# Week5: Intro to Crypto

1. **Reading**

**11.2.1 Cryptography**

* 1. A cryptographic key is a parameter used in an encryption algorithm in such a way that the encryption cannot be reversed without knowledge of the key.
  2. Two main classes of usage:
     1. - shared secret keys: the sender and the recipient must share a knowledge of the key and it must not be revaled to anyone else
     2. -public/private keys: the sender of a message uses a public key – one that has already been published by the recipient – to encrypt the message; The recipient uses a corresponding private key to decrypt the message.

**11.2.2 Uses of cryptography**

* 1. **名为Kerberos认证协议的常见安全协议。该协议的目的是确保用户与网络服务之间的安全通信和访问控制:[目的：确保用户与网络服务之间的安全通信和访问控制]**
     1. 身份验证请求：Alice希望访问她所在组织本地网络上的服务器（Bob）上的文件。为了做到这一点，她需要证明自己的身份。因此，Alice向认证服务器（Sara）发送了一个未加密的消息，指示她的身份并请求一个用于访问Bob的门票。
     2. 认证服务器的响应：Sara，即认证服务器，会响应Alice的请求。Sara会用Alice的密钥（KA）对消息进行加密，并在响应中包含一个门票，该门票由Bob使用，以及一个新的会话密钥（KAB），用于与Bob进行通信。因此，Alice接收到的响应看起来像这样：{{门票}KB, KAB}KA。
     3. Alice解密响应：Alice使用她的密钥KA（通过将她的密码应用相同的转换方式生成）来解密响应。密码不会通过网络传输，一旦使用后，密码会从本地存储中删除以防止泄漏。如果Alice拥有正确的密码生成密钥KA，她就能获取用于访问Bob的有效门票以及与Bob进行通信时使用的新加密密钥。Alice无法解密或篡改门票，因为它是用KB加密的。如果接收方不是Alice，那么他们不会知道Alice的密码，因此他们无法解密消息。
     4. Alice向Bob发送门票：Alice将门票与她的身份和访问文件的请求一起发送给Bob：{门票}KB, Alice, R。
     5. Bob解密门票：门票最初由Sara创建，实际上是：{KAB, Alice}KB。Bob使用他的密钥KB来解密门票。因此，Bob获取了Alice的真实身份（基于Alice和Sara之间共享的Alice的密码的信息）以及与Alice进行交互时使用的新的共享会话密钥KAB（称为会话密钥，因为它可以安全地用于Alice和Bob之间的一系列交互）。这确保了Bob只允许真正经过身份验证的Alice访问他的服务。
  2. **通过公钥认证实现安全通信的过程(hybrid cryptographic protocol)[目的：确保Alice可以与Bob建立安全通信]:**
     1. Alice获取Bob的公钥: 首先，Alice需要获取Bob的公钥。为此，她访问一个密钥分发服务，以获得Bob的公钥证书。这个证书之所以称为"证书"，是因为它是由一个受信任的权威（通常是一个被广泛认为是可信的个人或组织）签名的。在检查了证书的签名后，Alice从证书中读取Bob的公钥，即KBpub。这确保了她获得了真正的Bob的公钥。
     2. Alice创建共享密钥KAB并加密：接下来，Alice创建一个新的共享密钥KAB，并使用Bob的公钥KBpub进行加密，使用公钥算法。然后，她将加密结果发送给Bob，同时发送一个唯一标识公/私密钥对的名称（因为Bob可能有多个这样的密钥对）。具体来说，Alice发送的消息格式为keyname,{KAB}KBpub。
     3. Bob解密并获取KAB：Bob收到Alice的消息后，他从自己的私钥存储中选择相应的私钥KBpriv，并使用它来解密KAB。需要注意的是，Alice发送给Bob的消息可能在传输过程中被损坏或篡改。结果可能是Bob和Alice不共享相同的密钥KAB。如果这是一个问题，可以通过在消息中添加一个约定的值或字符串来避免，例如Bob和Alice的姓名或电子邮件地址。Bob可以在解密后检查这些值来确保安全性。这确保了Bob和Alice能够建立共享的密钥KAB，用于后续的安全通信。

\*上述问题描述了一种中间人攻击的潜在威胁，其中攻击者Mallory可以冒充Bob并拦截Alice与Bob之间的通信。为了应对这一风险，必须采取以下措施：

* + 1. 使用已知和受信任的证书颁发机构（CA）签名：Alice必须确保Bob的公钥证书是由已知和受信任的CA签发的，而不是由攻击者Mallory签发的。这样，Alice可以信任证书中包含的公钥是Bob的真实公钥。
    2. 验证证书的签名：Alice必须验证Bob的公钥证书的数字签名以确保它确实是由CA签发的。如果签名验证失败，Alice应中止通信。
    3. 通过安全渠道获取CA的公钥：Alice还必须以安全的方式获取CA的公钥，以便验证Bob的证书。这可以通过安全渠道（例如面对面会议或已知受信任的网站）获得CA的公钥。
  1. **Digital Signature[用于验证消息或文件的完整性和真实性，以确定消息的发件人是合法的]**
     1. 数字签名定义：数字签名是一种机制，通过它可以验证某消息或文件是由签名者生成的未经篡改的副本，就像传统签名在第三方面前验证文件的真实性一样。
     2. 数字签名基础：数字签名技术基于将一个仅签名者知晓的密钥与消息或文件不可逆地绑定在一起。这可以通过使用仅签名者知道的密钥来加密消息，或更好地，使用消息的压缩形式，称为摘要(digest)，来实现。摘要是通过应用安全的摘要函数计算的固定长度值，类似于校验和函数，但非常不可能为两个不同的消息生成相似的摘要值。
     3. 数字签名过程：签名者使用他们的私钥生成数字签名，然后将签名与消息一起发送。任何接收者都可以使用相应的公钥来解密签名，以验证消息的完整性和真实性。
     4. 公钥加密：通常，公钥加密被用于数字签名。签名者使用他们的私钥生成签名，而接收者可以使用相应的公钥来验证签名。这种方法保护了签名的机密性。
     5. 公钥证书：验证方需要确保公钥确实属于声称是签名者的实体。这一问题通常通过使用公钥证书来解决，这是由可信任的证书颁发机构（CA）签发的，证明了公钥的真实性和与特定实体的关联。
     6. 电子签名的本质是通过私钥签署消息，然后通过公钥验证签名，而不涉及消息的加密和解密过程。电子签名的主要目的是确保消息的来源和完整性，而不是保密性。
     7. 计算文档的摘要：Alice首先计算文件M的固定长度摘要，称为Digest(M)。这个摘要是通过应用一个安全的摘要函数来生成的，通常用于创建文件的唯一指纹。
     8. 用私钥加密摘要：Alice使用她的私钥（KApriv）来加密文件摘要，然后将加密的摘要附加到文件M中，从而生成了签名的文件M，即M, {Digest(M)}KApriv。这样，签名被嵌入到文件中。
     9. 接收方获取签名文件：Bob收到了被签名的文件M，并从中提取了原始文件M。
     10. 验证签名：Bob计算提取的文件M的摘要，即Digest(M)。然后，他使用Alice的公钥（KApub）来解密文件中嵌入的摘要{Digest(M)}KApriv。如果解密后的摘要与他自己计算的Digest(M)匹配，那么签名是有效的。这表示Alice是文件M的合法发件人，文件未被篡改。
     11. 通过这个过程，Alice使用私钥签署文件，并Bob使用Alice的公钥验证签名。这确保了Bob可以确认文件的完整性和真实性，以及文件的发件人是Alice。这种方法通常用于数字签名和文件完整性验证。

**11.2.3 Certificates**

* 1. 如何使用数字证书来建立身份验证和确保通信安全的信任链:
     1. 背景：Bob是一家银行，他希望客户在与银行建立联系时可以确信他们正在与真正的银行交流，即使他们以前从未与银行联系过。同时，银行需要对客户进行身份验证，以确保他们可以访问自己的帐户。
     2. Alice的银行账户证书：Alice是银行的客户，她希望获得一份证书，证明她在银行有一个账户。这个证书包括了一些信息，如Alice的姓名、账号等，并且被银行使用他们的私钥签名以确保真实性。
     3. Alice的证书用途：Alice可以在购物时使用这个证书来证明她在银行有一个账户，以便商家可以验证她的身份。
     4. 信任链的引入：为了确保商家可以信任Alice的证书，商家需要知道银行的公钥，以验证银行的签名。为了确保银行公钥的真实性，引入了一个信任链，其中包括了Fred，代表了银行联合会。Fred的角色是认证银行的公钥，并他可以颁发银行的公钥证书。然而，为了确保Fred的公钥的真实性，商家需要知道Fred的公钥，并且这个公钥也需要得到认证。
     5. 信任链的解决：为了打破信任链中的递归问题，商家需要确保他们知道Fred的真实公钥，以验证银行的公钥证书。这可以通过确保商家以某种方式获取Fred的公钥，并且可以信任这个获取方式来解决，例如通过Fred的代表或通过商家信任的人，他们从Fred那里直接获得了Fred公钥的签名副本。这构成了一个信任链。
     6. 问题和挑战：这个场景突出了一些与证书相关的问题，包括选择受信任的权威机构的难度、私钥泄露的风险以及证书链的长度问题。信任通常不是绝对的，因此选择权威机构必须依赖于证书的具体用途。
  2. To make certificates useful, two things are needed:

• a standard format and representation for them so that certificate issuers and certificate users can successfully construct and interpret them;

• agreement on the manner in which chains of certificates are constructed, and in particular the notion of a trusted authority.

