2.摘要

Napster pioneered the idea of peer-to-peer file sharing, and supported it with a centralized file search facility. Subsequent P2P systems like Gnutella adopted decentralized search algorithms. However, Gnutella’s notoriously poor scaling led some to propose distributed hash table solutions to the wide-area file search problem.

Contrary to that trend, we advocate retaining Gnutella’s simplicity while proposing new mechanisms that greatly improve its scalability. Building upon prior research, we propose several modifications to Gnutella’s design that dynamically adapt the overlay topology and the search algorithms in order to accommodate the natural heterogeneity present in most peer-to-peer systems. We test our design through simulations and the results show three to five orders of magnitude improvement in total system capacity. We also report on a prototype implementation and its deployment on a testbed.

Napster（半集中式网络）开创了对等文件共享的理念, 并通过一个集中的文件搜索工具支持它。随后的 P2P 系统, 如 Gnutella 采用分散搜索算法。然而, Gnutella 的糟糕的结垢导致一些人提出分布式哈希表解决方案, 以解决广泛的区域文件搜索问题。

与这一趋势相反, 我们主张保留 Gnutella 的简单性, 同时提出新的机制, 大大提高其可扩展能力。在前面的研究的基础上, 我们建议对 Gnutella 的设计进行一些修改, 动态地调整叠加拓扑和搜索算法, 以适应大多数对等系统中存在的自然异质性。我们通过模拟测试我们的设计, 结果显示总系统容量的三到五级提高。我们还报告了原型实现及其在试验台上的部署。

2.DHT

分布式哈希表是一类最近开发的系统，它们在Internet规模上提供类似哈希表的语义。 DHT最初的基本原理（尽管不是全部）是为不可扩展的Gnutella文件共享系统提供可扩展的替代品。 在过去的几年中，该领域的研究活动风靡一时，有许多设计方案和建议的应用。 所有这些提议都使用结构化覆盖网络，其中数据放置和覆盖拓扑都受到严格控制。 DHT提供的类似哈希表的lookup（）操作通常仅需要O（log n）步骤，而相比之下，Gnutella需要O（n）步骤来可靠地定位特定文件。

鉴于DHT提供的这种性能提升水平，当DHT可用时，很自然会问为什么要打扰Gia。 为了回答这个问题，我们回顾了P2P文件共享的三个相关方面。

（1）P2P客户端非常短暂。Gnutella和Napster中的测量活动表明节点的中值正常运行时间为60分钟。对于大型系统，例如100,000个节点，这意味着每分钟来来往往超过1600个节点的流失率。对于使用非结构化覆盖网络的Gnutella和其他系统而言，流失几乎没有问题，只要对等体不会因丢失所有邻居而断开连接，即使在这种情况下，对等体也只能重复引导程序重新加入网络。相比之下，流失确实会导致DHT的严重开销。为了保持路由的效率和正确性，大多数DHT在每次故障后都需要O（logn）修复操作。如果节点在没有事先通知其邻居并转移相关状态的情况下发生故障，那么无格式故障需要更多时间并在DHT中工作以（a）发现故障并且（b）重新复制丢失的数据或指针。如果流失率过高，则由这些修复操作引起的开销可能变得很大，并且可能很容易使用低带宽拨号连接的节点淹没。

（2）关键字搜索比完全匹配查询更普遍，更重要。 DHT擅长支持精确匹配查找：给定文件的确切名称，它们将名称转换为密钥并执行相应的查找（密钥）操作。但是，DHT不擅长支持关键字搜索：给定一系列关键字，找到与它们匹配的文件。目前使用P2P文件共享系统，围绕共享音乐和视频，需要这样的关键字匹配。例如，为了找到麦当娜的歌曲“Ray of Light”，用户通常会提交“madonna light of light”形式的搜索，并期望文件共享系统找到与搜索查询中的所有关键字匹配的文件。这一点特别重要，因为P2P系统中的文件名没有明确的命名约定，因此通常相同的内容由不同的节点存储在几个（略有不同）的名字。

在DHT之上支持这种关键字搜索是一项非常重要的任务。例如，在频繁节点（因此文件）流失的情况下维护每个关键字构造倒排索引的典型方法可能是昂贵的。这只是通过额外的缓存算法进一步复杂化，以避免重载存储流行关键字索引的节点。这些问题中的一些可能在DHT中可以解决，如部署Overnet文件共享应用程序所示，基于Kademlia DHT。尽管如此，基于DHT的解决方案通常需要不遗余力地将查询模型纳入简单的完全匹配搜索之外。相比之下，Gnutella和其他类似系统毫不费力地支持关键字搜索和其他复杂查询，因为所有这些搜索都是在逐个节点的基础上本地执行的。

