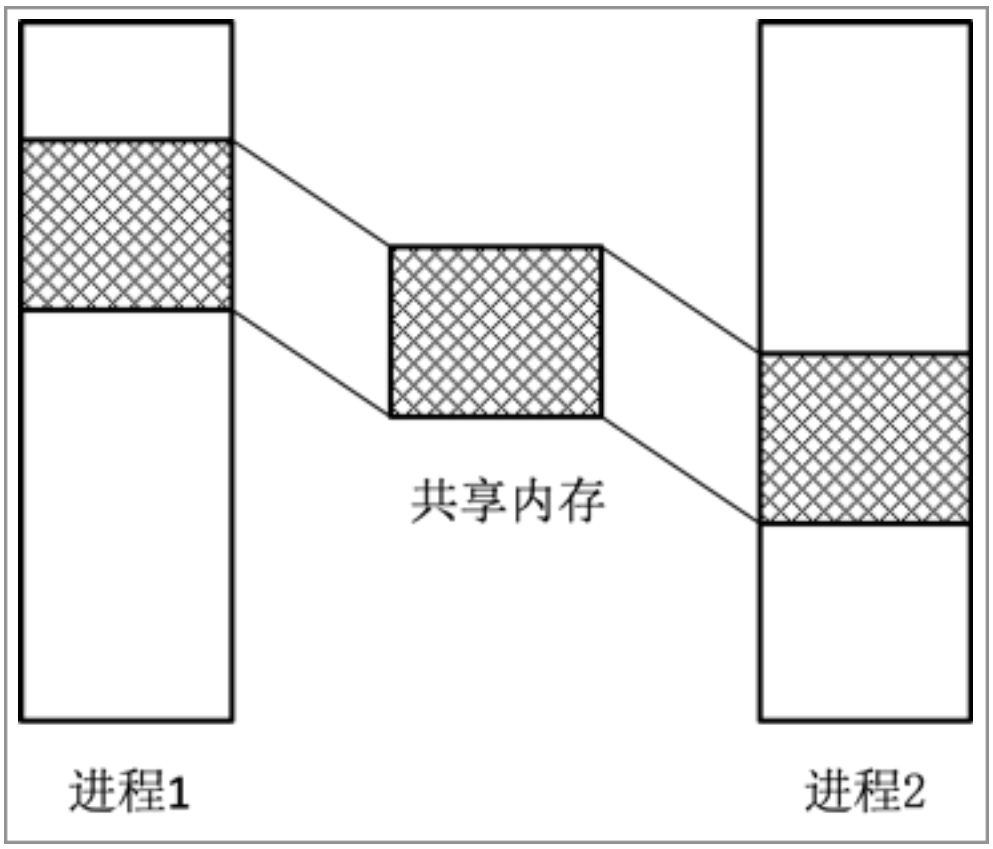
从操作系统管理的角度来说，进程是一个资源的容器，在操作系统中进程应该是严格隔离的，所以就有了虚拟内存的概念。两个进程的虚拟内存之间是独立的，相互之间没有任何的联系。在两个进程对同一个虚拟内存进行访问的时候，实际上是访问的不同的内存区域，不会产生干扰。在操作系统

内部，一般通过处理器内部内存管理器硬件的帮助，设置映射表的方法，将虚拟内存映射到物理内存。

共享内存的实现方法就需要涉及到操作系统的内存管理。操作系统控制着物理硬件，可以自己选择映射的方式，如果将同一块内存区域映射到两个不同的进程，就会形成两个进程各自访问自己的虚拟内存，而实际上访问的物理内存是同样的地址，这就形成了共享内存的机制。

操作系统将这样的功能按照编程接口的方式提供给程序员使用，使得两个进程之间可以自己建立共享内存的联系，进行进程间数据传递。从编程接

图2.14 同一块物理内存映射到不同进程的虚拟地址空间，形成共享内存



口上来说，也是通过内存映射的语义将操作系统分配的内存资源映射到进程自身的虚拟内存中。因此，共享内存的一般过程是先去申请一块内存资源，然后将这块内存资源映射到自己的虚拟内存空间的一段虚拟地址中。另外一个进程需要做同样的动作，将同样的一块内存映射到自己的虚拟内存地址空间中。这样的话，两块虚拟地址空间（属于不同的进程）实际被映射到了同

样的一块物理内存中，一个进程内部对这段内存的写入操作会被另外一个进程所读取。

图2.14是共享内存的概念图，同一块内存会被映射到两个进程中，并可以处于不同的虚拟内存位置，而实际的物理内存是同一个。因此，共享内存的编程接口包括了内存映射的内容。在System V中的共享内存映射的方式是通过shmat的方式进行映射，而POSIX共享内存的方式则直接通过内存映射文件mmap的方式获得。这两种方式都被绝大多数操作系统所支持，后面那种方式能够映射的内存会更大一些，因为其本身是对文件的映射。下面一段代码说明了如何使用System V的内存映射方式：

```
#define SHMSIZE     1024
main(){
    int shmid;
    key_t key = 1234;
    char *shm;
    shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666);
    shm = shmat(shmid, NULL, 0);
}
```

即通过shmget获得一段内存的访问，通过shmat（shared memory attach）进行内存的映射。另外一个进程需要同样的方式建立共享内存，注意要使用同样的key=1234，这样就可以对同一块内存进行读写。内存使用完毕后，可以使用shmdt（shared memory detach）来取消映射。

下面的一段代码说明了如何通过mmap进行内存共享：

```
int fd;
caddr_t result;
fd = open(“/home/chenkang/mysharedmemory”,O_RDWR));
```

result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

另外一个进程只需要通过同一个文件进行映射即可。当然，映射完成之后需要用close(fd)来关闭文件。如果不需要将文件进行保存，那么可以映射一些特殊的设备文件，例如/dev/zero。

共享内存的概念其实是非常容易理解的，单机内部实现共享内存的方式也非常直接。但是，在分布式环境中每一个机器都有自己独立的内存，这些内存从物理上就不是共享的，并且属于不同的操作系统实例去管理。在这样的模式下提供共享内存就比较困难了，因此在分布式环境中一般不使用共享内存的方式去提供多个处理单元的数据通信。

但是，在分布式环境中不使用共享内存的原因并不在于其实现的困难性，而在与性能以及扩展性方面的原因。实际上，由于共享内存概念的简便性，有很多人研究了如何在多个机器上提供共享内存的环境，即分布式共享内存（DSM：Distributed Shared Memory）。本书将使用几章的内容来讨论分布式共享内存。但是，这样的实现方式采用的网络无论是延迟还是带宽都远远无法与前端总线相比，而且在程序无法获知底层细节的情况下（这是提供分布式共享内存的出发点），性能上就非常难于优化。

以上就是对于多线程并行程序模型以及多进程并行程序模型的讨论。这部分内容是单机操作系统所提供的应用程序支持与运行环境，与建立分布式系统密切相关。当然，操作系统对分布式最直接的支持自然是对网络编程接口的支持，这是下一节讨论的重点。

操作系统对于网络编程的支持

网络的套接字编程接口

先回顾一下TCP层的网络四元组，即<源地址, 源端口, 目标地址, 目标端口>，其中地址是4个字节，而端口是2个字节。源地址和目标地址用以标识在网络中的两台需要通信的机器，而端口的组合则唯一标识了在对应的机器中的某一个特定的进程，这个特定的进程需要负责处理在这个TCP连接中所需要处理的数据。

套接字接口（Sockets）是当前网络通信的标准模式，首先被实现到类似UNIX的环境中，后来也被移植到完全不同体系的Windows系统中，因此通过套接字编程能够被编译运行到当前几乎所有的操作系统中。在编程语言上，各个编程语言都通过自己的方式对操作系统提供的套接字进行封装，提供给不同的程序员使用。

网络的套接字接口分为两个部分，一部分是服务器端的接口，另外一部分是客户端的接口。分为两个部分的原因也很简单，因为TCP协议本身也是依照这种方式执行的。客户端发起连接，而服务器端在接收连接。在网络程序中，服务器端的程序一般是去被动等待，因此需要服务器端的程序在一开始启动的时候进入一种监听的模式，等待客户端的连接。通过代码可以很容易看到这两部分编程的方式以及区别。

客户端编程的代码框架（具体的代码可以参考socket编程的书籍）：

```
fd=socket()
setsockopt(fd)
r=connect(fd,destination)
```

read(fd)/write(fd)

send(fd)/recv(fd)

close(fd)

继续看一下服务器端的代码框架：

listenfd=socket();

setsockopt(listenfd)

bind(listenfd)

listen(listenfd)

acceptedfd=accet(listenfd)

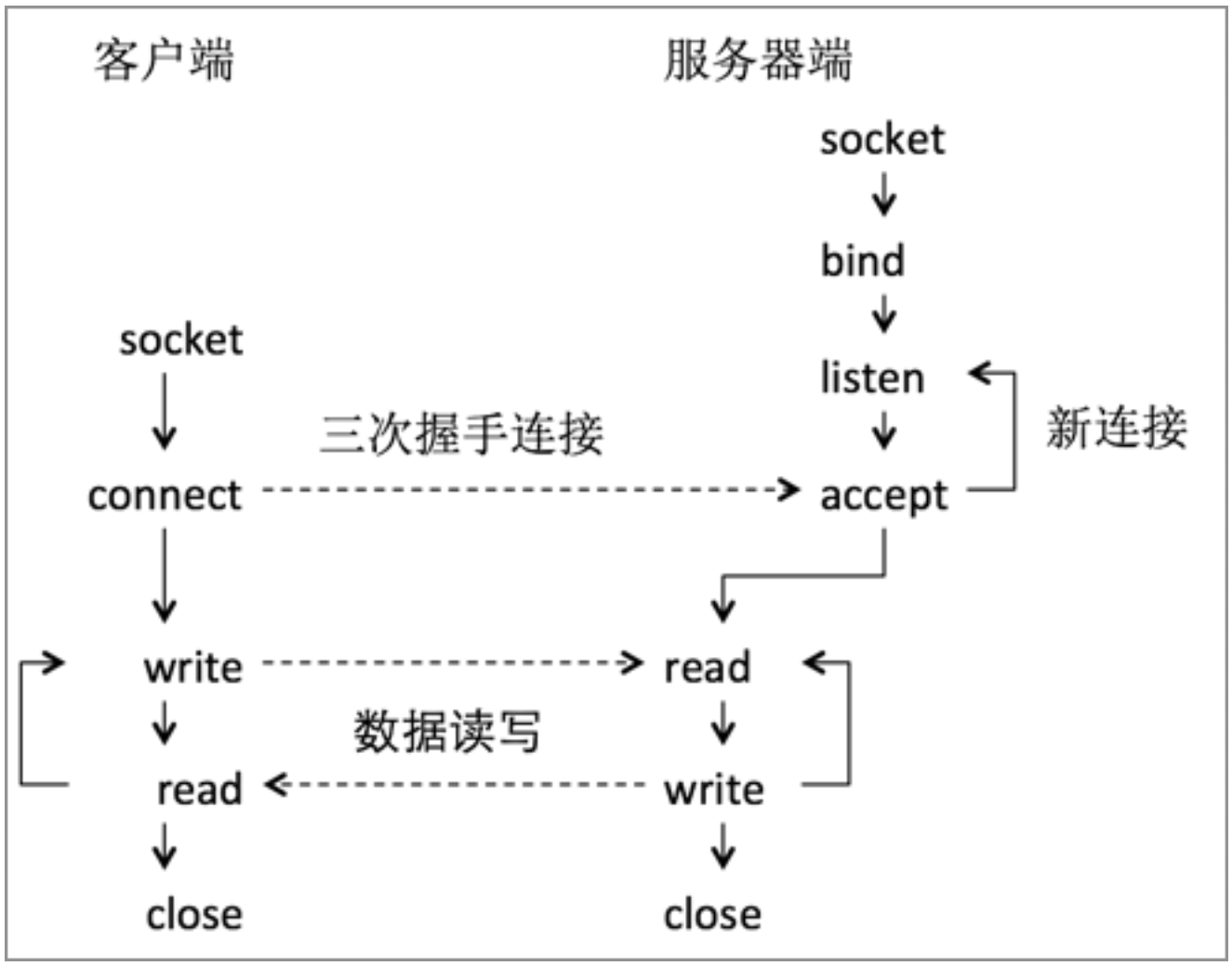
read(acceptedfd)/write(acceptedfd)

close(acceptedfd)

先解释一下客户端的代码。客户端在与服务器端进行通信的时候，首先要通过socket()来获得一个套接字，这个套接字也是最后进行读写的文件描述符（file descriptor）。然后可以通过setsockopt对这个套接字fd进行配置，之后客户端就可以通过connect函数来连接服务器了。此时，在服务器端必须准备好等待客户端的连接。这部分的工作交给了bind和listen。具体的说就是服务器端需要先通过socket()获得一个文件描述符listenfd，专门用于连接的监听接入。之后可以通过setsockopt对其进行配置。然后服务器端通过bind操作将这个listenfd绑定到需要监听的IP地址以及端口上。之后，服务器就可以进行监听的动作，即listen(listenfd)，在这个描述符上进行监听，这个函数的另外一个参数是在监听的同时，如果服务器正在处理一个连接，所允许的在队列中的等待的连接请求有多少个。之后，服务器端将调用accept系统调用等待客户端的接入。在这个时候，服务器端已经做好了接收连接的准备，客户端可以发送连接的请求（或者说，发送的请求可以被服务器端接收，否则的话会发生连接失败的情况）。在这个时候，accept系统调用就会返回，并且程序获得了另外一个文件描述符，即acceptedfd，这个是真正的与客户端进

行通信的文件描述符。之后，服务器端可以通过这个文件描述符使用read/write系统调用完成数据通信。

图2.15 套接字数据通信编程基本框架



上述网络编程流程可以显示在图2.15中。在数据传输完成之后的最后一步工作是通过close系统调用去关闭文件描述符，完成数据传输的最后一步工作，释放资源。在数据通信的时候，accept接口可以再次执行，去接收另外一个连接。

需要注意的是，上述的编程接口是一个通用的过程，在实际编程的过程中还会有需要有一定的变形，并进行配置。但是对于这个基本过程的理解是进行下一步编程工作的基础。

服务器端网络编程的模式

在实际的编程过程中，除了依据上述的基本的套接字编程方式之外，对于服务器端的服务程序来说，有一个非常重要的问题是如何支持多个客户端的连接以及在这种情况下的有效的数据传输。因为从客户端来说，相对的连接的服务器数目是有限的，每次只需要连接一个即可，而对服务器来说，每次可能需要服务的客户端数目会有数百上千个。因此，这就衍生了不同的服务器端的编程方法，以期待能够以更高的性能来支持更多的客户端程序。

多进程服务器端编程方式

在最早的编程模式中，在UNIX环境下面有一个标准的通过多个进程进行多任务处理的方式，这种方式也被用在服务器的网络编程方面，下面是这个多进程方式的标准做法。

```
acceptedfd=accept(listenfd);
pid=fork();
assert(pid>=0);
if(pid==0) //child process
    close(listenfd);
    do_some_thing_with_acceptedfd;
else
     close(acceptedfd);
     go_back_to_accept();
```

注意，这里是服务器端的代码，不是客户端的代码。这里一个关键的系统调用是fork()。这可能是Unix系统中最难理解的系统调用之一了。这个系统调用将生成一个新的进程，并且新的进程与原来的进程构成了父进程和子进程的关系。原有的进程为父进程，新生成的进程为子进程。父进程和子进

程具有完全相同的执行代码，完全相同的内存以及网络资源等，唯一不同的是fork()函数的返回值，在父进程中返回的是子进程的进程号，而子进程则返回0。fork()系统是建立进程的基本方式，因此，虽然使用了相同的代码，虚拟地址空间的内容完全一样，但是这些资源都是相互隔离的，之后的执行过程父子进程将独立执行。这样的话，可以据此理解网络服务器的代码。在上述的网络服务器处理代码中，子进程会关闭listenfd，因为它不需要进行监听的工作，子进程将使用accpetedfd来与客户端进行通信。而父进程不需要进行数据处理的工作，因此将acceptedfd关闭，继续使用listenfd等待另外一个客户端的连接。上述是服务器通过多进程的方式进行多服务器端处理的基本模式，也是apache服务器的处理方式。

多线程服务器端编程方式

通过多进程的方式进行并行处理，虽然能够对多个客户端进行处理，但是缺点也是显而易见的，因为使用多进程的方式进行处理，最大的问题是操作系统需要进行进程的上下文的切换。一般认为，这种切换方式需要使用较多的时间以及切换代价，例如一个最典型的负担就在于虚拟内存的切换，即需要从某一个进程的虚拟内存切换到另外一个进程的虚拟内存，这需要使用大量的指令才能够完成，并且由于虚拟内存映射关系的转换，需要将处理器内部的缓存进行清空，这也会造成处理速度的下降。因此，为了减少这一部分所带来的处理的额外负担，现代操作系统以及许多的网络服务器程序通过多线程的方式进行服务器的数据处理工作。通过多线程的方式进行处理的代码结构类似于多进程的方式，代码框架如下：

```
//在主线程中
acceptedfd=accept(listenfd);
thread=get_free_thread_from_pool();
set_thread_data(thread,acceptedfd);
```

```
activate_thread(thread)
go_back_to_accept()
//在子线程中
do_something(acceptedfd);
close(acceptedfd);
```

在这个代码中，主线程首先通过accept系统调用允许客户端的连接，并且获得了一个进行通信的acceptedfd，然后这个acceptedfd将会派发给一个新的线程进行处理。这个新的线程用acceptedfd进行与服务器端的通行工作，而主线程将返回到accept函数继续等待。

从上述的代码可以看到，多线程处理的代码与多进程处理的代码在结构上是类似的，但是在效率上要比多进程的代码要高一些，主要的原因是现在多个线程处于一个进程之内，因此，不需要通过fork的方式来创建多个进程，降低了创建的开销以及操作系统内部对于进程的数据维护。当然，多个线程的方式能够使用同样的虚拟地址空间，这就避免了虚拟内存的切换操作，也不必清空处理器内部的缓存，从而能够提高处理的效率。

通过select进行单线程多路描述符服务器端编程方式

当然，多线程的处理方式还是有一定的线程切换的开销，在操作系统内部以及多线程的库环境中也需要进行线程管理数据结构的维护，线程也需要一定的上下文，也需要通过上下文的切换来完成切换任务。这一部分的开销可以进一步通过下面所说的单线程多路描述符（file descriptors multiplex）的方式进行优化。在展开这部分的讨论之前，先要澄清一个概念，即在多核以及多处理器的环境下，必须要通过多线程或者多进程的方式才能够充分发挥多核多处理器的能力。因此，即使使用下述的多路描述符的方式来进一步提高服务器的性能，但是还是需要在多线程或者多进程的环境下进行处理。

在讨论通过单线程多路描述符之前，先讨论一下文件描述符的一项属性。这一项重要的属性即描述符是阻塞（blocking）访问的还是非阻塞访问的（non-blocking）。阻塞访问的描述符在进行read(fd)系统调用的时候，如果没有数据，将会阻塞到这个函数里面，并将调用这个函数的线程置于休眠状态。而非阻塞调用则在没有数据可用的时候，函数调用将立即返回，不会将对应的线程放置到休眠状态。因此，对于非阻塞调用来说，在一个线程中可以管理多个描述符，而不仅仅是一个描述符。这对于阻塞描述符来说是做不到的，因为一旦进入了数据读写的过程，线程将有可能会被阻塞，直到有数据或者被中断所打断。因此，下面进行讨论的多路描述符的方式需要将描述符使用fcntl或者setsockopt来将描述符设置为非阻塞的状态O_NON_BLOCK。

最先被使用的多路读写的系统调用函数为select，这个函数的原型（prototype）如下：

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
        fd_set *exceptfds, struct timeval *timeout);
```

在这个函数中，readfs是需要监听的读的描述符的列表，writefds是需要监听的写的描述符的列表，而exceptfds是需要监听的有可能出现异常的描述符的列表。nfds是上述的三个描述符列表中的最大的描述符的值+1。最后的一个timeout的结构是一个超时的结构，即通过多长的时间，这个系统调用必须要返回。因此，select系统调用的操作是阻塞的，只有在这几个描述符可以进行读写或者需要处理的时候才会返回。

以单线程进行多个描述符处理为例，通过select方式进行多路读写的操作的代码框架如下：

```
fd_set reafds,writefds;
struct timeval tv;
int retval;
```

```
FD_ZERO(&readfds); FD_SET(0, &readfds); FD_SET(x, &rfds);
FD_ZERO(&writefds); FD_SET(1, &writefds); FD_SET(y, &writefds);
tv.tv_sec = 5;  tv.tv_usec = 0;

retval = select(1, &readfds, &writefds, NULL, &tv);
if (retval == -1)
    perror("select()");/* some thing error, check errorno*/
else if (retval){
    printf("Data is available now.\n");
    if(FD_ISSET(0, &readfds)) do_read_something_with(0);
    if(FD_ISSET(x, &readfds)) do_read_something_with(x);
    if(FD_ISSET(1, &writefds)) do_write_something_with(1);
    if(FD_ISSET(y, &writefds)) do_write_something_with(y);
}
else
    printf(“timeout, no data”);
```

上述的例子给出了通过select的方式进行多路描述符读写的基本结构。程序首先是对一个名为fd_set，即描述符集合进行操作。这里的操作包括FD_ZERO即清零操作，以及FD_SET即设置操作。这两个操作用以在一个集合上设置程序感兴趣的那部分文件描述符，对于程序不感兴趣的文件描述符需要进行清零。文件描述符的集合有3个，分别为读文件描述符集合，写文件描述符集合以及异常文件描述符集合。之后，这三个集合会被传递给select函数作为参数进行调用。select函数的执行是阻塞的，即在这些文件描述符的集合中没有发生事件的时候函数不会退出，对应的线程处于休眠状

态。这一点实际上是被编程方式所允许的，因为在这种方式下，如果select没有执行完毕，那么本身就不会有输入输出IO发生，也就没有必要进行处理。当然，在select调用的最后一个结构是超时结构，可以方便程序员完成需要定时处理的任务。

使用select方式之后带来的一个重大的改进是现在可以在一个线程的内部就进行对多个描述符的输入输出处理，而不是需要为每一个描述符单独开一个新的进程或者新的线程。这能够大大降低整个系统的对于多进程以及多线程上下文的切换。因此，在进行网络服务器数据处理的情况下，可以通过select的方式提高整个服务器的处理能力。与select相类似的另外一个系统调用为poll，使用类似的数据结构以及检查机制来完成在一个线程中处理多个文件描述符。

多路描述符编程方式的改进

但是，从上述的代码中读者也可以看到，整个程序的结构中，如果从select函数调用完成之后，代码需要浏览一遍所有的描述符，观看描述符的状态是否是可读或者可写，这是一个循环的过程。在这种情况下，如果仅仅是一个描述符发生可以读写的转换，在进行检查的时候，仍然也是需要遍历所有的描述符，造成这一段代码的浪费。因此，可以想象，在实际的服务器程序中，如果有大量的客户端连接的话，程序的代码会将大量的时间浪费在检查描述符的状态上，从而会降低整个程序代码的效率。这个效率的根本原因是内核在监测到状态变化的时候，本身不会把可用的描述符搜集起来交给应用程序。因此，从效率上说，需要进一步改进的地方是需要内核的支持来一次性告诉应用程序可用的文件描述符。另外一个select的限制是文件描述符的限制，即一个进程最大的能够打开的文件描述符一般为1024，这对于普通的应用程序来说是足够的，但是对于大规模的网络应用服务器程序来说是远远不够的。

为了改善上述的select代码中对于描述符扫描带来的开销，在Linux内核（只在Linux2.6.19之后的内核）中引入了epoll的模式，能够大大改善在进行大规模服务器运行中的开销。通过epoll模式给出的系统调用，应用程序编码的时候就无需进行描述符扫描与判定，因为内核在返回的时候，就直接给出了能够进行处理的描述符的列表，而无需用户层的代码去扫描判断。在epoll的方式中，操作系统给出的编程接口包括epoll_create，epoll_ctl，epoll_wait三个调用接口。epoll_create这个系统调用帮助应用程序去建立一个新的描述符来代表/dev/epoll这样一个设备，这个描述符是可以通过close系统调用去关闭的。epoll_ctl用以向epoll方式的监听描述符集合中加入或者删除一个文件描述符，因此在系统进行读写之前，先要通过epoll_ctl这样一个系统调用将描述符加入到关心的集合中。这一点也非常类似在select方式下的fd_set的系统结构，不过，现在这个数据结构是通过内核去维护的，用户态的应用程序不会直接使用到这个数据结构。最后一个系统调用为epoll_wait，这个系统调用就相当于select系统调用，等待某一个文件描述符能够进行读写。下面是epoll的一个具体的例子程序（从Linux的手册中摘录）。

```
#define MAX_EVENTS 10
struct epoll_event ev, events[MAX_EVENTS];
int listen_sock, conn_sock, nfds, epollfd;
epollfd = epoll_create(10);
ev.events = EPOLLIN;
ev.data.fd = listen_sock;
epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev);
for (;;) {
    nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);

    for (n = 0; n < nfds; ++n) {
```

```
            if (events[n].data.fd == listen_sock) {
                conn_sock = accept(listen_sock,
                                   (struct sockaddr *) &local, &addrlen);
                setnonblocking(conn_sock);
                ev.events = EPOLLIN | EPOLLET;
                ev.data.fd = conn_sock;
                epoll_ctl(epollfd, EPOLL_CTL_ADD, conn_sock,&ev)
            } else {
                do_use_fd(events[n].data.fd);
            }
        }
    }
```

注意，在上述的代码段中删除了所有的错误处理的代码，实际的工作代码必须把错误处理的代码加入。

这段代码在使用epoll之前，先需要依据前面讨论的socket的编程模式去建立一个服务器端的监听的文件描述符（listenfd）。之后，代码通过epoll_create来获得一个使用epoll机制的文件描述符。在epoll_create函数中，MAX_EVENTS 10在内核中是一个指示，即需要监听的描述符规模，大概会被认为是可能的最大的描述符的数目，但是在当前的内核中是被忽略的。通过这个函数获得一个针对epoll的文件描述符epollfd。

之后，第一个被通过epoll_ctl(epollfd, EPOLL_CTL_ADD, listen_sock, &ev);加入到epollfd监视列表的是监听的文件描述符，即listen_sock，使用的操作位EPOLL_CTL_ADD，即加入到epollfd的监视列表中。对于listen_sock有意义的是输入数据的监视，即ev.events = EPOLLIN，即一个输入的事件。

整个程序的总体是一个循环的过程，这个过程与之前的多线程多进程以及select机制类似，即循环进行客户端的服务过程。循环的驱动函数为epoll_wait函数，需要等待这个函数的完成。这个函数返回的数据结构为struct epoll_event events[MAX_EVENTS]，并且可以看到返回值nfds就是内核告知应用程序发生事件的描述符的个数。现在，应用程序不必去扫描所有的描述符判断事件的发生，而是所有的返回的描述符都有事件发生。这也可以从代码中do_use_fd(events[n].data.fd);看到。通过这种方式，应用程序可以获得有效的进行读写的文件描述符，无需进行扫描，能够大大提高服务器程序的效率。

在使用epoll进行编程的时候，通常会遇到两个触发的概念：电平触发（Level-Triggered）和边沿触发（Edge-Triggered）。这两个概念原先是属于电子线路中的概念，是驱动时序电路的方式。传统的电子线路中通过电平触发的方式，即时钟信号为1的时候会触发电路中的各项功能单元的工作。现有的计算机电子线路中，通常使用的是边沿触发的方式，即在时钟信从0变为1的过程或者从1变为0的过程中触发电路中的功能单元。在epoll编程模型中，实际是借用了这样的一个概念，用以定义epoll_wait函数如何完成工作，触发应用程序的下一步的动作。

在电平触发模式下（Leven-Triggerd），epoll_wait函数一旦发生有文件描述符处于就绪状态（即有事件发生，如可以进行读写），那么就立即执行结束，通知应用程序可以进行读写。这是缺省的工作方式，可以看到，在传统的编程模式select以及poll都是这种工作方式。在这种工作方式下，如果应用程序不对文件描述符做操作，因为数据仍然可以读写，因此描述符的状态还是就绪状态（即处于电平1的状态），因此如果再次对对应的文件描述符调用epoll_wait，仍然可以获得通知。

另外一种编程方式就是所谓的边沿触发（Edge-Triggered）的方式，这种方式发生触发的条件是描述符从未就绪状态（non-ready）转变为就绪状态（ready），这就相当于在电子线路中时钟信号从0转变为1。在边沿触发的

模式下，内核如果发现描述符从未就绪状态转变为就绪状态，那么就会产生一个触发，告知应用程序可以进行读写了。之后应用程序可以对对应的文件描述符进行读写。此时，内核认为应用程序已经得到了通知，如果这个时候文件描述符一直处于就绪的状态，那么应用程序调用epoll_wait是不会得到通知的。内核不会在这种情况下发送更多的就绪通知，epoll_wait会持续处于阻塞的状态。应用程序在得到一个就绪的文件描述符的时候，应当对这个文件描述符进行持续的读写，直到这个文件描述符变为未就绪状态（例如，返回值为EWOULDBLOCK）。

边沿触发的方式只允许使用在非阻塞（non-blocking）的文件描述符中。如果使用阻塞的描述符，那么描述符要么得不到通知，要么会阻塞在某一个读写的过程中。电平触发的模式能够使得一旦某一个文件描述符处于就绪的状态，epoll_wait都会给出通知事件。因此，在电平触发模式下，可以使用阻塞的方式，将就绪的文件描述符交给epoll_wait，并继续获得就绪的通知。当然，在这样的情况下，务必需要保证写入或者读出之后，相对应的文件描述符还继续处于就绪状态，否则的话会阻塞程序的执行。

一般来说，通过边沿触发的方式可能会获得更好的性能，因为相对的从内核状态获得的通知事件会更少一点，从而减少了操作系统和应用程序之间的上下文的切换的次数。但是，这一点也是有代价的，边沿触发的方式只会在发生变化的时候会给出通知，从程序员的角度来说，特别是从电平触发的角度来说，可能会丢失事件通知，应用程序需要进行仔细的考虑，避免程序的执行阻塞在epoll_wait系统调用中。

多路文件描述符编程的退出机制（控制机制）

在这里，继续用一小段篇幅去注解一下网络编程的一个小细节，即如何从poll，select，epoll_wait中事件退出的问题。

以epoll为例，这个阻塞的等待事件的系统调用的形式如下，即：nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);。通过这一次函数调用可以看到，没有任何一种方式能够从这个系统调用中退出来，除非是发生了某一个事件，或者是发生了超时。这一点在某些情况下是不可接受的。例如，可能会为这个epollfd增加一个需要监听的文件描述符，以增加对一个客户端的服务。

如果使用超时机制的话，就会发生一段超时时间段内无法响应的情况，这就需要仔细衡量超时的设置。如果超时设置时间过长的话，客户端就会需要等待一段比较长的时间，如果超时时间短的话，就会造成大量的操作系统和应用程序的切换，这些切换会影响性能，并且这种加入一个描述符到epollfd的情况是非常少的，这些切换实际是一个浪费的调用。因此，希望通过应用程序的方式，能够在需要的时候，迅速结束epoll_wait的执行，以使得应用程序可以有机会进行下一步的操作，并且在正常的情况下不去打扰程序的正常执行，即让epoll_wait去继续等待下一次事件的到来。

因此，一个合理的做法是还是使用epoll_wait函数本身的特定，即这个函数在有事件发生的时候就会完成执行，返回到应用程序。可以通过制造一个事件来帮助epoll_wait的方式来完成执行，退回到应用程序，使得应用程序可以作下一步的执行。有许多读者可能已经猜到了，可以通过管道的方式在epoll_wait中植入管道的一端。之后如果希望从epoll_wait中退出来，可以在管道的另一端写入一定的数据，使得在epoll_wait中的管道描述符将状态从未就绪状态转变为就绪状态，从而使得epoll_wait完成执行进行通知。这部分的的代码示例如下。

```
int notification_pipe[2];
pipe(notification_pipe);//创建管道
ev.events = EPOLLIN;
ev.data.fd = listen_sock;
```

```
epoll_ctl(epollfd, EPOLL_CTL_ADD, notification_pipe[0], &ev);
/* 在这里，可以为epollfd加入任何需要监视的文件描述符 */
nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);//
```
进入事件循环，注意epollfd也同时关注了notificaiton_pipe[0]。

之后，应用程序使用同样的方式进行处理，只不过需要对 notification_pipe[0] 进行特殊处理，例如：

```
if(events[n].data.fd == notification_pipe[0]) {
  read_int_from_notification_pipe(notification_pipe[0]);//从中读出一个整数，说明获得了通知
  /* 这个时候，epoll_wait已经完成了工作，并且是由于管道退出的，需要依据应用程序的需求，做相应的处理 */
}
```

同时，在程序的另外一个部分，处于同一个进程内的多个线程，或者处于父子进程内，如果需要epoll_wait进行退出的话，那么就需要调用write(notification_pipe[1])来写入一定的数据，使得等待事件的线程能够获得执行的机会，来进行相应的处理。

当然，除了上述通过pipe的方式能够使得poll, select, 或者epoll_wait从等待的过程中退出，从而可以做相应的处理外，这不是唯一的从这些函数中退出的方法。实际上还有其它的方法，例如可以通过处理信号量的方式来获得通知。这个时候，为了能够正确处理信号，需要另外的一套处理函数，名为pselect，ppoll以及epoll_pwait。这些函数能够在保证安全性的前提下，使得函数退出的时候，要么是有事件发生，要么是获得了一个正确的操作系统信号。

在Linux系统中，还有另外一种方式能够完全替代pipe在这里的作用。从前面的讨论中也可以看到，pipe的真正作用是起到管道数据传输的作用，但

是在多路描述符服务器编程中，只是起到一个通知的作用，并不进行真正的数据传输，即并不关心pipe真正的工作。因此，在Linux系统中提供了一种新的机制，即eventfd，能够只用一个描述符来完成在这里对select，poll，以及对于epoll_wait的唤醒。eventfd的系统调用的接口为：

int eventfd(unsigned int initval, int flags);

可以看出，通过这个系统调用可以产生一个描述符，这个描述符具有一个初始的计数值，这个计数值可以用来当做一个semaphore使用，更多的时候是不具有实际意义的。在这里，获得的文件描述符可以被使用在上述的多路文件描述符的驱动函数中 ，即poll, select以及epoll_wait函数调用。事件描述符eventfd相比于管道减少了一个描述符的使用，概念上也更加清晰。

文件系统的异步处理方式

关于服务器端网络的编程模式还有一个重要的编程模式就是异步处理的方式。前面所述的方式无论是阻塞的，还是非阻塞的，或者将它们混合在一起，并通过多路文件描述符的方式进行编程，从本质上来说，都可以认为是同步处理的方式。同步处理与异步处理是编程中提供的两个非常重要的概念。同步处理的方式的本质特征是很容易理解的，即一个函数调用在返回的时候，这个函数所表明的动作已经完成，或者表明这个函数的功能可能没法完成，即函数调用失败。

同步方式的一个最大的特点是在函数调用的时候，程序的其它部分是不能继续工作的，而是只有等待函数调用完成之后才能够继续下一步的工作。这对于应用层的程序代码来说是没有问题的，因为应用层的程序代码都是需要完成一定的应用程序的功能而设置的，不会出现挂起服务器的情况。但是，如果一个函数调用涉及到系统调用，特别是涉及到输入输出的系统调用，那么这个函数调用可能就会挂起这个程序，等待输入输出的完成。例如，一个从网络中获得数据的程序在进行网络数据读取的时候，必然会调用

到read系统调用，而这个系统调用会等待网络中数据准备完毕才能够返回。在这个函数执行的期间，应用程序处于阻塞的状态，被操作系统挂起，直到数据准备完毕。如果是单线程程序的话，那么此时程序就完完全全处于休息状态，即使还有其它的工作需要做，也只能等待系统调用的返回。

异步的方式则提供了一种完全不同的处理方式。在异步方式下，一个函数调用并返回之后，并不是表明相应的工作已经完成，而是仅仅告诉函数的调用者，相关的功能已经被被调用者获知，被调用者会去完成指定的工作，至于何时完成，则在函数调用完成之后是不能保证的。

因此，如果采用同步执行的方式的话，为了能够进行多个功能的处理，必须采用多线程或者多进程的方式，或者采用前面所述的多路描述符的方式来进行任务的处理。整个程序在某一个时间点必须要去等待相应的任务的完成。而异步方式则采用了完全不同的方式，应用程序在调用完毕之后，无需等待，可以继续进行下一步的工作。操作系统则会在完成任务之后，去调用应用程序提供的函数来通知应用程序任务的完成。这里，这种通过应用程序提供给操作系统的函数被称为“回调函数”，因为应用程序是不会主动去调用这种函数的，而是通过底层的操作系统在完成一项任务的时候去调用。回调函数是一个非常重要的概念，属于异步操作的关键性的内容。前面所描述的信号处理的方式，实际上也是一种类似于回调函数的机制来起作用，应用程序没有明确的去调用回调函数，而是由操作系统在进行切换的时候，操作应用程序栈，给回调函数执行的机会。

以Linux下的异步输入输出函数为例（忽略错误处理），来进一步深化异步数据处理这一个概念。这里采用了aio的库，是对Linux底层异步处理函数的包装。（这里以一个实际的例子说明异步工作的方式。）