**11.3 Cryptographic algorithms**

* 1. 对称加密：
     1. Plaintext message(any sequece of bits) to a ciphertext(a different sequence of bits)
     2. The encryption transformation is defined with two parts, a function E and a key K. The resulting encrypted message is written {M}K:



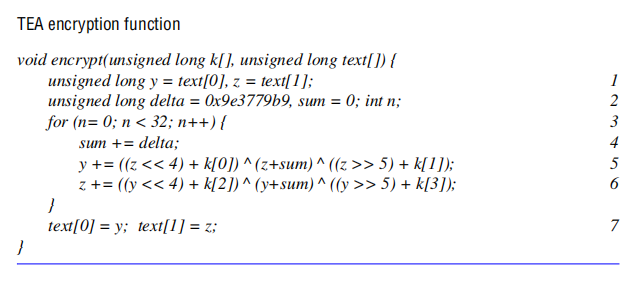
* + 1. The encryption function *E* defines an algorithm that transforms data items in plaintext into encrypted data items by combining them with the key and transposing them in a manner that is heavily dependent on the value of the key. We can think of an encryption algorithm as the specification of a large family of functions from which a particular member is selected by any given key. Decryption is carried out using an inverse function *D,* which also takes a key as a parameter. For secret-key encryption, the key used for decryption is the same as that used for encryption:



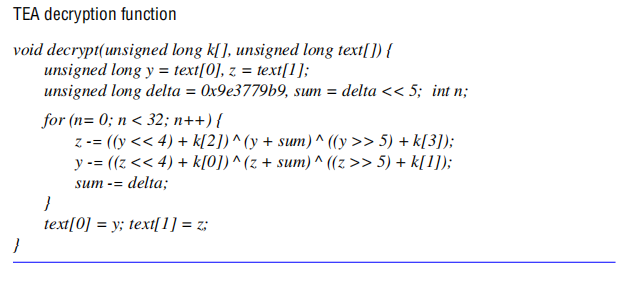
* + 1. 一次性函数（One-way functions）：对于强加密函数来说，计算FK(M)（将明文M加密为密文）相对容易，而计算FK^(-1)(M)（将密文M解密为明文）非常困难，以至于几乎不可行。这种类型的函数被称为一次性函数。这一特性对于信息加密的有效性至关重要，因为它保护了免受攻击，攻击者试图通过已知的密文M^和密钥K来发现原始明文M。
    2. 对称算法的强度：良好设计的对称加密算法，其强度抵抗了试图通过已知明文M和相应的密文M^来发现密钥K的尝试，这种攻击方式通常被称为暴力破解攻击。暴力破解的方法是尝试所有可能的K值，计算EKM(M)直到结果与已知的M^ ` K值匹配。如果K有N位，那么这种攻击平均需要2^(N-1)次迭代，最多需要2^N次迭代才能找到K。
    3. 密钥大小的重要性：对称加密算法的强度取决于密钥K的大小。较长的密钥通常更难被破解，因为暴力破解攻击所需的迭代次数是指数级的。这意味着随着密钥位数的增加，破解密钥所需的时间成指数增长。
  1. 非对称加密
     1. RSA算法(private-public key)
        1. 公钥可以看作是一个锁，任何人都可以用这把锁锁住（加密）信息，但只有拥有与之匹配的私钥的人才能打开（解密）锁。
        2. RSA的安全性基于一个数学难题，即大数的因式分解。它依赖于事实，当我们将两个大质数相乘时，很容易得到一个大数，但要逆过程，将这个大数分解回原来的两个大质数，却非常困难，尤其是当质数足够大时。【随机选择两个质数p和q，乘积为N】
        3. RSA还使用了一种数学运算，称为模幂运算，用于加密和解密信息。这是一种通过将数据提高到某个幂次方，然后取模得到结果的运算。
           1. 升幂运算：在加密过程中，明文会被升幂运算。明文M会被提高到公钥e次方，这是加密密钥的一部分。这个操作是明文与公钥的e次方的运算。
           2. 模取幂运算：在加密和解密过程中，都会使用模取幂运算。具体来说，加密时，升幂后的结果会对N取模，得到密文C。在解密时，密文C会被升幂到私钥d次方，然后对N取模，得到原始明文M。
     2. 公钥密码学的基础(public-key scheme)：公钥密码学是一种密码学方法，最早由Diffie和Hellman于1976年提出，它消除了通信双方之间的信任需求。其基础在于"陷阱门函数"（trap-door functions）的存在。陷阱门函数是一种具有秘密出口的一次性函数，它在一个方向上很容易计算，但在没有知道秘密的情况下，计算其逆向操作是不可行的。这一概念的提出使得公钥密码学变得可行。
     3. 陷阱门函数的应用：公钥密码学方案的基础是利用陷阱门函数，这些函数涉及到大数字的计算。在RSA算法中，这些大数字是由一对非常大的质数相乘而得。这个过程是一个一次性函数，因为从结果得出原始的相乘数是计算上不可行的，即因式分解。
     4. RSA算法：在RSA算法中，一对密钥由这一过程生成，其中一个密钥用于加密。RSA加密将明文视为二进制数字块，然后将其提升到密钥的幂次方，再对N取模运算，得到相应的密文块。
     5. 密钥和N的大小：RSA算法中，N和至少一对密钥的大小远大于对称密钥的安全密钥大小，以确保N不可因式分解。因此，RSA算法具有较高的抵抗暴力攻击的潜力，其安全性取决于无法因式分解N的计算不可行性。
  2. Block Ciphers
     1. 块密码：许多加密算法操作的数据块是固定大小的，通常是64位。消息被划分为这些固定大小的块，每个块都独立加密。第一个块在加密后可以立即传输。在简单的块密码中，每个密文块的值不依赖于前面的块，这可能存在弱点，因为它允许攻击者识别重复模式并推断它们与明文的关系。因此，大多数块密码算法使用密码块链接 (CBC) 模式来解决这些问题。
     2. Cipher Block Chaining (CBC) 模式：在CBC模式中，每个明文块在加密前与前一个密文块使用异或 (XOR) 运算结合在一起。在解密时，块被解密，然后前一个加密块（应该已存储为此目的）与它进行XOR运算，以获取新的明文块。CBC模式防止相同的明文部分加密为相同的密文，提高了安全性。为防止加密模式相同，需要在每个消息前插入不同的明文块，这个附加的文本称为初始化向量（IV），通常使用时间戳生成，以确保每个消息都以不同的明文块开始。
     3. 然而，CBC模式的使用受限于通过可靠连接传输的数据的加密。如果密文块丢失，解密将失败，因为解密过程将无法进一步解密其他块。
  3. Stream Ciphers
     1. 流密码（Stream ciphers）是一种加密技术，用于实时生成小块数据流的应用，例如电话通话的加密。这些应用中，数据以实时方式产生，可以非常小，甚至只有几个比特，因此将它们填充到标准大小（例如64比特）后再进行加密会很浪费。流密码允许逐位对明文数据进行加密，而不需要等到足够的数据块再一次性加密。
     2. 将块密码算法（Block ciphers）转换为流密码算法非常简单，关键在于构建一个密钥流生成器。密钥流是一个比特序列，它会逐位与明文数据流进行异或操作，以混淆明文。如果密钥流是安全的，那么生成的加密数据流也会是安全的。
     3. 密钥流生成器可以通过迭代数学函数来构建，这个函数会接受输入值并生成一系列输出值，这些输出值会被拼接在一起以构成明文块，然后使用发送方和接收方共享的密钥来加密这些块。生成的密钥流可以通过应用CBC（Cipher Block Chaining）模式来进一步混淆。生成的加密块会用作密钥流。
     4. 为了确保数据流的质量，密钥流块应该提前一段时间生成，以确保它们能在需要时立即可用。同时，生成密钥流的过程不应该占用太多的处理资源，以免延迟数据流。因此，原则上，实时数据流可以像批处理数据一样进行加密，只要有足够的处理能力来实时加密密钥流。
  4. 加密算法的设计
     1. 混淆（Confusion）：这个原则通过非破坏性的操作，如异或（XOR）和循环移位，将每个明文块与密钥相结合，产生一个新的位模式，模糊了明文块M与{M}K之间的关系。这样的操作使得分析基于字符频率的攻击失效，尤其是对于较大的明文块。混淆操作通过打破明文的常规模式来增加攻击的复杂性。例如，二战时的德国恩尼格玛机使用了单字母块的链接，最终被统计分析破解。
     2. 扩散（Diffusion）：明文通常包含重复和冗余。扩散原则通过改变明文块的部分内容来消除这些重复的模式。如果使用了CBC模式，冗余还会分布到更长的文本中。流密码由于没有块的概念，不能使用扩散。

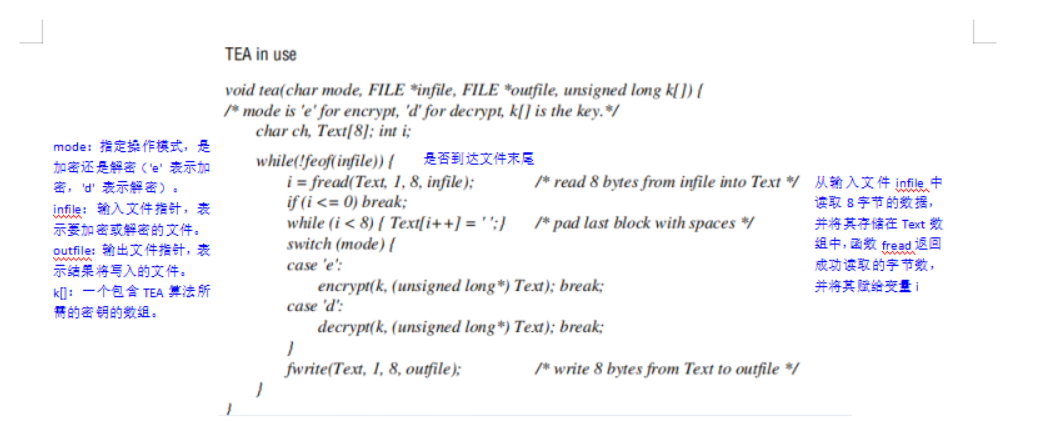
**11.3.1 Secret-key (symmetric) algorithms**

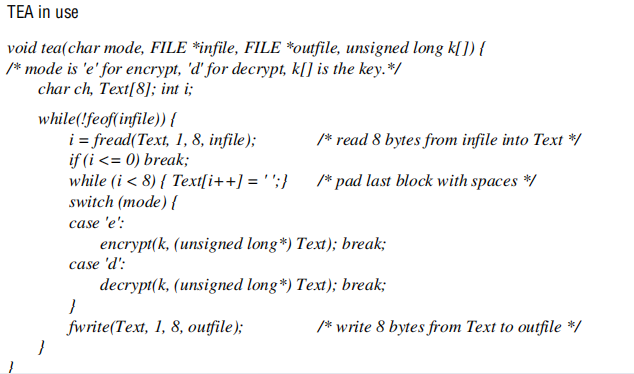
(1) TEA algorithm(Tiny Encryption Algorithm)



* + 1. 在TEA算法的32个轮次中，操作主要围绕着两个32位整数，即text[0]和text[1]，进行。这两部分被称为text的左半部分和右半部分。
    2. 与密钥的组合：在每一轮中，这两部分的数据会与密钥的不同部分进行组合。具体来说，text[0]和text[1]分别与四个32位密钥的部分（k[0]、k[1]、k[2]和k[3]）进行混合。
    3. 位移和异或操作：这两部分的混合包括位移和异或操作。位移操作（<<和>>）会对其中一部分进行位移，而异或操作（^）会将两部分的数据进行混合。
    4. 提供混淆：这些位移和异或操作提供了混淆，使得加密后的数据在数学上不容易逆向推导出原始的明文数据。
    5. 提供扩散：除了混淆外，TEA算法还需要数据扩散。这是通过对text[0]和text[1]的位移和交换操作来实现的。这些操作将确保明文的不同部分在加密后被均匀地分散到整个加密数据中。
    6. 非重复常数delta：每一轮中，算法会引入一个常数delta，该常数不会重复。这个delta常数与每部分text的数据一起进行组合，以确保即使部分文本不发生变化，密钥也不容易被破解。这增加了算法的安全性。