（3）大多数查询是针对干草，而不是针。 DHT具有准确的召回，因为知道文件的名称允许您找到它，即使系统中只有该文件的单个副本。 相反，Gnutella无法可靠地找到文件的单个副本，除非泛洪查询到达所有节点; 我们把这些文件称为针。 但是，我们希望流行的P2P文件共享系统中的大多数查询都是针对相对良好复制的文件，我们称之为干草。 根据P2P文件共享的本质，如果频繁请求文件，那么随着越来越多的请求者将文件下载到他们的机器，系统中将有许多副本。 我们称之为系统，其中大多数查询都是针对复制内容的大众市场文件共享系统。

Gnutella可以很容易地找到很好的复制文件。因此，如果大多数搜索都是针对干草而不是针头，那么Gnutella缺乏准确的召回并不是一个明显的缺点。为了验证我们猜测大多数查询确实是干草，我们使用仪表化的Gnutella客户端收集了查询和下载请求的痕迹。我们的跟踪工具抓取了Gnutella网络，搜索与前50个查询请求相匹配的文件。收集文件名和每个文件的可用副本数后，该工具转过来并提供相同的文件以便下载到其他Gnutella客户端。然后，我们测量了跟踪工具为此提供的内容看到的下载请求数。图1显示了下载请求的分布与可用副本的数量。我们注意到大多数请求对应于具有大量可用副本的文件。 5例如，一半的请求是针对具有超过100个副本的文件，而大约80％的请求是针对具有超过80个副本的文件的。

总之，面向瞬态的Gnutella式设计更加强大，并且支持一般搜索工具，这两者都是P2P文件共享的重要特性。他们在寻找针头时不如DHT熟练，但这可能并不重要，因为大多数P2P查询都是干草。因此，我们推测对于大众市场文件共享应用程序，提高非结构化P2P系统的可扩展性，而不是转向DHT基于系统，可能是更好的方法。

3.GIA设计

类似Gnutella的系统有一个基本问题：当面对高聚合查询率时，节点很快就会过载并且系统不能令人满意地运行。 而且，随着系统尺寸的增加，这个问题变得更糟。 我们设计Gia的第一个目标是创建一个类似Gnutella的P2P系统，它可以处理更高的聚合查询率。 我们的第二个目标是让Gia继续适应不断增加的系统规模。 为了实现这种可扩展性，Gia努力通过明确考虑其容量限制来避免任何节点过载。 在早期的研讨会论文中，我们提出了将能力意识纳入Gnutella的初步建议。 在我们目前的工作中，我们完善了这些想法，并提供了一个全面的设计，详细的算法和新系统的原型实现。 我们首先概述系统设计背后的原因，然后详细讨论各种组件和协议。

3.1设计原理

Gnutella协议使用基于泛洪的搜索方法在其P2P网络中查找文件。 为了定位文件，节点查询其每个邻居，然后将查询传播到其邻居，依此类推，直到查询到达原始查询器的特定半径内的所有客户端。 虽然这种方法可以定位文件，即使它们在极少数节点上被复制，但它有明显的扩展问题。 为了解决这个问题，Lvet al。 [12]建议用随机游走代替洪水。 随机漫游是一种众所周知的技术，其中在每个步骤将查询消息转发到随机选择的邻居，直到找到对查询的足够响应。 虽然它们比洪水更好地利用P2P网络，但它们有两个相关的问题：

（1）随机游走本质上是盲搜索，因为在每个步骤中，查询被转发到随机节点，而不考虑节点将对查询具有响应的可能性的任何指示。

（2）如果随机漫步者查询到达已经过载流量的节点，则它可能会在处理之前排队很长时间

Adamic et提出了第一个问题，建议不要使用纯随机游走，而是搜索协议应该偏向高度节点。这背后的直觉是，如果我们安排邻居知道彼此的共享文件，高度节点将拥有（指向）大量文件，因此更有可能得到与查询匹配的答案。但是，这种方法忽略了节点过载的问题。实际上，通过始终将随机游走偏向高度节点，如果高度节点没有处理大量查询的能力，则会加剧问题。

另一方面，Gia的设计明确地考虑了与P2P网络中的每个节点相关联的容量约束。节点的容量取决于许多因素，包括其处理能力，磁盘延迟和访问带宽。有充分证据表明，像Gnutella这样的网络中的节点在处理查询的能力方面表现出显着的异质性[22]。然而，之前关于缩放Gnutella类系统的工作都没有利用这种异质性。在Gia的设计中，我们明确地适应（甚至利用）异质性以实现更好的扩展。我们设计的四个关键组成部分总结如下：

（1）一种动态拓扑自适应协议，可将大多数节点置于高容量节点的短距离内。 自适应协议确保接收大部分查询的连接良好的（即，高度）节点实际上具有处理那些查询的能力。

（2）一种主动流量控制方案，以避免过载的热点。流量控制协议明确承认存在异质性，并根据可用容量将流量控制令牌分配给节点进行适应。

（3）指向内容的指针的一跳复制。 所有节点都保持指向其直接邻居提供的内容的指针。 由于拓扑自适应算法确保了高容量节点和高度节点之间的一致性，因此单跳复制保证了高容量节点能够提供对更多数量的查询的答案。