```
int fd, ret;
struct aiocb my_aiocb;
```

```
  fd = open( "file.txt", O_RDONLY );

/*下面的操作初始化一个异步处理的数据结构*/
  bzero( (char *)&my_aiocb, sizeof(struct aiocb) );
  my_aiocb.aio_buf = malloc(BUFSIZE+1);
  my_aiocb.aio_fildes = fd;
  my_aiocb.aio_nbytes = BUFSIZE;
  my_aiocb.aio_offset = 0;
  my_aiocb.aio_sigevent.sigev_notify = SIGEV_THREAD;
  my_aiocb.aio_sigevent.notify_function = aio_callback;/*这里是回调函数*/
  my_aiocb.aio_sigevent.notify_attributes = NULL;
  my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;/* 参数 */

/* 发起一个异步的操作 */
   ret = aio_read( &my_aiocb );
/* 完成，可以做其它的事情了 */
}

/* 下面是回调函数 */
void aio_callback ( sigval_t sigval )
{
  struct aiocb *req;
```

```
    req = (struct aiocb *)sigval.sival_ptr; /*获取参数，前面初始化的时候设置
*/
    if (aio_error( req ) == 0) {
      /* 检查是否正确，如果正确的话获得读取的长度，并进行处理 */
      ret = aio_return( req );
    }
  }
```

上述的代码是进行异步处理的总体的程序框架。在这个程序框架中，应用程序的代码需要读一个文件，但是是通过异步的方式去读取，因此先要为异步读取函数aio_read进行回调设施的处理，设置一个数据结构my_aiocb，包括缓冲区的设置，回调函数的设置等。之后就调用aio_read，这个函数将立即返回，但是不表示数据已经读取出来。之后，应用程序可以进行自由处理其它的工作。读取完成之后，操作系统会在用户线程中调用aio_callback函数进行获取数据的处理。异步处理的过程中，一旦下发任务后，后续的工作将不由应用程序控制，即何时调用aio_callback是未知的。因此，从代码上来看，要比同步的代码复杂很多，需要仔细设置各项预先的配置，以便异步调用能够成功进行。不过，相对来说，这种的复杂性是值得的，通过异步的输入输出方式通常能够得到更高的性能。应用程序的代码无需等待操作系统的通知，而操作系统自己会通知应用程序，并直接调用应用程序预先定义的回调函数来完成工作，这种方式会减少不必要的上下文切换以及等待。

有关于直接进行网络编程方式的大部分内容都在前面的讨论中，读者应当对这些内容有比较好的理解。相对于使用编程库来说，这里讨论的都是通过系统调用的方式直接调用底层的函数。如果能够进行良好编程的话，这是效率最高的一种方式，其它的方式实际上都是对这里所讨论的机制的包装。理解这些内容也能够帮助读者去理解现有的网络的一些编程框架，包括一些服务器的性能问题，例如Apache，Nginx以及Node.js的异同等。当然，使用

这种底层编程的方式对于应用程序员来说的要求非常高，如果处理不当的话，可能降低效率，并且难于调试。因此，必须要说的是对于大家平时使用的网络编程的话，还是使用现成的编程库进行编程比较方便，并且性能也不会有太大的问题。

关于远过程调用RPC的讨论

本章的最后部分讨论一下远过程调用（RPC: Remote Procedure Call）的相关问题。远过程调用可以说是建立在底层的网络编程模式之上的第一个抽象。从前面的讨论中读者已经可以看到，底层的编程方式需要涉及到许多的细节问题，因此需要对操作系统有比较好的理解才能够进行良好的编程，获得比较好的效果。从这些底层的功能提供开始，就有研究人员和开发人员试图提供上层的抽象来提高编程的效率，同时也希望能够维护较高的性能。其中，在不同的语言中进行套接字或者系统调用的绑定是必须的，例如在Java，Python，.Net中都提供类似的底层的抽象。

这一节需要讨论的内容是远过程调用，这个抽象能够应用在不同的语言层面上，例如在最底层的C/C++上有SUN RPC的调用，Java有RMI，在网络中有XML-RPC，JSON-RPC（能够绑定到不同的语言中，使用HTTP协议）。在RPC的核心的技术是网络上互相交互的协议，并且这个协议也是工业届常用通信协议，其中就包括SOAP，Google的Protocol Buffer以及Facebook的Thrift。

回到RPC的技术层面，RPC需要达到的目的是希望能够通过远过程的调用来实现编程的时候能够使得程序员获得类似本地调用的能力。例如，程序员在调用一个函数的时候：result=function(a1,a2)，这个函数对于本地以及远端进行透明化处理。从传统的程序的角度来看，function函数的调用者与被调用者应该处于同一个进程中，这样的话程序的执行控制从一个函数进入到另外一个函数。这对于程序员来说是一个友好的编程方式。但是，在涉及到网络的时候，就跟已经讨论过的一样，需要进行复杂的处理，才能够调用到远端的某一个进程的一段代码，并且需要等待结果返回。实际上，对于同一台机器上的两个程序来说，也是存在同样的问题，因为由于虚拟地址空间的

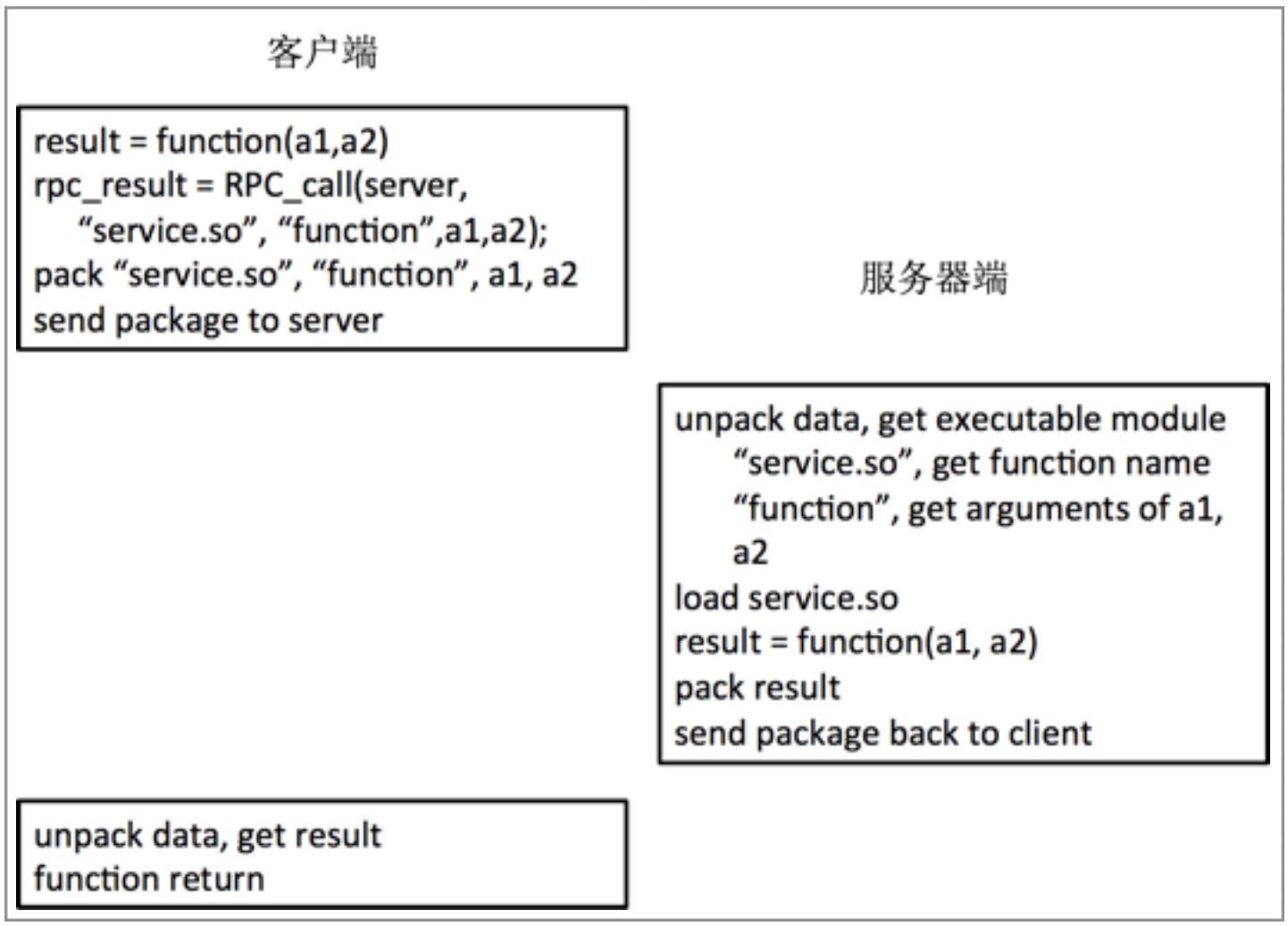
隔离，两个进程上的代码互相之间不能进行直接的调用，而是需要通过进程间通信（IPC: Inter-Process Communication）进行。这种方式也会带来较高的复杂性。因此，RPC的目的就希望能够提供一种底层的机制，使得多个进程，甚至进程运行的不同的服务器上都可以通过类似本地调用的方式直接进行函数调用，而不再需要涉及到底层的网络编程的工作。

具体地来说，result=function(a1,a2)这样的函数调用中，function这个函数可能处于和调用者同样的地址空间，可能位于同一台机器上的不同的进程中，甚至在不同的机器上的不同的进程（肯定是不同的进程）中。对于调用者来说，function位置不是它所关心的，它所关心的只是这个函数可以被调用，并且如同在本地调用一样获得具体的结果即可。而对于function函数的提供者来说，同样的，它会知道函数可能会被另外的服务器上的另外的进程所调用，但是它也不希望进行网络的复杂的处理。因此，对于RCP提供的库来说，就需要提供这样的底层的库进行抽象，使得函数的调用者可以直接调用，而函数的实现者也与直接实现函数的本地调用一样。因此，读者可能也猜到了，在这里，关键的一点就是进行包装的处理，将调用的位置透明化。下面就来具体分析一下为了实现上述的远过程调用的目的，底层的基础库需要做什么样的工作。讨论的场景是处在客户端机器中有一个进程（客户端进程p1），希望调用服务器端进程（p2）的某一个函数function，并使用了参数a1和a2，返回结果result。

分别来看看在客户端以及服务器端需要做什么样的工作。在客户端中，由于需要调用服务器端的代码，需要做的工作包括：1）确定服务器端的位置，需要调用的对象的服务器的地址；2）确定需要调用服务器端的具体的进程；3）将函数名称function，函数的参数a1，a1进行打包，通过网络发送给服务器端，等待服务器端的返回；4）在服务器端返回结果的时候，将打包的数据进行拆包，依据函数的原型提取结果，将结果返回给应用程序。这里1）和2）部分实际指示了通信的目标地址，可以通过IP:Port的形式表达出来。3）和4）实际是协议的打包和拆包的过程，在英文中也被称为marshal/

unmarshal，因此需要一套协议来规定参数以及结果的打包和拆包的处理方法和手段。这一段工作流程可以从图2.16中看出来。

图2.16 同步端远过程调用流程



从图中可以看出，一次远过程调用涉及到起码两次的网络交互，即一次从客户端到服务器端的请求过程，一次从服务器端到客户端的返回过程。从服务器端的角度来看，需要进行的工作就包括预先建立服务器程序，发布能够支持远过程调用的函数的名称，随后就进入监听的过程。在客户端连接进入，发出请求之后，依据客户端发送过来的参数，执行相应的函数，并返回结果。服务器端在这个阶段就包括1）提取网络数据后进行拆包，将参数从网络数据包中提取出来；2）执行函数，获得结果；3）打包，将返回的值依据RPC协议的要求进行打包，发送给客户端。

以上就是远过程调用的基本的流程以及讨论。当然，在实际的工作过程中还有实现的细节讨论，以及需要匹配相应语言的各种规范，但大致流程正如上述讨论的过程。下面将就两个方面展开讨论，一个是由于远过程调用可能会涉及到多个网络交互的过程，因此往往性能是比较差的，为了能够提高性能，这个时候可以采取一定的手段来屏蔽网络中的性能影响；第二个是远过程调用中的出错的处理，因为本地过程调用的时候，调用者和被调用者都处于一个进程内，无需考虑到跨进程间的出错的问题，而远过程调用则需要考虑这方面的问题，特别是这方面可能造成的正确性的问题。

关于远过程调用的性能的讨论

如同一般的网络编程类似，远过程调用由于涉及到了网络数据的交互，因此如果采用上述同步调用的处理方式的话，可能会造成性能低下。为此，在具体的实现上必须要考虑到这一点来提高远过程调用的性能。

第一个能够采用的方式是进行请求的批处理手段。在进行网络数据传输相关的处理的时候，性能的优化方面主要采用的手段一个是通过批处理的方式将多个请求来进行整合，这样的话就可以将多个请求整合为一个请求，在服务器端一次调用就可以执行多个函数，并且一次性的去返回结果，而不是一次一个请求进行处理，并每次去返回一个结果。这种方式可以屏蔽网络的延迟，即只需要一个延迟就可以为多个函数调用服务，批处理的方式也能够充分利用网络的带宽。但是，对于其中的一些函数调用来说，可能需要等待一段时间才能够获得与其它的函数调用进行打包的机会，会提高某一些函数调用的延迟。这种方式对于处理服务器端的处理性能在某些时候是十分必要的，因为往往服务器端会处理数目十分庞大的请求，服务器间的交互也是十分庞大的，通过批处理的方式能提高服务器之间的交互性能。但是，对于客户端程序来说，可能在大部分的条件下不会采用这种方式，因为请求数目相对来说还是比较少的。

另外的方式就涉及到了之前讨论过的网络数据处理的模型的问题。需要做的就是更改网络数据处理中的同步模型，将其更新为其它的模型。因此，在远过程调用中也会有不同的网络编程模型的区别，在这里被称之为调用模型的区别。同样的，将下面的两种方式称为异步调用模式以及回调函数模型。这两种处理方式没有本质的区别，唯一的区别就在于对于结果处理是应用程序自己去处理控制还是由RPC的运行时系统去控制。这两种方式的缺陷

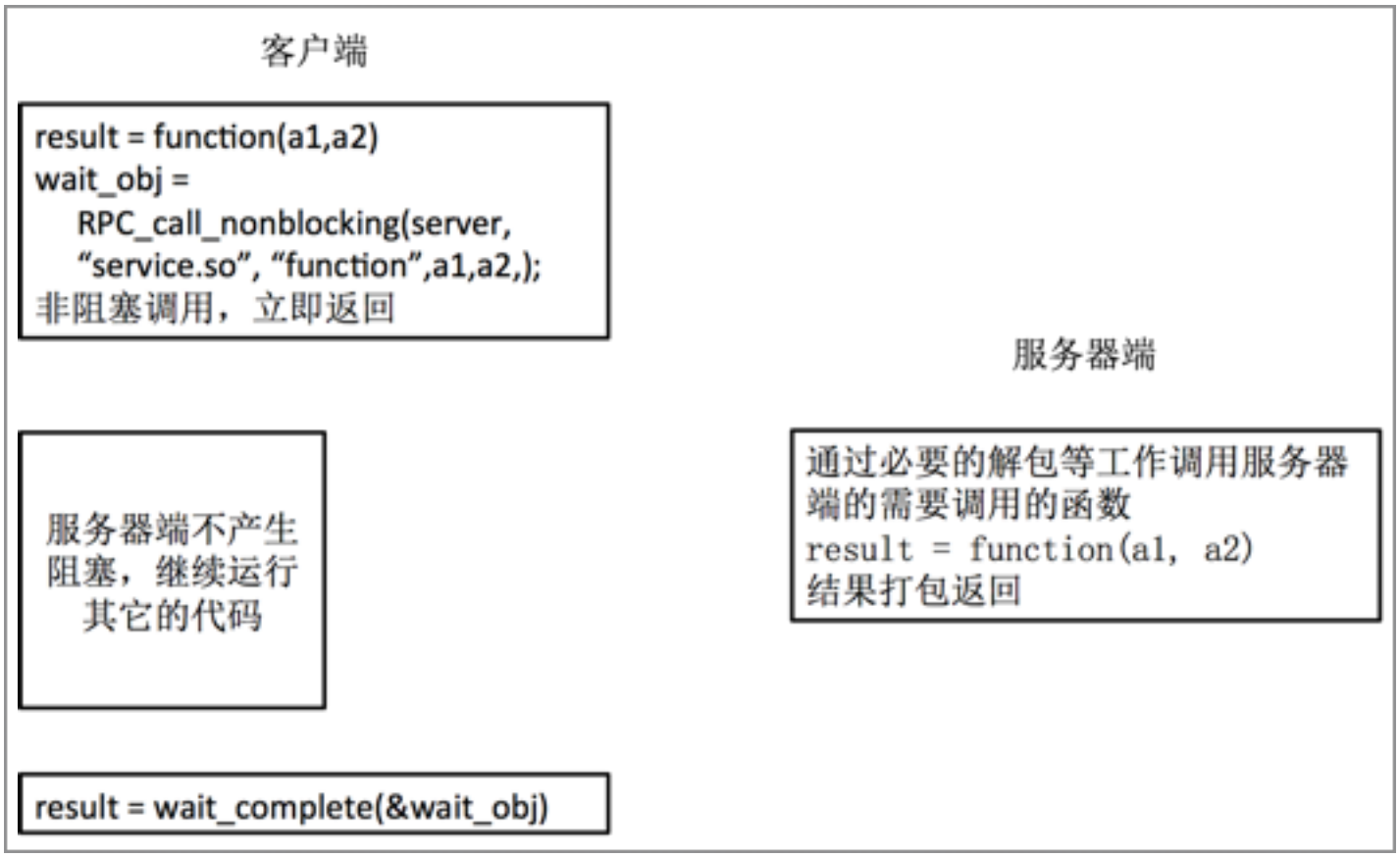


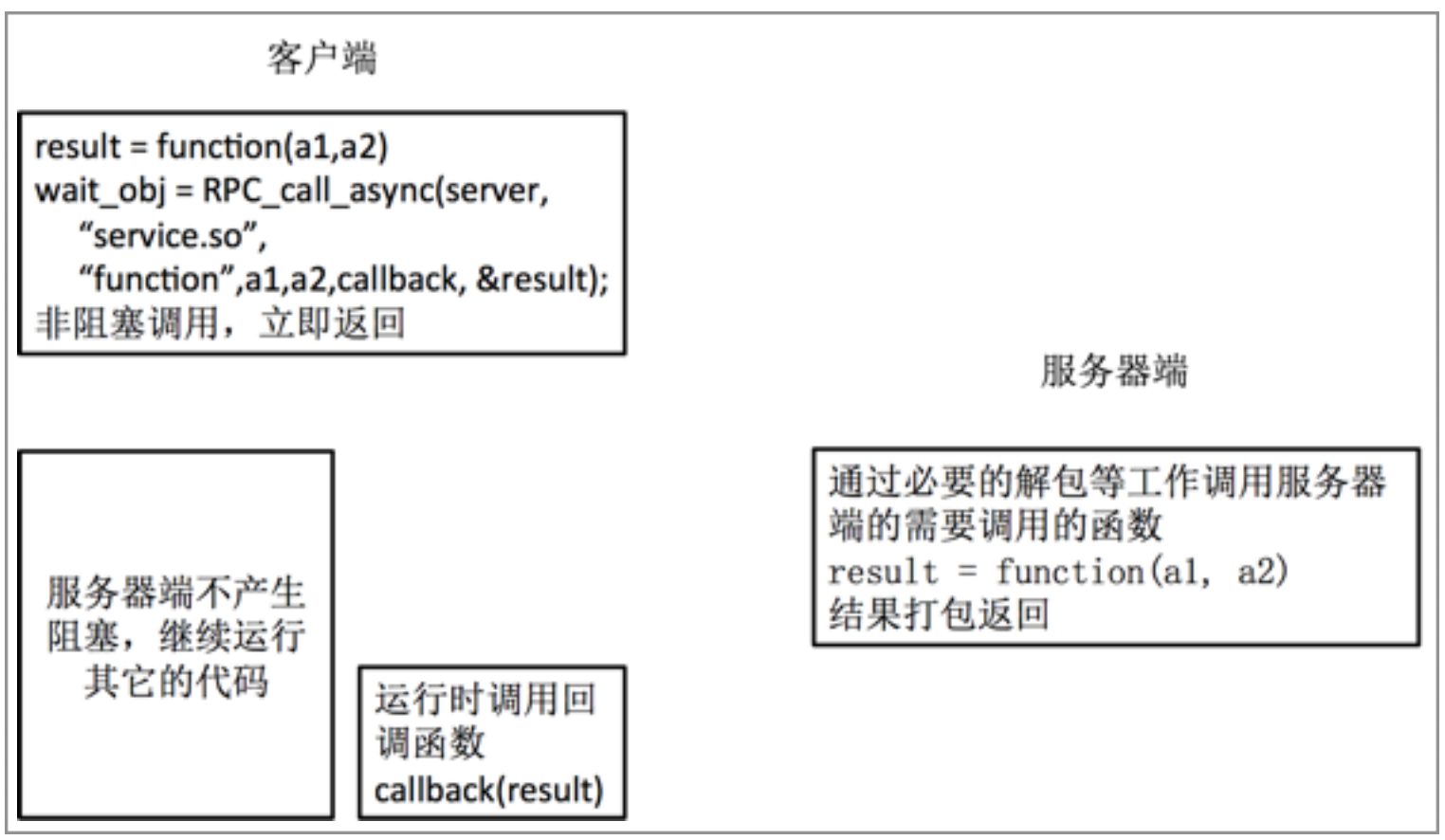
图2.17 应用程序自己控制的远过程调用

在于现在远过程调用不再是对应用程序透明的过程，而是对于应用程序上来说，它已经知道自己的函数调用过程中的一些细节，即可能会出现跨进程跨节点的调用。但是，对于提高性能来说，这种透明性的缺失往往还是值得的，也是通用的处理手段。

先来看一下由应用程序自己控制进行的异步远过程调用的流程，其直观的执行流程如图2.17所示。图中可以看出，客户端在执行的过程中，会发出一个远过程的调用。但是在发出这个远过程调用的时候，客户端不会等待函

数调用的功能返回（在这个时候，客户端不期待发出的执行过程已经完成），而是在函数调用返回的时候，进行下一步的处理。这些进一步的处理是不依赖于发出的远过程调用的结果，因此可以在结果还未出来的时候可以进行处理。之后，程序会在某一个同步点去询问底层的远过程调用的运行时的库，即需要进入一个同步点，在这个同步点中去确定等待远过程调用的结果。此时，因为下一步的操作需要依赖这个结构，因此程序除了等待不会进行其它的处理，在获得远过程调用的结果之后再进行下一步的处理。可以看到，这种调用方式能够使得应用程序在等待网络返回的时候做其它的一些处理，而不需要一味地进行远过程调用的等待，从而可以将计算和网络数据传输的时间重叠在一起。与此相类似的方式是可以使用多线程或者多进程处理的方式，能够在某一个线程进入到同步远过程调用执行的时候，切换到另外

图2.18通过回调函数去执行远过程调用



一个线程去进行处理，也能够达到计算与数据传输重叠的目的，但是需要加入一些上下文切换的开销。

上述的异步远过程调用应用程序实际上并不会在函数调用返回的时候得到通知，即进入同步点的时候，要么函数调用的结果值已经返回，要么还需要在同步调用函数中等待函数的返回值。由于远过程调用涉及到网络的处理，返回结果时间是不确定的，因此上述的两种情况是不可避免的。为了能够得到及时的通知，使用回调函数的方式可能是不可避免的。当然，可以采用其它的方式，例如使用信号等向应用程序发出通知，但是使用回调函数是一个更加自然的方式，并能够得到及时处理，而不需要在应用程序中得到通知的时候需要从原有的程序中退出来进行处理。通过回调函数进行远过程调用的流程如图2.18所示。

从图2.18中可以看出，在这种模式下，用户发出远过程调用的时候，就无需再去关心这个函数调用，因为应用程序知道之后的调用将由运行时来管理。在服务器完成函数调用过程之后，结果将返回给服务器端的远过程调用运行时，此时运行时知道这个远过程调用使用的是回调函数的方式，因此，在获得调用结果的时候就在应用程序的上下文调用用户预先定义的回调函数，完成相应的处理。

远过程调用对网络容错的处理

虽然远过程调用试图去避免网络带来的不透明性，争取将网络屏蔽在函数调用的背后。但是，实际上这一点是非常困难达到的。还是以result=function(a1,a2)这样一个远过程调用为例。在这个函数调用的形式中，result实际是function函数功能的返回值，包括执行的功能是否完成，完成到哪种程度，如果失败了，那么是什么原因。这些返回值都是与function函数的本身的实现有关的。但是，如果function函数是在远过程调用的环境下执行，那么实际上还需要处理网络出错的情况，即在result里面需要包含可能出现的网络的出错的情况。下面就讨论一下在网络中出现错误的时候，远过程调用函数的实现方式。

在这里，困难的地方在于，远过程调用可能在任意的位置都能够发生错误。在这里考虑一个简单的情况，即网络发生了丢包和重传的情况。在这样的情况下，观察一次RPC的调用过程，特别是观察服务器执行这个函数的前提条件。

第一种也就是最简单的情况就是一旦服务器接收到了请求，那么就立即执行客户端请求的功能。在这种情况下，客户端和服务器端的远过程调用运行时逻辑最为简单，即无需做特定的控制，一旦接受，就立即执行。如果网络与服务器正常的话，预想这种工作方案是也是效率最高的。

在这种情况下，考虑一下出现错误的情况。从客户端来说，依据底层的RPC的运行时，客户端可以获知调用没有成功，但是这个返回值只是表明在返回的时候，或者发送的时候可能出现了错误。无论是发送还是接收出现的错误都不能够表明服务器端的状态，即1）服务器端可能没有收到消息，2）服务器端可能收到消息之后尚未来得及执行函数，3）服务器端可能收到消息之后完成了执行但是在消息返回出现错误。

从客户端来说，这三种情况是无法区别的。如果出现错误的话，客户端通用的处理方法是进行重传，希望能够以重传的方式来完成远过程的调用。但是，这种重传的机制可能会出现问题，即一个函数从客户端来调用来看是进行了一次调用，但实际上在服务器端执行了多次。下面是可能的一个过程：

1）客户端调用result = function(a1,a2);

2）服务器端收到了请求信息；

3）服务器执行了请求；

4）服务器返回结果，但是由于网络错误，结果被网络丢失；

5）客户端发动重传机制，再次发出请求；

6）服务器端收到请求，会再次执行同一个函数调用。

上述场景下，客户端启动一次远过程调用，但是在服务器端会对同样的调用执行两次函数。在某些情况下，这种执行流程是可以接受的，但是在大部分情况下，这样的执行流程是不可接受的，其中一个典型的情况就是服务器端的加锁操作。如果对一个锁进行两次加锁操作，就会发生死锁，这种情况下，执行函数两次要比不执行的情况严重很多，因为不执行的话不会有死锁的发生，而执行两次则会发生死锁。不发生死锁能够让应用继续执行，而死锁的情况下应用程序无法继续执行。

从上述的讨论过程中可以看到，在对网络错误进行处理的时候，有可能对应用程序的正确性产生影响。为了应对这个问题，上层的应用程序需要进行特殊的处理，以避免出现应用逻辑出现矛盾的地方。但是在上述的请求一次就进行可能超过一次调用的情况下，读者可以看到，应用程序在进行相应方面处理的时候还是非常困难的，因为其不可避免的会造成服务端函数的多次执行。这就需要应用程序在每次调用之情都需要查验调用是否已经执行过了，如果没有执行过的话才发起一次执行。应用程序在这里可以采用的方式是对每一个函数调用进行标记，不同的函数调用具有不同的标记，并且需要应用程序的服务器端也对这些标记进行记录。这样的处理需要应用程序自己去处理，会降低远过程调用的透明性，并给应用程序的编写带来负担。

从上述加锁的例子来看，上述进行服务器端代码调用例如加锁解锁等的操作是一个普遍的操作。对于大部分的应用程序来说，其发出去的请求从客户端的角度来看，并不希望在服务器端多次执行。对于应用程序来说，一个更好的远过程的服务是能够保证同样一个调用请求最好在服务器端只能够执行一次，而不是多次，这样对于客户端来说也方便处理。当然，客户端也知道执行的环境是网络跨进程的远过程调用，不太可能每次调用都能够成功，因此一个更加合理的语义是执行0次，或者1次调用，或者说是最多一次的语义at-most-once。为了实现这种语义，实际上需要做的工作是把之前应用程序需要处理的工作移动到底层的远过程调用的运行时系统中。在运行时系统中需要标记每一次的调用。这就需要客户端的支持代码需要进行记录，并依

照次序将每一次的远过程调用进行编号，并在调用的过程中发送给服务器端。服务器端需要记录每一次调用的请求，如果发现了重复的请求，这说明是底层的网络数据重传机制对同一个调用进行了重传，导致重新调用，此时服务器端不应当继续去执行这个调用，而是直接将对应的执行的结果返回。因此，可以看到，每一次系统调用完成之后，都需要记录调用的执行结果，以便在相同的远过程调用发生的时候，能够将同样的一个结果返回给客户端。这里就会产生一个问题，即这种记录结果的行为会发生很多次，每一次发生远过程调用都会记录结果会造成运行时系统极大的负担。因此，服务器端需要做的工作是在执行的过程中，将不可能被客户端再次请求的远过程调用结果删除。当然，为了确认一个结果再也不会被客户端所请求，这也需要客户的配合来完成。具体的机制类似于TCP协议中的滑动窗口协议，客户端每次将自己的已经收获的确认的最大的远过程调用号发送给服务器端，而服务器端则依据此信息删除不必要的远过程调用的结果。

思考题

1 设计一种场景，能够同时出现死锁和活锁的情况

参考文献

系统编程，Linux，UNIX，Windows

网络编程

计算机网络

第 3 章

分布式文件系统

从本章开始将进行具体的分布式系统的研究。当然，实际的分布式系统是非常庞大与繁杂的，不可能一个一个举例说明，本书只能选取其中重要的分布式系统进行研究。与一个单机系统一样，一个可用的系统一般都需要建立一个存储系统用以存放数据。这在单机系统中表现为本地的文件系统以及数据库等，同样的，在分布式的环境中需要建立分布式文件系统或者分布式数据库用以存放数据。在这个基础之上就是对数据进行处理，关键一点是如何针对分布式的环境进行编程。这里对于计算与编程方面的考虑就类似于单机系统中的各种语言的编程环境与运行时。在分布式系统的环境中也需要这样的一个环境来支持应用程序的执行。

本章的重点内容是分布式文件系统，并通过 Google File System 文件系统这个具体的例子来帮助读者去理解分布式文件系统的各个方面。本章不会去描述这个具体文件系统的细节，而是从原理性角度来帮助读者去理解分布式文件系统的各个组成部分。本章包括以下的内容：

1）传统的文件系统回顾： 分布式文件系统是本地文件系统的扩展，共享了一些基本的架构以及处理方式，读者可以将分布式文件系统与本地文件系统的架构进行比较，以加深对这部分内容的理解。

2）简单的分布式文件系统NFS以及AFS：这两个文件系统是对单机文件系统的直接扩展，也是一种分布式文件系统的架构形式，将文件系统导入到分布式的环境中。

3）大规模数据存储的分布式文件系统：这一部分的讨论将带领读者研究如何设计一个用以存储大量数据的分布式文件系统，具体的例子就是 Google File System 这个分布式文件系统。需要注意的是这里的重点是各方面设计的背后的逻辑关系，而不是对与模块的功能性阐述。

读者除了需要关注分布式文件系统的设计与实现之外，也需要观察在设计和实现一个分布式系统的时候所需要进行考虑的固有特征。这就包括的系统的负载均衡，容错能力以及系统数据的一致性等。

3.1

文件系统的回顾与讨论

本小节首先回顾一下文件系统的基本概念，包括名字空间的组织，标准虚拟文件系统接口，分块大小，可靠性等。而后讨论在分布式环境下文件系统的组织，指出分布式文件系统是可以建立在本地文件系统之上的，完成的进一步工作是对资源在具体节点上的定位。最后讨论两个典型的分布式文件系统，包括NFS文件系统以及AFS文件系统，这两个分布式文件系统是对本地文件系统的直接扩展，与本地文件系统区别不大。无论是什么样的一个文件系统，其本质的功能就是为了定位数据，即可以通过文件名的方式找到对应的数据。这一点读者是必须要记住的。

磁盘的工作方式与文件系统的提出

文件系统是操作系统中一个非常重要的概念，可以说现代的操作系统都带有一个或者数个的文件系统的实现。文件系统本质上是一个翻译的程序，将上层的逻辑文件系统语义翻译成底层存储的原语。

文件系统的底层建立在磁盘接口的基础之上。在典型的硬盘结构中，磁盘中的数据通过磁头进行读写。在具体的操作上，需要经过定磁头寻找合适的盘面，寻道合适的磁道，旋转到合适的扇区等几个步骤，最后再进行读写。这是一个非常复杂的电器过程。好在现在硬件的厂商将这样的过程简化为磁盘块的操作，可以通过块的编号来去定位磁盘上的数据。虽然这已经进行了简化，但是直接操作磁盘的方式仍然非常复杂，对于应用程序员来说是不友好的。应用程序员如果直接使用磁盘的话，需要考虑许多与应用程序无关的因素，不方便使用。为了解决这个目的，文件系统就被提出来。

现代的文件系统向上（用户或者应用程序）都提供两个方面的接口，一个是树形的文件系统架构，使得用户可以通过目录树的方式来浏览文件系统中的文件，另外一个是文件的读写接口，可以对文件数据进行读写。

因此，文件系统是沟通上层的逻辑接口与底层的物理接口之间的桥梁。文件系统从中实现了翻译的工作，将用户层的文件系统语义逻辑翻译为对应的数据块的读写操作。

图3.1 磁盘与文件系统在系统中的位置

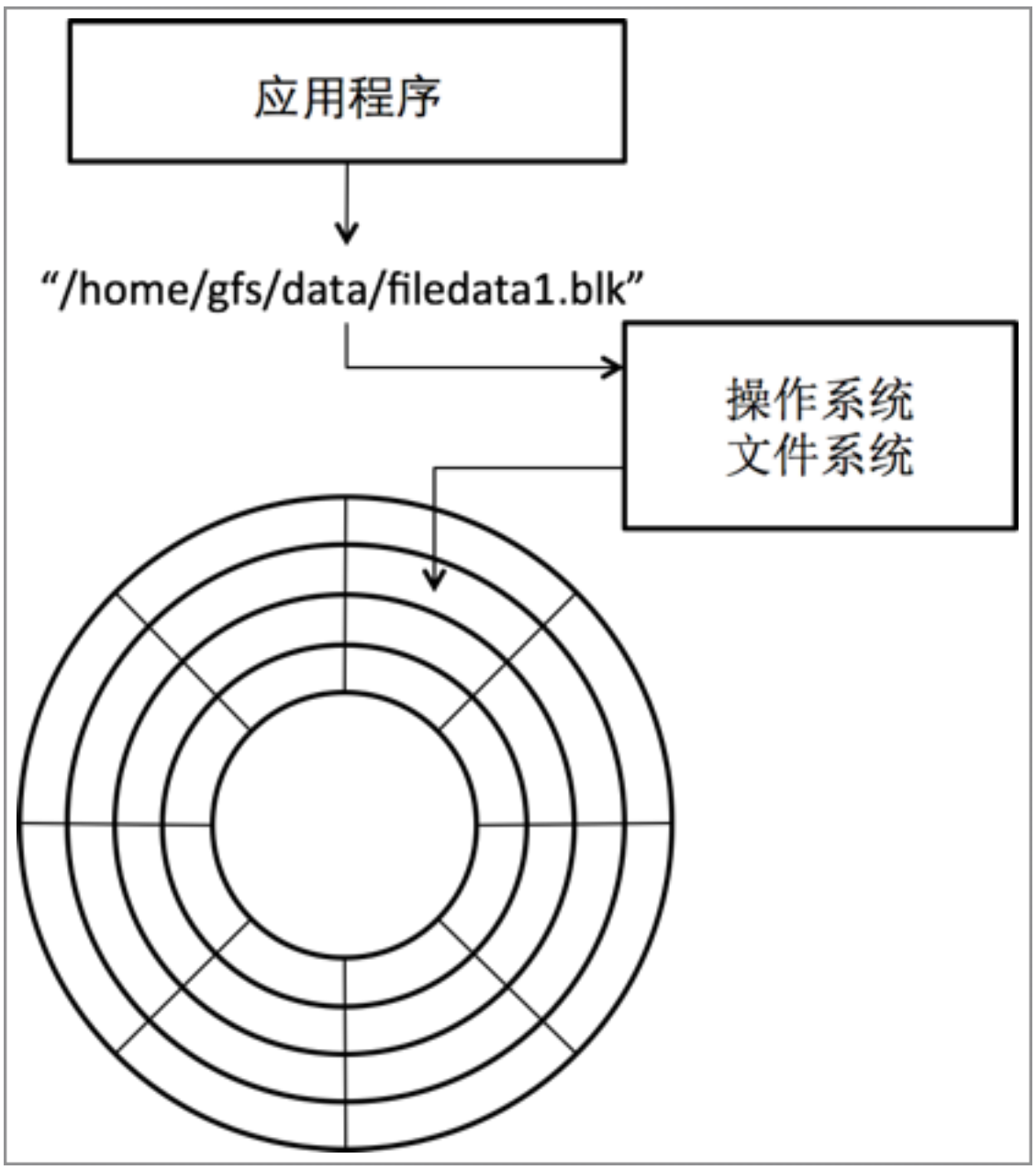


图3.1是文件系统的典型工作方式。文件系统完成了从上层的名字空间到底层具体物理位置的翻译工作。因此，从本质上说，文件系统是一个字符串到一个长整数的映射，file_name(type:string)==>address(type:long)。完成映射