从输入文件infile中读取8字节的数据，并将其存储在Text数组中，函数fread返回成功读取的字节数，并将其赋给变量i

是否到达文件末尾

mode：指定操作模式，是加密还是解密（'e' 表示加密，'d' 表示解密）。

infile：输入文件指针，表示要加密或解密的文件。

outfile：输出文件指针，表示结果将写入的文件。

k[]：一个包含TEA算法所需的密钥的数组。

**11.3.2 Public-key (asymmetric) algorithms**

(1) 在公钥密码系统中，有一对密钥，一个用于加密（公钥，通常称为Ke），另一个用于解密（私钥，通常称为Kd）。公钥可以被任何人获取，用于加密消息，而私钥必须保密，只有持有者才能用它来解密加密的消息。这种系统基于陷阱门函数，通常涉及对大数进行操作，如指数运算，以实现加密和解密。



* 1. RSA: 依赖于大素数的数学性质。RSA算法使用两个大素数生成公钥和私钥，其中一个用于加密，另一个用于解密。
     1. 选择两个大素数P和Q，它们的值都远大于10^100。然后将它们相乘以得到N，计算Z作为(P-1)和(Q-1)的乘积。

例如，文本中提供的示例数值：P = 13，Q = 17，则N = 221，Z = 192。

* + 1. 选择一个与Z互质（没有与Z共同因子的数）的整数作为私钥d。这个d将用于解密。

例如，文本中提供的示例数值：d = 53。

* + 1. 接下来，计算公钥e，使得e \* d ≡ 1 (mod Z)。这意味着e \* d 除以Z的余数等于1。

例如，文本中提供的示例计算：e \* d ≡ 1 (mod 192)，找到一个e值，使得e \* d = 385，然后e = 385 / 5 = 77。

* + 1. 确定k的值，其中2^k < N，通常在512到1024之间。这用于将明文分成块。

例如，文本中提供的示例：k = 7，因为2^7 = 128。

* + 1. 加密单个明文块M的函数为：E'(e, N, M) = M^e mod N。

例如，对于消息M，加密后的密文为：M^77 mod 221。

* + 1. 解密密文块c以生成原始明文块的函数为：D'(d, N, c) = c^d mod N。
    2. 最后，E'和D'是互为反函数的，即E'(D'(x)) = D'(E'(x)) = x，这适用于所有P在范围0 ≤ P ≤ N 的值。

在公钥密码系统中，接收方的公钥是公开的，任何人都可以获得它，因为它用于加密消息。然而，只有接收方拥有唯一且正确的私钥，因此只有接收方能够解密通过其公钥加密的消息。这确保了消息的机密性，因为只有拥有私钥的接收方才能够还原明文消息。

(弱点：公钥密码系统的一个潜在弱点是，因为公钥是公开的，攻击者可以轻松生成加密消息，并尝试通过枚举加密各种比特序列来破解未知消息，这被称为“选择明文攻击”。为了防止这种攻击，文本建议确保所有消息都比密钥长度更长，这样暴力破解加密消息的尝试将不太可行。)

# Week6: Cryptographic Protocols

1. **Reading**

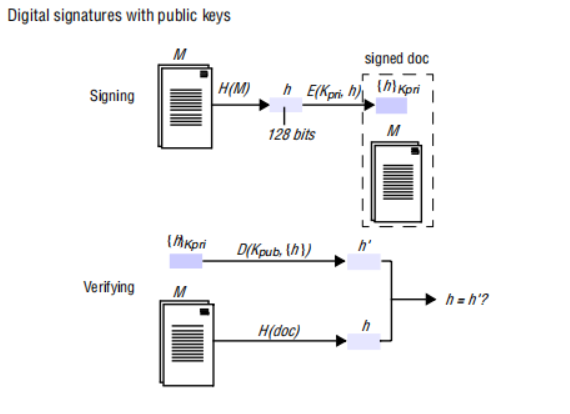
**11.4 概述**

1. Digital Signatures: An electronic document or message M can be signed by a principal A by encrypting a copy of M with a key KA and attaching it to a plain text copy of M and A’s identifier. The signed document then consists of: M, A, [M]KA
   1. If a secret key is used to encrypt the document, only principals that share the secret can verify the signature.
   2. If public-key cryptography is used, then the signer uses their private key and anyone who has the corresponding public key can verify the signature.[如果使用公钥密码学（public-key cryptography），那么签名主体使用自己的私钥来创建数字签名，将加密的文档 [M]KA 与 M 和签名主体 A 的标识信息一起附加在一份文档上。此时，任何人都可以使用签名主体 A 的公钥来验证签名的有效性。因为只有拥有与 A 的公钥相对应的私钥的签名主体才能够创建这个特定的数字签名，所以只要签名主体 A 的公钥是公开可用的，任何人都可以验证签名。这就是公钥密码学的强大之处，它允许广泛的验证，同时保护了签名的私密性。]
2. 摘要函数（Digest functions），也被称为安全散列函数（secure hash functions），通常表示为 H(M)。摘要函数的主要目的是将一个消息 M 转换成一个固定长度的散列值 H(M)。这个散列值应该满足以下要求：
   1. 设计精心：摘要函数必须经过精心设计，以确保对于所有可能的消息对 M 和 M'，H(M) 与 H(M') 不同。这是摘要函数的关键特性，以防止不同的消息映射到相同的散列值。
   2. 防止欺骗：如果存在一对不同的消息 M 和 M'，使得 H(M) = H(M')，那么可能会出现问题。恶意主体可以使用这一情况来进行欺骗，他们可以发送 M 的签名，但当面对问题时，声称实际上发送的是 M'，并声称 M 在传输过程中被篡改。因此，摘要函数必须确保这种情况极其罕见或不可能发生。

**11.4.1 Digital signatures with public keys**

(1) 步骤

* 1. A 生成密钥对：A 首先生成一对密钥，包括公钥（Kpub）和私钥（Kpriv）。然后，A将公钥 Kpub 发布到一个众所周知的位置，以便其他人可以访问它。
  2. 计算消息摘要：A 对消息 M 使用一个已经约定好的安全散列函数 H 进行摘要计算，得到消息的摘要值 H(M)。然后，A 使用自己的私钥 Kpriv 对该摘要值进行加密，生成数字签名 S = {H(M)}Kpriv。这个数字签名是由 A 创建的，用于验证消息的完整性和身份认证。
  3. 发送消息和签名：A 将消息 M 与数字签名 S 组合在一起，形成签名消息 [M]K = M,S。然后，A 将这个签名消息发送给 B。
  4. 验证数字签名：B 收到签名消息 [M]K 后，首先使用 A 的公钥 Kpub 来解密数字签名 S，从中获取摘要值 {H(M)}。然后，B 通过使用相同的安全散列函数 H 计算消息 M 的摘要值 H(M)。如果从数字签名中提取的摘要值 {H(M)} 与使用相同的消息 M 和安全散列函数 H 计算出的摘要值 H(M) 相匹配，那么数字签名是有效的，消息没有被篡改，且消息确实由 A 创建并签名。
  5. 这个过程说明了数字签名的基本原理，它用于验证消息的完整性和验证消息的发送方的身份。数字签名技术在信息安全和身份验证领域起着关键作用: \*RSA适用于数字签名的原因是，私钥只由签名者知道，这确保了数字签名的机密性。同时，公钥是公开可用的，任何人都可以使用它来验证签名的有效性。这种方式允许了身份验证和完整性检查，同时确保数字签名的机密性。



**11.4.2 Digital signatures with secret keys - MACs**

1. 使用秘密密钥进行数字签名加密：理论上，你可以使用秘密密钥算法来加密数字签名，但为了验证这样的签名，必须公开密钥，这会引发一些问题。
2. 安排安全地传输密钥：签名者必须安排密钥的安全传输给验证者，这可能涉及复杂的密钥分发过程。
3. 多个上下文下的验证：数字签名可能需要在不同的上下文和不同的时间进行验证，签名者在签名时可能无法确定所有验证者的身份。这可能需要将验证委托给可信的第三方，该第三方持有所有签名者的秘密密钥，但这会增加安全模型的复杂性并需要与可信第三方进行安全通信。
4. 保密密钥泄露的问题：透露用于签名的秘密密钥是不可取的，因为它会削弱使用该密钥进行签名的安全性。任何持有该密钥的人都有可能伪造签名，即使他们不是密钥的拥有者。
5. 因此，公钥方法通常是生成和验证数字签名的方便解决方案，因为它避免了上述问题。然而，在一些特殊情况下，当需要验证消息的真实性而又在安全信道上传输未加密的消息时，可以考虑使用共享秘密密钥生成低成本的签名，通常称为消息认证码（MACs）。这些MACs用于验证通信的真实性，基于双方之间的共享秘密密钥。
6. Sample steps:
   1. 密钥生成和分发：签名者 A 生成一个随机密钥 K，用于签名，并通过安全信道将该密钥分发给需要验证 A 发送消息的一个或多个受信任的主体。这些主体不会泄露共享密钥。
   2. 签名过程：对于要签名的文档 M，A 将文档 M 与密钥 K 连接起来，然后计算连接结果的摘要值 h = H(M || K)，其中 H 是摘要函数。然后，A 将签名附加到文档 M 上，生成带有签名的文档 M>K = M || h。
   3. 验证过程：接收者 B 也连接接收到的文档 M 与共享密钥 K，并计算摘要值 h' = H(M || K)。然后，B 验证签名，如果 h 和 h' 相等，则签名有效。

**11.4.3 Secure digest functions**

1. 摘要函数的特性：安全摘要函数 h = H(M) 应具备以下特性：
   1. 给定消息 M，容易计算出摘要 h。
   2. 给定摘要 h，难以计算出原始消息 M。
   3. 给定消息 M，难以找到另一消息 M'，使得 H(M) = H(M')。这种性质被称为单向散列函数（one-way hash functions）。
   4. 防止碰撞：防止两个不同的消息 M 和 M' 具有相同的摘要值 H(M) = H(M')。这是为了确保攻击者无法伪造签名，即使他们知道签名的摘要值。
   5. 防御生日攻击：生日攻击是一种统计悖论，涉及在一组中找到匹配的对的概率大于找到单个匹配的概率。为了防御生日攻击，摘要函数的输出长度通常需要足够长，比如至少128位，以减小碰撞的概率。
      1. "生日攻击"（Birthday Attack）是一种密码学中的统计攻击，得名于“生日悖论”。这个悖论表明，在一个相对较小的样本中，要找到其中两个元素具有相同的特性的概率要高于你可能预期的。具体来说，悖论指出，当你在一个集合中随机选择元素时，当选择的元素数量接近集合大小的平方根时，有一个相当高的概率会出现两个元素是相同的。
      2. 在密码学中，这一悖论与哈希函数的碰撞问题相关。哈希函数将输入数据映射到固定长度的输出（摘要），通常以二进制形式表示。因为哈希函数的输出空间是有限的，所以当你将不同的输入数据进行哈希运算时，有可能得到相同的哈希输出（碰撞）。
      3. 生日攻击的思路是，攻击者尝试生成大量不同的输入数据（如消息或文档），并计算它们的哈希值。攻击者在大量计算后，可能会发现两个不同的输入数据具有相同的哈希值，这就是碰撞。这对于密码学来说是危险的，因为攻击者可以使用这些碰撞来伪造数字签名或破坏消息完整性。
   6. 安全摘要函数的设计：安全摘要函数需要经过精心设计，通常使用位级操作和位逻辑操作。这些操作不需要保留信息，因为摘要函数不需要可逆性。通常，摘要中还包括源文本的长度信息。