（4）一种基于偏向随机漫游的搜索协议，它将查询引向高容量节点，这些节点通常能够最好地回答查询。

3.1.1拓扑自适应算法

拓扑自适应算法是将Gia客户端连接到网络其余部分的核心组件。在本节中，我们提供了适应过程的概述，同时保留了第5节中稍后讨论的一些特定机制的细节。当节点启动时，它使用类似于Gnutella中的引导机制来定位其他Gia节点。每个Gia客户端都维护一个主机缓存，其中包含其他Gia节点列表（其IP地址，端口号和容量）。主机缓存在客户端的整个生命周期中使用各种会合机制进行填充，包括联系众所周知的基于Web的主机缓存[5]并通过PING-PONG消息与邻居交换主机信息[6]。如果与这些主机的连接失败，则主机缓存中的条目将标记为已死。死亡的条目会定期老化。

拓扑自适应算法的目标是确保高容量节点确实是高度节点并且低容量节点在较高容量节点的短距离内。为了实现这一目标，每个节点独立地计算满意度（S） ）。这是介于0和1之间的数量，表示节点对其当前邻居集的满意程度。值S = 0表示节点非常不满意，而S = 1表示节点完全满足。只要节点不完全满足，拓扑适配就继续搜索适当的邻居以提高满意度。因此，当节点启动并且具有少于一些预配置的最小邻居数时，它处于不满意状态（S = 0）。随着它收集更多的邻居，它的满意度上升，直到它决定其当前的邻居集合足以满足其容量，此时拓扑适应变为静止。在5.2节中，我们描述了用于计算满意度的算法的细节。

在握手期间，每个节点基于其现有邻居和新节点的容量和程度来决定是否将另一节点接受为新邻居。为了接受新节点，我们可能需要删除现有的邻居。算法1显示了进行此确定所涉及的步骤。算法的工作原理如下。如果在接受新连接时，邻居总数仍然在预先配置的绑定最大值范围内，则自动接受连接。否则，节点必须查看是否可以找到适当的现有邻居来删除并替换为新连接。

如果Y的容量高于X的所有当前邻居，则X总是偏爱Y并丢弃现有邻居。否则，它决定是否保留Y，如下所示。从容量小于或等于Y的所有X的邻居中，我们选择具有最高度的邻居Z.如果X将其降低而偏向Y，则该邻居损失最小。仅当新节点Y具有比Z更少的邻居时才会丢弃邻居。这确保我们不会丢弃已经连接不良的邻居（可能断开连接）以支持连接良好的邻居。因此，拓扑自适应算法试图确保自适应过程向稳定状态前进。测量拓扑自适应过程的实验结果将在后面的5.4节中讨论。

3.1.2流量控制

为了避免创建热点或使任何一个节点过载，Gia使用主动流控制方案，其中只有当该邻居已通知发送方愿意接受来自发送方的查询时，才允许发送方将查询定向到邻居。这与大多数提出的Gnutella流量控制机制[16]形成对比，后者本质上是反应性的：接收器在开始变得过载时丢弃数据包;发送者可以推断出邻居根据他们从邻居收到的响应来发送数据包的可能性，但是没有明确的反馈机制。当查询在网络中泛滥时，这些技术可能是可接受的，因为即使节点丢弃查询，查询的其他副本也将通过网络传播。但是，Gia使用随机漫游（以解决泛洪的缩放问题）来转发每个查询的单个副本。因此，任意删除查询不是一个合适的解决方案。

为了提供更好的流量控制，每个Gia客户端定期向其邻居分配流量控制令牌。每个标记代表节点愿意接受的单个查询。因此，仅当节点已从该邻居接收到令牌时，节点才能向邻居发送查询，从而避免过载的邻居。在聚合中，节点以其可以处理查询的速率分配令牌。如果它接收查询的速度快于它可以转发它们（因为它过载或者因为它没有从其邻居收到足够的令牌），那么它开始排队多余的查询。如果此队列太长，它会尝试通过降低其令牌分配率来减少查询的流入。

为了激励大容量节点宣传其真实容量，Gia客户端根据邻居的容量分配令牌，而不是在所有邻居之间均匀分配令牌。 因此，广告处理传入查询的高容量的节点又为其自己的传出查询分配了更多令牌。 我们使用基于开始时间公平排队（SFQ）的令牌分配算法。 为每个邻居分配一个等于其容量的公平排队权重。 未使用任何已分配令牌的邻居被标记为不活动，并且剩余容量将在剩余邻居之间按比例自动重新分配。 当邻居加入和离开时，SFQ算法相应地重新计算其令牌分配。 令牌分配通知可以作为单独的控制消息发送给邻居，也可以通过捎带其他消息发送给邻居。

3.1.3 一跳复制

为了提高搜索过程的效率，每个Gia节点主动维护其每个邻居的内容的索引。当邻居彼此建立连接时，这些索引被交换，并且随着增量变化周期性地更新。因此，当节点接收到 查询，它不仅可以响应来自其内容的匹配，还可以提供来自其所有邻居提供的内容的匹配。 当邻居丢失时，无论是因为它离开了系统，还是由于拓扑适应，该邻居的索引信息都会被刷新。 这可确保所有索引信息在节点的整个生命周期内保持最新且一致。