解析之后，就可以通过具体的数据读写接口来读取或者写入数据。依据文件系统这个映射表，使得上层应用程序可以通过字符串而不是需要一个地址就能够对文件内容进行读写。应用程序员可以通过一种抽象的方式来使用磁盘，而不是与具体的磁盘数据块定位打交道。文件系统中一个最重要的概念就是命名空间，即这个文件名所在的空间是如何定义的。

文件系统的名字空间

通常看到的文件系统都是按照目录树的形式组织的，这样的组织形式往往被称为一个名字空间，或者命名空间（namespace）。图3.1中可以看到一种典型的文件系统文件的命名。通过名字空间目录树的方式，无论是应用程序还是最终用户都非常容易去定位一个数据对象。这个数据对象就是按照一个字符串表达出来的在名字空间中的一个具体的对象，例如在Windows的文件系统中，一个典型的全路径文件名有如下的字符串形式：“C:\Program Files\Google\bin\GoogleDesktop.exe”。从文件系统的上层语义来看，这就是具体的最终的数据对象在文件系统中的位置。但是，实际上从这个字符串来说，并不能表达出数据的真正物理存储位置，具体的存储位置应当是一个长整数，而不是一个字符串。这是一个抽象虚拟的位置，供上层的应用程序以及用户使用，屏蔽了具体的物理位置。文件系统将这个字符串作为输入，从文件系统的元数据中可以找到这个文件对象真正所在的物理位置，从而可以去找到真正的数据。

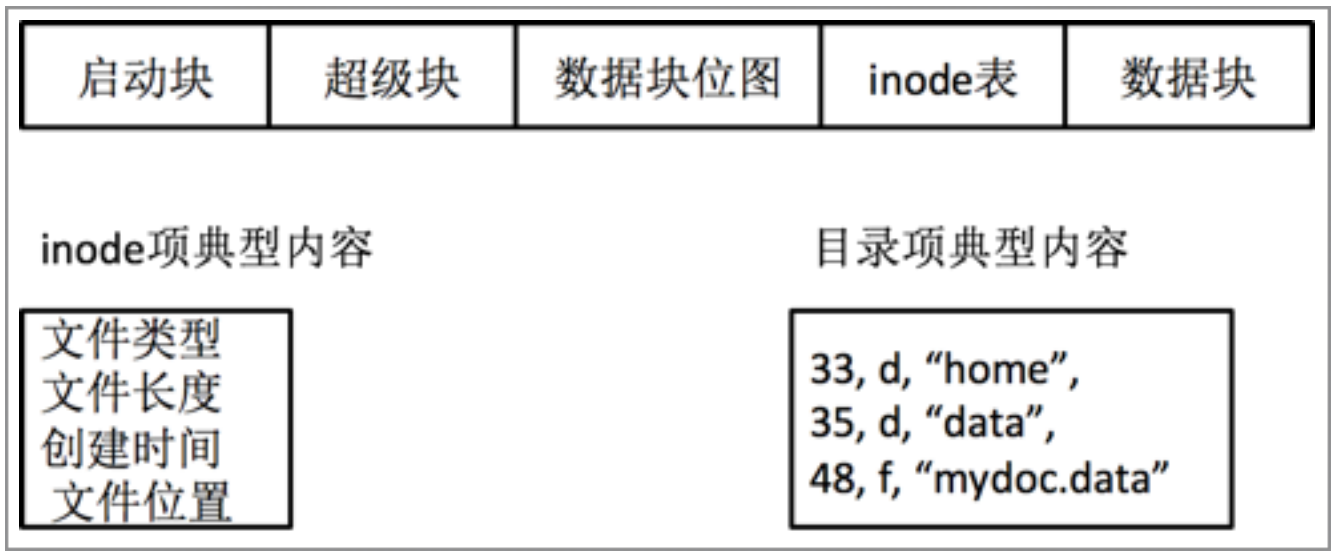
通常在看文件系统的时候，不同的文件系统可以看做是不同的名字空间。在同一个名字空间中，通常都有一个文件系统的“根”，这是找到所有文件对象的通用入口。不同的文件系统通常被认为处在不同的名字空间中，它们之间是相互独立的。一般在一个操作系统中的文件系统只有一个名字空间，这样的话通过共同的根，应用程序就可以定位到这个名字空间中的所有文件对象。在操作系统的内部，实际上除了文件系统这个非常庞大的名字空间外，通常还具有其它的不同的名字空间，例如在Linux内部的进程间命名

管道等各种进程间通讯的接口也处在本身的名字空间中；在Windows内部也有一个另外的重要的名字空间，即注册表所表达出来的名字空间。由于处理对象的不同，虽然有的时候名字空间有可能被合并成一个，但是出于功能上的考虑，将其认为是不同的名字空间会更加容易理解。

磁盘文件系统

文件系统是操作系统的必要组成部分，因此，一般谈到文件系统的时候，首先看到的就是磁盘上的文件系统。磁盘文件系统由于其重要性以及具有的长期发展过程，一般认为其技术已经相当成熟。但是，实际上磁盘文件系统也在不断的演进与发展的过程中，加入了很多原先没有的功能。例如，当前的磁盘文件系统都都加入了诸如访问控制，快照，日志的功能。这些功能丰富了文件系统的语义，能够给操作系统以及应用程序提供更加丰富的控

图3.2 文件系统的磁盘数据结构组织



制。现代的文件系统例如Linux下的Ext2, Ext3, Ext4等以及 Windows 下的NTFS已经具有了非常丰富的语义内容。

但是，说到底磁盘文件系统的最基本的功能还是进行数据的定位，将上层的名字空间中的字符串内容定位到具体的磁盘数据块中。由于组织成目录树的形式，数据的具体定位过程实际上是分层进行的。图3.2给出了一个典型的简化后的磁盘数据结构的组织。

一个磁盘被格式化完成之后，最先的启动块以及超级块会记录一些启动的信息以及元数据的信息。数据块位图则指示了数据块被分配的情况。一个文件系统中最重要的就是索引节点表，即图中的inode表。在这个表中的每一个表项，即一个inode项代表了一个目录或者代表了一个文件。图中也给出了inode中包含的信息，其中最重要的信息就是文件的位置。如果一个inode代表一个文件的话，通过这个数值，文件系统就获知文件在底层磁盘所存放的物理位置，进而可以完成读写操作。如果一个inode代表一个目录，那么这个文件位置的信息所指向的磁盘块将包含一个目录项的数据，而目录项的数据要么包含所有的子目录的名字以及对应的inode编号，要么这个目录里面所有文件的名字以及对应的inode的编号。

这样的数据结构可以帮助用户或者文件系统完成名字的解析。通过一个共同的根目录入口（inode编号为0），可以解析一级子目录以及对应的inode编号。依据一级子目录的inode可以获得一级子目录的目录项，可以继续进行名字解析，直到解析完成，定位到最终的文件。最终文件的inode项目会记录文件在磁盘中的位置。

分布式文件系统

由于通常磁盘文件系统都是在一个单机的环境中使用，只能给本地的应用程序以及用户提供数据。在分布式环境下面，由于有多个服务器的存在，就有了构造分布式文件系统的需求，帮助不同服务器上的应用程序以及用户去访问其它服务器上的数据。

一般来说，分布式文件系统可以构造在本地文件系统之上。由于参与分布式系统的各个服务器通常都是能够独立工作的完整系统，包括硬件设备以及操作系统应用程序等软件。此时，分布式文件系统就可以构建在现有的本地磁盘文件系统之上，而不需要直接去操作某一个具体磁盘的具体的数据块。这种方式大大加快了分布式文件系统的开发与部署，屏蔽了底层硬盘以及系统软件的差别。这说明了在分布式文件系统的实现上可以采用应用层服务程序的方式，而不是内核的方式，依赖与本地的文件系统完成磁盘的读写。这大大降低了分布式文件系统的实现难度，提供更高的灵活性。

在分布式文件系统中最重要的概念仍然是分布式文件系统的名字空间。同样的，分布式文件系统的名字空间起到的作用仍然还是对上层的应用程序以及用户提供一个方便的访问接口。通常分布式文件系统的名字空间与单机文件系统的名字空间类似，也提供了目录树的访问形式。前面说过，名字空间的最终目的是进行地址的翻译工作。在分布式环境下面，分布式文件系统的名字空间起到的最重要的作用也是进行地址的翻译。此时，名字空间对于多个服务器来说可能是相同的，多个服务器能够访问相同的名字空间。这与单机的名字空间不同，一般单机的名字空间只能给对应的服务器系统使用，两个不同的服务器的文件系统名字空间是相互独立的。在具体到文件数据定位的时候，分布式文件系统比单机的文件系统的所作的额外工作就是定位对应的服务器。这是由于分布式文件系统一般都构造在本地文件系统之上，只要定位到某一个具体的服务器，余下的工作可以交给非常成熟的本地文件系统去做。因此，可以说分布式文件系统的最重要的工作就是将名字空间中的字符串所对应的文件数据对象定位到具体的服务器中。这是最重要的分布式文件系统的元数据管理工作，不同的文件系统具有不同的做法。下面就介绍几种不同的分布式文件系统，及其对应的名字解析方法，如NFS，AFS以及Google文件系统的不同方式。

文件系统与分布式文件系统的参数特性

随着文件系统技术的不断发展，附加在文件系统上的概念也越来越多，对于文件系统来说也有很多不同的参数选择。本节介绍几个常用的描述文件系统特征的相关概念以及文件系统内部所具有的一些参数特征。不同的文件系统对于这些参数可能会有不同的选择。在后续的讨论中，也会讨论在分布式文件系统中对应参数的选择。

文件数据块的大小（Data Block Size）：文件数据块的大小是极其重要的一个参数。在单机文件系统中，文件数据块大小描述了保存在文件系统中最小的数据读写单元。在分布式文件系统中，数据块通常是表示进行元数据管理的一个最小的数据对象。作为文件数据的最小读写对象，数据块的大小影响了文件系统的多个特性。一个是读写的效率。由于数据是存放在磁盘上的，如果一个比较大的文件存放在磁盘的不同的磁道上，此时如果数据块太小，存放的位置比较分散，这就会很严重影响文件的读写效率。磁盘数据的随机读写要比顺序读写慢一个数量级以上。如果使用比较大的数据块，则能够达到顺序读写的目的，提高读写的效率。但是，如果数据块的大小过大的话，会严重影响磁盘空间的使用效率。每一个文件必须要起码占用一个数据块的大小，即使其内容只有1个字节。这个时候，如果在系统中存在大量的小文件的话，会严重浪费磁盘空间。另外，对于元数据管理来说，更大的数据块会减少元数据的数量，而更小的数据块则会增加元数据的管理负担，这对于分布式文件系统来说也是一个很重要的性能考虑。因此，出于不同的目的，不同的文件系统会选取不同的数据块的大小。通常的单机文件系统可以选择不同的数据块的大小，例如在NTFS中，用户可以选择将文件系统格式化成512字节，1KB，2KB，4KB或者更大的不同的数据块大小，其它文件系统也可以做相应的选择。

文件系统的可靠性与效率（Reliability and Efficiency）：可靠性与效率是当前文件系统的一个设计的重点。在完成文件系统名字空间到具体物理位置的功能性工作之后，不同的文件系统都会将设计的重心放在系统的可靠性和效率的设计上。

文件系统的可靠性在于如何处理可能出现的各种错误情况，使得在出错的情况下文件系统仍然能够进行正常工作或者降低错误造成的影响。文件系统面对的错误情况其实是很多的，从硬件的角度来看，磁盘上的磁道很有可能出现坏道，文件系统需要避免在坏道上进行数据读写。另外，整个磁盘也有可能由于掉电的原因突然停止工作，那么在进行重新启动的时候，设计上不进行可靠性考虑的文件系统非常有可能也遭到了破坏，无法继续工作。在分布式文件系统中需要面对更多的出错的情况，例如磁盘或者网络的工作状态都会影响分布式文件系统的可靠性设计，必须要考虑到服务器或者网络出现错误的情况。

文件系统的效率主要是指读写效率，即在进行数据读写的时候需要充分利用硬件的性能，不能由于文件系统本身的设计问题，消耗过多的资源，影响整个系统的运行效率。文件系统的效率通常与文件系统的可靠性会产生一定的矛盾。例如，为了提高文件系统的读写效率，很多情况下会利用到应用程序的时间局部性以及空间局部性，将数据内容缓存在内存中。这样的话，读写内存中的数据要远远快于去硬盘中进行读写。但是，这会降低整个文件系统的可靠性，因为如果出现掉电或者应用程序出错的情况下，数据还没有写入到磁盘，整个文件系统的数据就会出现不一致的情况。因此，在设计文件系统的时候，要同时兼顾到系统的可靠性与效率，在保证可靠性的条件尽可能提高整个系统的性能。在这两个方面不能够同时满足的情况下，系统也需要进行一定的取舍，如果有应用程序的语义的话，就能够帮助文件系统进行机制的选择。

文件系统的缓存（File System Cache）：在现代的文件系统中，文件系统的缓存是一个非常重要的机制。由于磁盘的机械特性，只能够进行旋转以及定位进行读写，读写速度受到了很大的限制，而内存是使用电路进行读写的，效率要比磁盘高数个数量级。因此，一般文件系统中一个非常重要的内容就是进行文件系统数据的缓存。在计算机系统中，通常的计算与数据读写具有局部性，包括空间局部性（访问到的数据其相邻的数据非常有可能在不

久的将来受到访问）和时间局部性（访问到的数据的本身也有可能在将来受到访问）。缓存的手段就是将经常访问的数据放置在内存中，这样的话大部分的数据访问可以在内存中进行，提高了读写的效率。

虚拟文件系统接口与POSIX语义

由于文件系统的重要性，并且几乎所有有用的应用程序都会使用到文件系统，这就涉及到文件系统访问接口的标准化问题。虚拟文件系统是类似UNIX操作系统的一个非常重要的概念抽象。虚拟文件系统并不是一个实际的文件系统，而是定义了一套接口语义。虚拟文件系统规定了一个文件系统所应当实现的接口规范。

图3.3 标准虚拟文件系统接口

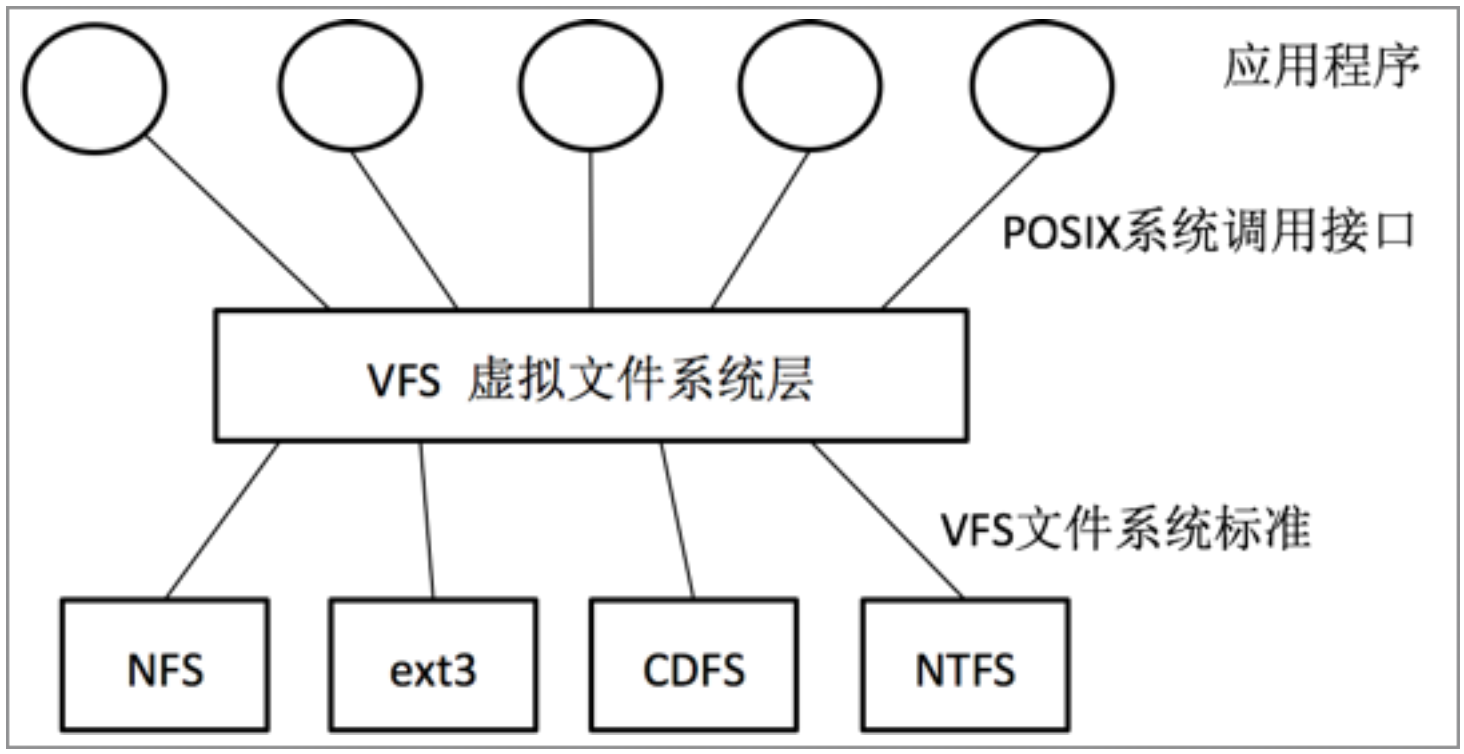


图3.3反应了虚拟文件系统的接口示例情况。在整个系统中可能会涉及到不同的文件系统实现，例如在Linux环境下，有针对磁盘的Ext3文件系统，针对光驱的CDFS文件系统个，有针对移动闪盘的FAT文件系统。这些文件系统都针对不同的介质进行数据组织。但是，如果每一个文件系统都提供不同的上层接口的话，就会造成应用程序以及系统的困扰。例如，一个文档处

理程序可能需要处理在不同介质上的文档，将文档从光盘中读出，进行编辑后写入到硬盘中。如果针对每一个文件系统都使用不同访问接口的话，整个应用程序的构造将会变得非常麻烦。幸好这些文件系统实现了虚拟文件系统的接口，所有的应用程序看到的实际上是同一个访问接口。应用程序只需要用标准的接口去访问文件系统即可，在虚拟文件系统的内部会根据不同的文件目录与文件名将命令转发给不同的文件系统去执行操作。通过虚拟文件系统的方式，屏蔽了底层硬件文件系统的不同，方便了整个系统的构造。虚拟文件系统也能够通过标准的方式在系统中安装不同的文件系统。而POSIX文件访问语义则标准化了应用程序编程的时候对数据空间的访问。通过这种编程方式使得应用程序具有了兼容性，能够运行于不同的文件系统，而无需考虑底层文件系统的细节。

文件系统的标准化以及标准访问接口带来了极大的方便性，这也是系统发展的过程中取得的重要贡献。但是，在某些情况下标准化的访问方式会带来其它的一些问题。例如，标准化的接口在满足程序能够运行于多样性环境的同时，也失去了特定系统所能够提供的特殊的功能。这里面最重要的体现一点是对系统的性能影响上。由于提供了标准的语义，那么在一些需要更加简化语义的环境下，会造成性能的损失。如果能够提供更加简单的访问接口的话，程序可能会获得更好的性能。在这里体现的情况就是如果一个文件系统能够满足所有的应用程序的话，那么很有可能对所有应用程序的满足都不“足够好”。在系统设计中经常会出现这种矛盾的选择，这就需要针对不同的应用做出不同的选择，有的时候甚至需要自己去设计合适的接口。这一点在Google File System中体现得特别明显，将在后面进行讨论。

思考题

思考题1：

假设在一个3000个节点的集群环境下运行一个类似于 Google File System 的分布式文件系统。为了保证可靠性，假设每一个文件具有3个副本。另外，有一个文件由3个数据块构成，即数据块A，数据块B以及数据块C。每一个数据块的大小为64MB。那么：a）对于某一个数据块A来说，如何保证3个副本都是一样的？b）一个写入操作x会写入到数据块A，数据块B以及数据块C，另外一个写入操作y写入到数据块B以及数据块C。这两个写入操作并发进行，列举出这两个操作都正确完成之后（在此期间没有其它的写入操作），所有可能的结果。

思考题2：

在 Google文件系统中，数据块是在多个系统之间进行副本复制以提高可靠性的，那么必然会带来一致性的问题。依据你的分析，描述在如下的过程中，文件系统是如何保证数据块的一致性的。

客户端从元数据服务器（ master）中获得了数据写入的位置，在这个时候出现了错误，分析下面两个独立的场景，分布式文件系统需要做何种处理。

（场景 1）考虑其中一个Chunk服务器出现错误，整个系统将会发生什么事情；

（场景 2）考虑这个时候客户端出错了，数据在写入完成之前客户端失败了，那么服务器端需要做什么事情继续维持数据的可靠性和一致性。

思考题3：

举例说明一下 Google 文件系统可能产生垃圾的情况，并设计分布式文件系统中的垃圾收集算法。

思考题4:

文件系统的快照是文件系统在某一个时间点的状态，快照能够用来恢复文件系统之前的一个系统的状态，用以恢复数据或者恢复误操作之前的状态。请阅Google文件系统的论文，提出你的分布式文件系统的快照算法。

第十八章 拜占庭错误一致性协商

分布式系统的一致性问题（Consensus）是分布式系统以及分布式算法的一个基本问题，在理解分布式系统中具有重要的地位。之前已经讨论了系统中存在停止错误情况下的一致性协商问题如何解决，即Paxos算法。本章继续研究这个问题在出现任意错误，即可能存在故意破坏者的情况下如何达到一致性协商。

在第一章中已经讨论了分布式系统中的出错模型，本章的内容将不再限制出错的类型。可能存在的最坏的情况是系统中的一部分节点会被一个攻击者所控制，那么这部分节点会联合起来去破坏一致性协商的过程。再次需要说明的是正确节点由于不知道对方的一个节点是否是错误还是正确的，因此不能进行协作。每一个正确节点只能根据本地的状态以及收到的消息做工作，并严格遵循相同的协议（即算法的流程）。

本章算法的过程与Paxos算法类似，也是通过投票的方式进行，并从大多数人给出的投票中进行推断系统的总体流程。下面的内容将从简单的算法开始，逐步建立能够完成一致性协商的过程，大致的思路是从最简单的情形开始，建立一些不完善的算法流程，并逐步进行修正，以获得最后的算法。算法的过程比较复杂，不太容易第一次就阅读明白，读者可以在有了一个大致印象之后，再次进行阅读，并且把细节填写清楚，可以完成对于算法的理解。

拜占庭一致性协商问题的描述

拜占庭将军问题与分布式一致性协商协议

Lamport 以一种有趣的方式描述了这一个分布式环境下的一致性协商问题（Consensus）。与描述Paxos算法一样（拜占庭问题描述得更早一点），Lamport通过讲故事的方式描述了这个问题。拜占庭帝国就是东罗马帝国，拜占庭即现在土耳其的伊斯坦布尔。拜占庭将军问题描述的是拜占庭军队攻占城市的场景。将军们驻扎在城外，每一位将军指挥一支部队。将军们通过通信员进行通信，共同决定是否进行攻击，即协商出一个结论，要么是攻击，要么是撤退。将军们之间通信通过信使进行传递。但是，这些将军中有叛徒，叛徒试图去阻止忠诚的将军达到一个一致的决定，这样某一部分忠诚的将军会决定攻击，而另一部分忠诚的将军会决定撤退，行动就会失败。信使的传递也不是可靠的，可能会迟到，可能半路就被叛徒的将军捕获了，导致信息不能送达。拜占庭将军问题就是为了解决在这样的条件下如何设计协议，使得忠诚的将军们能够达到一致的结论。当然，也需要解决有多少的叛徒就能够使得这样的一致性协商无法达成。

从上面的描述中就可以看出，拜占庭将军问题与分布式系统中的一致性协商协议具有紧密的联系。在这里，每一个将军就代表了一个进程，叛徒就是试图去破坏协议达成的攻击者。进攻或者撤退就是进程之间希望达到的下一步的操作值。而信使则代表了信道中消息的传递的情况。

同之前讨论的没有恶意节点的情况下的分布式一致性协商协议类似，下面将讨论逐步构造出在实际工作中可以采用的拜占庭问题投票的解决方案。这里的解决方案来自于这篇论文：Miguel Castro and Barbara Liskov, Practical Byzantine Fault Tolerance, OSDI1999。

论文给出的方案是通过投票方式来让一组节点都同意一个相同的操作，因此在基本方法上与Paxos算法类似。投票方式解决拜占庭问题的一个想法就是通过一组相互联系的进程对于一个提议进行投票，如果有足够多的进程同意了某一个提议，那么即使是有恶意节点进行捣乱，也会有足够的正确的进程来达到一致性的决定。下面的讨论将通过正确节点以及恶意节点来指代在系统中的参与节点，正确节点会严格遵循协议的流程进行计算，而恶意节点则希望通过互相协作来阻止正确节点达到一致。

关于拜占庭问题的一个理论的结果是这样的，即在系统中如果有f个恶意节点的话，必须有超过3f+1（包含）的节点才能够确保所有的正确节点得到一致的结论，也就是说正确节点必须有2f+1个或者以上。下面的讨论如无特殊说明，都是在这样极端的情况下展开讨论，即系统中有f个而已节点，2f+1个正确节点。

分布式系统的错误模型回顾

之前讨论的分布式系统面对的错误模型往往是停止错误模型，即在节点失效的时候就停止工作，或者最多再加上一个恢复的过程，节点在失效之后再重新启动，并按照算法的流程继续工作。错误恢复会依据保存在永久存储上的信息进行下一步的动作。本章对于错误的假设更加一般，节点将依据任意的错误方式进行执行，包括恶意节点之间可能相互串通来破坏算法的执行。在这种错误模式下，例如在Paxos下的失效节点变为恶意节点，那么这些节点即使是在最终提案被确定之后仍然会去响应一些新的提案，并进行投票动作。此时，其它的投票人就会依据恶意节点的响应继续工作，从而去破坏可能的已经被选定的提案。特别的是如果首领进程Leader是恶意节点的话，可以让一部分投票人投票同意一个提案，并且将相同（具有同样的提案序号）的提案提交给其它的投票人，但是具有不同的提案值。这个时候，Paxos算法便不能正确去选定一个提议。从另外一个角度来看，为了应对恶意的f个节点，Paxos算法的总数目2f+1个节点是不够的。恶意节点的错误模

型不同于前面所一直讨论的停止模型或者停止恢复模型。先来看一下在拜占庭错误下的恶意节点所具有的能力。

恶意节点的能力

正如前面所说的那样，本章算法的假设是恶意节点知道正确节点的相关信息，比如知道正确节点当前的状态，正确节点运行的算法程序等。另外也假设恶意节点是可以互相协作的，每一个恶意节点知道其余的恶意节点，恶意节点可以共同串通起来去愚弄正确节点。这种方式将恶意节点的能力最大化，在实际系统中也是可能出现的，例如一个攻击者通过主动攻击，获得了一些节点的控制权，将这些节点转变成为恶意节点。

恶意节点可以依据自己的意愿执行任意的算法来进行工作，包括去破坏一致性协商的结果或者去拖延一致性结果的达成。例如恶意节点收到请求后可能会完全忽略这个请求，自主决定自己的动作。恶意节点可能忠实响应某一些请求，因为它知道正确节点的算法。恶意节点可以用一个错误的值去响应请求。恶意节点甚至可以针对同一个请求给出不同的响应给不同的正确节点，以起到混淆的目的。另外，这里的讨论场景也是在副本状态机的上下文环境下。这在副本状态机中，恶意节点的某一些节点可以执行这些操作，而其它的则不去执行，或者执行一个错误的操作。恶意节点希望去破坏正确的副本状态机去执行相同的流程，从而达到使得协议失败的目的。恶意节点还可以对正确节点进行拒绝服务攻击。这种攻击可以使得某一些正确的节点不能对请求做出响应，或者延迟响应，可能会破坏一致性达成或者拖延一致性的达成。

从另一个方面可以看一下正确节点在拜占庭假设前提下的处境。正确节点只能依据自己的本地信息以及从网络中的其它节点发送过来的信息按照算法执行程序，进行自己的计算并依据算法来发送消息给其它的节点。正确节点是不知晓其它节点是否是正确节点的。这样的假设是合理的，因为正确节

点在与其它节点进行通信的时候，某一些恶意节点会假装依据协议的流程进行执行。这样的话，正确节点除了知道有恶意节点存在外，不知道任何信息，也不知道与自己通信的对方是不是恶意节点。这个假设也是必须的，如果正确节点知道其它的正确节点，只需要自身步调一致即可，就不存在拜占庭将军的问题了。

为了能够使得所有的正确副本状态机的每一个状态转换达到一致，对于正确节点所需要执行的代码来说，需要依赖两个事实，即所有正确节点都执行相同的代码，另外一个是恶意节点不会太多。这样两个事实就可以帮助使得正确的节点可以通过投票的方式获得同样的结果。

恶意节点的能力也不是无限制的，一个假设前提条件是恶意的节点不会占到大多数。更精确的，就是依据已经证明的拜占庭将军问题的下限，对于n=3f+1个节点来说，恶意节点不能超过f个，不能够更多，否则的话系统的一致性协商不能获得。另外，在下面的算法中所有的节点都依据自己的证书标明自己的身份，证书是基于密码学方式给出的，不可篡改（参考匿名网络那一章的密码学基本知识介绍）。恶意节点当然知道所有的恶意节点的身份证书，但是恶意节点不能够攻击密码学算法，这是当前理论中很难做到的事情，不能获得正确节点的密钥信息。这样的话，恶意节点不能够构造出一个正确节点的消息而不被发现。因此在后面的算法讨论中，读者需要明确的一点是每一个节点都通过密码学签名的方式来签名自己所发出的信息，恶意节点不能伪造正确节点的消息。

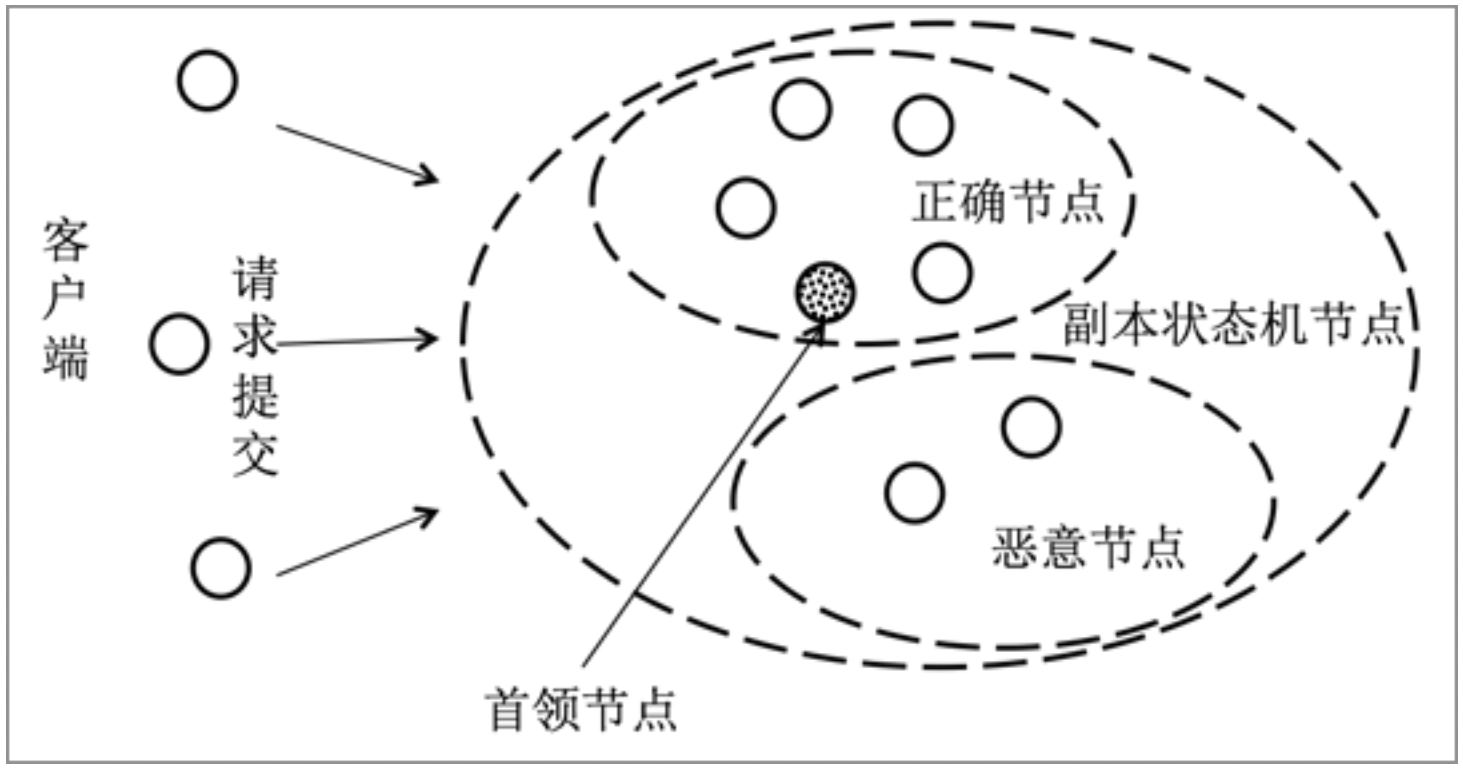
另外，为了实现在一个副本状态机下的一致性协商，还假设客户端是正确的，客户端不会去破坏副本状态机中的正确节点达到一致。这里隐含的假设是客户端不会去参与到一致性协商的过程中，而是只是去提供副本状态机所需要执行的请求指令。这种情况类似于Paxos算法中的提案人和投票人的关系，投票人决定所选定的提案，而提案人只提供提案的选择。

副本状态机与系统的总体结构