**11.6 Case studies: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi**

**11.6.1 Needham-Schroeder 认证协议**

* 1. 背景和需求：Needham-Schroeder认证协议是为了解决在网络中安全管理密钥（以及密码）的需求而开发的。当该协议首次发布时，网络文件服务刚刚兴起，需要更好的方式来管理本地网络的安全性。协议的目标是提供安全的密钥服务，以发放会话密钥。
  2. 秘密密钥和认证服务器：协议依赖于认证服务器，该服务器向客户端提供秘密密钥，用于安全通信。认证服务器的任务是以加密消息的方式与客户端通信，以提供对共享密钥的获取。这些密钥可以用于加密和解密通信。
  3. 协议描述：协议用于两个进程（A和B）之间的安全通信，但在客户-服务器系统中，通常A是客户端，B是服务器。秘密密钥以两种形式提供给A，一种用于加密发送给B的消息，另一种安全地传输给B。
  4. 票据和随机数：协议基于认证服务器生成和传输票据，这些票据包含用于A和B之间通信的秘密密钥。随机数（nonces）被用来展示消息的新鲜性，它们只被使用一次，通常由需求生成。
  5. 消息传输和验证：如果协议成功完成，A和B可以确保他们收到的消息来自对方，并且只能被对方或认证服务器S理解。这是因为使用KAB加密的消息只能使用A或B的秘密密钥进行解密。
  6. 协议弱点和解决方案：协议存在一个弱点，即B没有理由相信消息3是新鲜的，导致可能的安全问题。这个弱点可以通过向消息3添加随机数或时间戳来解决，以确保消息的新鲜性。
  7. 总的来说，Needham-Schroeder认证协议是一种早期用于安全通信的协议，它依赖于认证服务器和秘密密钥来确保通信的机密性和完整性。需要特别注意的是协议的弱点，需要采取额外的安全措施来防止潜在的攻击。

# 

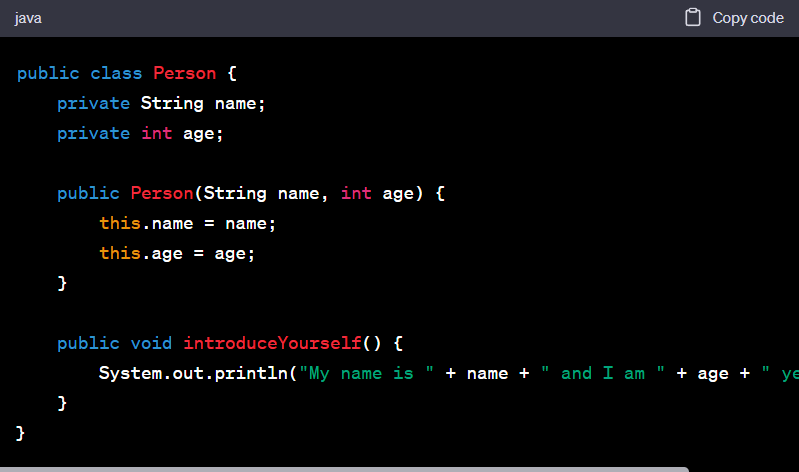
# Week7: RMI and Web Services

1. **Reading for RMI:**
   1. Pg. 15 define remote invocation
      1. 术语“服务器”可能对大多数读者来说很熟悉。它指的是在网络计算机上运行的程序（进程），接受来自其他计算机上运行的程序（客户端）的请求，以执行特定服务并做出相应的响应。请求的进程被称为客户端，这种整体方法被称为客户端-服务器计算。在这种方法中，请求是从客户端发送到服务器的消息，响应是从服务器发送到客户端的消息。当客户端发送请求以执行操作时，我们说客户端调用服务器上的操作。客户端和服务器之间的完整互动，从客户端发送请求到接收服务器响应，被称为远程调用。
      2. 同一个进程有时既可以充当客户端又可以充当服务器，因为服务器有时需要调用其他服务器上的操作。术语“客户端”和“服务器”仅适用于单个请求中扮演的角色。客户端是主动的（发出请求），服务器是被动的（仅在接收请求时激活）；服务器持续运行，而客户端仅在它们构成一部分的应用程序需要时运行。
      3. 需要注意的是，默认情况下，“客户端”和“服务器”这些术语通常是指进程，而不是它们运行的计算机，但在日常用语中，这些术语也可以指计算机本身。另一个区别是，在使用面向对象语言编写的分布系统中，资源可以封装为对象，并由客户端对象访问。这时，我们谈论客户端对象调用服务器对象的方法。
   2. Pg. 43 define remote invocation
      1. 远程调用（Remote invocation）代表了分布系统中最常见的通信范式，涵盖了一系列基于分布系统中通信实体之间的双向交流的技术，最终导致对远程操作、过程或方法的调用。远程调用是分布系统中的关键概念，允许一个实体请求另一个实体上的操作或服务，而这两个实体可以位于不同的计算机或设备上。这种通信范式在网络通信和分布式计算中起着重要作用，允许远程计算和协作。
   3. Pg. 44 Remote Procedure calls and access and location transparency & Remote Method Invocation
      1. 远程过程调用（Remote Procedure Calls, RPC）：远程过程调用是分布计算领域的一项重要概念，最初由Birrell和Nelson于1984年提出。在RPC中，远程计算机上的进程中的过程可以被调用，就好像它们是本地地址空间中的过程一样。底层的RPC系统隐藏了分布计算的重要细节，包括参数和结果的编码和解码、消息传递以及维护过程调用所需的语义。这种方法直接而优雅地支持了客户-服务器计算模型，其中服务器通过服务接口提供一组操作，客户端可以直接调用这些操作，就好像它们在本地可用一样。RPC系统因此提供（至少）访问和位置的透明性。
      2. 远程方法调用（Remote Method Invocation, RMI）：远程方法调用与远程过程调用类似，但适用于分布式对象的环境。在这种方法中，调用对象可以调用远程对象中的方法。与RPC类似，通常将底层细节隐藏在用户视线之外。不过，RMI实现可能会进一步支持对象标识以及在远程调用中传递对象标识符作为参数的能力。它们还受益于更紧密地集成到面向对象编程语言中。
      3. 这些技术都有一个共同点：通信表示发送方和接收方之间的双向关系，发送方明确将消息/调用发送到相关的接收方。接收方通常也知道发送方的身份，而且在大多数情况下，双方必须同时存在。相比之下，出现了一些技术，通过第三实体进行间接通信，允许发送方和接收方之间具有较强的解耦。特别是：
         * 1. 发送方不需要知道他们发送给谁（空间解耦）。
           2. 发送方和接收方不需要同时存在（时间解耦）。

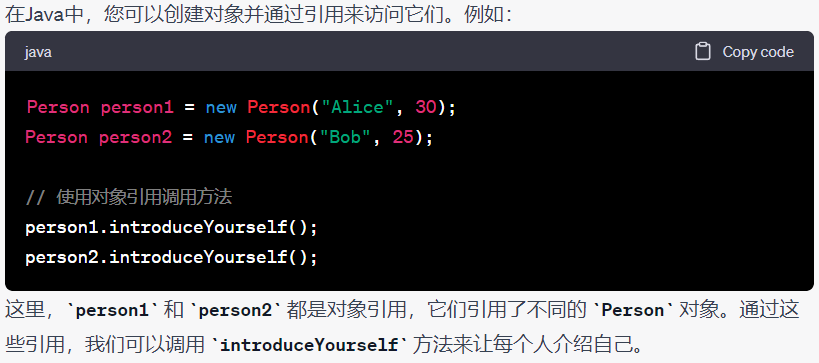
这些技术提供了更大的灵活性和松耦合性，适用于各种分布式系统的设计和实现。

* 1. Pg. 204 Section 5.4, 5.4.1, 5.4.2 to end of Page 211.
     1. 5.4 Remote method invocation
        1. 远程方法调用（Remote Method Invocation, RMI）与远程过程调用（RPC）密切相关，但扩展到了分布式对象的世界。在RMI中，一个调用对象可以调用一个潜在的远程对象中的方法。与RPC一样，通常将底层细节隐藏在用户视线之外。RMI与RPC的共同点包括：
           1. 它们都支持接口编程，从中获得相关好处。
           2. 它们通常都建立在请求-响应协议之上，并可以提供一系列调用语义，如至少一次和至多一次。
           3. 它们都提供类似的透明性水平 - 即，本地和远程调用使用相同的语法，但远程接口通常会暴露底层调用的分布性质，例如通过支持远程异常。
        2. 以下的差异增加了在编程复杂的分布式应用和服务时的表达能力。
           1. 程序员能够在开发分布式系统软件时充分发挥面向对象编程的表现力，包括使用对象、类和继承，并可以采用相关的面向对象设计方法和相关工具。
           2. 基于面向对象系统中的对象标识的概念，RMI-based 系统中的所有对象都具有唯一的对象引用（无论是本地还是远程的），这些对象引用也可以作为参数传递，从而提供比RPC更丰富的参数传递语义。
           3. 参数传递的问题在分布式系统中特别重要。RMI允许程序员不仅按值传递参数，还可以传递对象引用，这在底层参数很大或复杂的情况下特别有吸引力。在接收端，一旦接收到对象引用，就可以使用远程方法调用来访问该对象，而不必在网络上传输对象值。
     2. 5.4.1 Design issues for RMI
        1. 在设计层面，RMI在编程接口、调用语义和透明性等方面与RPC具有相同的设计问题, 我们首先描述传统的单一图像对象模型，然后描述分布式对象模型：
           1. 对象模型(Object model)：一个面向对象的程序，例如在Java或C++中，由一组相互作用的对象组成，每个对象由一组数据和一组方法组成。对象通过调用其方法与其他对象进行通信，通常传递参数并接收结果。对象可以封装其数据和方法的代码。一些语言，例如Java和C++，允许程序员定义可以直接访问实例变量的对象。但在分布式对象系统中使用时，对象的数据应该只能通过其方法访问。

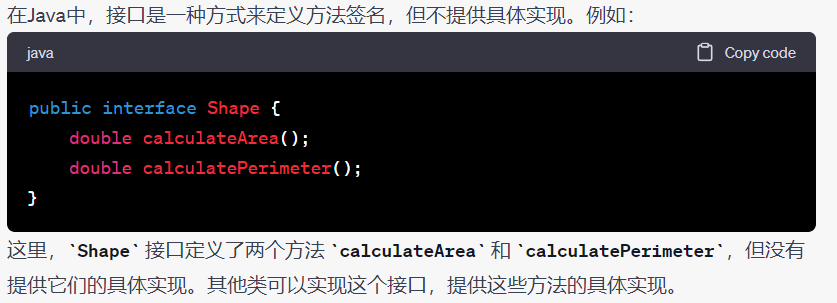
在java过程中，对象是类的实例：



* + - * 1. 对象引用(Object references)：对象可以通过对象引用访问。例如，在Java中，似乎持有对象的变量实际上持有对该对象的引用。要调用对象的方法，需要提供对象引用和方法名称，以及任何必要的参数。被调用方法的对象有时被称为目标，有时被称为接收者。对象引用是一等值，这意味着它们可以分配给变量，作为方法的参数传递，并作为方法的结果返回。