3.1.4搜索协议

拓扑自适应（其中高容量节点具有更多邻居）和单跳复制（由此节点保持其邻居的共享文件的索引）的组合确保高容量节点通常可以为大量查询提供有用的响应。因此，Gia搜索协议使用偏向随机游走：Gia节点不是将传入查询转发到随机选择的邻居，而是选择具有流量控制令牌的最高容量邻居，并将查询发送给该邻居。如果它没有来自任何邻居的令牌，它会对查询进行排队，直到新令牌到达为止。

我们使用TTL来限制有偏差的随机游走和簿记技术的持续时间，以避免冗余路径。通过簿记，每个查询由其发起者节点分配唯一的GUID。节点会记住它已经转发了对给定GUID的查询的邻居。如果具有相同GUID的查询返回到节点，则将其转发到其他邻居。这降低了查询两次遍历同一路径的可能性。为了确保前进，如果节点已将查询发送到其所有邻居，则会刷新簿记状态并开始重新使用邻居。

每个查询都有一个MAX RESPONSES参数，即查询应搜索的最大匹配答案数。除了TTL之外，查询持续时间受MAX RESPONSES限制。每次节点找到查询的匹配响应时，它会减少查询中的MAX RESPONSES。一旦MAX RESPONSES达到零，则丢弃查询。查询响应沿着与查询相关联的反向路径被转发回发起者。如果由于拓扑自适应而导致反向路径丢失，或者由于节点故障而导致查询或响应被丢弃，我们将依赖第5.3节中稍后描述的恢复机制来处理丢失。

最后，由于节点可以为其自己的文件或其邻居之一的文件生成响应，因此我们将转发的查询附加到拥有这些文件的节点的地址。这确保了查询不会为同一文件实例产生多个冗余响应;仅当拥有匹配文件的节点尚未在查询消息中列出时，才会生成响应。

4.

我们首先描述我们的仿真模型和用于评估算法性能的指标。 然后我们报告一系列模拟的结果。 我们的实验侧重于在各种条件下处理查询的能力方面的聚合系统行为。 我们展示了我们系统的各个组件（拓扑适应，流量控制，单跳复制和基于偏向随机游走的搜索）以及它们之间的协同作用如何影响总系统容量。 由于篇幅限制，我们不会提供评估每个设计组件内权衡的详细结果。

4.1系统模型

为了捕获查询负载对系统的影响，Gia模拟器对系统中的每个节点施加容量限制。我们将每个节点i建模为具有容量C i，其表示它可以每单位时间处理的消息的数量（诸如用于拓扑适配的查询和添加/删除请求）。如果节点以高于其容量C i的速率从其邻居接收查询（如在没有流控制的情况下可能发生），那么将多余的查询建模为在连接缓冲区中排队，直到接收节点可以从那些缓冲区。

对于我们的大多数模拟，我们根据Saroiu等人报告的Gnutella的测量带宽分布得出的分布为节点分配容量。我们的容量分配有五个级别的容量，每个级别相隔一个数量级，如表1所示。如[22]中所述，这种分布反映了Gnutella客户中有相当一部分具有拨号连接到Internet的现实，大多数通过电缆调制解调器或DSL连接，少数参与者具有高速连接。对于SUPER实验，容量为1000x和10000x的节点被指定为超级节点。

除了其容量之外，每个节点i被分配查询生成速率q i，其是节点i每单位时间生成的查询的数量。 对于我们的实验，我们假设所有节点以相同的速率生成查询（当然，它们的容量是有界的）。 当需要缓冲查询时，它们将保持在队列中。 我们将所有传入和传出队列建模为具有无限长度。 我们意识到，在实践中，队列不是无限的，但我们做出这个假设，因为删除查询和将其添加到任意长队列的效果基本相同。

查询被建模为搜索特定关键字。每个关键字都映射到一组文件。文件随机复制到节点上。与特定关键字关联的所有文件都是具有该关键字的查询的潜在答案。我们使用术语复制因子来指代查询答案所在的节点部分。因此，对复制因子为1％的关键字执行查询意味着可以在系统中1％的节点处找到对此查询的答案。在已部署的系统中，实际搜索流量将包括同时涵盖一系列复制因素不同的多个查询。但是，每个搜索过程都在很大程度上独立进行（除了队列内的延迟和流量控制的动作）。因此，我们不是必须选择特定的查询分布，每个查询都使用自己的复制因子查找关键字，而是专注于所有具有特定复制因子的查询流，并研究在更改复制因子时我们的结果如何变化。