下面的讨论也是基于一个副本状态机来完成对于一致性协商协议的建立，这与Paxos算法类似。关注的一个重点是如何达到某一个特定操作的一致。另外由于副本状态机的关系，也需要保证所有操作在所有的正确节点中按照相同的顺序执行。为了保证这一点，也需要一些额外的工作，即协商每一步操作的同时还需要考虑多组操作之间的顺序。随着算法讨论的展开，读者会认识到这一点。

从副本状态机的角度来看，假设在系统中通过n个节点来保存副本，构成一组副本状态机。下面的讨论从简单的情况开始，逐步建立一套协议处理过程，以解决拜占庭问题。

图18.1 拜占庭一致性协商的副本状态机结构



下面的一部分讨论围绕着一个非常简单的副本状态机过程。假设在这个副本状态机中只维护一个状态值x，这样每一个副本都保存了一个x的副本。在操作上，首先将a值写入到x，之后写入b，最后读取x的最终的值。如果按照正确的操作顺序，正确读出的值为b。但是，在拜占庭错误模式下，有可

能不同的x的副本是不一样的，从客户端读取的角度来看需要某一种读取规则来获得正确的读取结果。

既然是基于副本状态机来完成容错的工作，不可避免的需要处理多个客户端同时将操作提交给副本状态机的情况。在Paxos算法中，所有的操作会提交给一个首领节点，首领节点会定出一个顺序，以便使得所有的副本都按照同样的顺序进行操作。下面的投票过程解决拜占庭问题，也使用了相同的方式来完成对于首领节点的使用，用以解决多个操作之间的顺序问题。当然，这里还需要特别防护的是如何去应对首领节点是一个恶意节点的情况，这也是算法的一个关键的地方。

图18.1是即将讨论的系统模型。在这个模型中，客户端会将请求发送给副本状态机的节点中。所有的节点组成了一组副本状态机的系统，这个系统中会选出一个首领节点用以协调并发写入操作。在这组节点中，有一部分的节点是恶意节点。在图中的首领节点被标记在正确节点的集合中，需要注意的是首领节点也有可能是一个恶意节点。

建立在拜占庭错误模型下的一致性协商

在讨论完成需要完成的目标以及系统的配置模型之后，下面逐步开始建立可以应对拜占庭错误的一致性协商算法。先从最简单的情形出发，分析每一个情形与假设的差距以及每一个情形解决方法的缺陷，最终获得一个完整正确的算法流程。

协议流程1：n个正确节点进行工作

先看一个如果不存在拜占庭错误，也不存在失效错误的情况下如何进行一致性协商。这个问题将会变得非常简单，客户端可以将请求发送给所有的副本状态机节点，并且等待所有的节点返回结果。图18.2给出了这样的一个算法的流程。

图18.2 完全正确执行的n个节点完成一致性协商

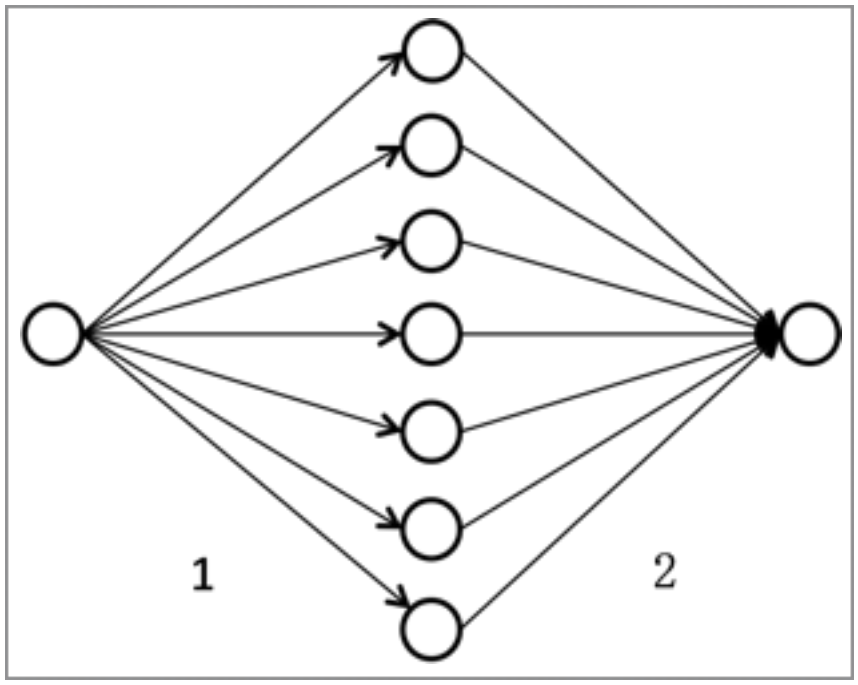


图18.2 中给出了这样的一个算法流程，即客户端在第1步将自己的请求发送给所有的节点，然后在第2步等待所有的节点返回请求。在有n个节点的情况下需要等待同样n个节点的响应，并且只有在这么多个节点所有的节点都

返回同样的响应结果之后才能够进行下一步的工作。当然，这样的操作对于单个的请求操作是没有问题的，对于多个并发的请求来说，需要首领节点进行协调。基本方式就是在出现并发的时候，首领节点去规定一下操作的顺序，基本流程还是这里所描述的提交和响应的流程。

当然这样一个最简单的设计，只能在所有节点都正确工作，网络也都正常工作的条件下才可以完成一致性的协商。任何一个模块出现错误，这里设计的算法都不能继续工作。对于拜占庭错误来说，恶意节点返回一个与其它节点不同的响应，那么客户端必然会知道这个不一致性，依据算法的设计，整个系统不能继续工作。

上述的算法的工作条件比较极端，因此实际上为了应对可能出现的恶意节点的情况，或者失效节点的情况，客户端实际上不能够等待所有节点的响应以及不要期望从所有节点中都获得同样的返回值。（从获得响应的数目来看，这一点实际上是与Paxos算法类似的，在Paxos算法中不需要等到所有的投票人的信息，只需要等待大多数即可。）在一个基于投票的系统中，由于会出现某些恶意节点，其行为不可预测，从所有的节点中获得一致的响应是不现实的。从所有的节点获得相同的结果这样一个预期就会导致哪怕一个节点不进行响应，整个协议的过程都会失效。因此，一个合理的预期是客户端如果能够从大多数的节点中获得一个一致的响应，那么可以认为这个一致的响应是真正执行的结果。为了保证系统存在一些恶意节点的情况下能够继续工作，客户端不能够等待所有节点的响应。

下面的内容对上面的设计进行改进，看看在出现恶意节点的时候，有什么样的信息可以利用，并对算法流程进行改进。

协议流程2：2f+1个节点，f+1个正确节点，f个恶意节点

由于上面的协议流程只是一个假设的场景，但是毕竟得出了客户端不能够等待所有响应的一个提示。可以对上述的场景进行扩展，假设一个类似于

之前讨论的Paxos算法的类似场景，观察是否会出现什么样的问题。在这里，系统的所有节点中，会出现f个恶意的节点，正确节点要比恶意节点多，那么正确节点至少有发+1个。在这个场景的配置下，实际上恶意节点只占据了一小部分，大多数的节点还是正确节点。现在客户端可以继续将请求发送给所有的副本状态机节点，但是不应该等待所有的节点给出响应，而是只需要等待大多数的节点给出一个一致的响应即可。正确节点将依据协议的过程严格执行，并给出正确的结果。

图18.3 大多数节点是正确节点的协议流程

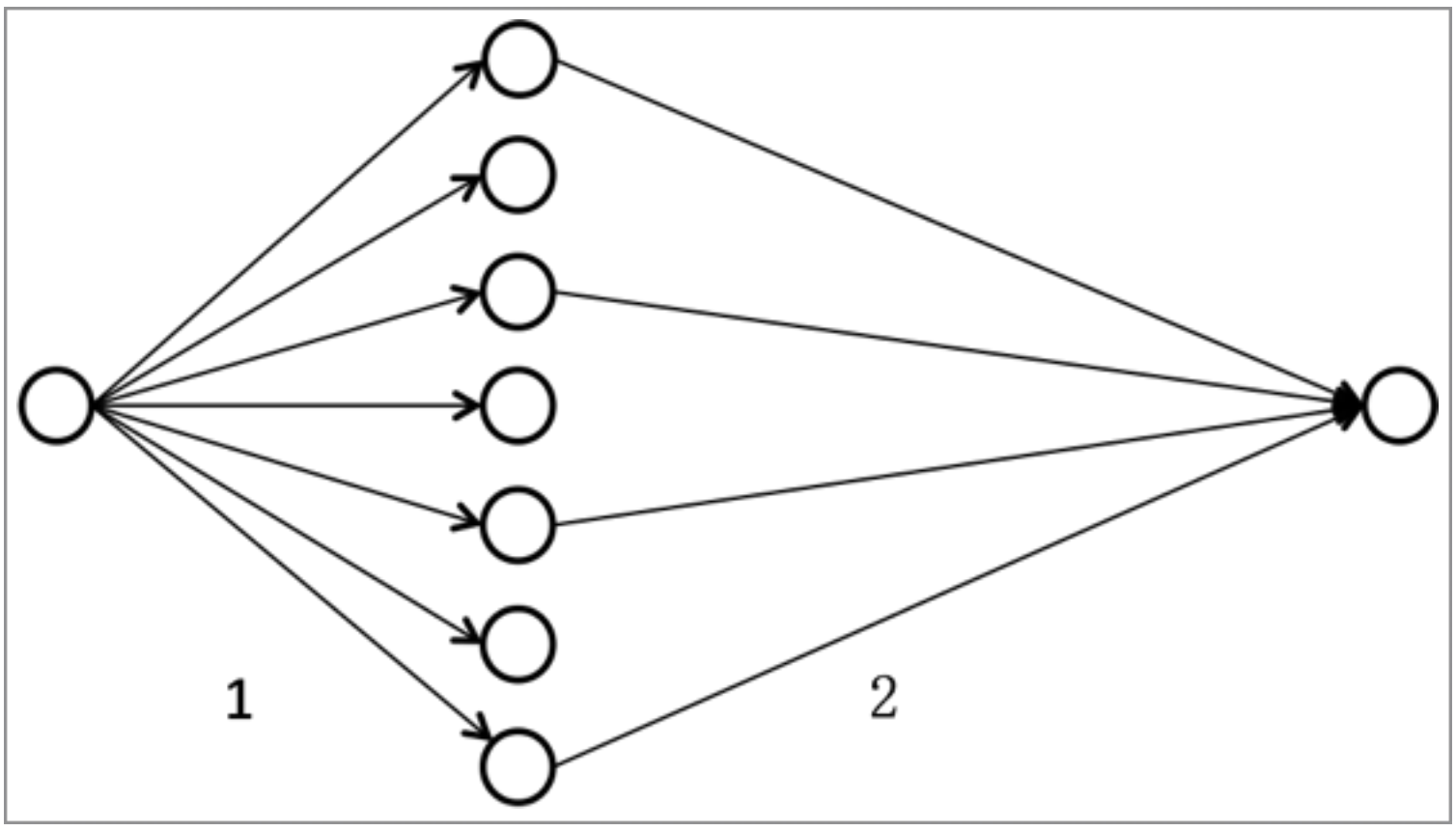


图18.3是在现在这种情形下进行工作的流程。客户端首先在第1步将请求发送给所有的副本状态机中的服务器，并且在第2步等待f+1个匹配的响应，即相同的返回结果，并以此作为这个操作的执行结果。在图中可以看到，客户端只需要获得f+1个一致的结果就可以认为是这部分工作已经完毕。在这种情况下，节点中即使出现了恶意节点，由于最多只有f个恶意节点，因此在获得f+1个匹配的响应的情况下，必然从起码一个正确节点中获得了响应。这样起码有一个的正确节点的事实就保证了所获取的节点的响应是正确

的。这样的一个协议是对前面的那个协议的一个改进，不会是因为哪怕是一个恶意节点也会造成整个系统不能继续工作。

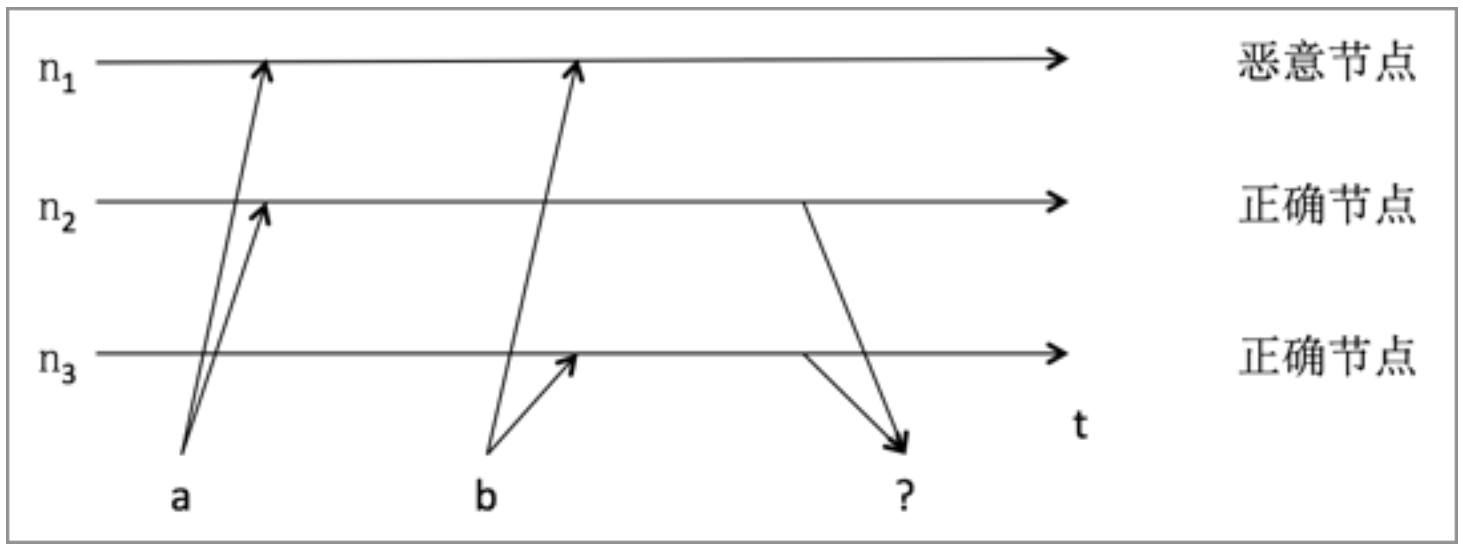
分析一下现在这种方法可能会造成的问题。对于一次操作来说，上述的场景下面确实能够保证从起码一个正确节点中获得响应，而这个响应就是这个正确节点的执行结果。这个时候，其它的恶意节点可能会给出相同的结果，但是这不影响客户端程序与这个正确节点的交互。由于恶意节点是知道正确节点的算法和数据的，因此这种由恶意节点去执行正确流程的做法是非常有可能的。但是，副本状态机是由一系列的流程组成的，需要所有的操作在正确节点中都以相同的顺序完成执行。从这一点上来说，保证一次操作执行是不够的。

考虑多个操作上出现的问题，在这里可能出现的一种场景是在某一步操作一个正确节点与所有恶意节点构成大多数，完成副本状态机的一步操作，而在下一步的时候，所有恶意节点与另外一个正确节点构成大多数。 这两个正确节点并不知道对方执行的是什么操作，正确节点内部的状态发生了不一致，违背了所希望达成一致性协商的目标。这个时候实际上正确的副本状态机节点起码出现了三种不同的状态，即前面的两个节点的状态以及剩余的其它正确的节点的状态。剩余的其它的正确的节点所处的状态不会代理严重的问题，因为此时它们会缺乏一些操作，而不是出现真正的分叉。如果存在某一种机制使得它们能够应用最新的操作以赶上工作的进度，这样的状态也是可以接受的。

上一段的描述还比较抽象，看一下前面给出的副本状态机的具体例子。假设这里的f=1，那么共有3个节点，分别为n1，n2和n3。其中n1是恶意节点，其它两个节点为正确节点（所有节点都应当维护一个唯一的状态x）。在第一步的时候，正确节点n2完成写入操作即写入a，并给予响应，另外一个响应来自于恶意节点n1。在这一步操作的过程中，n3没有参与工作，可能没有收到消息，或者是被网络丢失了，或者是被恶意节点n3通过攻击隐藏了这个写入消息。在第二步的时候，节点n3写入了正确的值b，节点n2由于

干扰没有收到这个操作消息。在这一步的大多数成员的构成同样来自于恶意节点n1。注意在这里这两步的正确节点是不同的两个节点。从客户端的角度来看，都获得了写入成功的响应，并且这个响应来自于大多数（f+1）个节点的一致响应。在每一个步骤中，客户端都认为命令得到了正确的执行。这里的计算流程如图18.4所示。

图18.4 大多数节点正常工作的场景



分析在图18.4中的计算流程，在完成上述两个步骤后，在第三步的时候，客户端会去读取这个x的值。这样，它有可能从两个正确节点n2以及n3中分别读取的值是a和b，此时恶意节点不进行响应。这种情况会造成客户端获得不同的响应值，即客户端获得的操作结果是不一致的，这就会造成客户端不能继续进行工作，协议失败。客户端也同样不能够从恶意节点中获得任何有价值的信息，因为恶意节点完全可以不发送任何信息或者是返回一个与前面两个答案不一样的信息，客户端只能获得不一致的信息。需要注意的是，在现在这种情况下，算法设计上不能让客户端去继续等待更多的状态机节点的响应，否则就会退化到前面所描述的所有节点都必须正确工作的场景下。

上面描述的是客户端探测到冲突的情形，另外一个微妙的情况是恶意节点可以返回结果a，这样的话，客户端获得了一个被大多数节点认可的响应（n1和n2的返回）。这个时候，客户端探测不到冲突，而是继续认为整个系统工作正常。这里的大多数节点包括了恶意的节点，虽然客户端可以继续

工作，但是正确节点中的副本状态机状态已经受到了破坏，产生了不一致。这种出现了不一致，导致副本状态机的状态出现了分叉，而不是仅仅少几个最近操作的情况是不可接受的，因为系统非常有可能不能恢复到一致状态。

从副本状态机的观点来说，正确节点进行了不同的操作，产生了不一致，违反了副本状态机的正确性要求。一方面，由于恶意节点的干扰，正确节点之间不能维持一个一致的状态，另外一个方面这种操作无论对于客户端还是对于状态机组成员的正确节点来说都是无法探测到的，即在一步操作完成之后没有办法确保这步操作可以在所有的正确节点中展开。

从上面的分析中看到，要在拜占庭条件下完成正确的副本状态机还是颇为困难的。分析上述协议出现失败的本质原因是前后两次协议流程中参与的正确节点是不同的，第一步是n2，而第二步是n3。当然，成员不同是完全有可能的，显然随着节点的增多，不能要求参与每一个步骤的正确节点都是相同的，参与操作的正确节点发生改变是一个正常的流程。实际上，在这里的一个最关键的因素是前后的两次操作之间参与的正确节点集合之间没有交集，这就会导致每一步的结果信息在传递中出现丢失。后果就是造成了之后的正确节点没有能够获得前述协商的结果，就会出现状态分离的情况。因此后面算法对此修补的时候，要使得前后两次的操作参与节点集合的交集中有正确节点。这样的话就会使得操作能够在正确节点之间进行传递，避免出现状态不一致的情况。

协议流程3：3f+1个节点，2f+1个正确节点，f个恶意节点

继续对设计的协议进行完善。在这里，直接使用已经被证明了的拜占庭将军问题的理论解。这里包含的节点的数目是3f+1个，其中2f+1个是正确节点，f个是恶意节点。可以直接对上面的协议进行扩展。同前面的情况一样，客户端将请求发送给所有的节点，并等待足够多的匹配的响应。现在，足够多的节点响应的数目达到2f+1。这样的设计关键一点是使得每次提供正

确响应的正确节点数目达到了f+1个，那么前后相邻两次操作的共同的正确节点的数目起码有一个，不会是空集。这样的话，恶意节点试图将正确节点分成不相交的两个集合的企图就不会得逞了。

还是以之前设定的例子场景来分析上述的协议流程在执行的过程中是否能够真正完成一致性协商的工作，并维持所有的操作顺序一致，或者说是有什么样的改进。如果设定f=1的话，现在至少应该是4个节点，分别是n1,n2,n3,n4。在这里，仍然假设n1是恶意节点，n2,n3,n4都是正确节点。每一个正确节点维护了一个变量x的副本。

图18.5 更多的大多数节点操作

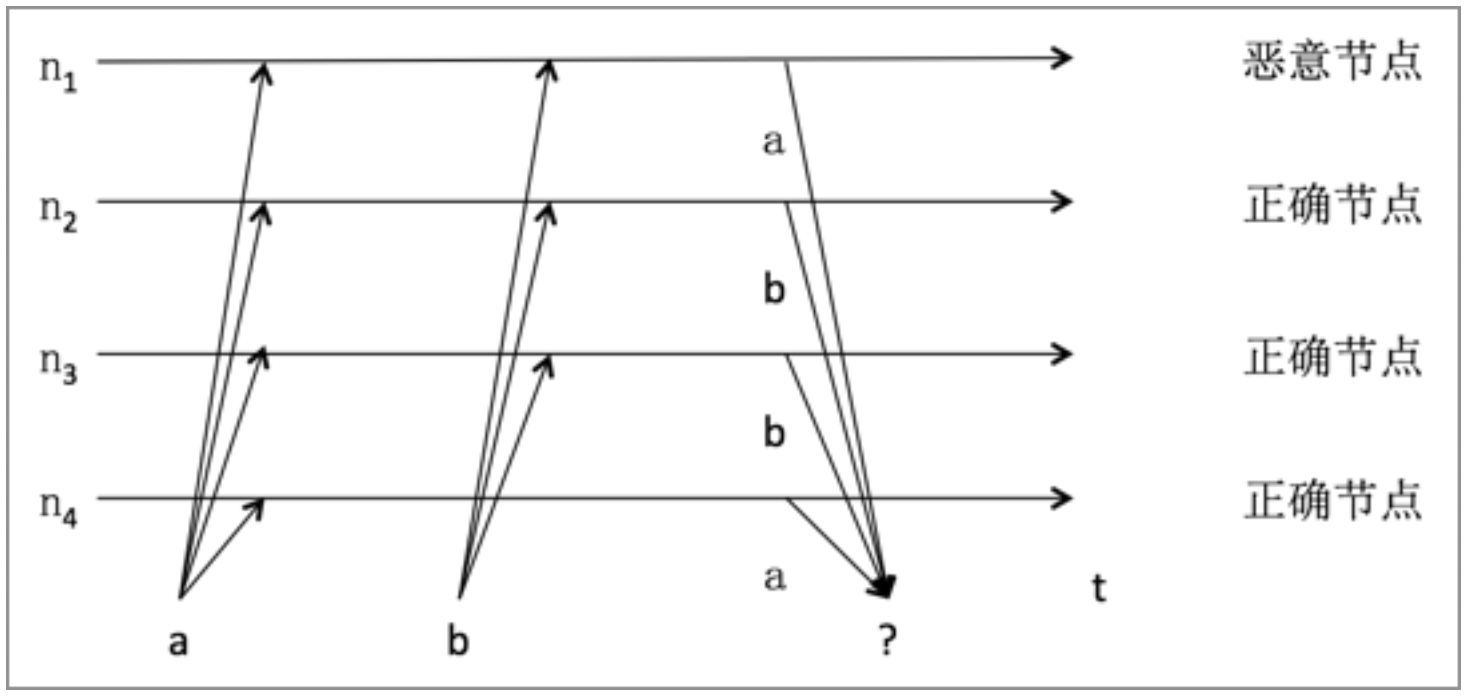


图18.5 中的操作经过了一下几个步骤进行操作：

第一步，每一个节点都收到了写入a的操作，并且正确节点进行了正确一致的操作，客户端也收到了足够多的匹配的响应，能够继续进行下一步的工作。

第二步，n1, n2, n3收到了写入b的操作，由于n2, n3是正确节点，因此忠实地进行了这一步的操作请求。此时，n4由于某些原因没有收到这个消息，没有执行这一步的操作。此时，可以发现正确节点已经出现了不一致的情

况。但是，由于恶意节点可以给出与n2, n3节点相同的响应，因此客户端也获得了足够多的匹配的响应，可以进行下一步的工作。此时整个系统中的正确节点的大多数（在这里是2个）维持在一个一致的状态上，少部分的正确节点由于忽略了某一些操作，状态要落后于大多数的正确节点。前面已经讨论过，这种缺乏几个最近几个操作的状态是可以接受的。由于是通过投票方式进行系统状态的维护，因此也可以期望只要有大多数的正确节点能够维持一个一致的状态，整个系统可以进行下一步的工作。这种情况下，期望所有的正确节点都能够执行每一步操作也是不实际的。剩下的少部分正确节点可以在将来的状态转移中获取相应的日志，完成对应的操作即可赶上其它节点的进度。

第三步，客户端需要读取变量x的值，此时恶意节点可以进行协议的破坏工作。由于是读取阶段，依照前面的讨论，客户端需要等待2f+1个匹配的响应。在这种情况下，n1和n4响应为“x的值为a”，而n2和n3响应为“x的值为b”。此时，客户端得不到足够多的一致的响应，无法确认操作是否在正确节点中执行完毕，将不能继续工作，协议失败，因为他要等待2f+1个匹配的响应。此时客户端实际上已经等到了所有节点的响应，不可能再有新的响应去打破这种平衡。从这个例子可以看出，通过扩大正确节点的数目进行投票，同样也是有可能造成一致性协议无法完成，会停止在某一个步骤。

但是，现在的协议设计由于有了更多的正确节点，对比之前的协议流程有了一定的改进。在前面的协议流程中，恶意节点的破坏能够让整个系统继续运行，客户端和服务器端都可以按照设计的协议继续工作，并且不能探测到攻击破坏行为的发生。在现在这个协议流程中，由于大多数正确节点在客户端得到足够多2f+1匹配响应的时候（至少有f+1个正确节点，至多有f个恶意节点），其内部的状态是一致的。之后，如果恶意节点不配合大多数正确节点的工作，就会导致副本状态机停顿，客户端不能够继续工作。这样，客户端在这个时候其实是可以发现系统里面出现了问题，虽然获得了所有副本

状态机节点的响应，但是没有足够的一致回答，不能够进行正常工作了，副本状态转换的过程中已经停顿。

前述的协议流程造成副本状态机无法继续工作的原因是由于恶意节点的破坏，使得在执行顺序上出现问题。在恶意节点的配合下，一部分正确节点无法获得最近的部分操作，而正确节点的大部分继续新的操作。如果恶意节点不参照协议流程执行，可以看到在上述步骤的最后一步，客户端最多只能获得2f个匹配的响应（f个正确节点，是正确节点的小部分，另外f个是恶意节点），没足够多的一致的响应，客户端不能进行进一步的工作。注意，这个时候f+1个正确节点能够执行倒数第二步的操作，并且在那一步中客户端也能够认为自己完成了工作。

因此，从这里的例子来看，即使将副本状态机的节点数目中正确节点的比例扩大到理论值，直接进行简单投票的方式无法维持副本状态机的正常工作。对此，需要其它的一些机制使得副本状态机能够按照步骤推进。为了解决这样一个问题，实际上这里的讨论还缺乏一个重要的一环，即如何定出操作的顺序的问题。操作的顺序不能按照时间戳来进行，而通用的手段是通过一个首领节点（Leader）来帮助完成。首领节点是进行上述协议修正的一个可以利用的因素。

操作的定序与提交

首领节点的最重要的作用是进行操作的定序工作，对于副本状态机来说，就是每一步需要做什么样的操作。看一下并发操作的情况，到现在为止客户端的所有操作都发送给所有的节点，并且每一个节点都知道操作的先后顺序。但是，在副本状态机的系统中，客户端不会只有一个，而是需要支持多个客户端的并发执行。这个时候，通过客户端自身来确定操作的序号是不能工作的，因为所有的序号必须要按照自然数的方式提交到系统中，所有的序号必须连续，不能在其中空出来一部分的操作。客户端有可能失效，有可

能有延迟，也可以自由加入以及推出发操作动作，在这种情形下维护每个客户端严格按照需要发送操作是不可能的。因此，副本状态机会选择出一个首领节点，由这个首领节点确定具体的序号并完成转发工作。之后，其它的节点依据这个需要进行操作。这种工作方式与之前讨论的Paxos算法类似，当然，现在这里的首领节点可能是个恶意节点。因此必须要处理首领节点是一个恶意节点的情况。

另外一个值得注意的是关于完成操作执行的过程，从前面的讨论中可以看到，在之前的协议设计中，都是让状态机中的每一个副本都直接执行操作。但是，这种执行模式可能是不正确的，因为一旦操作被执行，在后续的过程中如果发现不合适的话就难于撤销。因此，所有的操作必须要在一致性协商完成之后（完全确认）才能够进行执行，否则的话就会造成正确节点的状态发生偏离，出现不一致情况。有一种不一致的状态是可以接受的，前提条件是大多数的正确节点都处在一致的状态，小部分正确节点没有获得最新的操作而落后于其它的正确节点。这样的情况可以接受的原因是在这种情况下，小部分的正确节点可以在之后的信息交换中补充执行一些漏掉的操作以将状态调整到最新的状态即可。

讨论到这里可以看到，操作顺序是一个关键的问题，而操作顺序的确定则依赖于首领节点的工作。另外，对于正确节点来说也只有在完全确认一致性协商之后才可以执行对应的操作，修改状态机的状态，否则出现不一致因为无法回滚操作也就不能够保证状态的一致。在后续的对于协议的修改过程中，将主要围绕这两点来展开，使得协议过程可以避免由于恶意节点的破坏造成停止的情况。由于操作的顺序需要通过一个首领节点定义出来，在下面的讨论中，读者需要特别注意首领节点在达成一致性协商中的作用以及对于首领节点是恶意节点的处理。

通过投票进行分布式环境下的一致性协商

首领节点是恶意节点能够进行的破坏操作以及防护办法

所采用方法的基本出发点还是通过节点之间的投票来达成一致性协商，并且这其中的操作顺序将由首领节点负责定义，因此首领节点将在协商协议中起到重要的作用。但是由于正确节点并不能知道当前的首领节点是正确节点还是恶意节点，因此有必要分析一下在投票协议的流程中首领节点是恶意节点的情况下如何去进行破坏协议的工作。（这里当前的含义是首领节点不是一成不变的，例如发生首领节点下线的时候，需要新的首领节点工作，这与Paxos算法的情况类似。）

首领节点的主要工作是接收客户端的请求，然后将客户端的请求进行排序，为每一个请求编入一个序号，再将带有序号的请求操作转发给副本状态机中的每一个成员。其它的成员会依据序号来完成状态转换工作。对于一个恶意的首领节点来说，首先需要看一下这样一个首领节点能够进行什么样的破坏。首领节点面对的是客户端以及其它的在状态机中的正确节点，因此可以看一下首领节点面对这两类节点来说分别能够做什么样的破坏，以及客户端与其它的正确节点能够做什么样的防护。

针对客户端来说，首领节点的主要功能是接收请求以及返回响应结果。一个恶意的首领节点可以完全忽略客户端的请求，或者在获得客户端的请求之后进行篡改再发送给其它的副本状态机成员，或者在之后的返回中返回一个错误的结果，或者根本就不返回结果。针对这种情况，客户端能够做的防护工作是将请求发送给副本状态机的所有成员，而不是仅仅是首领节点，这就避免了首领节点对于请求的忽略或者篡改的问题。通过客户端的签名工作，也可以避免恶意首领节点的篡改问题。客户端也需要从其它的副本节点

获得结果，而不是仅仅从首领节点获得结果，获得一组一致的结果的方式能够避免首领节点给出一个错误结果的问题。同时，客户端与副本节点中的正确节点通信以获得一致的结果，这样也同时避免了首领节点不做出响应的问题。由于每一个节点都是通过自己的证书对响应进行签名的，因此首领节点并不能对正确的节点的信息进行篡改，也不能伪造正确节点的响应或者正确节点与其它节点之间的通信消息。可以看到，客户端针对恶意节点实际上是可以进行防护操作的，但是需要一个详尽的协议来达到这个目的。由于正确节点都可以和客户端直接进行通信，恶意客户端就不能停止定序工作，否则的话正确节点会发现长期存在没有定序的操作，就可以推测出首领节点工作不正常（不一定是恶意，可能下线），就可以通过重新选择的方式来更换首领节点。

在上述的客户端的工作改进之后，恶意的首领节点不能给其它的副本节点发送不存在的操作，也不能发送经过篡改的操作，也不能忽略操作不进行转发。此时，恶意首领节点能够做的工作是给不同的副本节点发送不同顺序的客户端请求操作，即给一个特定操作对不同的正确节点定不同的序号。由于序号是首领节点规定的，其它的节点不进行这样的工作，因此这是首领节点在进行针对其它副本节点操作的时候能够做的破坏工作。另外，首领节点也可以响应副本节点说某一个操作已经完成，但是实际上却没有完成，这也是首领节点的定序不一致。对于这个问题的防护就是正确节点不能立刻相信首领节点给出的序号，而已正确节点互相交换从首领节点获得的序号信息，如果发现不一致，显然首领节点是一个恶意节点（每个人都进行消息签名，除了恶意节点，正确节点不会去假扮恶意节点）。每一个正确节点必须要保证所获得的序号在所有（至少是大多数）的正确节点中都取得一致，才能够将这个操作按照给出的序号执行。这种方式也就遏制了首领节点进行的破坏工作。同样的，这只是一个初步的想法，需要一个细化的协议来达到目的。

协议流程4：具有首领节点的投票协议

在分析完毕首领节点如果是恶意节点的情况下能够进行的破坏性工作以及对应的防护措施之后，下面来看一下如何构造一个协议，使用首领节点来规定操作的序号。下面的协议设计中，一个关键的点是副本节点拿到一个从首领节点转发（匹配客户端直接发出的请求）请求的时候，不能立即将这个请求操作进行执行。因为这个时候，首领节点是恶意节点的话，拿过来的操作（序号）是不一定正确的，直接执行会出现一致性状态的破坏，无法恢复。因此，需要叠加一个额外的副本节点之间的消息通信轮次来确认得到了一个一致的消息序号。

在这个场景中，总共的节点的数目是3f+1个，其中2f+1个为正确节点，f个为恶意节点，其中首领节点可能是正确节点或者是恶意节点。下面是协议的操作流程。

第一步：客户端需要将请求操作发送给所有的节点，包括首领节点。同前面所分析的一样，这一步避免首领节点对于客户端操作的忽略或者篡改。

第二步：首领节点定出一个操作的顺序，并将序号与当前的请求进行映射。首领节点发送PRE-PREPARE(op, n)给所有的副本节点。在这个流程中，之所以被命名为PRE-PREPARE，原因是此时的需要是临时的序号，副本节点之间还需要进一步的准备以及确认。

第三步：每一个副本节点发送PREPARE(op,n)给所有的副本。为了避免由于首领节点是恶意节点，造成给不同的节点发送不同的操作序号的问题，副本节点之间相互交换从首领节点获得的序号信息进行确认，确认的流程也是对PREPARE消息的搜集过程。

第四步：如果一个副本节点获得了足够多的PREPARE(op,n)，那么这个副本可以确信op是序号为n的操作，就可以进行操作，可以改变状态，发送响应给客户端。在这里，足够多的数目是2f+1个（包括它自己），那么就可以确信大多数的正确节点(f+1个)都同意将op的序号定为n。这样可以确保大多数的正确节点执行相同的操作，并且前后的两次操作的正确节点集合之间的

交集不为空。如果这个时候正确节点不能够获得足够多的响应的话，不能够继续进行操作，不能将操作直接执行去更新副本状态机的状态。

前面的第三步以及第四步的流程希望达到的目标是使得正确节点能够确信某一步的操作序号，这样的话如果副本状态机的状态需要进行更新的话，就可以确信无疑进行更新。