* + - * 1. 接口(Interfaces)：接口提供了一组方法的签名定义（即其参数、返回值和异常的类型），而不指定它们的实现。如果一个对象的类包含实现该接口方法的代码，那么该对象将提供特定的接口。在Java中，一个类可以实现多个接口，而接口的方法可以由任何类来实现。接口还定义了可以用于声明变量的类型或方法的参数和返回值的类型。请注意，接口没有构造函数。

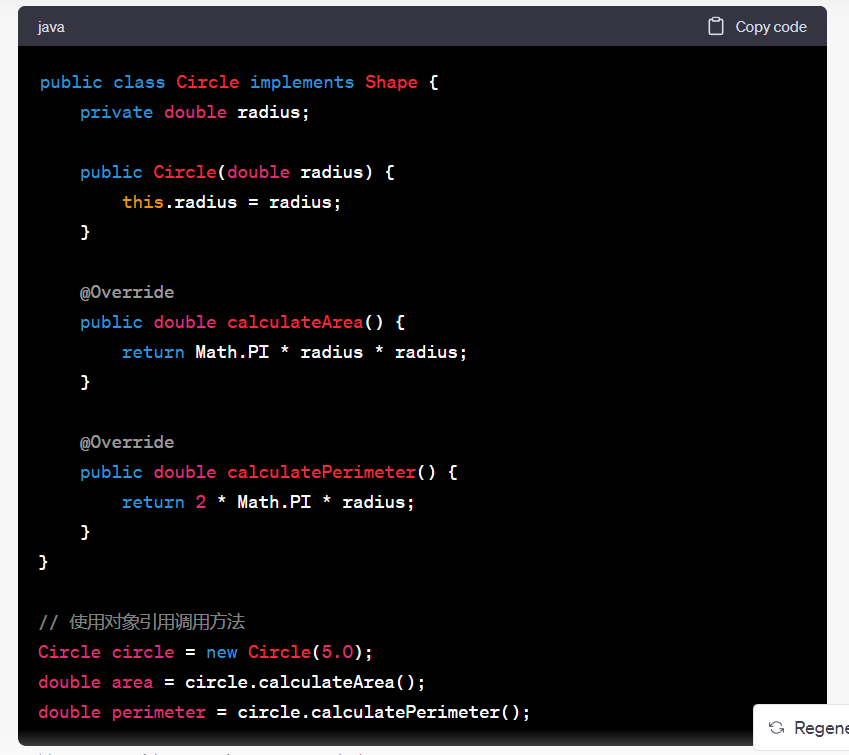


* + - * 1. 动作(Actions)：在面向对象的程序中，动作是由一个对象调用另一个对象的方法来启动的。调用可以包括执行方法所需的额外信息（参数）。接收者执行适当的方法，然后将控制返回给调用对象，有时提供结果。方法调用可以具有三种效果：

接收者的状态可能会更改。

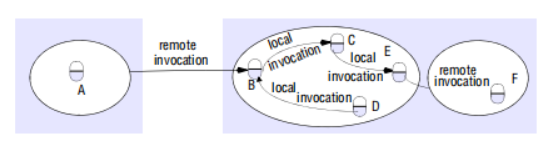
可能会实例化新对象，例如，通过使用Java或C++中的构造函数。

可能会发生其他对象的方法上的进一步调用。



\*由于调用可能导致在其他对象的方法中进一步调用，因此动作是一系列相关的方法调用，每个调用最终都会返回。

* + - * 1. 异常(Exceptions)：编程中经常会遇到各种不同严重程度的错误和意外情况。在方法执行过程中，可能会发现许多不同的问题，例如对象变量中的不一致值，或者尝试读取或写入文件或网络套接字时的失败。当程序员需要在其代码中插入测试来处理所有可能的异常或错误情况时，会降低代码的清晰度。异常提供了一种干净的方法来处理错误条件，而不会使代码变得复杂。此外，每个方法的声明明确列出了它可能遇到的异常，允许方法的用户处理它们。一段代码可以被定义为在特定的意外条件或错误出现时抛出异常。这意味着控制会传递到另一段捕获异常的代码，而不会返回到引发异常的地方。
        2. 垃圾回收(Garbage collection:)：在编程中，需要提供一种手段来释放不再需要的对象占用的空间。例如，像Java这样的语言可以自动检测对象不再可访问时，回收该空间并使其可用于分配给其他对象。这个过程被称为垃圾回收。当一种语言（例如C++）不支持垃圾回收时，程序员必须处理分配给对象的空间的释放。这可能是错误的一个主要源头。
      1. 分布式对象模型(Distributed objects):



在分布式对象模型中，对象被分为两种类型：

* 可以接收本地和远程调用的对象：这些对象可以在本地进程中调用它们的方法，也可以通过网络远程调用它们的方法。这意味着这些对象既可以在本地使用，也可以在远程计算机上使用，使它们适用于分布式系统。
* 只能接收本地调用的对象：这些对象只能在本地进程中调用它们的方法，无法通过网络进行远程调用。它们在本地上下文中可用，不适用于跨网络的分布式通信。
* Method invocations between objects in different processes, whether in the same computer or not, are known as remote method invocations. Method invocations between objects in the same process are local method invocations.
* 在这张图中，B和F就是remote objects(remote objects are objects that can receive remote invocations)
* All objects can receive local invocations, although they can receive them only from other objects that hold references to them(例如在这张图里C可以invoke E的方法，但前提是C需要有E的reference)
* Remote object references: Other objects can invoke the methods of a remote object if they have access to its remote object reference. For example, a remote object reference for B must be available to A.

假设有两台计算机A和B，它们位于不同的地方，并且彼此通过网络连接。在计算机A上有一个对象（远程对象）的实例，我们将其称为remoteObj，该对象具有某些方法，例如doSomething()。

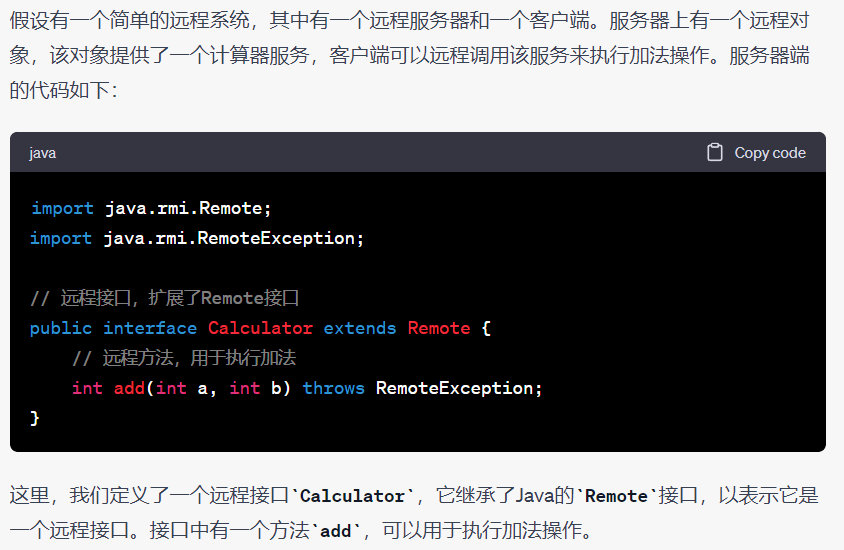
获取远程对象引用：计算机B上的程序需要远程调用remoteObj上的doSomething()方法。为了实现这一点，首先需要获取remoteObj的远程对象引用。

远程对象引用：远程对象引用是一个标识符，用于唯一标识remoteObj，它是计算机A和B之间通信的关键。在这个示例中，远程对象引用可以是一个包含remoteObj的网络地址或其他标识信息的标识符。

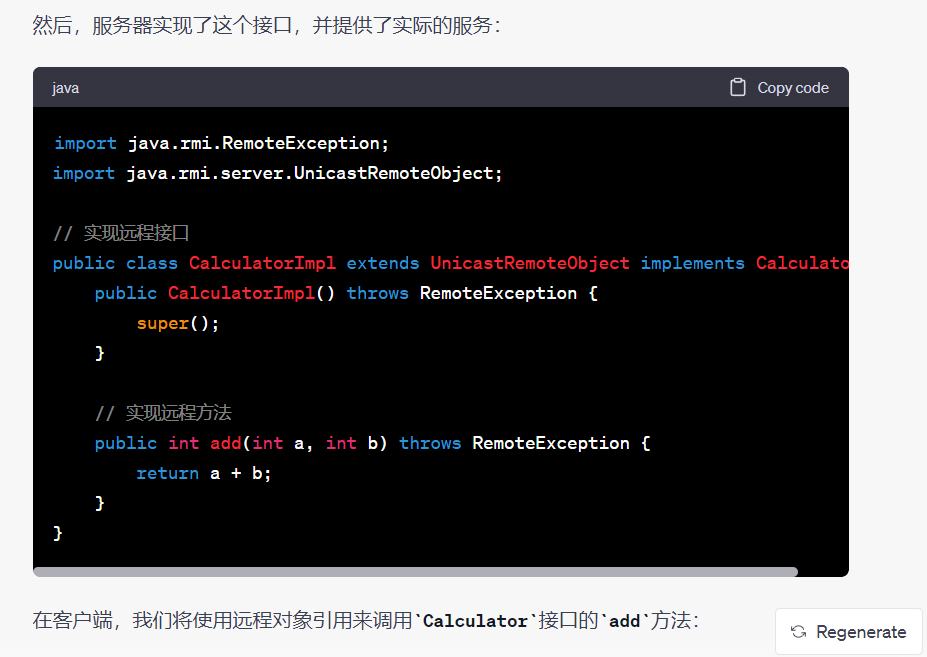
远程方法调用：在计算机B上，程序使用remoteObj的远程对象引用，调用doSomething()方法。计算机B通过网络将请求发送到计算机A，请求执行doSomething()方法。

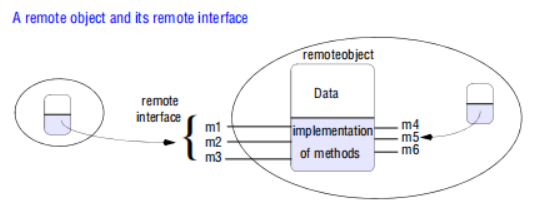
计算机A的响应：计算机A接收到请求后，执行remoteObj上的doSomething()方法，然后将结果发送回计算机B。

通过这个过程，计算机B能够远程调用计算机A上的对象的方法，而不需要在本地创建该对象的实例。远程对象引用充当了标识符，用于唯一标识和引用远程对象。

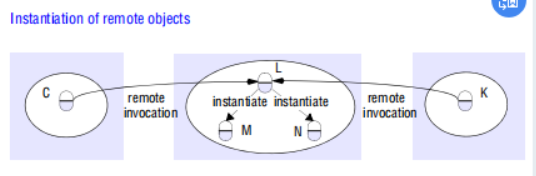
* Remote interfaces: Every remote object has a remote interface that specifies which of its methods can be invoked remotely. For example, the objects B and F must have remote interfaces.