我们使用随机连接的拓扑开始我们的模拟。GIA仿真使用拓扑自适应来重新配置此初始拓扑。算法使用两个预先配置的参数：min nbrs和max nbrs。对于所有的实验，我们使用min -brs = 3.我们设置max nbrsto 128.但是，max nbrs还有一个额外的约束。为了避免中等容量节点或低容量节点聚集这么多邻居以使其容量过于细分，我们要求C numnbrs≥一些min alloc，其中min alloc表示我们愿意分割的最细粒度级别。节点的容量。有了这个附加约束，我们注意到对于每个节点，max nbrs = min（max nbrs，？C min alloc？）。在针对不同的min alloc值测试GIA拓扑自适应性能的一些初步模拟之后，我们确定了min alloc = 4. 由拓扑自适应和其他组件生成的所有控制流量被建模为消耗资源：每个消息一个容量单位。 因此，模拟器间接地捕获控制流量对系统整体性能的影响。

对于RWRT和FLOOD，没有拓扑适配;我们使用随机图。事实上，我们知道Gnutella网络具有类似于幂律图的特性[20]。但是，无法保证偏斜的Gnutella分布中的高度节点也是高容量节点。实际上，在没有高容量和高度的明确一致性的情况下，随机游走将导致高度节点过载。将这种拓扑上的随机游走与GIA进行比较会不公平地将结果偏向随机游走。因此，对于RWRT，我们选择使用具有均匀度分布的纯随机拓扑，这减轻了这个问题。这种均匀随机图上的RWRT性能与各个节点的度数无关. 将以相同的概率访问所有节点。另一方面，FLOOD的表现确实取决于度数，实际上会更高程度地恶化。对于我们的实验，我们因此选择均匀随机图，平均度为8。这种选择是临时性的，但反映了避免不必要地偏向RWRT和FLOOD的决定。

平均而言，随机图的直径为7.因此，对于FLOOD和SUPER，我们将查询的TTL设置为10，以确保查询不会受到人为限制。对于RWRT和GIA，TTL设置为较大的值（1024），但在这种情况下设置正确的TTL值并不重要，因为随机行走终止时

他们找到了所需数量的回复。

尽管模拟器模拟了第3节中讨论的各种协议的行为，但它不会捕获单个数据包级别的行为，也不会解释由后台流量引起的网络行为中的任何变幻莫测。我们这样做是因为我们的观点不是要在现实世界中量化算法的绝对性能，而是要评估各种设计选择的相对性能。在第5.4节中，我们提供了一些初步结果，报告了我们在广域互联网中实施和部署Gia的经验。

4.2性能指标

为了测量负载对系统的影响，我们研究了系统性能的三个方面作为提供负载的函数：成功率测量为成功定位所需文件的查询分数，跳数计量为数字查找所请求文件所需的跃点数，以及查询从开始到结束所用时间的延迟。图2显示了运行Gia系统的10,000节点网络在增加查询负载下的成功率，跳数和延迟。对于这些图，就像在我们的其余模拟中一样，当我们提到一个0.1的查询负载时，我们的意思是系统中的每个节点每单位时间发出0.1个查询（当然，这取决于节点的容量）。如图中的每个图表所示，当查询负荷增加时，我们注意到曲线中有一个尖锐的“拐点”，超过该拐点，成功率急剧下降，延迟迅速增加。跳数保持稳定直到拐点和然后减少。这种减少的原因是只对成功的查询测量跳数;在增加的负载下，成功的查询往往是所请求的文件位于距查询的发起者几跳之内的那些查询。这些图描绘了GIA模型中膝盖的存在;我们使用RWRT，FLOOD和SUPER对一系列复制因子进行的模拟显示了相同的行为，尽管在不同的查询负载下。

理想情况下，我们希望系统能够在保持低跳数和延迟的同时实现高成功率。 为此，系统必须在上图所示的膝盖之前操作。 因此，我们定义以下指标用于评估：

4.3性能对比

我们比较了GIA，RWRT，SUPER和FLOOD在不同复制因子和不同系统大小下的行为，最多10,000个节点。我们在增加的复制因子下测量了CP和CP-HC。在图3和图4中，我们概括了具有5,000和10,000个节点的系统的结果。具有额外系统尺寸的实验产生的结果与此处所示的结果一致;为清楚起见，我们在图表中省略了它们。对于10,000节点系统，我们模拟低至0.01％的复制，因为这对应于整个系统中针对任何查询的单个匹配答案。同样，对于5,000个节点，我们模拟低至0.05％的复制。我们认为复制因子为0.01％，其中只有10,000个节点中的一个拥有查询的答案，这代表了一种相当悲观的测试场景。这些实验中的每个查询都会运行，直到找到一个匹配答案。这表示查询发起者将查询中的MAX RESPONSES参数（参见第3.2.4节）设置为1的情况。实际上，大多数用户希望查询返回多个答案;我们稍后会看一下那个场景。对于GIA和RWRT，我们测量所有查询的平均跳数。因为对于SUPER和FLOOD，查询在每一跳都被复制，所以很难为整个查询定义一致的跳数;因此，我们将跳数计算为找到第一个答案所花费的跳数。