现在，由于正确服务器的每一步操作都需要在大部分的正确服务器之间进行确认，可以确保状态转移都是一个严格正确的流程。因此，对于客户端来说并不需要从2f+1个节点中获得匹配的响应，而是只需要从f+1个副本节点中获得匹配的响应即可。f+1个匹配的响应能够保证起码从1个正确节点中获得响应，那么这个响应必然反映了正确的状态转换过程。

图18.6 通过两阶段投票进行的一致性协商

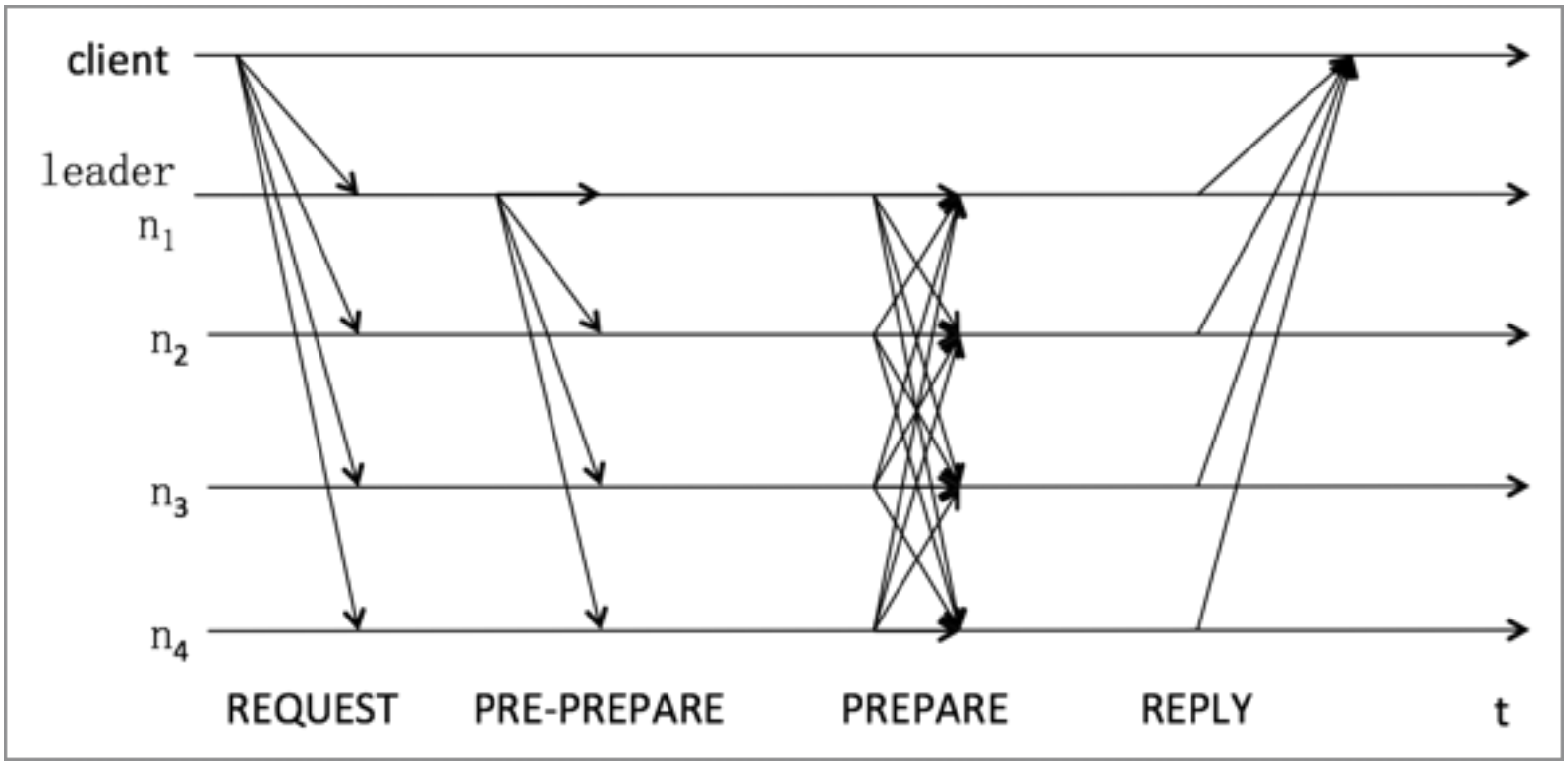


图18.6给出了刚才设计的协议的流程，可以看到现在这里的这个协议就比较复杂了。从图中可以看出，协议将通过多轮次的信息交换来达到系统操作定序的目的，并且最终将定序的结果应用到副本状态机的状态更新中。这里

的协议流程也是之前上一个协议流程的直接扩展，扩展的部分是利用一个首领节点并尽量对首领节点是恶意节点的情况进行防护。

由于现在的协议对比之前的协议已经大大复杂了，直观理解协议正确与否变得比较困难，需要仔细进行分析。为了理解上述的协议是否真的能够工作，或者说是否是正确的，需要进行下面的分析。下面的分析包括两个部分，一个是如果首领节点是正确节点的话，看看整个系统是否能够正常工作以及正常工作的情况如何；另外一个是如果首领节点是恶意节点的话，是否能够正常工作，以及正常工作的情况如何。当然，还需要分析如果不能够正常工作的话，整个正确节点应当可以利用什么样的信息进行进一步的修正，以最终能够完成一致性协商。

首领节点是正确节点的情况

首先来看一下首领节点是正确节点的情况。在这种情况下，主要是观察一下恶意的节点是否能够阻止正确流程的执行。

基于之前的假设，恶意节点不能够伪造首领节点的信息，所有的节点包括首领节点都会通过密码学签名的方式对所有的信息进行签名，因此信息是不能伪造的。如果剩余的正确节点能够正确收到首领节点的消息的话，那么依据协议的流程，一致性协商的结果是可以达成的。所有的正确节点在这个时候可以从首领节点中获得正确的相同的需要，互相交换消息之后，可以获得足够多的PREPARE消息，那么每个人都可以正确更新自己的状态。在这种情况下，客户端也可以获得足够多的REPLY消息，并可以继续工作。

对于恶意节点的破坏来说，恶意节点可以想方设法阻止消息的传输，但是它们也不能永远阻止消息传输。恶意节点可以处在静默状态，不对外界进行响应，但是协议流程中实际上也不需要它们的响应来构成节点的大多数。处于这种环境下的恶意节点可以参与协议的流程中，并且遵循协议的流程来发送PREPARE消息。这种方式下恶意节点实际上对于外界来说表现得像一

个正确节点。恶意节点通过这种方式可以使得某一部分正确节点尽快达到2f+1的大多数，并且通过拒绝服务攻击的方式阻止其它的（最多f个小部分正确节点）正确节点收到操作的信息。这样的话，小部分的正确节点将收不到操作，可能在状态流程中落后于大部分的（至少f+1个）正确节点。但是，这样的状态是没有问题的，因为这部分的正确节点在将来的某一个时间可以从这个正确的首领节点中获得错过的操作，通过将这些操作进行执行就可以将状态更新到最新。总而言之，在首领节点是正确节点的情况下，恶意节点不能够阻止一致性协商结果的达成，而最多只会延迟一致性协商结果的达成。当然，这里还有另外一种情况，即首领节点可能并不是恶意节点，只是在一段时间内出现了停止错误，整个协议也需要处理这种情况，通过选举出新的首领节点使得整个一致性协商过程可以继续下去。从分布式算法的性质来说，首领节点是正确的节点的情况下，整个系统的安全性（Safety），即不发生不可逆转的状态不一致是可以保证的；而整个系统的活跃性 （Liveness） 也比较乐观，只要恶意节点无法阻止正确节点之间的通信，整个系统可以不断完成状态转换。

这里对于首领节点是正确节点的情况分析，也为解决首领节点是恶意节点的情况提供了一种思路。通过这里的分析可以看到，首领节点是正确节点的情况下，整个系统通过投票协议就可以完成一致性协商。因此，如果在首领节点是恶意节点的情况下，如果一致性协商不能达成，那么正确节点如果能够探测到这种情况，就可以通过选举另外选择一位首领节点，并一直选择直到一个正确节点称为首领节点为止。之后在这个正确节点的带领下，整个一致性协商可以继续进行下去。因此，在下面的讨论中，重点是两个方面的内容，一个首领节点是恶意节点的话，如何探测到这种情况；另外一个是如何选举出新的首领节点，并且在首领节点出现变化的过程中维持系统状态的一致，不至于出现状态分叉的情况。读者也可以从后面的内容中可以看到，解决恶意节点所采用的方法同样可以用来解决节点节点失效以及恢复的情况。因此，在理解所叙述的内容的时候，读者可以首先将网络和节点都设置

为正常工作的情况，此时恶意节点会做出一些破坏的动作，在这个基础上去理解协议的正确性。之后可以在加上正确节点的停止恢复错误，可以帮助读者理解整个协议流程。

首领节点是恶意节点的协议流程分析

下面需要重点分析一下首领节点是恶意节点的情况，主要的重点是观察正确节点是否能够在恶意节点进行破坏的时候能够观察到这一点，以及正确节点如何去保证状态不会被破坏。同样的，状态被破坏的标志是两个正确的节点在同一个序号状态转换的时候做了不同的操作，造成状态不一致。正确节点的一部分落后几个操作不认为是状态遭到了破坏，因为这样的正确节点可以在其它正确节点的帮助下（通过日志）将状态更新到最新的状态。前面已经分析过了，首领节点（现在是恶意节点）不能伪造客户端的操作或者忽略客户端的操作，因为已经通过客户端的签名以及广播到所有的副本节点。首领节点可以进行的破坏包括将不同的操作顺序发送给不同的副本节点，或者说告诉某一些节点一部分的操作已经完成了，而实际上这个操作并没有完成（这种情况实际上也是给了同一个操作不同的序号）。

下面的分析分为3种情况进行讨论，三种情况的划分是最终获得2f+1个PREPARE消息的正确节点的数目。在下面的情况分析中，首先给出的是对应情况可能发生的证据，然后再分析发生之后产生的效果。

第一种情况：

所有的正确节点都获得了2f+1个匹配的PREPARE消息。这种情况是可以发生的，只需要首领节点依据协议的信息正确执行即可，并在发送PREPARE消息不作为。这个时候，正确服务器之间可以互相交换获得的PRE-PREPARE消息就退化为现在这种情况。

这种情况下，所有的正确节点获得的操作应当都是相同的。显然，对于任意两个正确节点来说，每一个都获得了2f+1个匹配的PREPARE消息，其中就起码有f+1个匹配的消息是从正确节点获得的。这样的话，这两个节点之间必有正确服务器的交集，因此这两个节点将获得相同的操作，并且与第三个正确节点给出的操作相同。以此类推，所有的正确节点拿到的操作都是一样的。正确节点获得这部分信息之后，可以将操作在状态中进行执行。此时，对于客户端来说，整个副本状态机中的所有正确节点都可以继续执行，那么此时对于客户端来说，只需要获得f+1个匹配响应即可，作为操作的结果进行下一步操作。

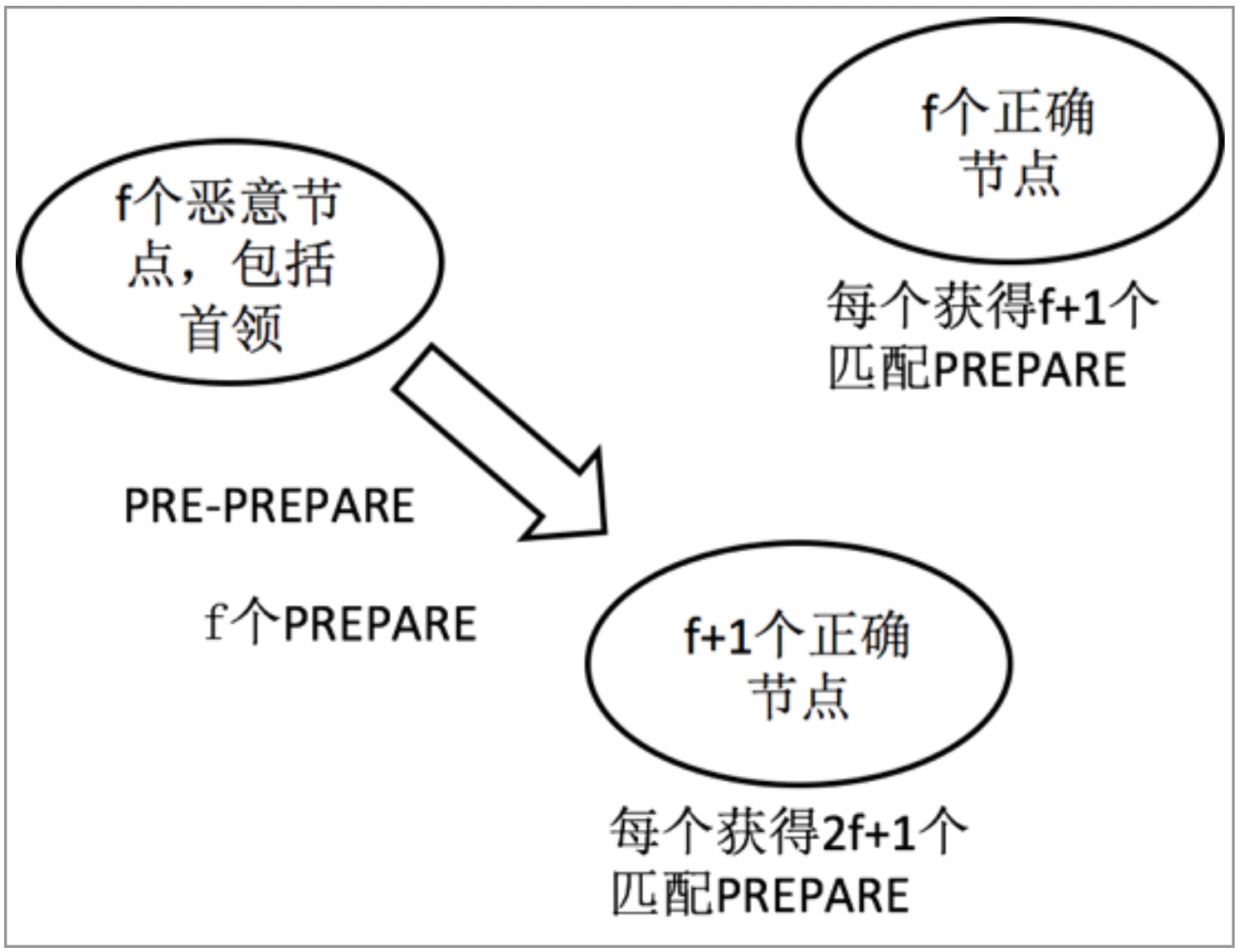
第二种情况：

这里有超过≥f+1个正确节点获得了2f+1个匹配的PREPARE消息。在这种情况下，大部分的正确节点都获得了匹配的2f+1条PREPARE消息，只有一小部分的正确节点没有获得足够多的匹配的PREPARE消息。这种情况也是可能发生的，下面特别以正确节点f+1个收到2f+1个匹配的PREPARE消息为例。此时，恶意首领节点将消息只发送给f+1个正确节点，这f+1个正确的节点互相之间交换PREPARE消息，剩下的匹配的PREPARE消息都由恶意节点处获得。图18.7给出了这种情况的发生。至于数目超过f+1的情况，只需要恶意节点减少发送PREPARE消息的个数即可，可以获得任意≥f+1并且小于2f+1个正确节点收到2f+1个匹配的PREPARE消息的情况。

在这种情况下，小部分的正确节点获得不到信息而不能参与这个操作的工作。这里就会产生这样一个问题，即大部分的正确节点是否可以执行这个操作去改变状态机的状态。如果仅仅是让小部分的节点能够在将来的时间赶上状态变化的进度是可以的。但是，问题是如果小部分正确节点与恶意节点结合在一起，将这个决定的信息进行推翻，那么就会出现大问题，即状态不可能进行回滚了，此时不同的正确节点之间的状态产生了分离，不可恢复。现在看看这样的情况是不是真的可以发生。举个例子来说，恶意的首领节点

给f+1个正确节点发送写入x=a的命令，而向另外f个正确节点发出读取的命令。如果f+1个正确节点能够获得2f+1个PREPARE响应（如何获得在前面讨论过了），那么实际上另外的f个正确的节点是无法获得足够的PREPARE响应的。这样的话，这里的最多f个节点不会获得足够多的PREPARE消息，因此不能达到一致性的协商。

图18.7 有f+1个正确节点获得2f+1个匹配的PREPARE消息的情况



f+1个节点能够完成协议的所有流程，从前面的分析中也可以看到，f+1个节点协商完成的操作也是完全一样的。另外，其它的f个正确节点由于获得不了足够多的PREPARE消息，是不可能推翻f+1个正确节点的协商决议的。这样的话，这个时候f+1个节点实际上可以将协商完成的操作对状态进行操作转换。

因此，对于这种情况的结论就是：所有获得2f+1个匹配的PREPARE消息的正确节点（超过≥f+1个）都可以认为已经完成了一致性的协商，本序号的操作在这个阶段已经完成了协商。由于剩下的最多f个正确节点以及恶意节点一起也不能够构成足够多的投票去取消已经决定的操作，整个系统可以继续运行去协商下一步的操作。但是这些小部分的正确节点毕竟不知道当前的协商一致的操作情况，这种状态会导致一部分的正确节点的状态不能够进行转换，造成了状态滞后的情况。在协议的后续设计中，需要考虑这种情况，使得这些节点在将来的执行过程中能够恢复对遗漏的操作的执行，以追赶完成状态的转换。

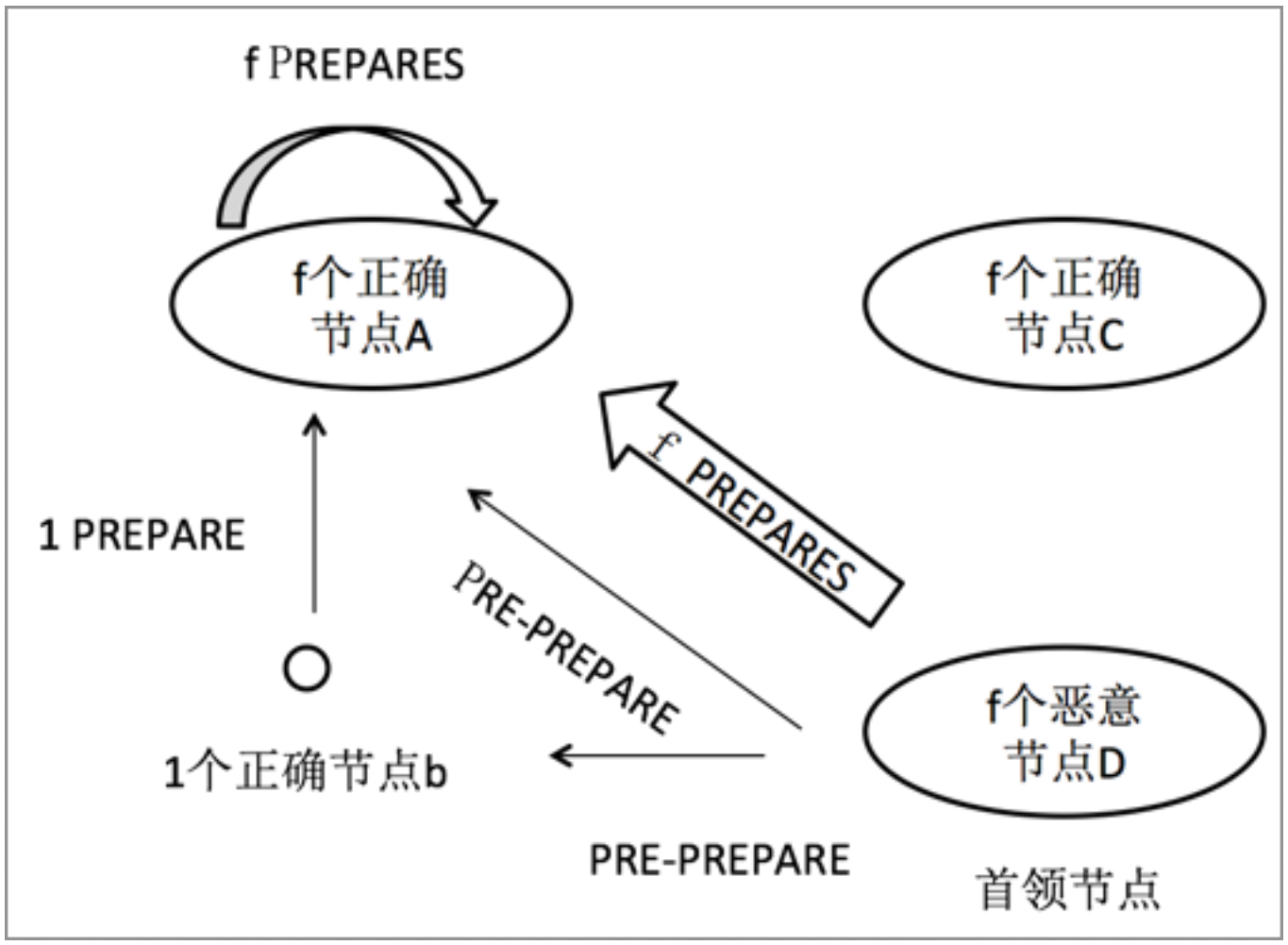
另外，在这种状态下，对于客户端来说，是可以获得f+1个匹配的响应的。这f+1个匹配的响应不会从f个没有达到一致性协商的正确节点中获得，因为这些正确节点已经陷于停顿，不能够响应客户端了。匹配响应的结果只能来自于f+1个已经执行了状态转换操作的正确节点中，或者一部分的响应来自于恶意节点。从恶意节点中获得一致的结果不会影响整个系统的状态的正确性，因为客户端起码需要获得f+1个匹配的结果，而这其中起码有一个来自于正确节点，而此时大部分的正确节点的状态与这个正确节点的状态是相同的。

第三种情况：

第三种情况就是只有≤f个的正确节点获得2f+1个匹配的PREPARE响应。这种情况也是非常有可能发生的。例如，可以设计场景使得f个正确节点获得2f+1个匹配的PREPARE响应。对于这f个正确节点来说（设为集合A），它们之间可以互相交换，以获得f个匹配的PREPARE响应。另外f个匹配的响应来自于恶意节点。最后一个PREPARE响应可以来自于另外一个正确节点（设这个节点为b）。这样的话，这f个正确节点可以获得2f+1个匹配的PREPARE响应。剩余的f个正确节点（设为集合C）从A以及b中获得f+1个匹配的PREPARE，而恶意节点，给出另外的信息。节点b从集合A以及自身获

得f+1个匹配的PREPARE，从恶意节点以及集合B中获得其它的不同的信息。图18.8反映了这种情况，恰好有f个节点获得了2f+1个匹配的消息。

图18.8 f个正确节点获得2f+1个匹配的响应

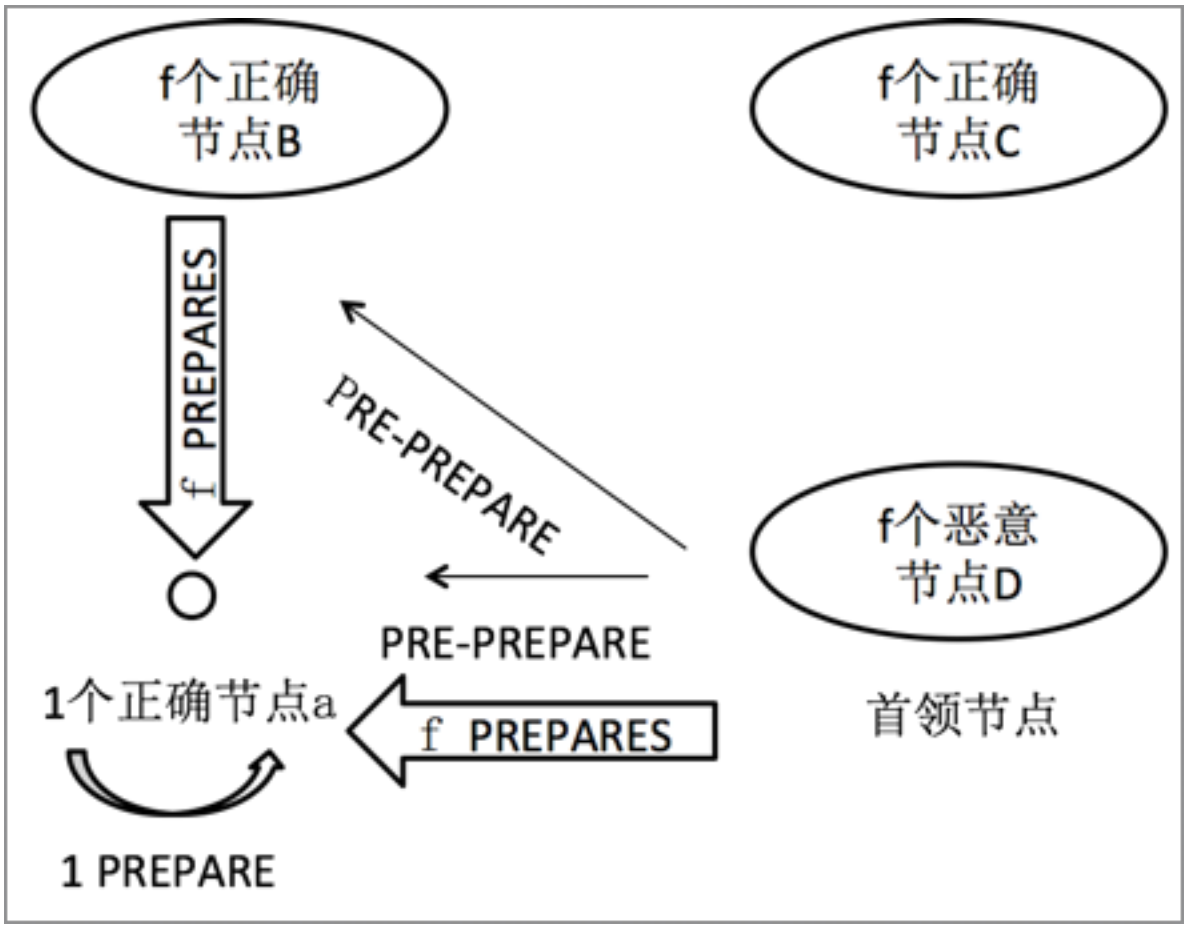


作为另外一个极端的情况，也可以设计另外一个场景，使得只有一个正确节点获得2f+1个匹配的PREPARE。可以将这个节点设为a，一部分的正确节点的集合为B，另外一部分正确的节点的集合为C，剩下的恶意节点的集合为D。可以给出图18.9中的方式来构造出只有一个节点a获得了2f+1个匹配的，首领节点在f个恶意节点里面。这里的情况与上面的情况是类似的，只不过现在f个恶意节点将配合这个唯一能够获得2f+1个匹配PREPARE的正确节点，使得只有一个节点能够手机足够多的PREPARE。

在这种情况下，从副本节点的角度来看，大部分的正确副本节点由于拿不到足够多匹配的PREPARE而不能继续工作。对于客户端来说，最有可能的情况是从收到2f+1个PREPARE的正确节点以及恶意节点那里获得超过f+1

个的匹配的操作结果。这个操作结果也是所有获得2f+1个PREPARE消息的正确副本的操作结果，其它的大部分正确节点由于搜集不到足够的信息，不能够进行下一步的工作。

图18.9 只有一个节点获得2f+1个匹配的响应



因此，在这种情况下整个系统可能会有一小部分的正确节点（在恶意节点的配合下）能够继续工作，其它的大部分正确节点会停止在某一步的操作中。并且客户端由于拿不到足够多的回答，操作不能完成。

但是，实际上如果恶意节点不进行配合的话，例如在上图恶意节点发送给b的PRE-PREPARE操作不是对应的操作号，而是另外一个任意的数，那么整个系统将出现所有的节点都没有收到2f+1个匹配的PREPARE消息，整个系统会停止下来，客户端也收不到足够的f+1条消息而不得不停止下来。

总和上面所讨论的情况，在协议流程4的计算过程中，所有的情况都会发生，可以存在任何数目的正确节点获得2f+1个PREPARE消息的情况。另外，依据现在的协议，会有任意的数目的正确节点会完成状态机的转换过

程。分析到现在为止，只能得出这样的结论：即在这里的协议中，一部分的节点能够进行状态的转换，另外一部分节点则无法进行状态的转换。至于这里出现的一部分节点状态没有转换的问题，尚不清楚如何进行处理，因此如果这个协议流程4能够工作的话，起码需要弥补这一方面的缺陷。

所有的情况中，有一种情况是永远不会发生的，即存在两个不同的正确节点会同时收到2f+1个匹配的PREPARE消息，但是这两个PREPARE消息是不同的。这是不可能的，原因是这两批消息的集合的交集中必然有正确的节点，而这个正确的节点是不可能给两个其它的节点响应不同的PREPARE消息的。因此，在当前的协议设计中，不会出现两个正确的节点进行不同的操作的情况以造成状态的实际分裂。

到目前为止，协议流程4会有以下的一些结论：如果首领节点是正确节点的话，那么整个系统可以持续的工作，恶意节点不能够对整个系统造成破坏。如果首领节点是恶意节点的话，整个系统中的一部分正确节点（可能是大部分正确节点，也有可能是小部分正确节点，任意数目的正确节点都有可能）依据本地的信息以及网络中传过来的消息可以认为达到了一致性的协商。其它的正确节点由于拿不到足够的响应消息，不能够继续工作。这也是继续设计协议流程的时候必须要考虑的问题。但是，无论如何对于某一个序号的操作来说，整个协议在不同的正确节点中不会决定出两个不同的操作内容出来。

对于协议流程4的分析可以先到这里为止，由于这里最严重的问题是系统在最后一种情况下不能够继续执行，只能通过选举出新的首领节点来继续下一步的工作。

协议修正与系统配置更改（View Change）

首领节点的选择

之前为了应对首领节点是恶意节点的情况，设计了前述的协议流程4。但是从之前的分析来看，这个协议流程还存在一些缺陷。在首领节点是正确节点的时候，可以保证所有的正确节点都可以正确执行操作，获得一致性协商的结果。这种情况下副本状态机可以继续工作。但是，在首领节点是恶意节点的时候，会有任意多的正确节点会得不到一致性协商的结果，也会有任意多的正确节点（有可能是少部分正确节点）得到了一致性协商的结果。副本状态机有可能可以继续工作，也有可能不能继续工作。虽然在可以继续工作的情况下，一部分的正确节点不能继续状态转换，但是还是有可能存在副本状态机不能继续工作的情况。因此，对这个协议进行修补的关键是能够通过某种方式，在不能继续工作的时候选举出一个新的首领节点，寄希望于新的首领节点是正确节点，带领所有节点进行正确工作。当然，新选出来的首领节点还有可能是恶意节点，那就需要选择一个算法继续进行选举，直到选举出一个正确节点为止。这里进行继续选择的过程可以被称为一个系统配置更改 （View Change），也有被称为视图更新的，但是用系统配置更改更能表达这个过程的含义。

之所以被称为是系统配置更改，这是因为当前的首领节点是哪一个节点实际上是整个系统配置的一部分。在这里的协议中，不会去考虑节点变化的情况，因此整个副本状态机的系统成员是不变的，不需要将新的节点加入到系统配置信息中，也不能把节点从系统中删除。因此，系统配置更改的情况只在于首领节点的更新。另外，为了区分每一次的系统配置更改情况，可以对所有的系统配置情况形成一个列表并可以编出一个序号。因此，每一个系统配置的情况可以表达为一个二元组：（View Number，Leader），即系统

配置序号以及对应的首领节点。可以将这样的信息保存在每一个参与的副本节点中，对于正确的节点来说使用这样的信息从首领节点接收对应的操作的序号，而对于恶意节点来说这部分信息可以用来有针对性的进行攻击操作。

为了阻止每次在进行系统配置更改的时候，都会造成恶意节点被选出作为首领节点的情况，需要规定一个轮流当选首领节点的顺序。有一个简单的方案就可以避免上述的情况（每次选出来的都是恶意节点），即如果总体的节点的数目为n的话，对于系统配置编号View Number = v的系统配置来说，规定其首领节点为编号为取模运算v mod n的结果。这样的话，首领节点就会在所有的节点之间进行轮转，最多经过f轮的系统配置更改之后，剔除了f个恶意节点，就会将首领节点传递给一个正确节点。这里是对于首领节点进行更新的策略，下面是首领节点进行更新的触发操作。

系统配置更改的触发与响应

由于在系统中的所有节点都是对等的（即使是首领节点，也是各个副本节点轮流承担的），因此任意一个节点都可以发起系统配置更改。对于恶意节点来说，如果认为有可能造成破坏的话，可以在任何一个时机都可以发起系统配置更改。但是对于正确节点来说，只需要在有必要的时候进行系统配置更改。如果一个正确节点通过自己的观察，如果认为整个协议能够继续执行，完成一致性协商，并可以更新自己的状态机，那就没有必要发起系统配置更改。因此，对于正确节点来说，如果工作在上述的协议流程4，发起系统配置更改的条件是：节点从客户端获得了一个操作，但是经过一段设定的超时时间之后，并没有获得匹配的2f+1条PREPARE消息。对于这个节点来说，在这种情况下当前的正确节点的状态机不能够继续进行下去，只能发起系统配置更改看看是不是有恶意节点充当首领节点的情况发生。

那么，随之而来的情况就是是否一旦有节点提出了系统配置更改，那么其它的节点就必须响应这样的请求去同意这样的配置更改？或者说，此时就

让所有的正确节点都同意系统配置更改呢？这也是不合适的，因为如果是这种情况的话，恶意节点会不断发起系统配置更改的请求，使得整个系统的状态机处在无穷的系统配置更改流程中，真正的工作无法继续进行下去。因此，这里必须有足够多的成员发起系统配置更改的请求才能够成功完成一次系统配置更改，进入下一个系统配置的周期（即一个更大的系统配置序号View Number）。

系统配置更改方案1：

基于上述的分析，直接将上述的方法表达为一系列的协议流程，可以形成下面的一个最原始的系统配置更改方案，在这个方案中，由下一轮的首领节点来搜集足够多的配置更改请求，并完成系统配置的更改，通知其他的副本节点成员。这个方案的具体流程可以通过以下的步骤进行操作（注意，下面的协议流程是针对正确节点来说的，恶意节点可以执行任何操作）：

1.由于超时而得不到足够多的2f+1个匹配的PREPARE消息，某一个副本节点发送一条VIEW-CHANGE消息的请求给新的首领节点。新的首领节点的选择方法是所有的节点都已知的方案（即取模mod n方案）。这个时候，如果发起系统配置请求的是正确节点，那么这个正确节点将不会继续工作，因为其内部的状态机已经卡在当前的操作中，需要等待新的首领节点发出信息来使得其继续工作。

2.新的首领节点等待足够多的VIEW-CHANGE消息请求，因为不是任意少数的节点发起系统配置更改，配置更改就会成功的。这里还没有对足够多提出明确的定义，后面会通过分析足够多能够满足的一些性质来判断什么样的数目才能够被称之为足够多。

3.新的首领节点在搜集到了足够请求之后，发起一个新的系统配置，即发送NEW-VIEW消息给所有的其它的副本节点。在这个消息中还需要附带所有的收到的VIEW-CHANGE消息，这些消息将作为当前这个 NEW-VIEW 的

证明，即已经有足够多的节点同意发起系统配置更改。这个过程使用了签名的方法，是的收到NEW-VIEW消息的正确节点能够判断出确实是有足够多的节点同意发起系统配置更改。

4.所有的正确节点在收到NEW-VIEW之后，将会保存这个新的系统配置，之后就会进入到一个新的序号的系统配置中进行工作。在原有序号中出现的停止状态更新的情况将被取消。如果出现了状态停止更新的情况，就需要新的首领节点来推动。

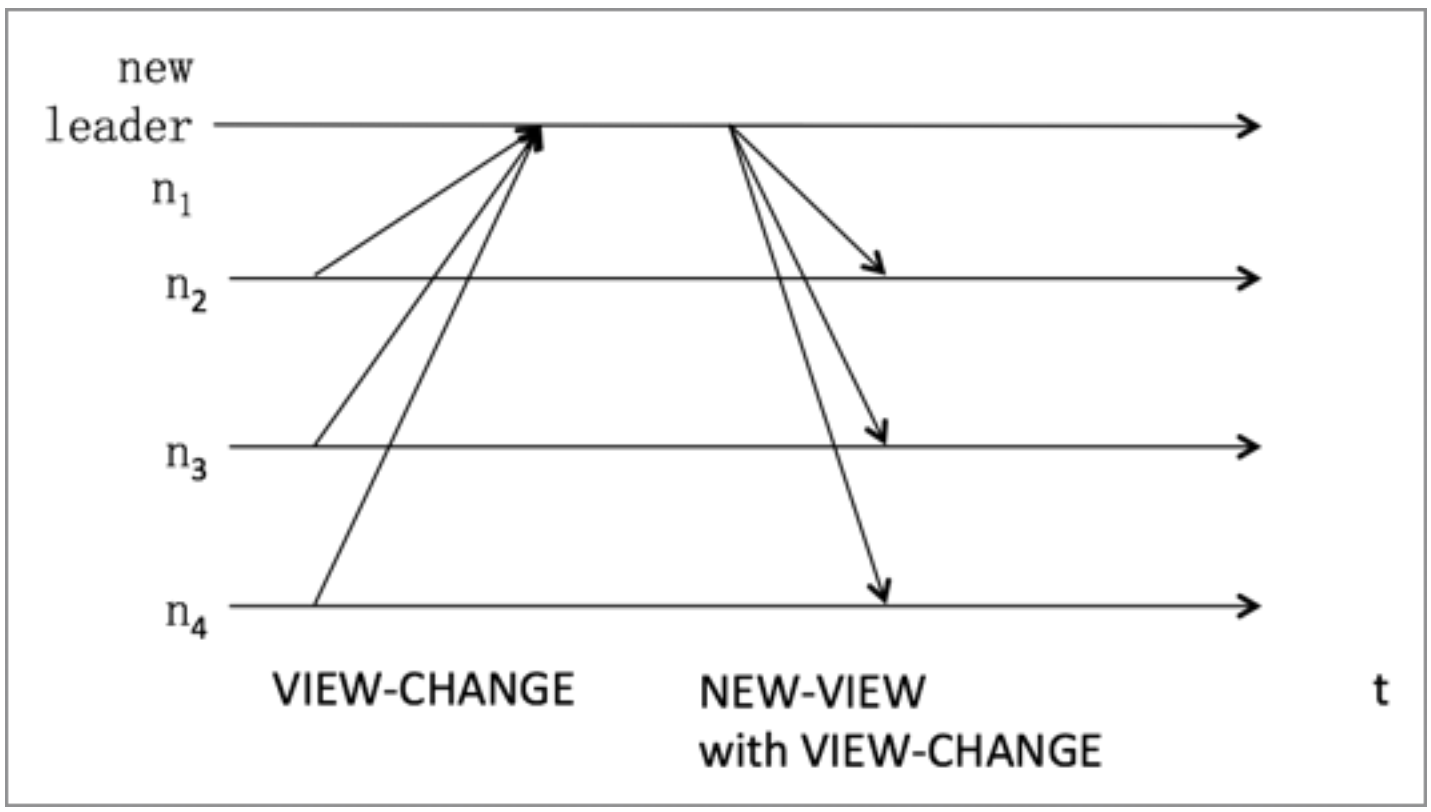


图18.10给出了系统配置更改方案1的执行流程。上述是系统配置更改方案的大致流程，可以期望在完成配置更改之后，如果新的首领节点已经看到了序号为n的操作已经完成，即从首领节点自己的角度来看已经收到了2f+1个PREPARE，那么首领节点就可以开始从n+1开始排列后面的操作。当然，这个过程需要详细分析，观察这样的这换是否可以正确执行。

在此之前，先来看看这里的配置更改方案中的足够多应该需要多少个节点发起VIEW-CHANGE。少于等于f个是不够的，这种情况下恶意节点相互配合就可以使得状态机只做无用功。超过2f+1个也是不行的，因为这个时候

如果恶意节点不参与的话，永远不会系统配置更改成功，副本状态机会停滞。因此，足够多应该在f+1（包含）个到2f+1（包含）个之间。可以看到，实际上起码需要大多数的正确节点发起系统配置更改才可以真正完成系统配置更改，这也是合理的。至于具体的数目是多少，现在还没有信息可以用以决定，将会在后面进一步进行分析。

下面来分析上面的配置更改方案1配合前面的协议流程4是否可以在执行过程中正常工作，或者说能够正确工作的条件是什么。从首领节点的工作来看，其主要目的就是去确定操作的编号，推动状态的转换。因此，这里的关键一点是随着系统配置更改之后，所有的正确节点都应该看到操作的编号在两个配置之间正确连续传接下去，不可以跳过一些序号。

分析这里的系统配置更改方案1的情况，结合前面的协议流程4的协议操作流程。由于新的首领节点需要搜集足够多的VIEW-CHANGE消息以发起一个配置更改，这就需要起码大多数的（f+1）正确节点的支持。而协议流程4里面的三种情况中，如果超过f+1个正确节点能够获得2f+1个正确的PREPARE消息，前两种情况将不会出现大多数正确节点发起VIEW-CHANGE的过程。只有最后一种情况中会出现大多数正确节点卡在某一步的情况，在这种情况下大部分的正确节点会发出VIEW-CHANGE的请求。如果此时能够发出足够多的VIEW-CHANGE消息，那么有可能完成系统配置更改的工作。当然，在这种情况下，恶意节点应该不会配合进行配置更改，因为使得一个恶意节点成为一个首领节点对于破坏协议的流程是有利的。这里也看到之前的协议流程4里面的问题，即有可能导致协议流程执行不下去。

那么，如果出现这里所说的配置更改的情况，是不是整个系统可以在新的配置下正常工作呢？这里就涉及到新的首领节点的后续工作。有一个问题是在前面协议流程4的情况下，如果小部分的正确节点收到了足够多（2f+1）的PREPARE消息，这些小部分的正确节点的状态会进行更改。而如果新的首领节点（正确节点）完成配置更改之前没有收到足够多的 PREPARE

消息，新的首领节点是不会去更新自己的状态的。这样在系统配置更改完成之后，新的首领节点由于没有这一部分的信息，新的首领节点在新配置下的操作将从原来的序号开始。这会造成状态的分裂，因为一部分的正确节点执行了一项操作，而另外一部分正确节点开始新的同样序号的定序操作工作。这样就会形成不一致的状态，协议协商失效。

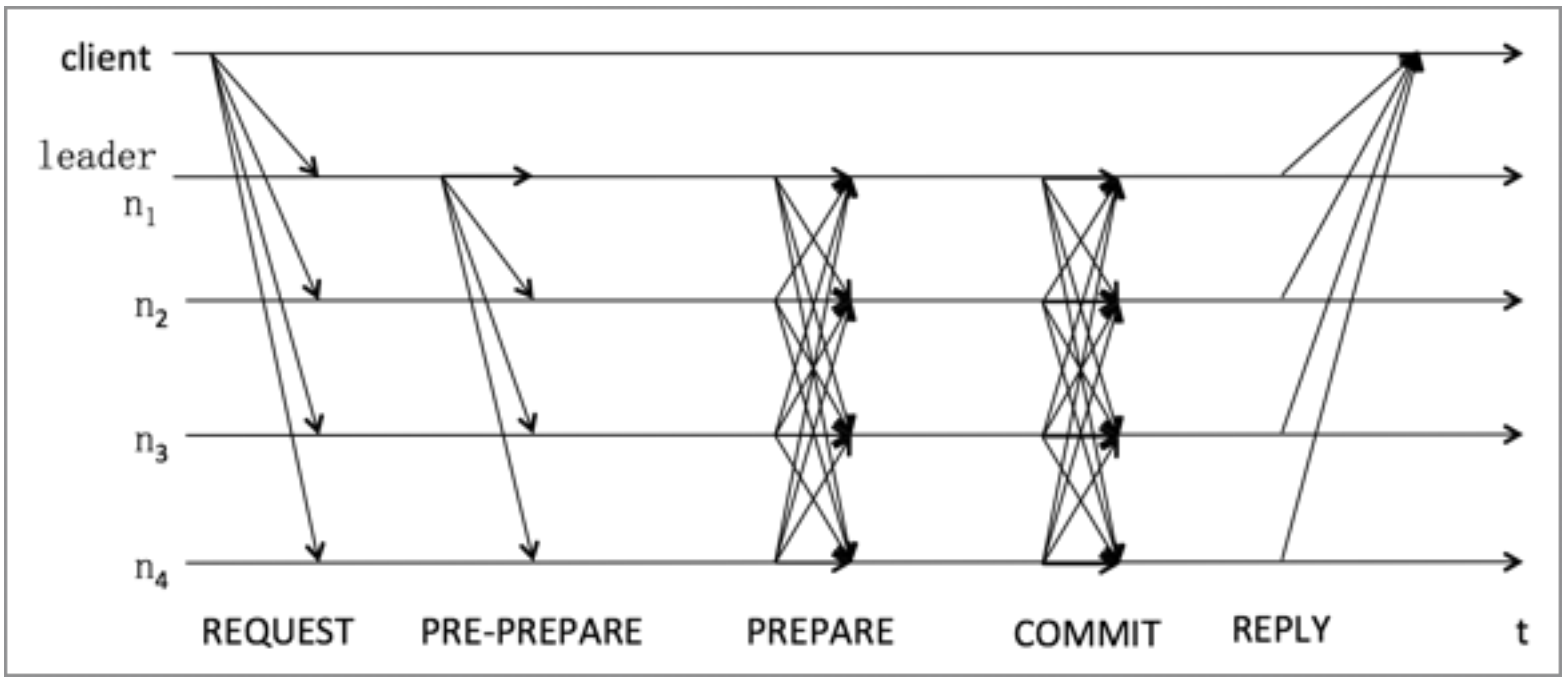