在这个示例中，客户端使用远程对象引用来查找和调用服务器上的远程对象的方法。

远程对象引用充当了标识符，用于唯一标识和引用远程对象。客户端可以在不同的计算机上通过网络调用服务器上的远程方法。



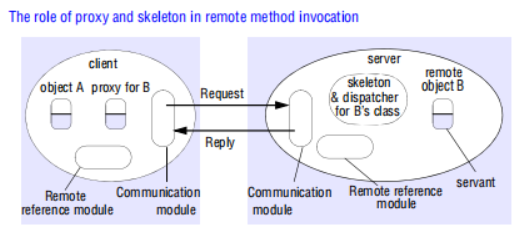
* + - * 1. 动作(Actions): 在分布式系统中，对象之间的方法调用可能会导致对象引用的传递，使得对象A需要持有对象B的远程对象引用。远程对象引用可以作为远程方法调用的结果获得，这意味着对象A可以通过调用对象B的方法来获得对象F的远程引用。当一个操作导致新对象的实例化(instantiation)时，通常新对象将存在于请求实例化的进程内，例如在使用构造函数的地方。如果新实例化的对象具有远程接口，那么它将成为一个远程对象，并具有一个远程对象引用。分布式应用程序可以为远程对象提供方法，用于实例化可以通过RMI访问的对象，从而有效地提供了远程对象的实例化效果:



对象L包含一个用于创建远程对象的方法，那么来自C和K的远程调用可能会导致分别实例化对象M和N。

这允许分布式系统中的对象通过远程方法调用来创建新的对象实例。

* + - * 1. Garbage collection in a distributed-object system: 如果编程语言（例如Java）支持垃圾回收，那么分布式对象系统应该允许对远程对象进行垃圾回收。分布式垃圾回收通常是通过本地垃圾回收器与额外的模块合作实现的，这个模块执行一种基于引用计数的分布式垃圾回收。如果没有垃圾回收机制，那么不再需要的远程对象应该被删除。
        2. Exceptions: 远程方法调用可能由于被调用对象位于不同的进程或计算机中而失败。例如，包含远程对象的进程可能崩溃或繁忙无法回复，或者调用或结果消息可能会丢失。因此，远程方法调用应该能够引发异常，例如由于分布的原因而引发的超时异常，以及在调用的方法执行期间引发的异常。后者的示例包括尝试超出文件末尾读取数据，或者在没有正确权限的情况下访问文件。
    1. 5.4.3 Implementation of RMI



* + - 1. 通信模块（Communication Module）：通信模块是协同工作的两个模块，执行请求-回复协议，用于在客户端和服务器之间传输请求和回复消息。请求和回复消息的内容如图所示。通信模块只使用前三个项，这些项指定了消息类型、请求的ID以及要调用的远程对象的远程引用。操作ID以及所有编组（marshalling）和解编组（unmarshalling）的工作是由下面讨论的RMI软件来处理。通信模块共同负责提供指定的调用语义，例如至多一次（at-most-once）。
         1. 服务器的通信模块：服务器的通信模块选择要调用的对象类的调度程序（dispatcher），将本地引用传递给调度程序，该引用是从请求消息中的远程对象标识符获取的，而该远程对象标识符则从远程引用模块获得。
         2. 这段文本强调了在分布式系统中实现远程方法调用涉及到多个模块和组件，每个组件都有其特定的任务和职责，以确保有效的远程通信和方法调用。通信模块扮演着关键的角色，用于处理请求和回复消息的传输，而服务器的通信模块选择适当的调度程序来处理远程方法调用。RMI软件负责处理编组和解编组等更高级别的任务。
      2. 远程引用模块（Remote Reference Module）: 负责在本地对象引用和远程对象引用之间进行转换，以及创建远程对象引用。为了支持其职责，每个进程中的远程引用模块都有一个远程对象表，用于记录本地对象引用和远程对象引用之间的对应关系（这些远程对象引用是全系统范围的）。该表包括：

对该进程持有的所有远程对象的条目。例如，在图5.15中，远程对象B将记录在服务器的表中。

对每个本地代理的条目。例如，在图5.15中，代理对象B将记录在客户端的表中。

代理（Proxy）的角色将在RMI软件部分的子节中进行讨论。远程引用模块的操作包括：

当一个远程对象首次被传递为参数或结果时，将要求远程引用模块创建一个远程对象引用，并将其添加到表中。

当一个远程对象引用出现在请求或回复消息中时，将要求远程引用模块提供相应的本地对象引用，这个本地对象引用可能指向代理或远程对象。在远程对象引用不在表中的情况下，RMI软件会创建一个新的代理对象，并要求远程引用模块将其添加到表中。

* + - * 1. 远程引用模块由RMI软件的各个组件在编组和解编组远程对象引用时调用。例如，当请求消息到达时，表被用来确定要调用哪个本地对象。
        2. 远程引用模块的主要作用是管理远程对象引用的映射，以便在分布式系统中进行远程调用时，能够正确地定位和识别相应的本地对象或代理。这有助于实现远程方法调用的透明性和有效性。
      1. Servants：Servants是一个类的实例，它提供了远程对象的主体。在RMI中，Servants负责处理远程请求，它们存在于服务器进程中，当远程对象被实例化时创建，并在不再需要时被垃圾回收或删除。
      2. RMI软件：RMI软件是位于应用级别对象和通信以及远程引用模块之间的一层软件。RMI软件包括代理（Proxy）、调度程序（Dispatcher）、和骨架（Skeleton）等中间件对象，它们扮演以下角色：
         1. 代理（Proxy）：代理的作用是使远程方法调用对客户端透明，它表现得像本地对象一样，但不是执行调用，而是将调用封装在消息中发送给远程对象。代理隐藏了远程对象引用的细节、参数的编组（marshalling）、结果的解编组（unmarshalling）以及与客户端的消息发送和接收。每个远程对象对应一个代理，代理类实现了远程对象的远程接口的方法，以确保远程方法调用适用于远程对象的类型。
         2. 调度程序（Dispatcher）：服务器拥有一个调度程序和一个骨架，每个骨架对应一个表示远程对象的类。调度程序接收来自通信模块的请求消息，使用操作ID来选择适当的骨架中的方法，并将请求消息传递给骨架。调度程序和代理使用相同的操作ID分配来表示远程接口的方法。
         3. 骨架（Skeleton）：骨架是远程对象的类的一部分，它实现了远程接口的方法。这些方法与代表远程对象的Servants中的方法在实现上有很大差异。骨架方法会解编组请求消息中的参数，并调用Servants中相应的方法。然后，它等待调用完成，将结果和异常一起编组到回复消息中，以返回给发送请求的代理方法。
      3. 远程对象引用的编组：远程对象引用在编组时包括关于远程对象的远程接口的信息，例如远程接口的名称或远程对象的类。这些信息使得代理类能够确定要创建新代理时的类型。例如，代理类名可以通过将远程接口的名称附加"\_proxy"来生成。
      4. 代理、调度程序和骨架类的生成：代理、调度程序和骨架类是由接口编译器(interface compiler)自动生成的，这些类用于实现RMI中的远程方法调用。

-----------------------------------------------------------------------------------------RMI 知识点总结---------------------------------------------------------------------------------

\*\*RMI（远程方法调用）概述：\*\*

1. RMI是一种分布式计算技术，用于在不同计算机或进程之间进行远程方法调用。

2. 它允许客户端应用程序调用位于远程服务器上的对象的方法，就像调用本地对象一样。

3. RMI用于构建分布式系统，允许对象在不同计算机上通信和协作。

\*\*RMI的关键组件：\*\*

4. \*\*远程对象（Remote Object）\*\*：是服务器上的对象，它的方法可以被远程客户端调用。

5. \*\*远程接口（Remote Interface）\*\*：定义了远程对象可以提供的方法和服务。

6. \*\*远程对象引用（Remote Object Reference）\*\*：用于标识和定位远程对象的引用。

7. \*\*代理（Proxy）\*\*：代理在客户端上充当远程对象的本地代表，使远程方法调用对客户端透明。

8. \*\*骨架（Skeleton）\*\*：骨架在服务器端上充当远程对象的本地代表，负责将远程方法请求分派到远程对象。

9. \*\*Servants（Servants）\*\*：Servants是远程对象的实际实现，它们包含远程方法的实际代码。

\*\*RMI的工作原理：\*\*

10. 客户端通过代理对象调用远程方法，代理对象将请求封装成消息发送给服务器。

11. 服务器上的骨架接收消息，将其传递给相应的Servants来执行远程方法。

12. 执行结果和可能的异常被封装成响应消息，发送回客户端。

13. 通信模块负责消息的传输，远程引用模块管理本地和远程对象引用之间的映射。

\*\*RMI中的概念：\*\*

14. \*\*分布式对象模型\*\*：RMI采用面向对象的分布式模型，对象之间可以远程通信。

15. \*\*远程接口和实现\*\*：远程接口定义了远程对象提供的方法，实现类包含方法的具体代码。

16. \*\*远程方法调用\*\*：客户端通过代理对象调用远程对象上的方法。

17. \*\*远程对象引用\*\*：用于唯一标识远程对象的引用，传递给客户端以进行远程调用。

\*\*其他重要概念：\*\*

18. \*\*垃圾回收\*\*：确保不再需要的远程对象被释放以节省资源。

19. \*\*异常处理\*\*：处理在远程调用期间可能出现的异常，包括分布相关的异常。

20. \*\*代理、调度程序和骨架的生成\*\*：这些组件是自动生成的，用于支持RMI的远程方法调用。

-----------------------------------------------------------------------------------------RMI 知识点总结---------------------------------------------------------------------------------

1. **Reading for Web Service:**

概述：Web服务概述：Web服务提供了一种服务接口，使客户端可以以比Web浏览器更一般化的方式与服务器交互。客户端通过使用XML格式的请求和回复，通常通过HTTP传输，来访问Web服务接口中的操作。

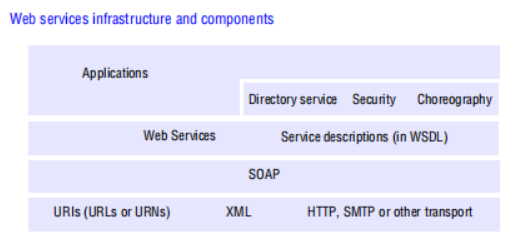
Web服务的使用方式：与CORBA（Common Object Request Broker Architecture）不同，Web服务的访问方式更加灵活，可以更容易地用于Internet范围的应用。它们支持更加自由的访问方式，适用于互联网上的各种应用。

接口描述语言（IDL）：与CORBA和Java一样，Web服务的接口可以使用IDL进行描述。但对于Web服务，需要提供额外的信息，包括使用的编码和通信协议以及服务位置等。

XML安全性：用户需要一种安全的方式来创建、存储、修改文档并在Internet上进行交换。传输层安全性（TLS）的安全通道并不能提供所有必要的要求，因此XML安全性的概念被引入以填补这一缺口。