回想一下，我们设计Gia的第一个目标是使其能够处理比Gnutella更高的聚合查询率。图3和图4中最明显，最重要的观察结果是总体系统容量（由崩溃点定义）比FLOOD或RWRT高3到5个数量级。即使与超级节点方法相比，Gia也是如此。特别是在更高的复制率下表现更好。这并不奇怪，因为超级节点中使用的泛洪技术限制了它们的可扩展性。因此，我们明确实现了通过Gia提高系统容量的目标。我们的第二个目标是Gia保留了处理任意大小系统的高聚合查询率的能力。从图中可以看出，这一目标也得到了满足。 GIA（和RWRT）的缩放行为由复制因子决定。也就是说，在固定的复制因子下，CP和CP-HC在很大程度上不受系统大小的影响。这是预期的，因为复制因子是答案所在节点的百分比。因此，我们在这里展示的性能数据适用于任意大的系统尺寸。

这些实验清楚地证明了GIA相对于RWRT，SUPER和FLOOD的可扩展性。 但是，这些实验仅限于在找到单个匹配答案后搜索终止的查询。 实际上，大多数用户希望查询返回多个答案。 我们现在看一下（a）我们的结果如何概括超出这个案例，以及（b）不同的设计组件如何为GIA的巨大性能提升做出贡献

4.4多重搜索响应

在本节中，我们将根据查询所需的响应数量，查看不同系统模型的折叠点（以及相关的跳数）的变化情况。 回想一下3.2.4节，查询包含一个MAX RESPONSES参数，该参数指示在结束查询之前应该将多少响应发送回查询的发起者。 MAX RESPONSES参数仅在GIA和RWRT的上下文中有用。 对于FLOOD和SUPER，查询会通过网络泛滥，因此MAX -RESPONSES对其行为没有影响。

表2显示了复制因子为1％的10,000节点系统的所有四种系统模型的CP。对于RWRT和GIA，MAX RESPONSES的较高值意味着查询需要在结束之前更长时间地搜索网络。这样可以为每个查询提供更高的有效跳数，从而使每个查询都可以利用更多的可用系统容量。如表中的CP值所示，这有效地降低了整体系统容量。正如预期的那样，不同的MAX RESPONSES对SUPER和FLOOD模型没有影响。

如前面图3所示，崩溃点也取决于复制因子。当文件以较少的节点复制时，查询必须平均访问更多节点才能找到它们。结果，崩溃点随着复制因子的减少而下降。实际上，我们发现在r的复制因子下对k MAX RESPONSES的查询的性能等于在相应较低的r / k复制因子下对单个响应的查询的性能。这在表3中描述。对于所有四种系统模型，以1.0％的复制因子搜索10个答案产生的CP几乎与通过以0.1％的复制因子搜索单个答案而获得的CP相同。同样，在1％复制时搜索20个答案会产生与在0.05％复制时搜索单个答案相同的CP。

鉴于此结果，我们对其余的GIA和RWRT模拟进行了建模，以便在查找第一个查询答案后终止搜索。这并不会改变我们的结果的性质，但使分析系统变得更简单，并且足以揭示各种设计之间的显着差异。

4.5 因素分析

我们在4.3节中的结果表明，就可以成功维持的查询负载而言，GIA的性能优于RWRT，SUPER和FLOOD几个数量级。我们现在将注意力转向研究GIA的各个组件（拓扑自适应，流量控制，单跳复制和偏置随机游走）如何影响性能增益。许多研究人员提出了使用一种或多种GIA组件来改善Gnutella可扩展性的方案。 GIA与大多数其他方案的区别在于将所有组件组合成一个全面的系统设计，与以前的工作不同，它使每个组件适应“容量敏感”。

在本节中，我们展示它不是任何单个组件，但事实上，它们的组合都为GIA提供了这一巨大的性能优势。我们展示了GIA的每个设计组件对其性能至关重要，然而，在RWRT中添加任何单个GIA组件并未显着缩小GIA和RWRT之间的性能差距。我们不考虑FLOOD，因为从FLOOD到GIA的主要设计飞跃是从使用洪水到使用随机游走的过渡，其效果已经被基本的RWRT捕获。同样，SUPER只是迈向GIA设计的一步，其中包括一些单跳复制和节点异构性的即席意识。在这里，我们在检查GIA的性能时，一次一个地删除它的四个设计组件，并将它与RWRT的行为进行比较，如果我们一次一个地添加这些设计组件的话。

表4显示了在0.1％复制时10,000个节点的因子分析结果。乍一看，可以得出结论，GIA从单跳复制中获得了大部分性能提升，因为从GIA中删除单跳复制会严重影响其性能。但是，向RWRT添加单跳复制只能将CP提高一个数量级，而GIA作为一个整体提供的CP比RWRT高出四个数量级。除了单跳复制之外，它还是拓扑自适应，偏置随机游走和流量控制的组合，使GIA在RWRT上获得了巨大的性能提升。

偏向随机游走似乎对GIA的表现没什么影响。这是因为在高查询负载（即，接近CP）时，流控制组件用于将负载转向任何可用容量（通常在高容量节点中），因此功能类似于偏置行走。但是，在较低的查询负载下，当所有节点都轻载时，偏向的行走有助于将查询快速定向到高容量节点。