针对这个问题，在这里的一个正确系统配置状态转换的条件就是一旦一个操作被某一个正确节点执行了（即状态机的状态进行了更新），那么新的首领节点必须知道这件事情，可以进行相同工作的定序。这样就可以使得新的首领能够正确地编排跨系统配置更改之间的操作序号。有一个手段是让新的首领节点在进行任何操作之前去询问系统中的其它节点的操作情况，以便获得最后一个可能的被执行的操作的序号。但是，同样结合上述的协议流程4给出的协议，因为新的首领节点只会去等待2f+1个回答，而不是去等待所有的回答。此时，从f+1个正确节点中获得的消息以及f个恶意节点中获得的消息的总和还是有可能没有覆盖到可能进行操作的正确节点。还是会存在新的首领节点没有收到最新的操作的情况。但是，从这里的分析也可以看出，为了使得新的首领节点能够获得最新的执行的操作已经不是新的首领节点自己能够解决的问题，而是需要其它的正确节点的配合。换一句话说，对于前面所描述的协议流程4也需要进行修改。

通过前述的分析，需要给执行流程中加入一个限制，即除非已经确认有可能的新的首领节点已经知道了这个操作的情况，否则就不要执行一个操作去更新状态机的状态。前述的协议流程4的协议流程设计还需要进一步的修正，并且这里的系统配置更改方案也需要修正。实际的协议工作流程分为两个部分，分别是协议执行的流程以及系统配置更改的流程。

协议执行流程5

在这里的协议流程中，由于需要确信新的首领能够接收到已经被执行的操作，两轮的协议流程已经不够了，需要加入新的提交轮，即形成3个轮次的协议流程，包括PRE-PREPARE，PREPARE以及COMMIT。注意，同前面一样，下面的协议流程也只是正确节点所需要执行的操作，恶意节点可以

图18.11 三个阶段进行提交的协议流程



做任何想做的事情。

1）客户端将操作发送给首领节点（以及其它所有的参与工作的副本节点）。

2）首领节点给当前的操作选择下一个序号，发送PRE-PREPARE 消息给所有的副本节点。

3）任意一个副本节点收到消息之后，发送 PREPARE 消息给所有的副本节点。

4）任意一个副本节点收到2f+1个匹配的PREPARE消息之后，发送 COMMIT 消息给所有的副本节点。

5）任意一个副本节节点收到了2f+1个匹配的 COMMIT 消息之后，就可以执行这个操作。

上述的过程如图18.11所示。来看一下上面协议执行的一些性质。同前面所描述的协议执行流程4类似，如果首领节点是一个正确节点，那么可以期望整个协议流程可以正确执行下来，不需要恶意节点的帮助，2f+1个正确节点相互收发消息就可以使得操作能够在所有的正确节点之间完成协议流程，达到一致性协商的目的。

再来看一下首领节点是恶意节点的情况。下面的分析同协议执行流程4的情况分析类似。

情况1：如果2f+1个正确节点都收到了2f+1个匹配的PREPARE消息，那么也可以收集到2f+1个匹配的COMMIT操作，整个系统可以正常执行下去。在这个时候，恶意首领节点的行为和正确节点一样，没有阻止整个副本状态机的执行过程。

情况2：如果超过f+1个正确节点能够收到2f+1个匹配的PREPARE。假设最差的情况现在只有f+1个节点收到了，那么这f+1个节点互相之间可以交换COMMIT消息。这个时候，剩下的f条COMMIT消息只能中恶意节点来。对于某一个正确节点来说，它是可能在恶意节点的帮助下，收到足够多的COMMIT消息，因此在这种情况下可能出现操作被执行。另外，如果恶意节点不进行配合的话，也有可能所有的正确节点都不能够进行提交操作。这个时候，2f+1个节点会提出配置更改的要求。在这种情况下，因为卡在协议流程的某一步，不能够执行提交的操作，并且因此发起系统配置更改的正确节点数目会从f个一直到2f+1个都有可能。

情况3：如果小于<f+1个正确收到了2f+1个匹配的PREPARE，那么任意一个正确节点收集到的匹配的COMMIT消息数目就会少于2f+1个，这样的话，所有的正确都不会收到足够多的COMMIT消息。在这种情况下，所有的正确节点都不能够执行对应的操作。所有的正确节点都卡在当前的操作

中，要么在等待足够的PREPARE消息，要么在等待足够的COMMIT消息。所有的正确节点在这种情况下将发起系统配置更改请求。

从上面的所有情况分析来看，一旦有一个正确节点执行了操作，那么它必然收到了2f+1个匹配的COMMIT消息，而这其中的f+1个必然来自于正确的节点。这样的话，如果一个操作被执行，那么必然是大多数（超过≥f+1个）的正确节点都收到了2f+1条匹配的PREPARE消息。因为这里的正确节点只有收到了足够多的2f+1条匹配的PREPARE消息才能够发出COMMIT消息。因此，操作被执行的情况会出现在这里所分析的情况1以及情况2。而由于正确节点无法继续执行，进而发起系统配置更改请求则会出现在这里所描述的情况2以及情况3。后面两种情况都有可能出现2f+1个正确节点一起发起系统配置更改的请求。

从客户端执行的角度来看，与前面的协议执行类似，客户端只要能够获得匹配的f+1个回答，就可以获得当前执行的结果，使得系统可以继续执行下去。因为如果能够保证正确的节点都去执行相同的操作，那么获得f+1个匹配的请求是没有问题的，起码一个请求来自于正确节点。

这里的协议流程5在匹配系统配置更改方案1的时候，还是会出现新的首领节点尚未知有可能完成提交的更新节点的信息的情况，因此需要修改一下系统配置更改的方案，通知可能的新的首领节点已经完成提交的操作的情况。

系统配置更改方案2：

配合上面的协议执行流程5，需要设计一套系统配置更改的方案，其目的就是使得如果存在一个正确节点可能搜集了足够多的COMMIT的话，那么这个操作的情况必须要被新的首领节点知道。结合前面已经讨论过的系统配置更改方案1，新的首领节点需要搜集足够多的信息来定出下一个操作的顺序。以下是修改之后的系统配置更改方案。

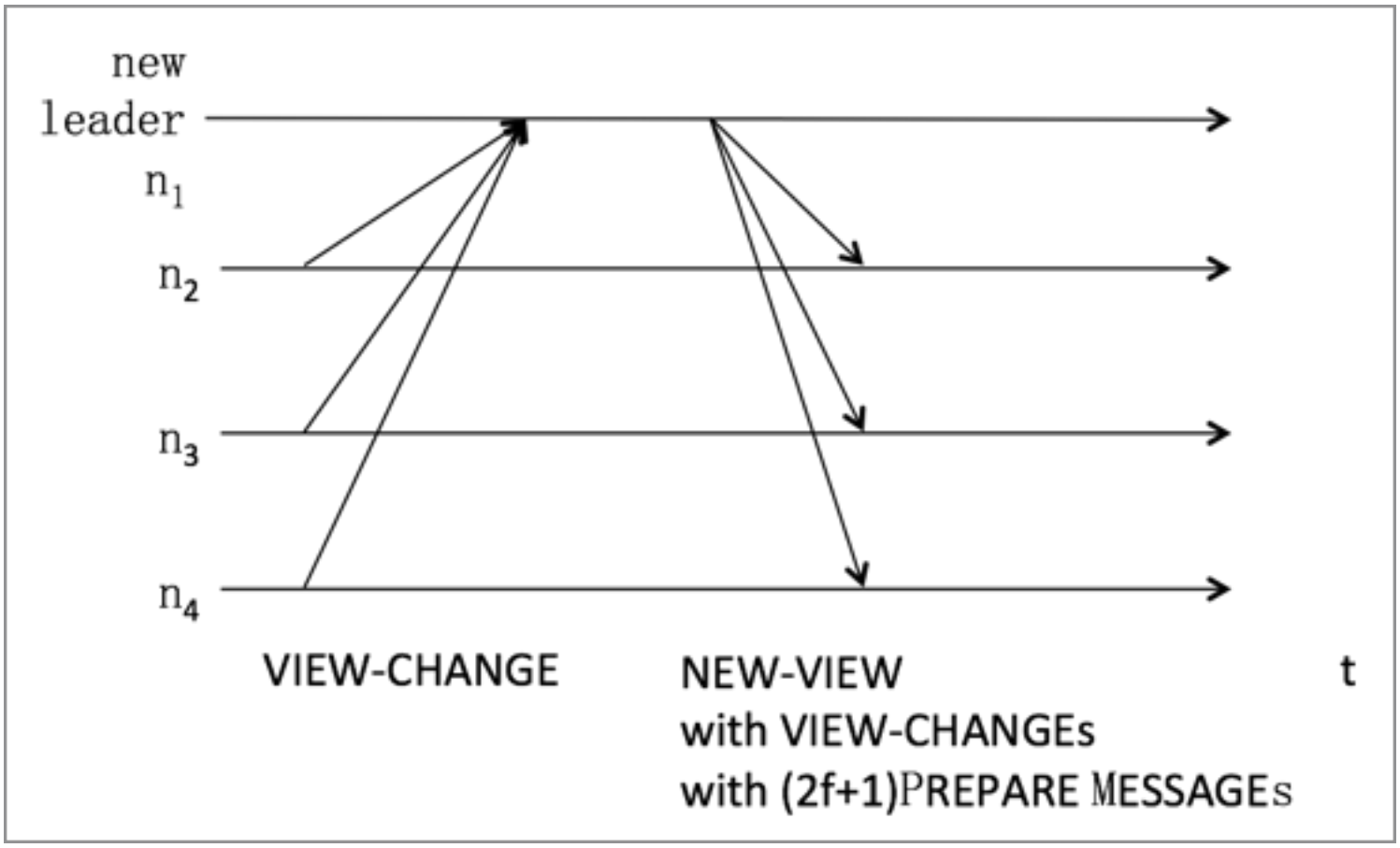
1）　每一个正确节点在发起配置更改的时候，如果这个时候其已经接收到了2f+1个匹配的PREPARE消息，那么VIEW-CHANGE的消息发送给新的首领节点的时候将附带这2f+1个匹配的PREPARE消息。

2）　新的首领节点等待2f+1个VIEW-CHANGE请求消息，这些请求消息中，一部分包含了对应的PREPARE消息，另外一部分没有附带，因为那些正确节点没有收到足够多的PREPARE消息。

3）　在新的首领节点搜集到足够多的VIEW-CHANGE消息之后，发送NEW-VIEW 消息给所有的副本节点，其附带的消息包括所有的 VIEW-CHANGE 消息以及如果某一个 VIEW-CHANGE 包含了2f+1个匹配的PREPARE消息的话，也附带这些2f+1条匹配的PREPARE消息。

可以看到，这里的系统配置更改方案基于与前面描述的系统配置更改方案基本相同。不同的地方在于为了使得新的首领节点能够接收到可能进行的操作，在发送VIEW-CHANGE的消息的时候，顺带还需要发送接收到的2f

图18.12 系统配置更改方案2的执行流程



+1个匹配的PREPARE消息。这些消息就包含了旧的系统配置中的最后一个操作动作。整个系统配置更改的过程如图18.12所示。

下面来分析一下，结合协议执行流程5以及系统配置更改方案2，如果一个操作被执行了，是不是新的首领节点就肯定能够知道这个操作呢？这里只需要考虑前面一个首领节点是恶意节点的情况。实际上，在前面描述的情况1（所有节点都收到2f+1条匹配COMMIT消息）是不会发生系统配置更改的。因此，出现系统配置更改并且附带对应的PREPARE消息只会出现在情况2以及情况3中。在情况3中所有的正确节点都不会执行对应的操作，因此这里一个操作被执行，并且在VIEW-CHANGE中附带对应的PREPARE消息只会出现在上述的情况2中。如果一个操作被执行了，那么其对应的正确节点必须收到2f+1个匹配的COMMIT消息，这些消息必然来自于至少f+1个正确节点，这些正确节点都收到了匹配的2f+1条PREPARE消息，这些正确节点起码发出f+1条COMMIT消息。而新的首领节点的产生也必须等待2f+1条对应的VIEW-CHANGE消息，这其中也必然有至少f+1条消息来自于正确节点。这样的话，对于执行操作的节点来说对应着f+1个发出COMMIT的正确节点，对于新的首领节点来说，也对应着f+1个发出VIEW-CHANGE的正确节点，这两个集合之间必然有交集，这个交集中的节点将向新的首领节点发送2f+1个匹配的PREPARE消息。首领节点必然会收到旧的系统配置中的最后一个操作。

在这里，足够多的VIEW-CHANGE消息被定义为2f+1个。对于小于等于f个以及大于2f+1个的情况前面已经分析过了。考虑一个极端的情况，例如将足够多的消息定义为f+1个，因为这里的f+1个可能是由1个正确节点以及f个恶意节点组成的。此时，这1个正确节点可能不会附带任何的PREPARE消息，因为其没有获得匹配的2f+1条PREPARE消息。但是，在系统中还是会出现可能，即某一部分的正确节点进行了操作，但是由于没有将PREPARE消息转发给新的首领节点，导致首领节点不知道旧的系统配置中的最后一个操作，会导致错误的操作顺序。以此类推，将足够多的VIEW-CHANGE的

消息定义为从f+1到2f个都是不能保证新的首领节点获得最后一个操作的。因此，需要将足够多的数目定义为2f+1个。读者也可能注意到了，这样足够的数目定义要求是比较高的，因为如果恶意节点不配合的话，这就是要求所有的正确节点都参与这个工作。这种情况会导致活跃性（Liveness）的问题，会在后面对这个问题继续进行讨论。

在整个系统完成上述的系统配置更改之后，新的节点可以依据获得的信息继续工作。如果新的节点仍然是一个恶意节点，那么他实际上可以为同样的序号定义不同的操作，期望导致正确节点状态出现分叉。但是，由于前面的协议执行流程5的保证，这样的情况不会发生。而对于一个正确的新的首领节点来说，如果在配置更改的过程中没有任何的附带的PREPARE消息，那么新的首领节点可以依据本地的信息，为尚未完成的操作选择下一个操作定下序号。例如，如果已经完成的操作的序号为n，那么对于没有完成的操作的序号将被定义为n+1。当然，如果在系统配置更改中出现捎带 PREPARE 消息的情况，新的首领节点将为附带在消息中的操作仍然定义序号为n，并且发出PRE-PREPARE消息，在新的系统配置中完成对应的操作。在这种情况下，这个已经被某一些正确节点执行过的操作，它的序号将仍然为n，对于已经执行过的正确节点来说，只需要做一次判断就可以了，不会影响状态的转换。另外，前面的分析也可以看到，在任意情况下都不可能出现某两个正确节点收到2f+1条匹配的PREPARE消息，但是这两个操作不一样，这就保证了协议的安全性。因此，新的首领节点可以安全的发出 PRE-PREPARE 消息而不需要去考虑同一个序号下 PREPARE 消息中出现两个不同操作的情况。

总结一下，看一下前面完成的协议流程针对前面描述的协议流程4的改进情况。当然，在性能上协议流程4由于少了一轮提交信息的流程，减少了消息所以会快一些，但是协议流程5在正确性上进行了保证。当然，两个协议在首先是首领节点是正确节点的情况下，都能够得到正确的结果，不会出现系统停滞不前的情况。

在首领节目的是恶意节点的情况下，也分为三种不同的情况，即情况1所有的正确节点都收到了2f+1条匹配的PREPARE消息；情况2是大部分正确节点都收到了2f+1条匹配的PREPARE消息；情况3是只有小部分节收到了2f+1条匹配的PREPARE消息。

1）对于情况1来说，两个协议的执行流程类似，因为恶意的节点不影响正确节点的执行，所以实际上是恶意节点帮助正确节点完成工作，系统不会停滞不前。

2）对于情况2来说，协议流程4能够使得大部分正确节点能够继续工作，小部分正确节点将滞后一部分操作。协议执行流程5中，如果大部分的正确节点都完成了提交，那么整个协议将可以继续执行，客户端可以收到超过f+1个回答，客户端也可以继续执行。但是，实际上也会存在只有一小部分正确节点获得足够的COMMIT消息的情况，在这个时候，这一小部分正确节点会去执行操作，剩下的大部分正确节点处于停滞状态，不能够继续执行协议流程。另外，从客户端的角度来看，实际上在这个时候也收不到足够多的回答，只能收到一小部分的回答，客户端将不能继续工作。此时，由于已经有一部分的正确节点完成了对应的操作，按照协议的流程以及系统配置更改的流程，这部分的正确节点是不会发起配置更改请求的。即使剩下的大部分的正确节点发起了配置更改请求，由于恶意节点不会配合正确节点的工作，新的首领节点不会收到足够多的VIEW-CHANGE消息，配置更改不能成功。在这个时候，协议流程将停滞下来，一致性协商失效。这个情况也就是前面所说的协议在活跃性方面的问题。下文将举出一个具体的例子来说明这里所出现的问题，并且对上述的协议做一个小的修补以解决活跃性问题，使得协议在这个时候能够继续执行下去。

3）对于情况3来说，协议流程4可能会陷入停滞的状态，即恶意节点可以进行破坏，使得客户端得不到足够数目的响应，整个系统不能够继续执行。但是在新的协议流程中，由于得不到足够多的COMMIT消息，所有的正确

节点都无法提交以及执行对应的操作。由于所有的正确节点将提出系统配置更改，整个系统将在新的系统配置中完成工作，避免出现停滞的状态。

协议执行流程的停滞状态避免与系统配置更改方案2的修正：

下面以一个具体的例子说明在上述的协议流程以及配置更改方案出现了什么样的问题以及如何进行修补。在这里，首领节点是一个恶意节点，标记为f，恶意节点的集合为F，这个成员的数目为f。（注意，这里集合用大写，首领节点用小写，数目也为f，但是应当不会引起混淆。）所有的正确节点的集合分为两个部分，一个部分有f+1个正确节点，标记为集合A；另外一部分有f个节点，标记为集合B。将协议流程按照下面的方式执行一遍，可以参考图18.13中的流程：

在PRE-PREPARE阶段：

f发送(n, op1)给A，发送(n,op2)给B。

实际上就是f会将同一个编号赋给两个不同的操作。

在PREPARE阶段：

A发送op1给B，B发送op2给A，F发送op1给A，F发送op2给B

从A的角度来看，它们获得了 2f+1个op1以及f个op2。

从B的角度来看，它们获得了 f+1个op1以及2f个op2。

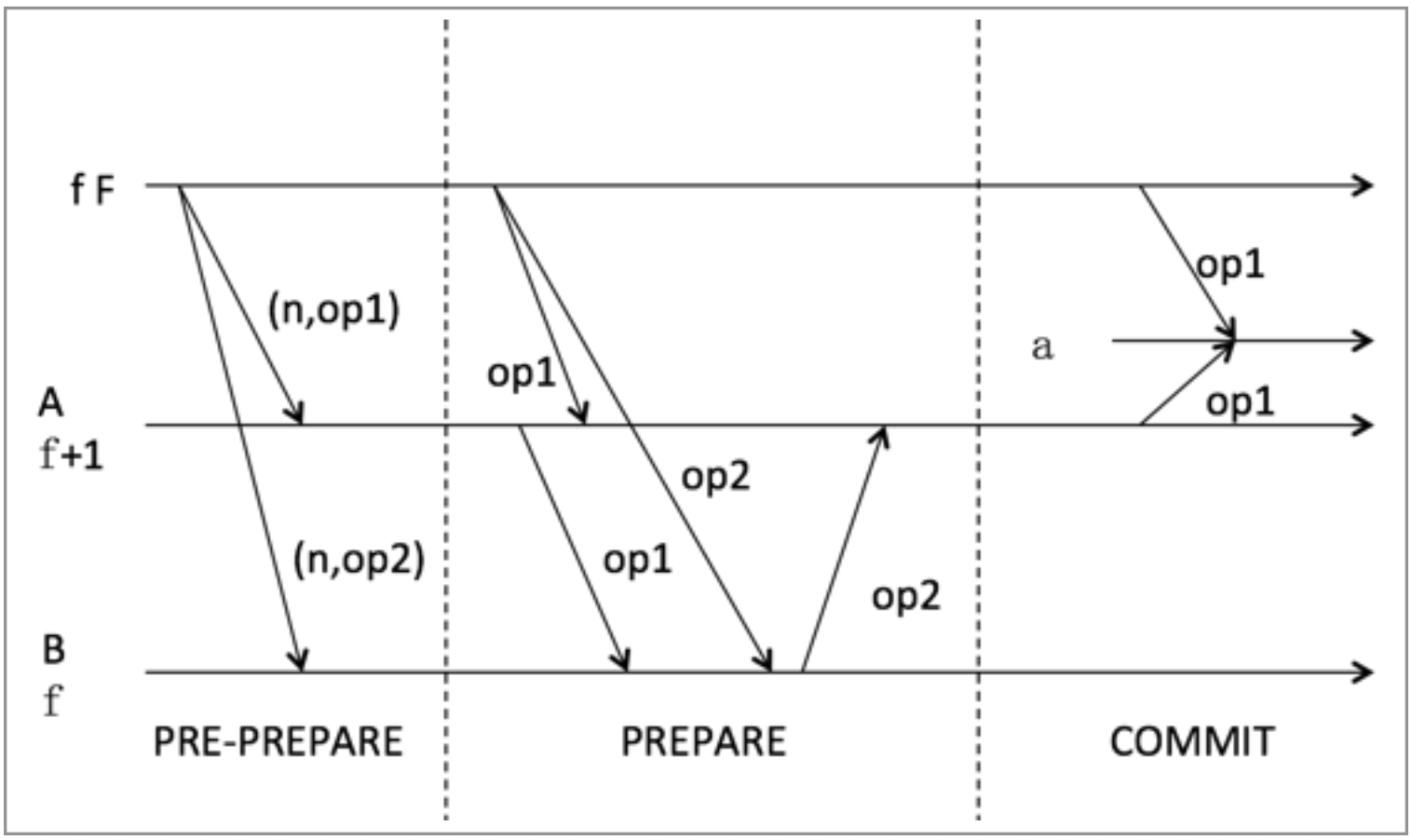
可以看到，在这个时候，只有A获得了足够的2f+1个prepare，可以进入到COMMIT阶段。

在COMMIT阶段：

从A的角度来看，里面的所有格正确节点都可以获得f+1个COMMIT消息。

再从A里面分出来得1个正确节点a，F给它补充了f个COMMIT，那么a可以执行op1，剩下的A-a里面的节点由于接受不到足够的COMMIT消息，不能执行操作op1。

图18.13 协议出现活动性问题的情况



在这个情况下，2f个正确节点(A-a,以及B)会发起系统配置更改即发出VIEW-CHANGE消息。但是，这个情况下VIEW-CHANGE消息数目是不够的，因为一次成功的系统配置更改（需要攒2f+1个），而此时的恶意节点不会发起系统配置更改。另外，从客户端的角度来看，现在这个情况只能获得一个回答，即节点a的回答，其它的节点都停滞不前了。

系统配置更改方案3：

可以看到，在协议执行流程5以及系统配置更改方案2的流程中还是会出现整个系统停滞的情况。从这里的分析可以看到，协议执行流程5能够使得新的首领节点获知旧的系统配置中的最后一个操作的情况。但是，对于系统配置更改方案来说，获得足够的VIEW-CHANGE消息的数目是2f+1个，在上述的特例中，已经有2f个正确节点发出VIEW-CHANGE还是不能够成功推动系统配置更改。结合之前的分析，系统配置更改几乎需要所有正确节点的参与。从这个角度来看，系统配置更改所需要的要求实在是太高了。实际上，如果一旦有一个正确节点发出系统配置更改的请求，实际上也就是意味着整个副本状态机处于一个非正常的状态，其它的正确节点应该力促系统配置更改的发生，而不是阻止这件事情发生。因此，在这里为了解决系统停滞的问题，对于正确节点系统配置更改进行如下的修正，即在发出 VIEW-CHANGE 消息的时候，不仅发送给新的首领节点，而且还要发送给所有的其它节点。这样的话，正确节点在收到f+1个 VIEW-CHANGE 消息的时候，可以判断起码有一个正确节点发出了系统配置更改的请求，那么这个节点如果自己本身没有发出 VIEW-CHANGE 的话，它应当也发出 VIEW-CHANGE 请求。这样就形成了最后的系统配置更改方案3（图18.14）：

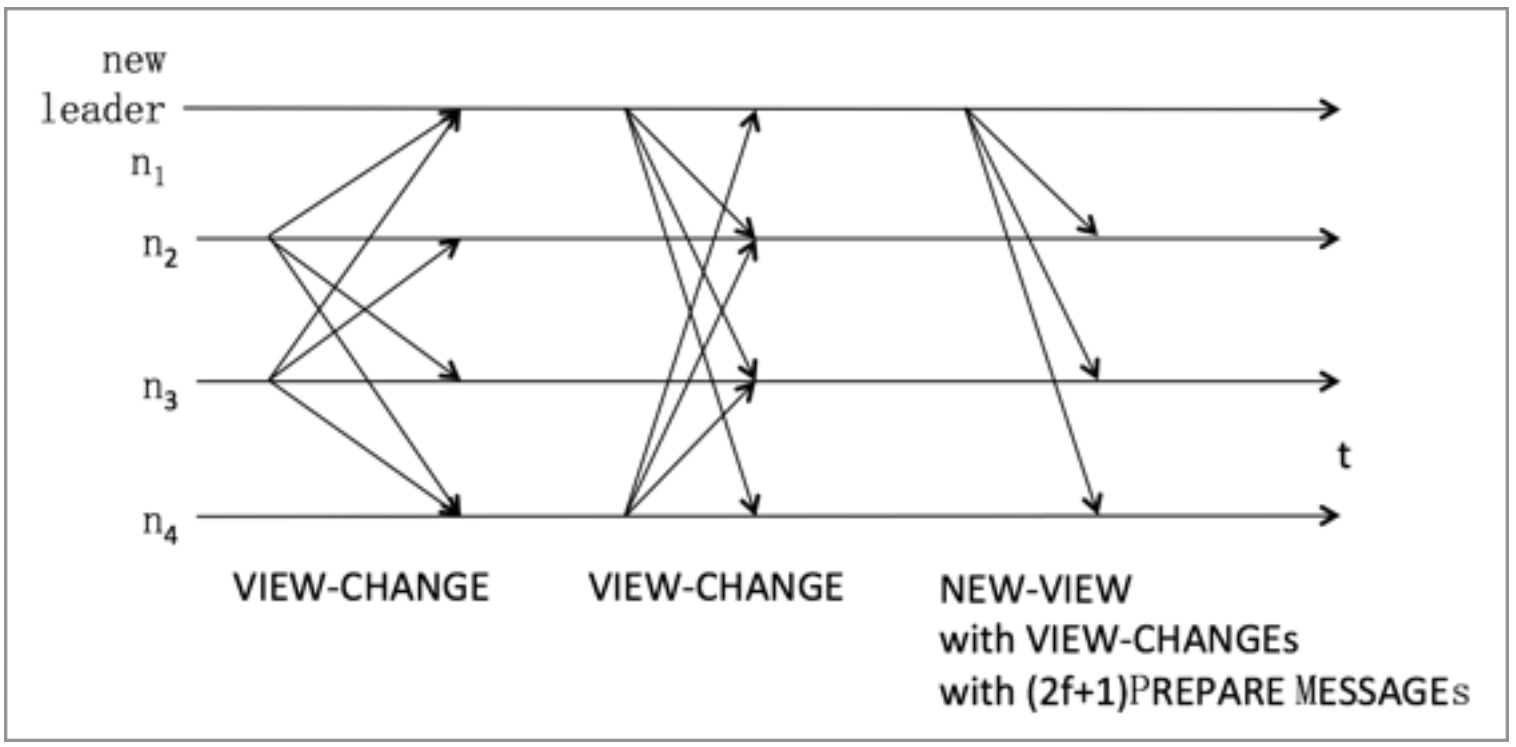
1）每一个正确节点在发起配置更改的时候，如果这个时候其已经接收到了2f+1个匹配的PREPARE消息，那么VIEW-CHANGE的消息发送给新的首领节点的时候将附带这2f+1个匹配的PREPARE消息。正确节点也将系统配置更改的消息广播给系统中的所有节点。

2）每一个正确节点如果收到了f+1个VIEW-CHANGE的消息，那么这个正确节点如果没有发出过对应的VIEW-CHANGE消息，它就应当也参与发出自己的VIEW-CHANGE消息。

3）新的首领节点等待2f+1个VIEW-CHANGE请求消息，这些请求消息中，一部分包含了对应的PREPARE消息，另外一部分没有附带，因为那些正确节点没有收到足够多的PREPARE消息。

4）在新的首领节点搜集到足够多的VIEW-CHANGE消息之后，发送NEW-VIEW消息给所有的副本节点，其附带的消息包括所有的VIEW-CHANGE消息以及如果某一个VIEW-CHANGE包含了2f+1个匹配的PREPARE消息的话，也附带这些2f+1条匹配的PREPARE消息。

图18.14 系统配置更改方案3的执行流程



通过这样的修正，特例的系统停滞的情况就被解决了，即现在节点a实际上可以收集到2f(>f+1)个VIEW-CHANGE消息，那么它也发出自己的 VIEW-CHANGE 消息，使得消息数目的总数达到了2f+1。系统配置更改的过程可以成功完成，系统消除了停滞的状态，在新的系统配置条件下继续执行。

一般的，如果在COMMIT阶段，大部分的正确节点能够获得足够多的COMMIT 消息，进而执行操作的话，整个系统可以继续执行，并且客户端会收到足够多（f+1个）的匹配的响应，也可以继续执行。但是，如果只有一小部分的正确节点收到足够多的COMMIT消息，那么这小部分会执行操作，剩余的大部分≥f+1的正确节点将发起系统配置更改的请求。依据前面修正完成的系统配置更改方案3，已经执行操作的小部分正确节点会收到超过（≥f+1）个VIEW-CHANGE消息，那么也会参与到系统配置更改的过程

中。此时，系统配置更改会成功进行，整个系统不会停滞下来。

协议执行流程5以及系统配置更改方案3是最终完成拜占庭错误下分布式系统一致性协商的正确协议。

分析到现在，关于一个可用的拜占庭将军问题协议方案已经完整的提出以及进行了非形式化的证明。整个协议包括协议执行流程5以及系统配置更改方案3的总体还是相当复杂的，但是整个协议的讨论过程还是具有很大的启发意义，建议读者仔细体会。

思考题

1 如果客户端是不可靠的，那么系统会发生什么情况?

参考文献

本章的内容实际上与经典的拜占庭问题是有一定差距的。经典的拜占庭问题可以参考这个文献：本章的内容来自于下面的论文。本章的内容主要是分析一个协议方案的正确行，但是在实际的系统中，不光需要考虑正确性，还需要一个高效的实现。在原始论文中还考虑了对于上述协议流程的优化，超出了本书的范围不再赘述，建议有兴趣的读者可以参考原文进行分析。

第二十二章 匿名网络

本章将讨论一个非常重要的系统，即Tor匿名网络。Tor网络往往被形象地称为是一个洋葱网络，在本章的后面读者将会感受到为何这样一个网络被称为是洋葱网络。

本章以及下一章与现代密码学密切相关，可以说是现代密码学在分布式系统中的一个重要的应用。有一些读者可能不是非常了解现代密码学的结构，本章将首先介绍一些现代密码学的相关内容。由于本书不是现代密码学的具体研究，因此本章的内容只是现代密码学一些算法属性的分析，有进一步了解需求的读者可以翻阅相关的密码学专业书籍。

本章的主要内容包括以下三个方面，如果读者对于现代密码学已经非常了解了，可以直接阅读匿名网络的相关内容。

- 现代密码学的算法与特性：如果需要理解匿名网络，一些密码学的基本知识是必须的。深入的知识可参考密码学专门书籍。
- 网络代理与安全数据传输：所有的数据在匿名网络中都以密文形式传输，TLS协议是结合了密码学的安全信道传输，也是建立Tor网络的基础。深入的知识可参考TLS的规范文档。
- 匿名网络：匿名网络是进行网络匿名访问的基础，这部分介绍Tor网络所使用的各项技术。

现代密码学的算法特性

本书不是现代密码学的详细介绍，但是为了理解匿名网络以及进一步理解由现代密码学构造的分布式系统，必须要对现代密码学的整个体系结构有一个完整的了解。现代密码学也构成了许多分布式系统的理论基础，理解整个现代密码学的知识体系是非常必要的。

现代密码学的主要内容分成两个重要的组成部分，一个是用以信息的加密与解密的密码学，另外一个是密码分析学。密码学主要用以对信息的加密以及解密，使得通信的双方可以通过加密的方式进行通信，即使信息被中间截获了，由于缺乏密钥信息，截获的信息不能获得解密，从而可以保证信息不会被泄露给除了通信双方的第三方。而密码分析学则从另外一个方向进行研究，即如何对加密解密系统进行破解，进而可以分析加密解密算法的安全特性，帮助理解加密解密算法的抗攻击强度。在这里只对密码学的体系进行一个简单的介绍。

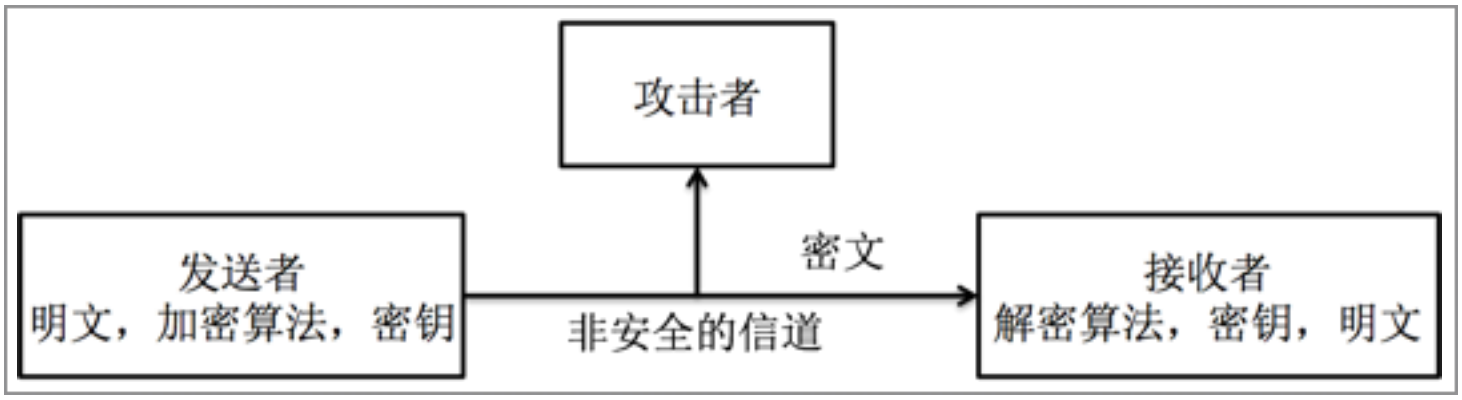
在现代密码学中，主要的密码算法分为三个类型，分别是对称密钥加解密算法，非对称密钥加解密算法，哈希算法。另外，值得注意的一点是，现代密码学的算法都是公开的，任何一个人都可以检视密码学算法的正确性以及可以着手进行分析。现代密码学的安全性不依赖于算法的不公开特性，通过将算法公开，使得任何人都可以对算法进行检查，实际上也就对算法的特性进行了检查。对于加密获得的密文带来的解密的难度依赖于所使用的密钥。密钥的长度越长，一般认为安全性也越好。

22.1.1 对称密钥加解密算法

加密通信的基本形式

数据的加密和解密是传统的数据安全通信的一个基本的手段。为了说明数据安全通信的流程，在这样一个场景下面会涉及到几个运行中的角色，包括消息的发送者，消息的接收者以及可能的攻击者。

图22.1 数据的安全通信的参与者



在图22.1中，消息在发送之前需要先经过加密的方式，在网络中以密文的形式在网络中传输数据。数据经过加密之后，即使在传输的路径中被攻击者截获，那么攻击者获得的是加过密后的数据，攻击者没有密钥不能进行解密。数据接收者获得数据的时候需要进行解密，将密文数据解密为明文数据。加密算法除了算法本身之外，还需要密钥参数。密钥会保留在发送者以及接收者两端，其它人不会知道，因此也就保证了密文数据的秘密特性。加密算法一般是公开的，公开也能够保证算法能够被更多的人进行检验。

对于攻击者来说，当然是希望获得明文的信息。在加解密通信的模型中，攻击者一般会被假设为能够进行网络监听，获知一部分甚至全部的加密的信息。如果能够获得密钥的话，那么攻击者将获得明文信息，达到攻击的目的。因此，加密算法就需要保证攻击者在没有密钥的时候无法恢复出明文的数据，而通信的双方则需要保证密钥的安全性，不能泄露密钥。

由于密钥的安全性是由密钥的使用者进行保证的，这就有可能使得攻击者通过某一些技术手段甚至社会工程学的手段来获得密钥。例如，通过软件的漏洞可以获得密钥，此时可以对截获的数据进行解密。甚至如果进行历史数据累积的话，也可以对历史数据进行解密，这就说明了密钥的重要性。为