Web服务的重要性：Web服务在分布式系统中变得越来越重要，它们支持全球Internet上的互操作性，包括重要的业务对业务整合领域，以及允许第三方开发者在现有服务基础之上创造创新软件的“mashup”文化。此外，Web服务还为网格计算和云计算提供了基础的中间件。

* 1. **9.1 Introduction**
     1. Web增长的有效性： 在过去的20年中，Web的迅猛增长证明了在互联网上使用简单协议作为广泛区域服务和应用的基础的有效性。这意味着互联网的简单协议架构为广泛的在线服务和应用提供了支持。
     2. HTTP请求-响应协议： HTTP（Hypertext Transfer Protocol）是一种协议，用于在互联网上传输超文本和数据。它的主要功能是允许通用客户端（例如Web浏览器）向Web服务器发送请求以检索网页和其他资源，然后接收服务器的响应。
     3. 通用浏览器的限制： 通用浏览器虽然非常灵活，但在处理特定需求和复杂应用时存在限制。即使通过下载特定应用程序（applets）来增强其功能，它们仍然不能满足某些特定应用的需求。
     4. 客户端-服务器模型： 原始的客户端-服务器模型中，客户端和服务器都有特定的功能。Web服务重新引入了这个模型，其中特定于应用程序的客户端与提供具有特定功能接口的服务进行交互。
     5. Web服务的互操作性： Web服务提供了一种基础设施，使不同组织中的客户端程序能够无需人工干预而相互交互。它们允许开发复杂的应用程序，通过提供集成多个其他服务的服务来实现。
     6. Web服务与浏览器的区别： 由于Web服务提供了更广泛的互操作性，通用Web浏览器无法直接访问它们。
     7. Web服务的提供： Web服务的提供基于使用HTTP请求来执行程序。当HTTP请求中的URL指向可执行程序时，例如搜索，该程序将生成结果并返回。Web服务是Web的扩展，可以由Web服务器提供，但它们的服务器不一定是传统的Web服务器。
     8. Web服务器和Web服务的区别： Web服务器提供基本的HTTP服务，而Web服务提供了基于其接口定义的操作的服务。
     9. XML中的数据表示： 客户端和Web服务之间的消息交换使用XML进行数据表示和编组。XML是一种文本表示方法，虽然相对冗长，但由于其可读性和易于调试，已经被广泛采用。



* + 1. SOAP协议： SOAP（Simple Object Access Protocol）协议规定了使用XML打包消息的规则，例如用于支持请求-响应协议。SOAP用于封装这些消息并通过HTTP或其他协议（如TCP或SMTP）进行传输。Web服务通过部署服务描述来指定服务的接口和其他方面，以使潜在的客户端能够了解服务的特性。
  1. **9.2 Web Services**
     1. Web服务支持两种通信模式：
        1. 同步请求-响应模式：客户端发送请求，服务端立即返回响应。
        2. 异步消息模式：请求可能需要较长时间才能完成，因此可以采用异步文档交换，客户端发送请求后，定期接收状态信息，最后接收完成的详细信息。或者甚至可以使用事件模式，其中客户端可以注册感兴趣的事件，以在某些事件发生时（例如服务到达或离开）获得通知。
     2. 松耦合（loose coupling）:松耦合是指减小各个服务之间的依赖关系，以建立一个灵活的底层架构，从而减少一个服务的更改对其他服务产生连锁影响的风险。
        1. 使用接口编程：通过将接口与实现分离（也支持在选择编程语言和平台等方面的异构性），接口编程提供了一种松耦合的级别。接口编程被大多数分布式系统范式采纳，包括分布对象和组件，以及Web服务。
        2. 使用简单、通用接口：分布式系统中的接口趋向简单和通用，这在世界范围网（World Wide Web）和Web服务的REST方法中得到了体现。这种方法通过减少对特定操作名称的依赖来促进松耦合。这也导致了数据比操作更重要，通常在数据中包含了交互操作的语义（例如，Web服务中的相关XML文档定义）。
        3. 使用多种通信范式：Web服务支持多种通信范式，包括请求-响应通信、异步消息传递以及间接通信范式。不同通信范式的选择直接影响了松耦合程度。例如，在请求-响应通信中，两方之间本质上是耦合的；异步消息传递提供一定程度的解耦（在第6章中称为同步解耦），而间接通信还提供了时间和空间的解耦。
     3. REST，或表述性状态传输（Representational State Transfer），是一种基于网络体系结构的设计风格，用于创建分布式系统和网络应用程序。REST的核心思想是通过使用HTTP协议的不同方法（例如GET、POST、PUT、DELETE）来进行资源的创建、读取、更新和删除，以实现松耦合的分布式系统。以下是REST的一些关键原则和特点：
        1. 资源和标识： 在REST中，一切都被视为资源，每个资源都有一个唯一的标识符（通常是URL）。资源的状态可以通过资源的表示（通常是XML或JSON）来表达。
        2. 状态无关性： REST架构是无状态的，每个请求都包含了足够的信息，服务器不需要保留客户端的状态。这使得系统更容易扩展和管理。
        3. 统一接口： REST使用统一的接口，包括HTTP方法（GET、POST、PUT、DELETE）来执行各种操作，使得客户端和服务器之间的通信更简单和一致。
        4. 资源的自描述性： 每个资源都包含了足够的信息来描述其自身和与其他资源之间的关系。这使得客户端可以理解资源并进行适当的操作。
        5. 按需获取： REST允许客户端按需获取资源的一部分，而不是整个资源，从而提高效率。
        6. 无连接性： REST请求是无连接的，每个请求都是独立的，服务器不需要保留有关客户端的信息。
  2. **Section 9.7.3 starting on page 417**
     + 1. 云计算： 云计算是一种基于互联网的服务，提供足以支持大多数用户需求的应用程序、存储和计算服务，从而使用户能够几乎完全摆脱本地数据存储和应用软件的需求。云计算也倡导将一切视为服务，从物理或虚拟基础设施到软件，通常按使用量付费而不是购买。这个概念与一种新的计算业务模型密切相关，其中云供应商为客户提供计算、数据和其他服务，根据他们的日常使用需求，例如通过互联网提供足够的存储容量以充当存档或备份服务。
       2. Grid计算与云计算的重叠： Grid计算的发展先于云计算的出现，是其出现的一个重要因素。它们共享在互联网上提供资源（服务）的共同目标。然而，Grid通常专注于数据密集或计算昂贵的高端应用，而云计算更为通用，为个人计算机用户到高端用户提供一系列服务。与云计算相关的业务模型也是一个区分特征。因此，可以说Grid是云计算的早期示例，但云计算自那时以来已经得到了显著的发展。
       3. Web服务和云计算： 一切视为服务的观点使Web服务成为云计算的自然实现途径，许多供应商都采用了这种方式。其中最著名的是Amazon Web Services（AWS），云计算的另一种方法，即Google基础设施和相关的Google App Engine，它们采用了比Web服务更轻量、性能更高的方法。
       4. Amazon Web Services（AWS）： AWS是建立在Amazon.com拥有的庞大物理基础设施上的一套云服务。最初为支持其电子零售业务的内部用途而开发，Amazon现在将许多设施提供给外部用户，使他们能够在该基础设施上运行独立的服务。AWS的实现解决了关键的分布式系统问题，如管理服务可用性、可扩展性和性能，允许开发人员专注于使用他们的服务
       5. Amazon Web Services（AWS）的服务： Amazon提供了广泛且可扩展的一系列服务，其中EC2（Elastic Compute Cloud）是一个更详细的服务。EC2是一种弹性计算服务，其中“弹性”指的是能够根据客户的需求提供可调整大小的计算能力。EC2向用户提供虚拟机，称为实例，根据其规格的需求。例如，用户可以请求以下类型的实例：
          1. 标准实例，适用于大多数应用程序。
          2. 高内存实例，提供额外的内存容量，例如用于涉及缓存的应用程序。
          3. 高CPU实例，用于支 持计算密集型任务。
          4. 集群计算实例，提供具有高带宽互连的虚拟处理器集群，用于高性能计算任务。