4.6异质性影响

由于GIA明确设计为对节点容量敏感，因此我们现在研究异构性对系统性能的影响。 表5比较了GIA和RWRT的性能与从Gnutella类容量分布中得出的节点容量，以及所有节点具有与Gnutella分布的平均节点容量相同的容量的情况。 当节点具有异构容量时，GIA中的CP会得到改善。 相比之下，我们看到RWRT不能容忍异质性，相对于统一容量情况，CP下降超过两个数量级。虽然两种情况下CPR HC对RWRT保持不变（如人们所预料的），但是GIA的跳数下降了，由于有偏差的随机游走开始将查询引向高容量节点。

4.7鲁棒性

到目前为止，我们的结果表明Gia的性能明显优于以前的非结构化P2P文件共享系统。在本节中，我们展示了Gia可以在节点故障面前维持这种性能。

节点故障模型。我们通过为每个节点分配从[0，MAXLIFE-TIME]随机均匀选择的正常运行时间来模拟节点故障，其中MAXLIFETIME是模拟参数。当节点的正常运行时间到期时，节点将重置。也就是说，它与其邻居断开连接，关闭，并通过最初连接到随机数量的邻居立即重新加入系统。这类似于建模现有节点关闭和离开系统，而其他新节点同时加入系统。当节点关闭时，它在本地排队的任何查询都会被最初生成它们的节点删除并恢复。最后，当节点加入和离开系统时，拓扑自适应开销由每个节点的自适应操作消耗节点内的容量的事实捕获。

图5和图6分别绘制了在MAXLIFETIME增加的情况下10,000节点Gia系统的CP和CP-HC。我们看到，相对于静态情况，当MAXLIFETIME减少到10.0个时间单位时，CP下降大约一个数量级，而跳数增加大约五倍。请注意，在10个时间单位的MAXLIFETIME中，大约20％的节点在每个时间单位复位。即使在这种极度紧张的测试中，GIA的性能也只下降了不到一个数量级。在静态条件下，这仍然比RWRT，SUPER和FLOOD提高了2-4个数量级。

4.8 文件下载

上面的结果表明，Gia可以支持比以前提出的分布式文件搜索方法更高的查询负载，并且即使面对高节点流失也可以保持这种性能优势。 Gia卓越的性能改进源于其独特的设计组件组合以及将工作汇集到系统中的高容量节点的能力。

因此，我们的结果使我们得出结论，分散式P2P系统中的搜索不再需要构成难以克服的扩展问题。如果是这样，我们推测限制可伸缩性的下一个瓶颈可能是文件下载过程。如果最近的测量研究表明文件大小继续增加，那将尤其如此。我们相信Gia能够以对各个节点的约束敏感的方式利用容量，这也会对下载产生有益的影响。即便如此，Gia也会帮助用户下载，如果存在文件，则通常会将用户定向到文件的高容量副本。但是，除非高容量节点还存储更多文件，否则此优势不太可能显着。因此，为了让Gia能够协助文件下载，我们必须扩展Gia中使用的一跳复制，以允许文件本身的主动复制（而不仅仅是指向文件的指针）。一种简单形式的主动复制将用于过载的低容量节点，以在其一跳邻域中的较高容量节点处复制流行文件。这可以按需方式完成，其中高容量节点仅在接收到查询和对该内容的相应下载请求时才复制内容。

为了衡量这种主动复制可能有用的程度，我们对一个给定文件可用的所有节点的总容量进行了简单计算，无论是否有这种主动复制方案。结果数字列在表6中。我们看到主动复制将服务于给定文件的节点的总容量增加了38到50倍。这看起来很有希望，尽管需要更多的分析和模拟来验证这种方法的有用性。

5.

**5.1容量设置**

在我们到目前为止的讨论中，我们假设节点的容量是一个数量，表示节点每秒可以处理的查询数。对于低带宽客户端，查询处理容量受客户端访问带宽的限制。另一方面，对于具有高速访问连接的节点，其他问题（例如CPU的速度，磁盘延迟等）可能会影响容量。我们的原型实现忽略了CPU速度和磁盘延迟对查询容量的影响。我们假设容量是接入带宽的直接函数。节点可以让其最终用户配置其访问带宽（通过用户界面，如在许多Gnutella客户端中所做的那样），或者通过执行从众所周知的站点下载大块数据的配置脚本来自动确定访问带宽在互联网周围，根据下载的平均时间来衡量带宽。此外，节点的广告容量可以根据节点在系统中的时间加权。这确保了网络中连接良好的高容量核心主要由稳定节点组成。在未来的实施中，我们计划尝试使用自动配置脚本，该脚本除了考虑网络带宽和节点生命周期之外还考虑其他因素，以确定客户端容量。