了保证一次数据通信的安全性，这里面会涉及到一个被称为前向安全性（forward secrecy 或者是完美的前向安全性 perfect forward secrecy 即PFS）的概念。将在讨论一些加密算法之后来讨论如何达到前向安全性。

对称加密的基本形式

对称加密的是最容易理解的一种加密的形式，也是传统的密码学中延伸到现代密码学的概念。之所以被称为对称加密，原因是用以加密的密钥和用以解密的密钥是相等的。

图22.2 对称加密与解密算法流程

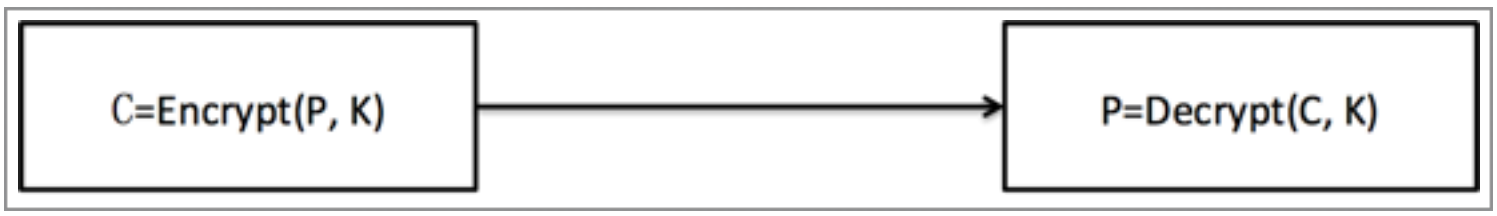


图22.2是对称加密的图示，通信双方都用同样的加密算法进行加解密，使用的密钥也是相同的密钥。在进行通信的时候，发送端使用加密算法Encrypt以及密钥K对一段明文P进行加密，并获得一个密文C。密文C会在网络上进行传输。在接收端，收到密文C之后，接收者会使用解密算法Decrypt以及相同的密钥K对密文C进行解密，并获得原文P。当前通用的对称加密的算法包括DES（已经废弃），IDEA，AES加密等算法，通常会在实际系统中使用AES加解密算法。在这里只是一个方向的通信，而实际的系统中是存在两个方向的通信的，另外一个方向的通信与这里的方式一致，通过同样的加解密的过程完成通信。

在对称加密中，密钥的长度是一个重要的参数。一般来说，密钥的长度越长，可以认为获得的密文的强度越强，也就越不容易被攻击者所攻破来获得明文。这是比较容易理解的属性，即对于密钥的攻击中，一个重要的攻击

方式就是进行穷举，而穷举的范围与长度的指数相关，长度越长，穷举的空间将按照指数的方式爆炸。

在对称加密中，有一个通信的双方都需要进行认可的密钥，这个密钥必须在通信之前进行由通信的双方持有。当然，这个密钥不能在通信的信道中进行明文传输，否则会被攻击者截获，使得安全通信无法实现。对称加密密钥交换一种方式就是通过离线方式进行交换。但是离线的方程会带来额外的密钥分发和管理的负担。在后面讨论加密通信的例子的时候，可以通过公钥密码体系来完成这样一个密钥信息的交互。

22.1.2 非对称密钥加解密算法

现代密码学带来的一个重要的发展就是非对称加密形式的提出。这里面有一个非常著名的算法，即RSA算法。本章不去讨论RSA算法的本身，读者可以从任意一本密码学课本中获得RSA算法的细节。在这里仅仅讨论一下非对称加密的基本形式与性质。非对称加密顾名思义就是加密密钥和解密密钥是不相等的。信息的发送者通过一个其中一个密钥进行加密的工作，信息的接收者必须通过另外一个密钥才能够进行解密的工作。

图22.3 非对称加解密算法的流程

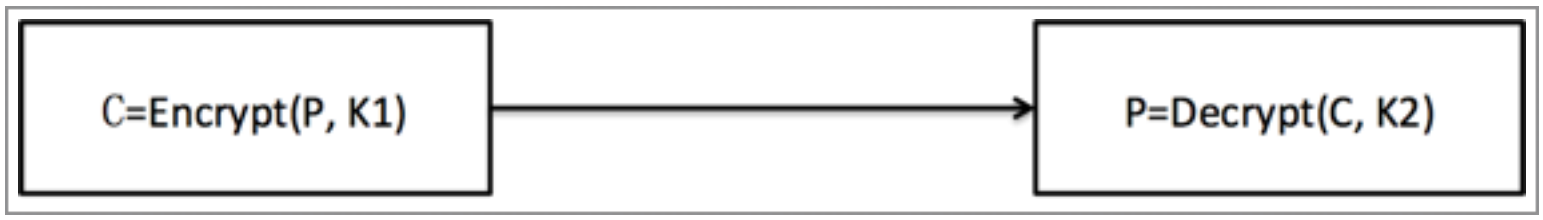


图22.3是非对称加密的结构图。加密是通过一个密钥K1进行的，而解密是通过另外一个密钥K2进行的。在实际的工作过程中，对于信息接收者来说，是不需要密钥K1的，因为这个密钥对于解密没有什么帮助，信息接收者只需要保存K2即可。因此，对于信息的接收者R来说，他可以预先生成一对密钥K1以及K2。R可以将K1公布出来，让所有的人都知道。任何一个人

希望往R那里发送消息，只需要通过K1进行加密即可。由于只有R才持有K2，因此只有R才能够对这个数据进行解密，任何其它人获得密文之后并不能对数据进行解密。这里，对于接收者R来说，K2被称为其私钥，因为R是私人持有，不公开；而K1被称为是公钥，公布出来被所有人知道。这样的话，非对称密钥也往往被称为是公私钥，这样的一个加解密体系也被称为是公钥密码体系。

公私钥的方式能够对密钥管理带来好处，原因是可以公开公钥而不再需要进行密钥的线下或者不安全的交换。实际系统中为了避免中间人攻击而需要建立一个公开密码认证体系。（PKI：Public Key Infrastructure）

对称加密与非对称加密是两种不同的加密通信方式。一般认为，非对称加密的方式对于信息的保密性方面要优于对称加密的方式。在非对称加密的方式中，密钥的长度也是一个重要的参数，长度越长也能够保证更好的安全性。当前的公钥密码体系中，RSA仍然是一个重要的算法，RSA的密钥长度现在更多的被设置为2048位，这样一个长度是很难通过穷举的方式进行攻击的。可惜的是，非对称加密体制在带来更好的安全性的同时，也带来了加解密计算的低性能。在进行实际数据加密的时候，对称加密方法对于数据的加密过程要比非对称加密方法要快几个数量级。因此，实际的通信需要将两个加密体系融合在一起，达到安全性以及速度的平衡。

证书与身份认证

证书以及身份认证是公钥密码体系提出之后一个重要的系统性应用。由于公钥体系中，每一个实体R都具有一个私钥公钥对<K1,K2>，同前面讨论的一样，这里的K1为公钥，K2为私钥。因此可以用这一个公私钥对作为这个对象的唯一的标识。公私钥对可以形成电子签名，即通过K2可以对一段信息I进行加密，并且将明文以及密文一起呈现出来，形成这样的信息：<I, RSA(K2, I), K1>。世界上的任何一个人都可以对这一段信息进行验证，即通过公钥K1对加密的信息进行解密，看看这两段与明文信息是否一致，一致

的话就可以确保信息I是由R生成并签发的。签名是不可伪造和不可抵赖的。因为只有R才能够签名出上述的信息，任何其它人都不可能形成这样的信息，不能进行伪造。同样是因为只有R才持有私钥K2，因此R不能对信息I进行抵赖。

通过公私钥对带来的电子签名特性，可以建立一整套认证体系。认证中心（CA：Certification Authority）一般是公开被认可的可信任的机构。认证中心通过自己的公私钥对可以对用户颁发数字证书，这个数字证书直接表明了用户的身份信息以及用户的公钥。用户的私钥不需要提交给认证中心，而是需要自己进行安全维护。数字证书就表明了一个用户在网上的身份信息，其它的用户可以依据自己对于认证中心的信任程度来决定自己的操作。由于证书包含了用户的公钥，其它用户可以明确知道自己的通信对象。

图22.4 服务器证书的验证过程

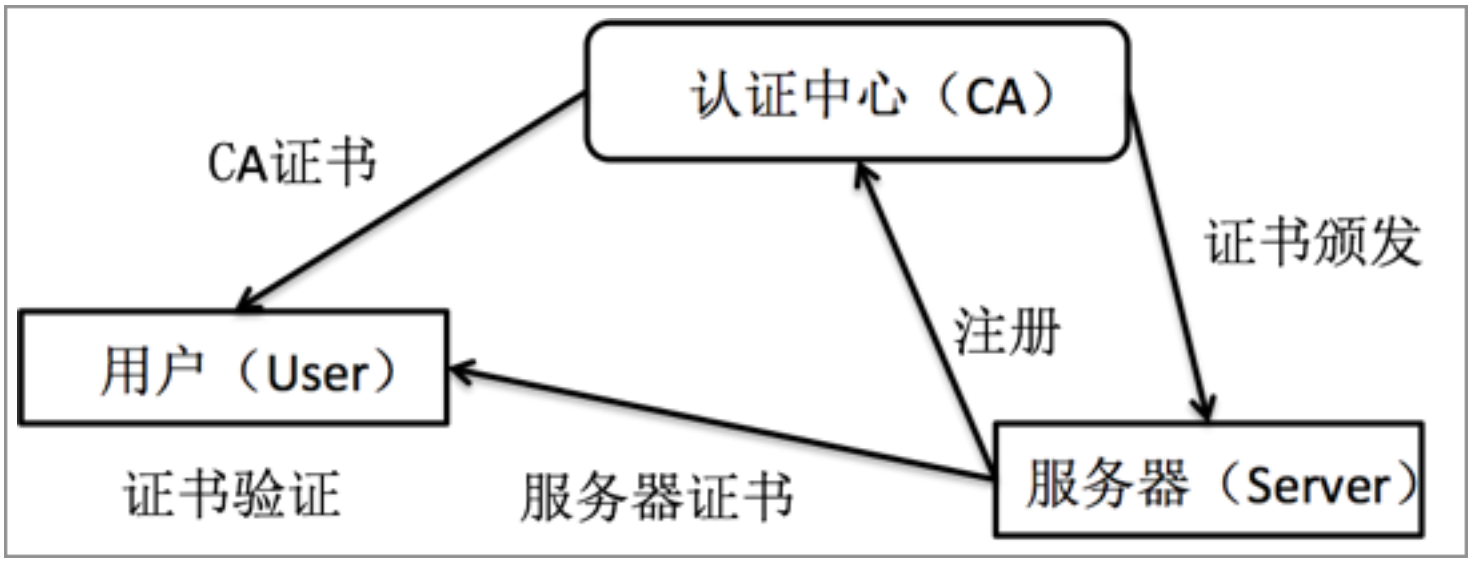


图22.4给出了进行身份认证的必要模块以及认证的流程。在图中，一个用户试图与一个服务器进行通信。服务器会预先与一个认证中心进行注册，获得认证中心的服务器证书。服务器证书包含了服务器的公钥，并通过认证中心的私钥进行签名。用户也会预先装入认证中心的证书。在通信之前，用户可以通过认证中心的证书对服务器证书进行证书验证，从而确认服务器的真实身份。这种方式也可以避免前面所说过的中间人攻击，因为通信的时候可

以确认通信对方的真实身份，而不用去理会中间人给出的证书。

再来看一下性能方面的情况，在实际使用中，对称加密的速度要远远快于非对称加密的速度。对称加密的速度往往可以达到线速，即不影响网络数据传输的速度，而非对称加密的速度则不能够跟上这样的速度。由于这样的特点，实际的应用也体现了这一点，非对称加密使用最多的场合是签名与认证，而对称加密则更多用于数据的加解密。由于非对称加密对于密钥分发以及密钥管理上的特性，实际系统往往将这两者结合在一起。在通信的发起阶段通过非对称加密算法交换出一个会话使用的对称密钥，之后使用这个密钥进行通信，例如传输层安全协议TLS: transport layer security就是这么工作的。

21.1.3 密码学哈希

在现代密码学体系中，有一大部分内容是关于哈希算法的。哈希算法实际上是一种信息摘要的算法，即将一段规模比较大的数据转换为比较短的信息摘要，例如信息摘要的长度为256个字节。哈希算法的一个重要的作用是用于信息的判重工作，看看两段数据是否一致。如果两段数据的哈希值不一样，它们包含的数据必然是不一样的，因此不需要进行进一步的比较。

在密码学中的信息哈希算法还有一个哈希强度的特性，即通过哈希算法之后获得的信息摘要与原有的信息没有直接的联系，从哈希值中不能获得任何关于原来信息的线索。从哈希值中将不能够获得原始数据的任何信息，恢复原始数据变得不可能。即使是依据一个哈希值来构造出与原文相同的哈希值的另外一个数据也是非常困难的。

由于哈希函数能够将一块规模大的数据缩减为规模较小的数据，而对于较小的数据进行非对称加密能够获得比较短的时间，哈希算法可以被用来用于数字签名。一般的方式是对需要签名的信息进行哈希函数计算，获得信息的摘要，然后对这个摘要信息使用私钥进行签名。之后在进行验证的时候也

进行同样的哈希函数计算，通过公钥对签名进行解密，进行比对之后就可以验证签名是否有效。

21.1.4 前向安全性与Deffie-Hellman密钥交换协议

关于现代密码学的内容最后一项需要介绍的是前向安全性以及与之相关的密钥交换协议。关于前向安全性的一个假设是通信的内容（经过加密的内容）有可能被攻击者在通信的信道中截获，这样截获的信息可能会被攻击者长期保存。随着时间的推移，如果某一部分的密钥信息被泄露了，那么保存的信息就会被解密。为了避免这种情况，需要一种前向安全性（Forward Secrecy） 的加密与使用方法，使得将来某一个事件持有的永久密钥（例如证书和私钥）发生了泄露也不会对历史上的通信数据产生威胁。

由于一般在通信的时候使用的是对称加密的方法，密钥需要在通信之前双方协商完成。如果将这样的密码保存在永久存储（例如硬盘）中，那么就有将来泄露的可能，使得已经完成的通信在将来某一天被解密。因此，这样的密码最好不要在永久存储中出现，一种方式是每次通信前进行随机会话密钥的生成（即这里的的对称密钥）。因为有了公钥体系，通信双方可以依赖于公钥体系每次通信之前互换一个临时的会话密码，并且在使用完成之后这个会话密钥就进行丢弃处理。由于会话密钥不会在永久存储中出现，因此不必担心会话密钥的泄露。但是，这里的会话密钥需要在每次通信之前通过网络进行交换，交换的时候通过证书的公钥进行加密，因此可以说是会话密钥依赖于证书所对应的私钥。如果将来私钥也发生了泄露，那么通过私钥可以解密出会话密钥，因为这部分公钥加密的会话密钥曾经在网络中出现过。可以说，这种方式也不具有前向安全性。

Deffie-Hellman 密钥交换协议能够完美解决这个问题，因为通过这里的密钥交换协议保证会话密钥不会以任何的形式出现在网络中，从而也避免了保存数据以便以后进行解密的可能。Deffie-Hellman密码交换协议依赖于数论的相关知识，计算流程可以通过下面的例子来说明。

通信的双方都认可了一个公共的信息：即使用一个质数p=23，以及一个底g=5（注意，这里的5是23的原根，关于原根的概念可以参考数论数据，即使不了解也不影响对于这个密钥交换协议的性质的理解）。

通信双方各自选定一个随机数，并且这个随机数不需要告诉对方以及任何的第三方，这样的随机数可以被认为是双方各自的秘密。假设双方的随机数分别为6和15。双方可以进行各自的计算，其中一方的计算方式是：$5^6 \bmod 23 = 8$；另外一方的计算方式是：$5^{15} \bmod 23 = 19$。之后双方交换这里的计算结果，即这里的8和15可以在网络上进行互换。互换完成之后，双方将结果与自己本地的秘密进行进一步计算，其中一方的计算为：$19^6 \bmod 23 = 2$；另外一方的计算为：$8^{15} \bmod 23 = 2$。可以看到，通过这种方式，双方计算的结果是一样的，这个计算结果将会作为会话密钥所使用。从性质上来看，由于双方都不知道对方的秘密，并且每次计算的时候都是随机生成自己的秘密，因此没有任何信息是可以在将来泄露，并威胁到上面的计算出来的会话密钥。因此，从这点意义上来说，即使是会话的所有信息都被攻击者所记录，并且双方将来的私钥（如果使用公钥体系的话）发生泄露。由于这些信息都不会用来获得每次临时的会话密钥，每次生成的各自的秘密以及对应的会话密钥都不会保存的永久存储中，因此这样的信息加密通信方法可以被认为是具有前向安全性的。

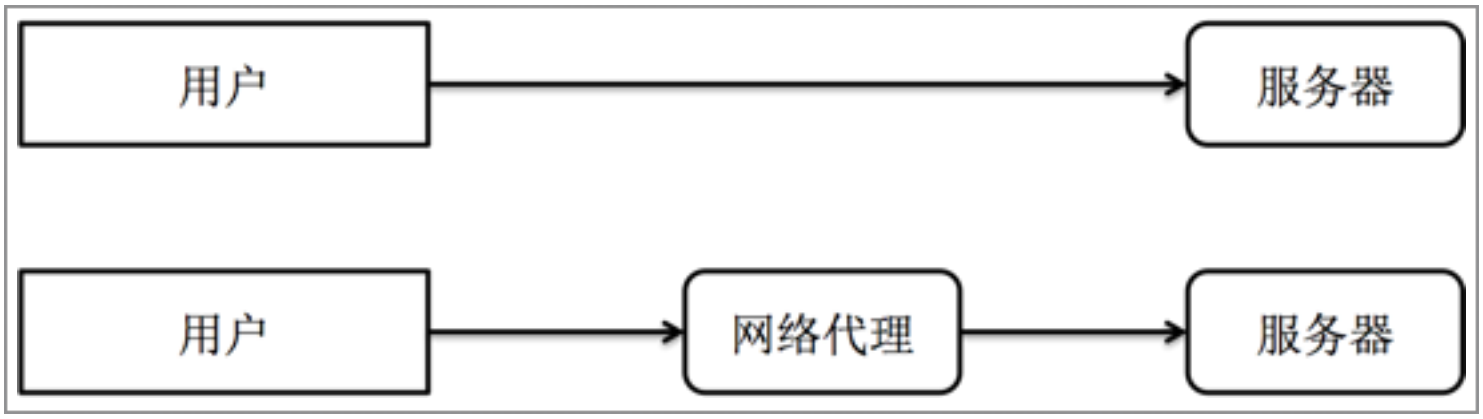
网络代理与传输安全协议

这一节介绍两个方面的内容，一个是网络代理，一个是传输安全层协议的工作流程。对于匿名网络来说，网络代理是必须的，因为不能够直接访问所需要访问的网站，否则的话就会泄露自己的访问企图。使用传输安全层协议也是必须的，因为如果明文在网络上传输，同样也就失去了安全性。

22.2.1 网络代理

网络代理的目的是避免让客户端直接访问服务器端。网络代理的结构如图22.5所示。

图22.5 直接访问与代理访问



从图中可以看出，正常的是直接访问，用户直接与服务器建立连接（例如TCP连接），之后用户可以与服务器进行直接的通信。通过代理服务器的话，用户首先跟网络代理服务器进行通信，之后网络代理服务器代替用户与服务器进行连接，所有的网络流量会首先通过网络代理服务器进行通信。使用代理，而不是直接连接服务器进行访问是有很多原因的，例如可以代理绕过防火墙特定的端口关闭。另外，有的时候通过网络代理的方式可以绕过网

络中特定的繁忙的传输线路，反而可以提高网络的访问速度。通过网络代理来提高访问速度也是经常采用网络代理的原因。

在后面讨论的匿名网络中，也需要使用网络代理的方式去访问目标服务器，而不是直接去访问目标的网站。这是一个必须需要做的工作，原因很显然，如果直接访问的话，通过网络窃听就立刻会知道哪一个用户访问了哪一个网站，也就没有匿名访问的特性了。

网络代理能够工作在多个层次上面，能够支持不同的应用程序。例如，可以在浏览器中设置HTTP代理，网络浏览就可以通过代理进行。但是，这种工作在应用层上的网络代理服务器只能支持特定的应用。例如这里的HTTP协议，而不能支持其它类型的应用。虽然HTTP协议已经可以支持一大类的网络应用程序，但是对于全面的网络应用程序支持还是不够的。为了能够使得网络代理支持更多的应用程序，需要让代理服务器工作在更加底层的网络中。在这些网络代理服务器中，一个典型的代理服务器就是SOCKS代理服务器。SOCKS代理服务器能够工作在TCP的层次，这样的话所有的基于TCP的网络应用程序都可以使用SOCKS代理进行转发。在匿名网络中也将会使用SOCKS代理来帮助用户将流量匿名化，能够支持尽量全面的网络应用程序。这不仅仅是应用方便性的要求，也同时是匿名性自身的要求，只有足够多的用户使用匿名网络，才能够保证足够的匿名性。否则，如果只有少数的用户，通过全量的分析就很容易将用户以及流量对应起来。

关于网络代理的内容就先介绍到这里，这部分的内容足够能够帮助读者理解匿名网络的工作原理。网络代理的协议以及使用方面的内容可以参考相关的专门书籍。

22.2.2 传输安全协议

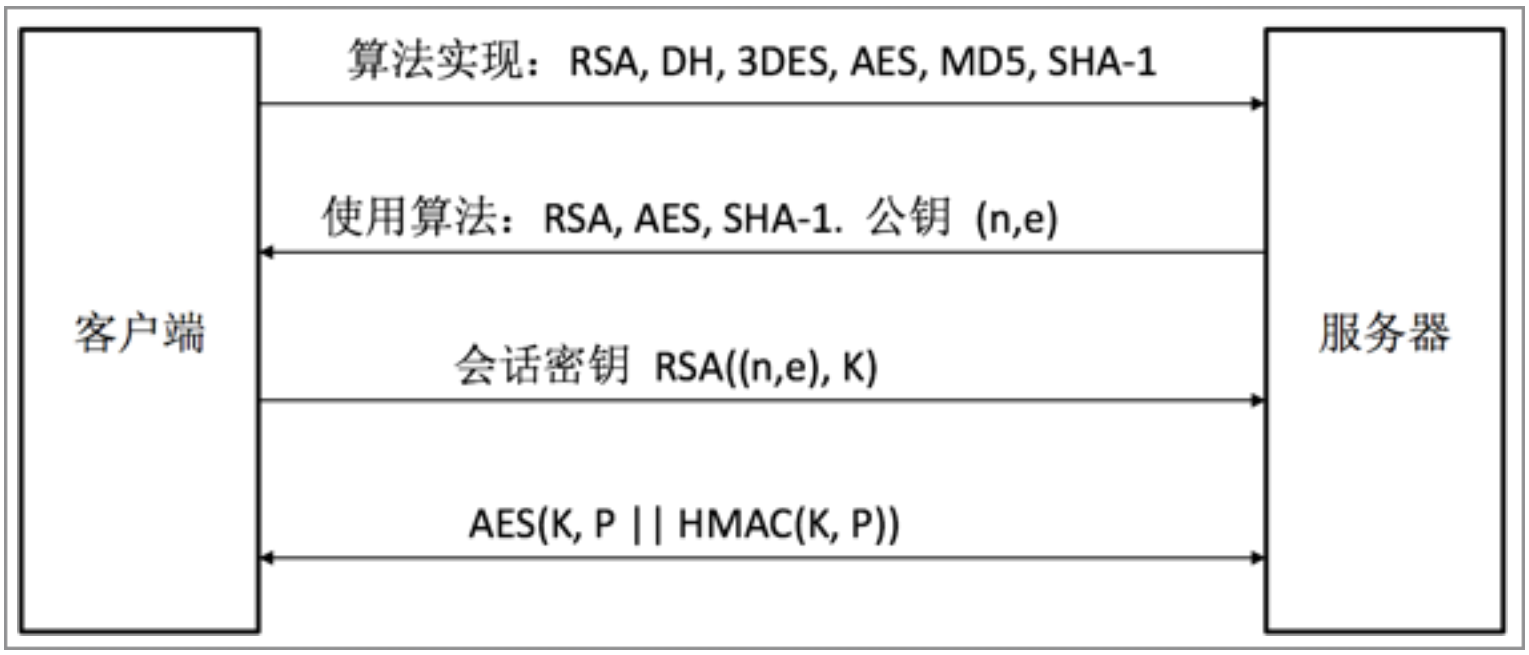
传输安全协议有的时候被称为是SSL（Secure Socket Layer），有的时候也被称为是TLS（Transport Layer Security）。实际上它们是一回事，可以认为TLS是SSL的升级版。SSL协议是Netscape开发的用以保护网络通信的协

议，在这个协议中使用了大量的加密解密算法，数据签名认证等方面的内容，现在已经被认为是网络中数据安全加解密方面的标准实现，在客户端以及服务器端有着广泛的应用。匿名网络也建立在TLS的基础之上对所有的数据进行加解密的工作，这样也就无需自己建立一整套安全数据传输的机制。在匿名网络中使用数据加解密技术，使得所有在网络上的数据都以加密的形式传输，这也是必须的，因为如果明文传输的话，非常明显会对匿名性构成威胁。

下面对传输安全协议的总体进行一个简单讨论，目的是帮助用户去理解匿名网络结构中的数据传输流程。

在传输安全协议中，包括了各种的密码学算法，概括来说，包括了对称加密，用以真正的数据传输；非对称加密用以进行用户认证；密码学算法用以辅助签名和验证等工作。图22.6是SSL协议的握手过程。

图22.6 传输安全协议的握手过程



从图中可以看出，握手过程会分为几个步骤。分别解释如下：

1）客户端到服务器端的算法协商：由于客户端和服务器端是分别独立实现的，每一端实现的密码学算法并不相同，为了能够使得双方能够用相同的

算法进行通信，客户端会将自己已经实现的算法告诉服务器。值得注意的是，在对称密钥算法，非对称密钥算法和哈希算法类别中，起码实现其中的一个。

2）服务器选择算法以及提供公钥：服务器会依据自己的情况，从客户端实现的算法当真选择自己认为合适的算法，将算法的选择情况告诉给客户端。服务器端会尽可能选择认为更加安全的算法。在返回信息中，服务器也向客户端展示了自己的公钥。为了防止中间人攻击，客户端需要通过公钥体系验证服务器的证书。

3）客户端在获得服务器端响应的时候，可以通过随机数发生器产生一个会话密钥，这个会话密钥只是会在当前的会话中使用，使用完成之后就会被丢弃。在这个时候，客户端通过服务器的公钥对这个会话密钥进行加密，并返回给服务器。服务器通过解密就可以获得这个会话密钥。

4）在上述的过程中，已经建立了一个会话密钥，那么客户端和服务器双方就可以通过这个会话密钥进行加密通信了。由于对称密钥加解密算法的良好性能，可以保证数据的快速加密传输。为了保证数据的完整性，在通信的时候也对数据进行完整性校验的工作，即通过HMAC方法进行完整性的传输和校验。

这里一个新的概念是关于HMAC的机制。HMAC的全称是Hash-based Message Authentication Code，即哈希运算消息认证码，可以用以内容的认证。例如，传统的密码验证的时候要么将密码传输到服务器端进行验证，要么将密码的哈希或者加密的形式传输并由服务器验证。但是这些方法会带来重放攻击的风险（使用相同的数据包再次进行验证）。HMAC机制则由服务器先生成一个随机数发送给客户端，之后客户端使用与服务器预先商量好的密钥K对密码进行加密，并与这个随机数进行混合计算哈希，将哈希值传输给服务器端。服务器使用相同的方法来获得哈希值，并进行验证。在网络中传输的是哈希值，而不是密码或者任何密码的加密形式。可以看到，对于

攻击者来说，只能记录的是随机数以及HMAC的哈希结果，并且每次都不一样，因而无法依据这样的信息恢复密钥，也就不能构造出合法响应来通过验证。

以上的流程是一个简化的流程，大致刻画了SSL握手相关的工作流程。在具体的实现中，会在上述的基本框架中进行调整，例如上述过程可能会导致前向安全性的缺失，因此在协商会话密钥的时候，会使用 Deffie-Hellman 密钥交换协议来替代这里的使用公钥进行传递的方式。另外的调整也包括对于客户端的身份验证，上述的过程中仅进行了服务器的身份验证。

在完成上述的握手协议之后，下一步的工作服务器和客户端将通过所建立的会话密钥进行数据通信。

在匿名网络中，将使用每一个服务器作为匿名的网络代理，并形一个代理服务器网络。所有的流量都将通过这里所说的传输安全协议进行加密。对于外部的攻击者来说，用户将流量导入到一个代理网络中，并且代理网络将数据导出到目标网络中，所有的流量都是经过加密的。这样的话，从外部的攻击者来说，就很难将一个进入到代理网络中的流量与流出代理网络的流量联系起来，从而起到了匿名的目的。在下一节中，将开始讨论匿名网络的机制，读者可以看到在设计匿名网络的时候，是如何综合利用各种技术来达成最终的目的。

匿名网络

这一节内容是匿名网络的设计。匿名网络可以有不同的实现形式，本节的重点是介绍当前正在部署使用的匿名网络，即Tor网络。Tor网络的全称是The Second-Generation Onion Router，即第二代的洋葱网络。洋葱形象地描述了网络中进行多层加密多层解密的结构。本节将首先从第一代的洋葱网络开始分析，进而深入到现在所使用的匿名洋葱网络。

22.3.1 网络流量监控攻击与混合网络

匿名路由的需求

随着互联网中的监控设施越来越完善，在互联网中隐藏自己的身份信息也变得越发的困难。在这里，隐藏身份信息包括对于内容的隐藏，更重要的是隐藏自己的网络访问行为，即尽量避免攻击者将某一个流量与某一位具体的用户或者某一个具体的IP地址练习起来。

具体的说，在匿名网络中隐藏自己的身份信息是一种常用的手段，虽然这种手段在近几年来也受到了政府规模监听的挑战。通过这个网络访问互联网的用户希望去隐藏自己访问互联网的行为，例如隐藏自己的源地址ip，隐藏自己访问的目标网站，隐藏自己的各种身份信息例如浏览器的标识等。总之，客户希望能够隐藏自己的一切信息，以至于攻击者不能够将一次或者多次的网络访问行为与一个特定的用户联系在一起。

流量分析攻击

匿名网络访问的防护方式是与攻击者的具体攻击行为相关的，匿名网络访问的行为试图避免的是受到流量分析等的被动式攻击。可以将网络中的攻

击技术粗略分为主动式攻击以及被动式攻击两种，当然，具体实施的攻击往往是两种攻击技术的结合。主动式攻击行为是由攻击者主动发起，利用对方网络以及主机的一些漏洞，进入到对方的主机系统，并获取，或者破坏对方机器中的一些信息，从而达到攻击的目的。被动式攻击行为主要的目的不是为了破坏对方网络中的系统，而是对网络中的内容以及行为进行监听。流量分析是一种常用的被动攻击行为。从匿名网络的角度来看，主动攻击在攻击目标用户之后能够获得目标用户的所有的网络行为，访问行为匿名化也无从谈起。因此，在匿名网络中主要防护的是被动式攻击。在实际系统中，主动攻击往往比较困难，也容易被被攻击者所发现，因此如果试图进行网络行为监听的话，一般不会过多采用主动式攻击方法。被动攻击不存在这样的问题，被攻击者还往往没有意识到自己已经遭受了这样的攻击，因此被动攻击是网络监听，网络审查，网络行为分析的主要手段。

在流量分析中，攻击者往往会在互联网中的一个或者多个位置设置监听设备，从网络中获取流量，并将这个网络流量进行实时分析或者存储之后进行离线分析。在具体实施上，被动攻击可以通过多种手段进行：例如可以监听目标用户端的网络，控制用户附近的网络代理服务器，数据交换设备，甚至是路由器就可以获得网络流量。如果这个网络流量不是加密的，那么可以从流量中获得一些明文，进而也就获得了诸如用户名，访问网站等的一些信息，可以勾画出被攻击用户的网络行为。由于流量分析是不可避免的，因此匿名网络希望在访问网络之前就建立一种机制，使得任何的网络流量手段在获得网络上的流量数据之后不能够解析出任何与用户身份和访问行为的信息。从这里的分析也能得出第一个结论，即网络流量中不能包含有明文的信息，加密是一项达到匿名网络的必须技术。

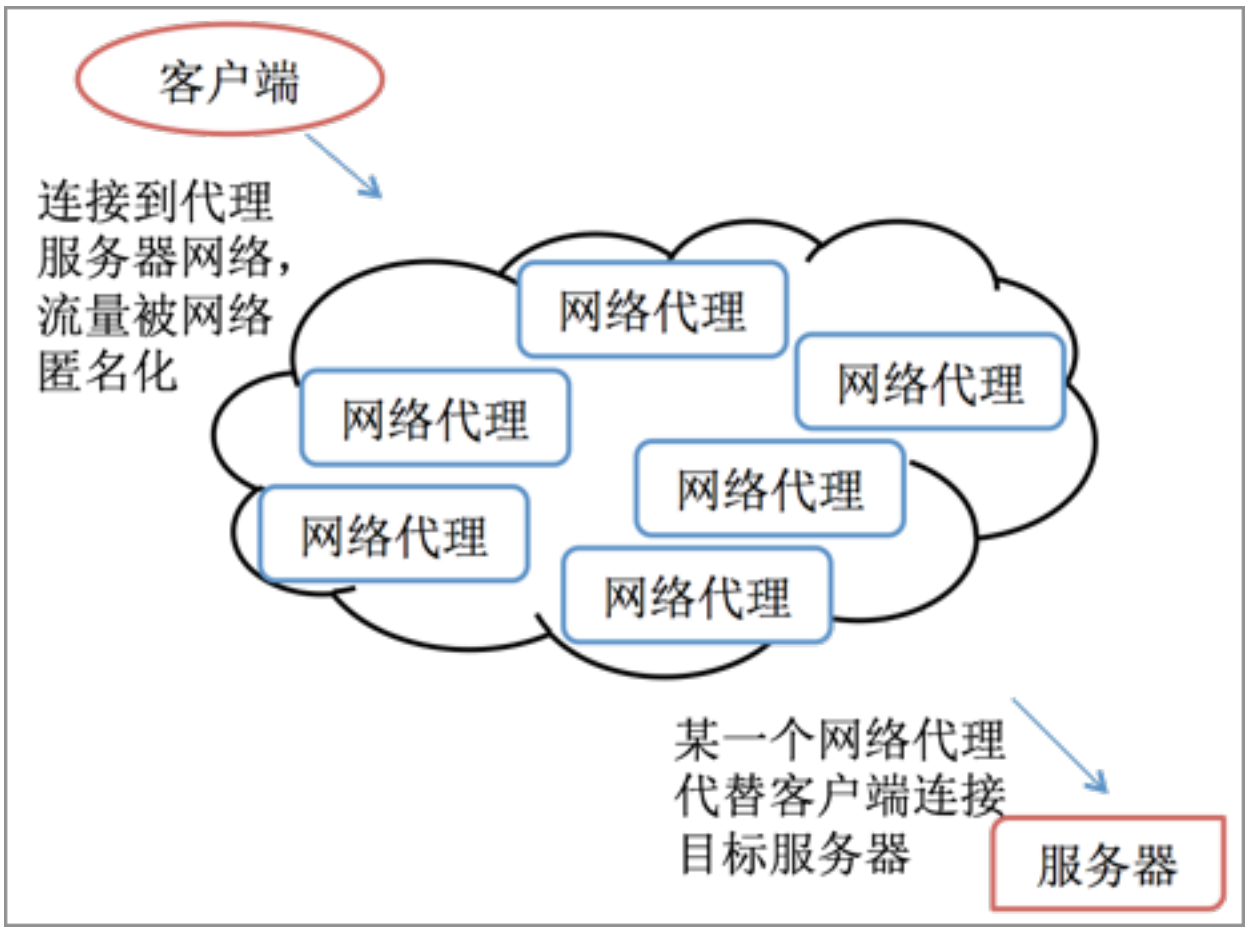
从攻击者的能力来看，假设流量分析的攻击者可以进行任何形式的监听工作，可以实施一定的主动攻击获得一部分主机的控制权，可以获得网络边沿（即用户端网络或者服务器端网络）的控制权，可以安排自己的服务器或者路由设备到网络上进行协助流量分析工作。但是，攻击者的能力不是无限

扩大的，由于互联网上设施的数目非常的庞大，攻击者只能去控制一小部分的设备，而不是大部分或者全部的设备。显而易见，如果攻击者能够控制全部的网络设备，那么只要监听用户的所有出入流量即可，沿着用户数据流量的路径就可以获知目标用户进行网络访问的信息。这里的对于部分网络可能被攻击者所控制的假设是对于普通的黑客行为是合理的，因为普通的黑客攻击不太可能获得数目非常庞大的网络设施的控制权。但是对于政府级别的有组织的监听来说可能这个假设就不成立了，有可能导致匿名网络在这种情况下不能发挥作用。

通过代理网络隐藏访问行为

从前面的分析可以看到，为了达到匿名访问的目的，直接访问目标网站是不合适的，因为这就会暴露用户的访问行为。在网络中明文传输也是不合

图22.7 利用网络代理服务器网络建立匿名网络



适的，因为明文中包含了太多有用的信息去揭示具体的某一个用户。这就提示要建立匿名网络的话，必须要构造一个代理网络以及将数据在节点之间加密传输。