# Week8: Peer-to-peer systems

1. **Readings**
   1. **10.1 Introduction**
      1. 对等系统 (Peer-to-Peer Systems): 旨在通过消除单独管理的服务器和基础设施要求，实现大规模数据和资源共享的系统。可以通过互联网或其他网络上众多个人计算机和工作站的资源来支持分布式服务和应用程序。
         1. 扩展性问题 (Scalability Issue): 当所有服务主机必须由服务提供商拥有和管理时，通过增加主机数量来扩展流行服务受到限制。管理和故障恢复成本以及可提供给单个服务器站点的网络带宽都是限制因素。
         2. 资源共享 (Resource Sharing): 对等系统旨在利用互联网边缘的可用资源，包括存储、计算、内容和人的存在。这种资源共享在各种个人计算机上的分布式应用中已经有所体现。
         3. 分布式系统 (Distributed Systems): 这些系统的设计使每个用户都能贡献资源，而所有节点具有相同的功能能力和责任。它们不依赖于中央管理系统的存在，实现了分散和自组织的服务。
         4. 中心化设计 (Centralized Design): 传统的客户端-服务器系统管理和提供对资源的访问，如文件、网页或信息对象，这些资源位于单个服务器计算机或小集群上。这种设计具有简单的资源放置和管理，但服务规模受到服务器硬件容量和网络连接的限制。
         5. 分布式信息对象管理 (Distributed Information Object Management): 对等系统需要考虑信息对象的放置和后续检索算法，以实现分散和自组织的服务，动态平衡存储和处理负载。
         6. 匿名性设计 (Anonymity Design): 对等系统可以设计为为资源的提供者和用户提供有限程度的匿名性。
         7. 数据放置算法 (Data Placement Algorithm): 对其有效运行而言，关键问题是选择一种用于在许多主机之间放置数据并随后访问数据的算法，以平衡工作负载并确保可用性，同时不会增加不必要的开销。
         8. 资源的不确定性 (Uncertainty of Resources): 由众多不同用户和组织拥有和管理的计算机和网络连接是不稳定的资源；他们的所有者不能保证保持开机、连接和无故障。因此，参与对等系统的进程和计算机的可用性是不可预测的。
         9. 复制资源 (Replicating Resources): 对等系统不能依赖于对个体资源的保证访问，尽管它们可以被设计为使对复制对象的访问失败的概率变得极小。然而，这种对等系统的弱点可以通过资源的复制来实现一定程度的抵抗恶意节点篡改（例如，通过拜占庭容错技术）。
         10. 多服务器可伸缩和容错体系结构 (Multi-Server Scalable and Fault-Tolerant Architecture): 早期的基于互联网的服务，包括DNS和Netnews/Usenet，采用了多服务器可伸缩和容错体系结构。这些体系结构提供了可伸缩性和容错性。
         11. 对等系统的发展 (Evolution of Peer-to-Peer Systems): 对等系统的发展经历了三代。第一代由Napster音乐交换服务推出，随后出现了第二代文件共享应用程序，提供了更大的可伸缩性、匿名性和容错性，包括Freenet、Gnutella、Kazaa和BitTorrent。
         12. 第三代对等系统具有中间件层，用于在全球范围内以应用程序无关的方式管理分布式资源。
         13. 对等系统中的中间件平台，如Pastry、Tapestry、CAN、Chord和Kademlia，用于在互联网上广泛分布的计算机上放置资源，以及为客户端路由消息。
         14. 这些系统提供资源的可靠交付保证，并在网络跳数内提供请求的可靠交付。
         15. 资源通过全球唯一标识符（GUIDs）进行标识，通常是从资源的状态中派生的安全哈希。
         16. 安全哈希技术使资源能够自我认证，以抵御不受信任节点的篡改。
         17. 对等存储系统天生适合存储不可变对象，但对于值不断变化的对象，需要添加可信服务器来管理版本并标识当前版本。
         18. 设计要求高可用性的应用程序需要避免同时使对象的所有副本不可用，可以通过使用随机分布的GUIDs来减小此风险。
      2. Overlay routing versus IP routing
         1. IP路由是互联网上常用的方式，它类似于邮递员送信件的方式，信件要经过一系列邮局或节点，最终到达目的地。
         2. 覆盖路由是一种在对等系统中使用的不同方式，其中每个计算机都可以充当邮递员，帮助将数据传送到需要的地方。这就好像人们在社区中互相帮助传递信息，而不仅仅依赖邮递员。
         3. 文本提到，尽管覆盖路由和IP路由有一些相似之处，但它们之间也有一些不同之处。这些不同可能部分是因为互联网一开始的设计方式，但由于互联网的遗留方式太强大，因此需要额外的应用层路由机制，以更好地支持对等系统中的应用程序。
      3. Distributed computation
         1. 分布式计算涉及利用大量终端用户计算机的计算能力来执行各种计算任务。
         2. SETI@home项目是一个著名的分布式计算项目，旨在搜索地外智慧生命的迹象。它将射电望远镜数据分发给志愿者的计算机，然后对结果进行协调和处理。
         3. SETI@home项目规模巨大，吸引了数百万台个人计算机的参与，以处理大量工作单元。
         4. 与其他分布式计算不同，SETI@home计算不需要计算机之间的通信或协调，结果通过短消息传递给中央服务器。
   2. **10.2 Napster and its legacy**
      1. 分布式计算涉及利用终端用户计算机上的闲置计算能力来执行各种计算任务。
      2. Napster是一个数字音乐文件下载的先驱，它展示了对等解决方案的需求和可行性。
      3. Napster采用了分布式文件共享方法，允许用户在个人计算机上存储和分享音乐文件。
      4. Napster的成功在于提供广泛分布的音乐文件，履行Shirky的格言。
      5. Napster最终因法律诉讼而被关闭。
      6. 具有许多节点的对等系统中，对于共享数据和资源的接收方和提供方的匿名性是设计者的关注点。
   3. **10.3 Peer-to-peer middleware**
      1. 在对等应用程序设计中，提供客户快速和可靠地访问分布在网络中的数据资源是一个主要问题。
      2. Napster维护了一个统一的文件索引，使用户能够查找文件的位置。
      3. 第二代对等文件存储系统使用分区和分布式索引，但每个系统都使用特定的算法。
      4. 对等中间件的功能需求包括支持分布在广泛分布的网络中的许多主机的服务构建。
      5. 对等中间件的非功能性需求包括全球可扩展性、负载平衡、本地互操作性、动态主机可用性、数据安全性和匿名性。

--------------------------------------------------------------------------------------Peer-to-peer Systems Summary-------------------------------------------------------------------------------------

* + - 1. 当谈到对等应用程序时，我们实际上在讨论一种网络应用程序的设计方法，它允许互联网上的不同计算机（称为“对等节点”）之间直接共享数据和资源，而无需依赖中央服务器。这种方法旨在实现以下目标：

1. \*\*去中心化：\*\* 对等应用程序不依赖于单一的中央服务器来管理数据和资源。相反，它们允许每个对等节点扮演服务提供者和服务消费者的角色。

2. \*\*共享资源：\*\* 对等应用程序使用户能够共享各种类型的数据和资源，如文件、音乐、视频等。这些资源通常存储在对等节点的个人计算机上。

3. \*\*分布式：\*\* 对等应用程序通过将资源分散在整个网络中的不同计算机上来实现数据的分布式存储和访问。

4. \*\*自组织：\*\* 这些应用程序允许计算机自动加入和离开网络，而无需中央协调。它们能够自动平衡资源的负载。

一些相关术语包括：

- \*\*对等节点（Peers）：\*\* 互联网上的不同计算机，它们共享数据和资源，并相互合作来实现应用程序的功能。

- \*\*中央服务器（Central Server）：\*\* 通常情况下，对等应用程序不依赖于中央服务器来协调操作。然而，有时中央服务器可能用于辅助通信或协调操作。

- \*\*资源共享：\*\* 对等应用程序的核心概念之一是资源共享，用户可以上传和下载文件、数据或其他资源。

- \*\*分布式存储：\*\* 数据和资源分布在不同计算机上，而不是集中存储在单一服务器上。

- \*\*自组织性：\*\* 对等应用程序可以适应新计算机的加入和现有计算机的离开，无需人工干预。

总的来说，对等应用程序是一种允许用户共享和访问数据和资源的网络应用程序，而无需中央服务器的干预。它们的设计目标是实现去中心化、分布式、自组织和资源共享。

在对等应用程序中，中间层（Middleware）是一种软件层，用于简化对等应用程序的设计、开发和运行。它提供了一些重要的功能和服务，以帮助对等节点更轻松地共享数据和资源，并实现应用程序的各种操作。以下是中间层在对等应用程序中的作用和功能：

1. \*\*资源管理：\*\* 中间层可以帮助管理分布在网络中的资源，包括数据文件、信息对象等。它可以跟踪资源的位置和可用性。

2. \*\*位置服务：\*\* 中间层可以提供资源的位置服务，使客户能够找到所需资源的位置，而无需了解具体的网络拓扑。

3. \*\*通信支持：\*\* 中间层可以处理节点之间的通信，包括请求资源、传输数据等。它可以处理网络通信的复杂性，使应用程序更容易编写。

4. \*\*负载平衡：\*\* 中间层可以分配资源的负载，以确保各个节点之间的工作负载均衡，从而提高性能。

5. \*\*安全性和隐私：\*\* 中间层可以提供安全性功能，包括身份验证、加密和访问控制，以保护数据和资源的安全性和隐私。

6. \*\*动态管理：\*\* 中间层能够处理网络中节点的动态加入和退出，以确保系统的稳定性和可用性。

总之，中间层在对等应用程序中充当了关键的角色，帮助应用程序更好地实现去中心化、分布式和资源共享的目标。它简化了开发过程，并提供了许多必要的功能，以使对等应用程序更有效地运行。

--------------------------------------------------------------------------------------Peer-to-peer Systems Summary-------------------------------------------------------------------------------------

* 1. **10.4 Routing overlays**
     1. 在对等系统中，中间层扮演着重要角色，特别是在处理分布式资源的管理和路由方面。这个中间层通常被称为"路由覆盖层"（Routing Overlay），它负责处理如何定位节点和对象，以便客户能够快速、可靠地访问这些对象。
     2. 以下是有关路由覆盖层的一些关键概念：
        1. 路由覆盖层的定义： 路由覆盖层是一个分布式算法，负责确定如何定位节点和对象。它被称为"覆盖层"，因为它实现了一个路由机制，与网络层的其他路由机制（如IP路由）完全独立。
        2. GUID： 路由覆盖层使用全局唯一标识符（GUID）来标识节点和对象。这些标识符是"纯"名称，因为它们不透露有关其引用对象位置的信息。
        3. 路由请求： 客户端想要操作一个对象时，它会将请求发送给路由覆盖层，包括对象的GUID。路由覆盖层将请求路由到包含对象副本的节点，以满足客户端的请求。
        4. 对象的插入/删除: 路由覆盖层不仅负责路由请求，还负责管理对象的插入和删除。当节点希望将新对象提供给对等服务时，它会计算对象的GUID并通知路由覆盖层，以确保其他客户端可以访问该对象。当客户端请求删除对象时，路由覆盖层需要将对象标记为不可用。
        5. 节点的加入和移除： 对等系统中的节点（计算机）可能随时加入或离开系统。当节点加入时，路由覆盖层会协调它承担一些其他节点的责任。当节点离开系统时，路由覆盖层需要重新分配其责任给其他节点。
        6. 分布式哈希表（DHT）： 路由覆盖系统通常使用哈希函数来生成GUID，以确定对象的位置和检索它们。这种路由系统有时被称为"分布式哈希表"，因为它使用哈希表的概念来管理对象的位置和访问。
  2. **10.5 Overlay case studies: Pastry, Tapestry**
     1. Pastry
        1. 128位GUIDs： Pastry中的所有节点和对象都有128位的全局唯一标识符（GUIDs）。节点的GUID是通过应用安全散列函数（如SHA-1）计算的，通常使用节点的公钥作为输入。对于对象（如文件），GUID是通过应用安全散列函数计算的，通常使用对象的名称或对象的一部分存储状态作为输入。这些GUIDs在0到2^128-1的范围内随机分布，不会泄露有关其计算来源的信息，且GUID之间的冲突极不可能发生。
        2. 路由算法： Pastry使用一种路由算法，可以在O(log N)步内正确路由到任何GUID。如果GUID标识的是当前活动的节点，消息将被直接传递给该节点；否则，消息将被传递到与其数值最接近的当前活动节点。活动节点负责处理其数值附近所有对象的请求。
        3. 本地性度量： 为了最小化不必要的传输路径延长风险，Pastry使用基于底层网络中的网络距离的本地性度量来选择适当的邻居节点。这些度量可以包括跳数或往返延迟等。
        4. 自组织： 成千上万的位于广泛分布的站点的主机可以参与Pastry覆盖层，而且它是完全自组织的。新节点加入覆盖层时，它们可以从现有成员那里获取构建路由表和其他所需状态的数据。如果节点故障或离开，剩余的节点可以检测到其缺席，并在相似数量的消息中协作重新配置以反映路由结构中所需的更改。
        5. 这段代码描述了当Pastry系统接收到一条消息（M），并要将它传递给一个特定的目标节点（D）时的步骤。为了做到这一点，Pastry使用了叶子集（leaf set）和路由表（routing table）来帮助确定下一个节点，以便消息最终到达目标。
           1. 首先，Pastry检查目标节点（D）是否在当前节点的叶子集范围内。如果是的话，它会将消息(M)直接发送给叶子集中与D最接近的节点。如果D正好等于当前节点的GUID，那么消息会被传递给当前节点。
           2. 如果目标节点（D）不在叶子集范围内，那么Pastry将使用路由表来决定下一个节点。它首先查找D和当前节点（A）的GUID之间最长的共同前缀，然后根据共同前缀的长度选择一个路由表中的位置(i)。然后，它会检查路由表中是否存在一个有效的下一跳节点（R[p,i]是否为空）。如果存在，它将消息(M)转发给路由表中指定的节点。
           3. 如果路由表中没有与目标D匹配的入口，Pastry将消息传递给叶子集（L或R）中具有相同前缀长度(p)但具有数字上更接近D的任何节点。这确保消息最终到达与目标D最接近的节点。