**5.2满意度**

在3.2.1节中，我们介绍了客户满意度的概念。满意度不仅决定是否执行拓扑适配，还决定它应该执行的频率。它是预先配置的min nbrs和max nbrs参数，节点的当前邻居集，它们的容量和度数的函数。邻居在最初相互连接时交换容量信息，并定期使用有关其当前程度的信息相互更新。算法2显示了计算满意度（）所涉及的步骤。它实质上是衡量节点所有邻居的容量之和（按度数标准化）与节点自身容量的接近程度。因此，具有较低程度的高容量邻居比具有相同容量但程度更高的另一邻居更有助于我们的满意度。这背后的直觉是，具有容量C的节点将在满负荷时每单位时间转发大约C个查询，并且需要来自其所有邻居的足够的输出容量来处理该负载。除了上面讨论的因素之外，还可以使用许多其他参数来计算满意度，例如，节点邻居的负载，网络位置等。但是，对于我们的原型，我们仅依赖于节点容量和计算满意度的程度。

满意度是决定节点进行本地拓扑适配的频率的关键。具有低满意度的节点比满意的节点更频繁地执行拓扑适配。我们使用满意度S和适应间隔之间的指数关系，I：I = T×K - （1-S），其中T是自适应迭代之间的最大间隔，K表示自适应的攻击性。在每个间隔I之后，如果节点的满意度等级<1.0，则它尝试添加新邻居。一旦节点完全满足，它仍然继续迭代适应过程，每T秒检查其满意度。

图7显示了攻击性因子K如何影响适应间隔。如所预期的，当节点完全满足（S = 1.0）时，无论K的值如何，适应间隔都是T.随着满足度降低，适应间隔变短。对于相同的满意度水平，较高的K值产生较短的间隔，因此引起更积极（即，更快）的响应。在5.4节中，我们将看看拓扑自适应速率如何在具有不同K值的实际系统中发生变化。

**5.3查询弹性**

如前所述，Gia搜索协议使用偏向随机漫游来跨网络转发查询。使用随机游走而不是泛滥的一个缺点是它更容易受到网络故障的影响。如果一个节点收到一个查询并在它可以将查询转发给邻居之前死掉，那么该查询将永远丢失。这与查询被多次复制的泛洪形成对比，因此即使节点在没有转发查询的情况下死亡，查询的其他副本也很可能仍存在于系统中。

为了克服这个问题，我们依赖于查询保持活动消息。发送回查询发起者的查询响应充当隐式保持活动。此外，如果查询转发的次数不足而没有发送任何响应，我们会发回一条显式的保持活动消息。这是作为没有实际匹配的虚拟查询响应消息实现的。如果查询发起者暂时没有收到任何响应或保持活动消息，它可以重新发出查询。

另一个问题是，响应通常沿着查询最初到达的相同路径转发。如果查询的响应被发回之前，如果查询路径中的一个节点死亡或拓扑由于适应而发生变化，那么这些响应将被丢弃。当节点死亡并导致响应被丢弃时，查询发起者会注意到它，因为没有保持活动响应。因此，如有必要，它可以重新发出查询。另一方面，拓扑自适应是一个受控制的过程，因此我们可以做得比等待超时更好。当由于拓扑自适应决定而断开连接时，连接在一段时间后不会关闭。它停止接受任何更多传入的查询，但继续转发需要沿该路径发送的任何延迟查询响应。

**5.4部署实施**

我们在PlanetLab [17]上部署了我们的原型实现，这是一个遍布北美，欧洲，亚洲和南太平洋的广域服务部署测试平台。为了测试我们的算法在多样性面前的行为，我们通过明确设置各个节点的容量，人为地将异质性引入系统。这些实验决不是要强调Gia系统的所有各种组件。我们在这里将它们作为一组早期结果呈现，以证明该方法在实际部署中的可行性。

我们在83个PlanetLab节点上实例化了Gia客户端，并提供了一系列容量。我们允许系统运行15分钟，然后关闭客户端。在实验过程中，我们跟踪Gia拓扑中的变化，以评估拓扑自适应过程的行为。图8显示了四个不同节点中每个节点的邻域随时间的变化。从四个容量类别（1x，10x，100x和1000x）中随机挑选这些节点。我们注意到，最初当节点都“不满意”时，它们会迅速聚集新的邻居。随着节点的满意度上升并且拓扑结构最终稳定到其稳定状态，拓扑适应速率减慢。

在上述实验中，1000x容量节点需要大约45秒才能达到其稳定状态。该时间间隔与拓扑自适应中使用的攻击性水平密切相关。回想一下，自适应间隔I是节点满意度S的函数，其侵略性因子K：I = T \* K - （1-S）。在上面的实验中，我们将T设置为10秒，K设置为256.我们进行了其他实验，以了解拓扑自适应的响应性如何随K的不同值而变化。图9显示了1000x容量节点对不同值的行为。 K.从图中可以看出，拓扑自适应确实响应了K的变化，并且在我们减小K的值时不那么激进。因此，这个参数给了我们一个旋钮，我们可以用它来折衷速度哪些节点达到其目标拓扑到整个网络中的流失率。