从使用代理服务器访问网络的角度来看，显然使用一个网络代理服务器是不够的，因为一旦这个代理被攻击者所控制，匿名性也就被破坏了。实际系统中会使用数目众多的代理服务器来形成一个代理服务器网络。如果使用多个代理服务器形成网络的话，在系统的总体上就会形成如图22.7这样的结构。可以看到，在这个网络中，与之前所描述的一样，客户端连接的是一个代理服务器网络而不是目标网络，对于攻击者来说，很难将一个特定的从代理网络出去的流量与一个进入代理网络的流量对应起来，这样就完成了匿名网络的基本的功能。

对于加密通信来说，可以使用之前所描述的SOCKS代理。SOCKS代理能够对应用的流量进行透明加密处理，应用不需要做任何的修改，适应了各种应用程序，进而可以吸引更多用户来使用匿名网络。

代理网络的最重要的功能实际上是路由的功能，即如何将用户的数据包在各个网络代理服务器之间进行路由。底层的物理路由已经通过物理的路由器进行了接管，这里的路由指的是在网络代理服务器之间的逻辑路由。一种方法是可以通过代理服务器网络帮助进行数据包路由，例如在代理服务器内部记录数据包的路由信息，并依据这个路由信息将数据交给下一个代理服务器或者最终发送给目标网络服务器。但是，这样的建立在代理服务器网络内部的路由方法是不合适的，因为这种情况下每一个网络数据包需要在代理服务器的协助下在代理服务器网络中进行转发，攻击者只需要沿着这条路径就可以获得用户访问的信息。这样会构造出某一个用户具体的路由路径，完成了对于匿名路由的攻击。因此，从匿名路由的角度来看，不能去信任任何其中的一个特定代理服务器，需要保证每一个代理服务器只能获知其前序的节点以及其后序的节点，而不知道整个网络路径的其它的节点信息。

如果路由信息不能够由代理服务提供，那么路的信息只能由需要进行匿名访问的用户提供，即在进行网络访问的时候利用网络中的代理服务器构造一个数据的路由路径，并通过某种机制保证数据就按照这条路径传递。由于针对某一个具体的数据包来说，某一个代理服务器只需要知道数据包的下一个代理服务器就可以完成数据的转发，因此对某一台代理服务器来说仅知晓数据包的来源服务器以及数据包的目的服务器，不会知道数据包的全部路由信息，从而也就限制了某一台代理服务器所能够获得的信息。攻击者控制了某一台服务器只能够获得局部的信息，而不是全局的信息，从而也就达到了匿名的目的。

当然，上述路径的最后一个节点是需要与目标网络服务器进行通信的，这部分的网络通信是数据流量离开代理服务器网络的流量。因此，路由中的最后一个节点也往往被称为退出节点（Exit Node）。从退出节点发出的流量必须要符合原有网络应用程序的特性，如果原有网络应用是通过明文进行传输的话，从退出节点到目标网络服务器的流量也是明文的。如果明文中包含了有关用户的一些信息的话，匿名访问的特性也会被破坏。这也就提示网络服务器如果想提供基于用户的信息服务的话，也需要采用数据加密的方法。当然，在这里如果只是类似于提供新闻服务的网站的话，使用明文也是没有任何问题的，因为这样的访问不会泄露任何的用户信息。

下面来看一下用户端如何为代理服务器网络提供路由信息，以及如何控制代理服务器进行路由工作。由于需要由客户端自己构建数据包在网络中的路由情况，所以客户端必须要获得可用的代理服务器的列表，这需要一些服务器来充当目录服务器提供这样的列表。后面讨论Tor网络的时候会讨论在Tor网络中的目录服务器。客户端在获得这样的列表之后，可以自行构造所需要的路由。由于每一个客户端都是独立选择在代理服务器网络中的路由，这就使得网络路由基本上可以认为是均分在所有的代理服务器上，加大了流量分析的难度。

在客户端选择完成自己的路由之后，例如，客户端选择了三台代理服务器P1，P2，P3，路由的路径按照P1，P2，P3这样的顺序，其中P1是客户端第一个联系的节点，P3是退出节点。依据之前讨论的公钥加解密系统的方式，可以按照下面的方式来控制数据包在代理网络中的路由。即对于一个数据Data来说，客户端需要准备的数据如下（选用RSA加密方法）：

D3=RSA（（目标服务器地址，Data），P3的公钥）

D2=RSA（（P3地址，D3），P2的公钥）

D1=RSA（（P2地址，D2），P1的公钥）

在这里，RSA加密方法的第一个参数是对应的数据，第二个参数是对应节点的公钥。数据中的第一项是需要传输的对方的地址，第二项是传输给对应地址的数据。可以看到，三个数据包D1，D2，和D3只能被对应的服务器P1，P2，和P3所解密。在这种模式下，客户端将数据包D1发送给P1，P1获得信息之后可以使用自己的私钥进行解密，获得数据包D2以及下一个数据代理点P2的地址。随后代理服务器P1可以将数据发送给代理服务器P2，而后者也可以用同样的方法来将里面的数据包传输给代理服务器P3。代理服务器P3是退出节点，现在可以使用客户端传输过来的数据与目标服务器进行通信，完成与目标服务器的一次通信。

但是，以上的过程的一个问题是数据如何返回给客户端。一种方法是使用相同的方法返回，即可以让目的服务器选择路由到客户端，但是这种方法是需要将客户端的地址泄露给目的服务器，这是不允许的。因此，这里的数据返回路径也只能通过客户端来控制，整个网络中的任何代理服务器以及目标服务器都不知道完整的路由信息。例如，在前面的例子中，如果希望数据包按照原路进行返回，客户端可以准备如下的返回地址，与请求数据包一起到达退出节点，退出节点可以据此来发送返回的数据。

（P2的地址，（P1的地址，（客户端的地址）P1）P2）P3

注意，在这种情况下，返回的消息也需要通过路由器的一层一层加密，否则的话就会被路径上的所有代理服务器获得通信的内容。上述的返回数据包可以帮助代理服务器来层层解密获得下一步的路由地址，最终将数据返回给客户端。

可以看到，上述的方式能够构造出一个匿名的路由系统。在这个系统的数据传递路径上，就像剥洋葱一样一层一层剥除，最终获得内部的实际数据。因此，这种方式的路由也被形象地称为是洋葱路由。上述的代理混合代理网络可以被称为是第一代的洋葱路由。这种洋葱路由有一个缺点会影响其实用性，即其依赖于非对称加解密算法，而这样的算法的计算速度是非常慢的，会影响数据的传输，这对于一些流式数据来说几乎是不可用的。因此，从性能角度来看，还需要设计更好的匿名网络来支持更多的应用正常的执行。下一节就会分析第二代的洋葱网络，即Tor网络。

22.3.2 Tor匿名网络系统架构

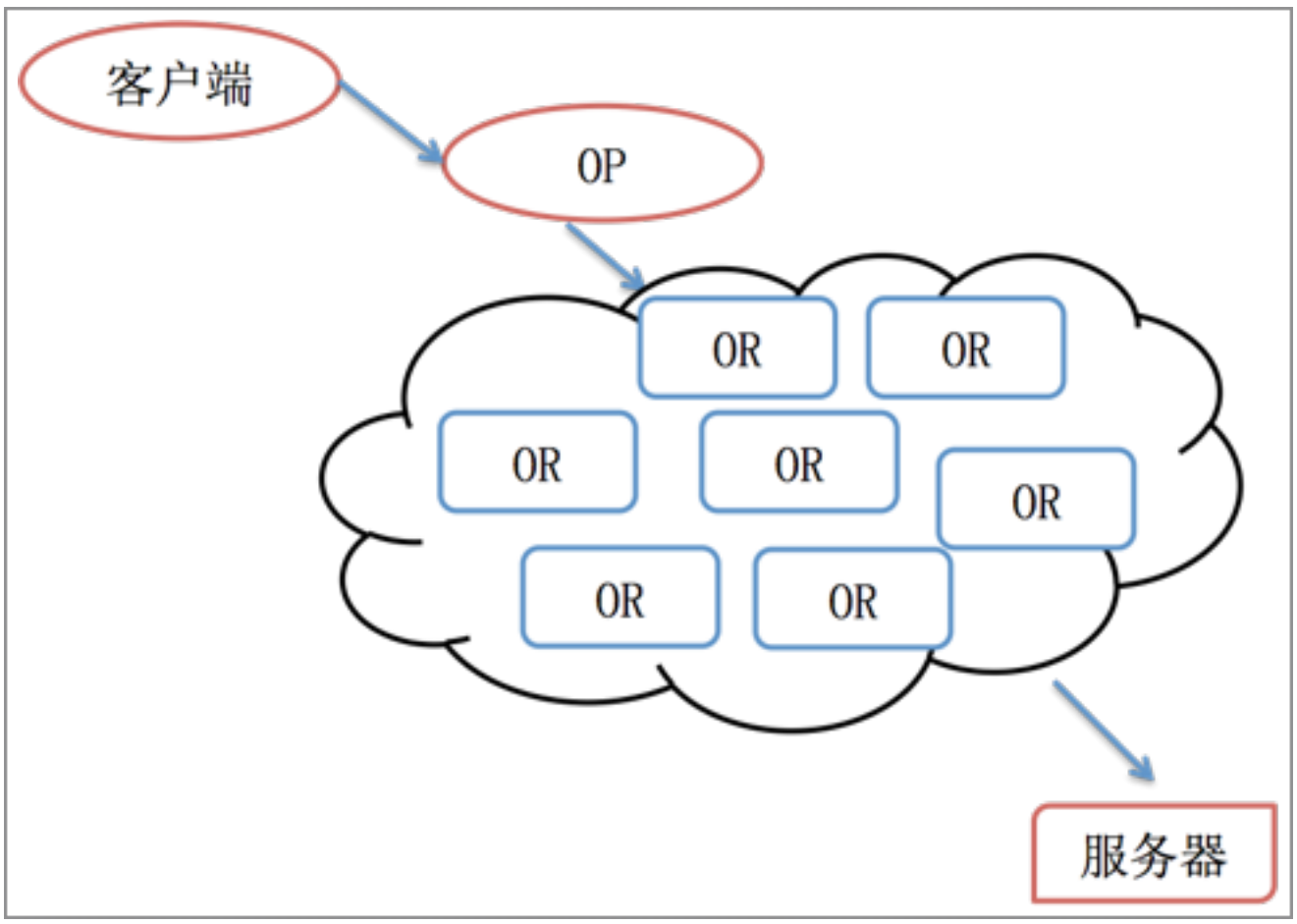
Tor的基本结构

在Tor网络中，也采用了类似于前述混合代理网络的形式来完成对于流量的转发。Tor网络对于上述的代理服务器方案的一个突出的改进是通过对称加密的方式提高数据传输的性能。另外，Tor在客户端提供了SOCKS代理的方式，可以匹配大部分的网络应用程序而不需要应用程序自己进行修改。这样可以吸引尽可能多的用户参与使用，也就能够在更高的程度上实现访问的匿名性。

在Tor网络中，客户端的应用程序与本地的一个代理服务器进行通信，这个代理服务器被称为OP（Onion Proxy：洋葱代理）。注意，OP是本地的代理，与应用程序运行同一个物理节点之中。之后，信息会通过本地的OP发送给Tor网络中的代理服务处，所有的代理服务器也被称为是OR（Onion Router：洋葱路由）。网络流量在Tor网络中流动，最后会在一个退出节点

（Exit Node）中将流量导向到目标网站。图22.8给出了Tor网络的代理服务器网络体系结构。

图 22.8



可以看到，这里的Tor网络与之前讨论的代理服务器网络几乎是一样的。这里除了OP需要是SOCKS代理之外，所有的OR可以不是SOCKS代理，只需要作为一个网络流量转发工具即可。从网络流量转发来说，对于基于TCP的网络应用，现在OP需要和某一部分的OR建立TCP连接，而OR之间可以建立TCP连接，最终由退出节点与目标网络服务器建立TCP连接，完成网络数据流量的转发。下面将讨论在这个结构下的工作细节。

Tor网络中的线路（Circuit）与数据流（Stream）

从前面的分析中也可以看到，在匿名路由中需要由客户端自己建立连接的路由过程，在Tor网络中也不例外。客户端需要通过目录服务器获知可用的OR列表，并从中选择所需要的路由服务器。由于需要使用对称加密的方

法来建立数据的连接，因此客户端在收发数据之前与每一个被选择的路由服务器协商所使用的加解密所需要用的密钥。例如，某一个OP选择了三个路由器OR1，OR2，OR3，并且协商完成了三个密钥K1，K2以及K3。在这其中，OR3为退出节点。这种方式建立起来的数据转发路径被称为Tor网络中的线路（Circuit）。在Tor网络中，多个TCP连接可以共享一条线路，在这里称每一条TCP连接为一个数据流（Stream）。在OR之间以及OP和OR之间的连接都使用了TLS连接，为了区分不同的线路，即可以使用同样的TLS连接来支持多个不同的线路，每一个OR都为每一个线路维护唯一的标识号，即CircID。同样的，由于对于每一条线路来说，可以支持多个用户层级的TCP的连接，为了区分不同的TCP的连接，也需要为每一个数据流分配标志符，被称为是StreamID。每一个Tor的数据包格式为512个字节，具有两种不同的格式，分别用于线路管理以及数据传输。

图22.9 Tor网络中的数据包格式

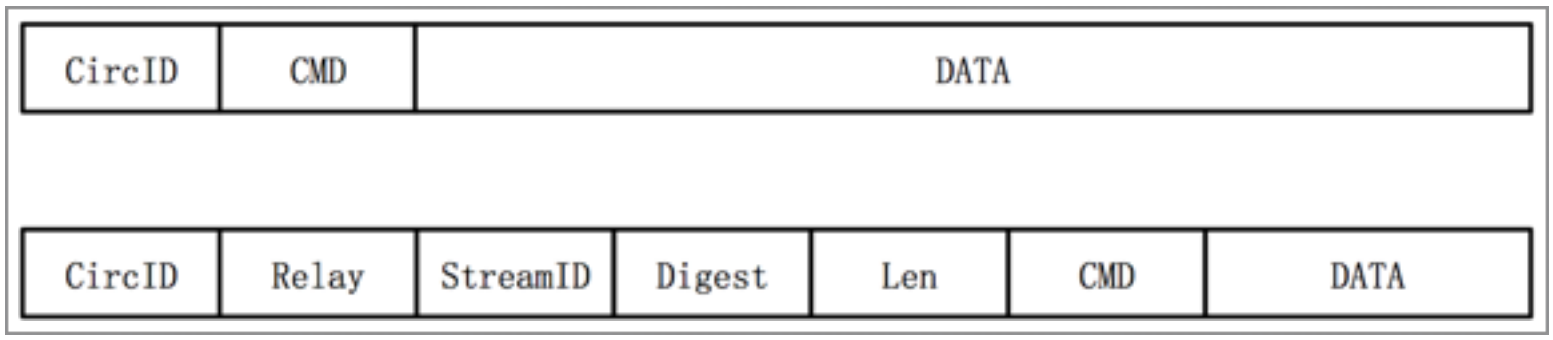


图22.9是Tor网络中的数据结构，清晰显示了在Tor网络中的数据格式。可以清楚看到CircID以及StreamID在数据包中的位置。在上述讨论的基础上，为了理解Tor的工作原理，无非就是去理解一下线路的创建以及数据流中数据包是如何转发的。

线路的创建

由于在客户端，应用程序首先连接的是本地的洋葱路由OP，并且由洋葱路由去连接Tor网络中的洋葱路由器，与前面的讨论一样混合网络代理服务器一样，这里可以由OP去选择路由。可以假设由OP选择了三个洋葱路由器OR1，OR2，OR3，并按照顺序建立线路。

在Tor网络中，创建线路的过程是逐步创建的。OP第一步将建立从OP到OR1的线路，创建线路可以使用一个特殊的命令，比如CREATE消息来完成。在这一步创建线路的时候，主要工作就是通过Diffie-Hellman来交换密钥，商议密钥K1。完成这一部分工作之后，所有通过OP发送给OR1的数据都通过K1进行加密。之后，这条线路可以被延长到OR2，方法是发送一条EXTEND的消息到OR1，跟随的是OR2的地址，附带于这条消息的是OP已经选定的Diffie-Hellman的一半握手消息。OR1将这部分数据拷贝到一个CREATE命令中，这样线路会被延长到OR2。在返回的时候，OR1会将返回的数据返回给OP，这样就会在OP和OR2之间协商完成一个密钥K2。同样的方法会被用来将线路延长到OR3。

通过上述的流程，在OP和OR1，OR2，OR3之间个协商出来一个对称加密的密钥K1，K2，K3。由于使用了Diffie-Hellman的密钥交换协议，获得密钥都是临时的，并在本次会话完成之后就会被丢弃，从而达到了前向安全性。

从上述线路建立的过程可以看出，攻击者获得一部分的流量信息，也无法获得全部的线路路由信息。特别的，如果攻击者控制了一个路由器（通过主动攻击的方式，使得这个路由器完全听从攻击者的指令），也只能获得这个路由器的前序和后继，而不能获知其它的线路信息。并且在网络中，可能会有多个线路会通过这个路由器，攻击者不能够确定下一步的攻击对象。这就提示攻击者只能够获得局部而不是全局的信息。当然，攻击者可以控制这个路由器将数据流量转发到另外一些攻击者控制的设备中，但是由于流量的密钥是通过Diffie-Hellman密钥交换算法与对应的路由器建立会话密钥的，除了指定的路由器，其它的路由器都不能对数据流量进行解密。这就保证了

数据流量必须按照OP指定的路由转发而不能被控制的路由器转发到任意的位置。

当然，这里可能还存在一个潜在的问题，即如果使用固定的CircID 的话，攻击者可以遵循一个固定的CircID 来按照顺序去进行攻击，获得一条路径。这种方式与后面将要谈的StreamID的匿名性类似。这里的一个处理办法是一条线路一旦经过了一个路由器OR，那么就立即更换对应的CircID。在每一个路由器内部都维护进入的CircID以及离开的CircID的对应关系，使得CircID是一个局部的信息而不是全局的信息。另外一个防护方式是线路的建立和维护不是长期的，而是一个有时效性的过程。客户端会不断创建新的线路，一旦创建完成，那么就立即切换到新的线路中，并将旧的线路进行废除。由于线路的创建以及线路的使用可以同时进行，这样的工作方式并不会非常影响性能，但是增加了攻击的难度。攻击者获得的信息往往是过时的信息，在攻击进入到下一个路由器的时候，信息已经不存在了，也就失去了攻击的方向。

数据流数据包在线路中的路由转发

通过建立的线路可以使得多条数据流公用一条线路，使得多个TCP连接可以通过一条线路来完成数据的传输。这样做而不是为每一条TCP建立一条新的线路的原因是线路的建立过程是一个耗时的过程。在建立线路的过程中需要一步一步的延长线路，这就会使得经过一个比较长的时间之后才能够使用线路中的退出节点连接目标网络服务器。通过复用的方式可以让多个TCP数据流可以分摊这种建立线路的负载，进而可以提高系统的性能。

以前面建立的线路为例，从OP发出的数据包按照下面的格式发出（假设所使用的对称加密方法为AES）：

AES（AES（AES（Data, K3），K2），K1）

上述的数据会首先发送给OR1，而OR1会通过K1进行解密，依据内部维护的CircID的信息来选择一对进入和离开的TLS连接，并在通过这个路由的时候更换对应的CircID。数据最终会到达退出节点OR3，OR3获得了最终的解密信息Data，将最终与目标网络服务器进行通信。完成通信后，会从服务器获得数据，OR3会依据掌握的K3对这个数据进行加密，选取合适的CircID，将数据按照原路返回。在数据返回的路径上的每一个路由器都依据相应的密钥对数据进行加密操作，并更新CircID。最终数据返回给OP，由于OP掌握了所有的密钥，因此可以对数据进行解密工作。可以看到，这里的工作方式也与剥洋葱类似，进行层层加密以及层层解密。

在数据流的支持方面，多个数据流（应用程序的TCP连接）可以公用一条线路。为了进行不同数据流的区分，使用的方法就是使用不同的StreamID进行区分。这个StreamID对于退出节点是有用的，用以建立与目标网络服务器不同的TCP连接。在数据返回的时候，依据不同的TCP连接对应到不同的StreamID，最终将服务器返回数据标记不同的StreamID，最终交给OP来将数据返回给应用程序。

在安全性方面，这种方式使得多个可能不同的数据流产生联系，长时间共同占用同一个线路会给攻击者留出足够长的时间进行分析。与前面讨论的一样，为了降低这个方面的影响，在Tor中采用了定时切换线路的方式。这种方式是在正常工作的流程中，Tor不断试图在后台建立新的线路，比如经过一定的时长，就建立新的线路。一旦新的线路建立完成，数据流就可以使用新的线路进行数据的转发。原来的线路就可以进行逐步拆除的工作，对应的路由器将来也可以重新使用。这种方式使用线路使得数据流可以过一定时间就通过不同的线路路由进行转发，提高了数据流的匿名路由特性，提高了攻击者的难度。

目录服务器

以上讨论了在Tor网络中的线路的建立以及如何进行数据的传输。但是前面所描述的客户端构造代理服务器网络路由的方式还缺乏一个基础设施的支持，即如何让客户端OP获得Tor网络每一个服务器的信息。在实际的系统中可以使用目录服务器的方式来提供这一部分信息。在目录服务器中保留了代理服务器网络中的所有的可用服务器的信息，包括服务器的公钥信息以及在物理网络中的域名与IP地址。在安装Tor客户端软件的时候，会预置一些目录服务器的信息，之后OP会首先跟目录服务器进行通信，获取代理服务器的信息。

通过目录服务器可以完成信息的基础服务工作，但是目录服务器也可能是匿名网络中一个遭受攻击的地方，因为如果将目录服务器中的服务器替换为攻击者所控制的服务器，就会使得攻击者有大量可用的代理服务器，降低用户的匿名特性。

在现实中，目录服务器可以使用多种方式进行安全保护。一个是目录服务器需要进行手工的维护，只有管理员确认的代理服务器才能够被接受成为网络的一个组成节点，并通过管理员的证书进行签名认证。这种方式能够保证代理服务器不会出现大量的攻击节点，但是会带来维护的负担，维护一个数千个节点的代理服务器网络是可能的，但是更大的网络规模造成维护非常困难。

目录服务器可以对所有的代理服务器进行签名，客户端在使用代理服务器的时候必须要先进行身份验证。目录服务器防止攻击的另外一个手段是进行数据副本的维护，即在多个目录服务器中维持相同的服务器数据。在多个服务器之间维护相同的数据能够降低目录服务器被攻击的危险，客户端必须与多个目录服务器进行通信，如果发现数据的不一致，那么就说明有服务器上的数据不正确，应该避免去使用不一致的代理服务器的信息。

以上的两点也是在匿名网络Tor中维护目录服务器的方式。如果不使用一个可信的代理服务器的列表，造成的后果就是降低网络的匿名特性，但是这

里也依赖于对代理服务器列表的管理员的信任。这个问题当前是通过线下来解决的，匿名网络的使用者信任网络的建设者试图提供匿名访问的功能，不会故意去做流量分析等攻击动作。

Tor网络中的攻击和防护

前面的内容讨论了在Tor网络中如何建立连接以及在这个连接中如何对网络的数据流进行转发。这些基本的设施就构成了Tor网络的基本功能，并且在功能上可以帮助用户进行网络数据的传输和浏览。下面讨论一下在上述的方案中如何应对网络的攻击以及如何在使用Tor网络的时候，不因为用户自身的失误来造成身份信息的泄露。

在*Tor*网络中是否还是需要使用*HTTPS*协议：对于Tor网络的使用者来说，如果访问的明文网站的话，就会在退出节点的位置将访问的请求解密为明文。如果这个时候网络请求中包含用户的信息（例如在HTTP的请求头中），那么不可避免的是会泄露的。因此，通过Tor访问的明文网站要避免泄露用户的信息，比如即使是一些新闻网站可能也会记录一些Cookie信息，这也会在一定程度上泄露部分用户信息。对于特别需要用户注册的网站，就需要通过HTTPS类似的协议来完成访问工作，即使是退出节点被攻击者所控制，由于不能解密HTTPS的数据流量，用户信息也不会产生泄露。

在*Tor*网络中的*DNS*名字解析问题：另外一个由于用户的原因产生的泄露是用户在使用网络的时候进行DNS名字解析。DNS系统在互联网中是通过UDP协议完成的，并且数据是未经加密的，那么一旦用户使用了DNS系统，其希望访问的目标网络就被攻击者所获知。这样的话就会产生一个矛盾，即如果进行网络访问的话不可避免需要通过DNS进行域名解析，但是一旦发生了解析就会失去匿名性。这个时候，就需要借用Tor网络来完成域名解析，而不是通过客户端本身的名字解析软件进行解析。值得注意的是，有

许多的Tor用户所使用的软件并没有意识到这一点，而是自发提前进行了解析，无形之中就降低了访问的匿名特性。

***Tor*网络对于主动攻击的防护**：对于主动攻击来说，如果攻击者直接去攻击目标的客户端，那么客户端本身将泄露自己的所有访问行为，就没有任何的匿名性可言。因此，Tor网络对于客户端的攻击没有匿名能力，其假设也不在于这一点。如果主动攻击的是路由器，那么攻击者往往的攻击能力是有限的，只能获得局部网络的控制能力，例如获得十几台的路由器的控制权。前面也研究过这样的问题，攻击者只能获得局部的信息，而不是全局的信息，因此对于匿名访问不会构成本质上的威胁。另外，所有的数据在Tor网络中都是通过加密以及完整性检查传输的，攻击者不能对数据进行篡改。在数据传输过程中，使用的Diffie-Hellman密钥交换算法具有前向安全性，避免攻击者进行线下分析以及临时的密钥避免了重放攻击（记录加密数据，并且在以后进行再次提交进行攻击）以及加密方式避免了注入攻击（在数据流中插入攻击数据）。

***Tor*网络对于被动攻击（流量分析）的防护**：对于流量分析攻击来说，Tor网络正好是对这样的攻击进行设计的，因此维持匿名性是没有任何问题的。攻击者只能监听整个网络的局部，而不能监听整个网络的全体，攻击者只能看到客户端连接了一个代理服务器网络以及代理服务器网络去连接目标网络。所有的数据都是加密的。这种方式使得攻击者很难将进入到网络的流量和从网络中流出的流量进行匹配起来，保证了匿名访问特性。

对于目录服务器的攻击：前面已经对目录服务器的攻击以及相关的防护方式进行过研究，在此不再赘述。值得注意的是，如果对目录服务器发起拒绝服务攻击的话，是有可能造成服务出问题的。但是，目录服务器所实现的功能是相当简单的，没有任何的计算，只是提供一系列只读的列表。并且当前的目录服务器可以服务整个网络，因此对它们进行拒绝服务攻击需要很强的计算能力。另外，在客户端访问的时候，已经获得了具体的代理服务器目

录，因此之后再也不需要与目录服务器进行通信，这会大大降低对于目录服务器进行攻击带来的影响。

拒绝服务攻击：Tor网络没有对拒绝服务攻击提出特殊的设计。在实际的运行过程中，需要对拒绝服务攻击进行一定的防护。对于Tor网络的拒绝服务攻击有可能影响一个局部的代理服务器网络，OP可以自己监控所需要的路由器的性能，并选择性能良好的路由器，避开遭受拒绝服务攻击的一部分服务器。在这里同样假设了攻击只能发生在局部，而不是全局，因为试图通过拒绝服务攻击拖垮整个Tor网络所需要的计算能力和网络带宽是非常巨大的，往往得不偿失。

以上讨论了Tor网络可能面临的一些攻击情况。可以看到这些攻击都只能影响一部分网络而不会影响全局的网络。但是，这些情况随着互联网的发展，网络带宽的提高也可能造成问题。例如，近年来越来越多的出现了对于Tor网络的全局攻击，全局攻击是Tor网络无能为力的。

匿名网络的全局攻击

当然，对于Tor网络的攻击可能还远远高于前面所描述的各种攻击方式，有一些攻击是Tor网络不能够防护的。一些攻击在之前看起来不可能，但是随着有大规模机构（比如政府与军队）的参与，Tor网络的匿名特性就会被破坏。其中一种方式是进行全局的监听。这以前被认为是超出了攻击者的能力，但是现在也会成为一个问题。全局监听的方式是在Tor网络的周边进行监听并获取流量。由于流量是被加密的，因此全局监听的方式也无法对流量进行解密工作。但是，可以通过入流量和出流量的特性分析匹配入流量以及出流量。例如，通过分析流量随着时间变化的特征，观察各个出入口流量之间的相关性，就可以匹配两部分流量，进而可以进行下一步的分析。对于这样的全局的流量分析Tor网络来说没有防护能力。一种可能的防护方法是使

用掩护流量，即发送一些不相关的流量来掩盖自己真实的访问目的。但是，这种方式也不能完全避免全局的流量分析，特别是攻击者获得了数量众多的路由器的情况下，各个路由器可能会相互配合，对一些数据流构建时间流量模式。这种情况下，即使使用了掩护流量模式也不能够应对流量分析的攻击。这种方式对于攻击者提出了很高的要求，一旦被成功实施，实际上会造成Tor网络的失败。全局的攻击还存在另外一种形式，即进行Tor网络的全局拒绝服务攻击。虽然需要规模巨大的计算能力和网络带宽，但这并非完全不可能。

思考题

1. 为什么CircID每经过一个洋葱路由OR就需要进行替换?
2. StreamID是区分不同的TCP连接的，如果被攻击者知道了，就可以依据这样的信息循序攻击整个链路的路径，有什么样的防护方法?
3. 目录服务器如果手工维护的话，维护成本很大，但是如果不手工维护的话，会造成目录服务器中的代理服务器被替换为攻击服务器。如果整个系统规模继续增大的话，有什么办法能够避免系统的维护问题?
4. 对于全局攻击，思考有什么的办法能够应对全局的流量分析攻击?

参考文献

本章的参考文献主要包括的密码学的相关内容，网络代理与数据安全传输的内容以及相应的匿名网络的内容。关于密码学的内容读者可以参考任何一本密码学的教科书。关于混合网络即第一代洋葱网络的结构可以参考文献：对于Tor网络可以参考：。注意，关于Tor网络的论文里面还讨论了各种攻击的方式以及Tor网络对此的防护，许多都是实际的工程经验，值得进行参考与学习。