摘 要

Peer-to-Peer（简称为 P2P）计算模式现在得到越来越广泛的应用，针对不同用途的 P2P 应用正在迅速流行。然而所有这些应用都是独立开发，缺乏统一的通信协议和体系结构，这就制约了新的应用的开发和应用间的互操作。

为了简化 P2P 应用的开发，促进 P2P 计算的发展，我们开发了 P2P 网络基础设施平台。我们通过将 P2P 应用共同需要的功能抽象出来，为这些应用提供统一和简单的调用接口，使应用开发者能够专注于应用本身的设计。同时，统一的传输机制和 XML 消息的使用极大地方便了不同应用间交换信息。

本文首先介绍了 P2P 平台的体系、控制核心，服务框架。提出了 P2P 平台的基本结构。

接下来详细描述了平台各基本服务结构、设计和实现，涉及 XML 消息解析、独立于网络环境的通信体系以及分布式查找协议。

最后给出了利用这个平台实现的 P2P 文件共享应用实例，展现了基于这个平台构建 P2P 应用的便捷性和优势。

关键词： **对等网络、对等实体、XML 消息、协议绑定**

Abstract

The Peer-to-Peer computing mode has gained ground, with which many P2P applications for various scenes are booming. The applications are developed individually, however, without unified interfaces and conforming transport protocols, which handicap the development of P2P applications and break the inter-operability among the P2P applications.

To simplify constructing a P2P application and drive the P2P computing mode, we implement a P2P infrastructure platform. By abstracting common modules from the P2P applications, we provide the developers a simple and unified invoking interface, so that they can fix their mind on the business logic of the application. Furthermore, the transport protocol based on XML messages facilitates information exchange among the applications.

In the paper, I firstly introduce the P2P platform's architecture, control kernel and service's frame, which represent the basic structure of the platform.

In the following chapters, I discuss detailed the basic services' structures, design and implementation, including XML messages, transport architecture independent of specific network protocol, and distributed lookup protocol.

Finally, I use the platform to construct a simple P2P files sharing demo, by which I show the convenience of developing a P2P application based on the platform and the advantages of it.

Keywords: Peer-to-Peer network, peer, XML message, protocol binding

第一章 引言

1.1 Peer-to-peer—全新的计算模式

现在代表一种新的 Internet 计算模式的革新正在发生，它极大地改变了现有的计算方式和软件运行模式。

这场革新随着 Peer-to-peer（以下简称为 P2P）计算的出现而引发，由于 Napster、Scour、Freenet Project，Gnutella 和 SETI@home 的成功，P2P 计算也越来越引起公众的主意。（Peer—对等实体， 参与 P2P 应用的各种网络结点。）

P2P 计算可以简单的定义为通过直接交换信息来共享计算资源和服务。这种计算模式得到主流计算机用户和 PC 业界成员的推动。

- 1999 年九月出现的 Napster MP3 音乐文件共享应用到 2000 年中用户已经超过 2 千万。
- 截至 2001 年初，用于分布处理射电望远镜数据的 SETI@home 程序已经吸引了超过 260 万的用户，这些用户总共贡献了超过 50 万年的 CPU 时间用来寻找地外文明。
- 2000 年十月，250 家公司和机构的 350 多位代表召开了第一届 P2P Working Group 会议。

P2P 计算提供传统 Client/Server 体系的替代技术。在部署现有的网络、服务和客户基础设施的同时，P2P 提供了不同于 Client/Server 的计算模式。这两种模式可以共存、交叉和互相补充。

在 Client/Server 模式中，客户向网络中的服务器提出请求，服务器负责处理和响应请求。在 P2P 计算中，每个参与的对等实体作为客户的同时还具有服务层功能，这就使每个对等实体在特定的应用环境下既作为客户也作为服务器。P2P 应用可以实现如下功能：分布存储、分布式计算、消息服务、安全和文件分布共享，这些功能都通过对等实体间直接交换信息实现。

P2P 网络中的每个对等实体都可以发出请求，也可以响应来自其他对等实体的请求。这种与其他用户直接交换信息的能力使 P2P 用户从传统的对中心服务器的依赖中解放出来。用户可以有更高级别的自主权和对服务的控制权。

P2P 计算的最大的好处在于团体组织，用户可以组织一个 ad hoc 组进行有效地和安全地请求处理、资源共享、协同工作和通信。随着 P2P 计算的发展，我们可以预计将来会出现各种各样的在线团体。

图 1-1 描述了 Client/Server 模式和 P2P 模式的不同，同时也描述了两种模式如何叠加和共存。在 Client/Server 模式中，每次交换信息和通信都通过中心服务器，并由中心服务器管理。在 P2P 计算模式中，对等实体直接进行通信和交换信息。一些 P2P 应用也许在某些时候使用服务器，但 P2P 计算的总体影响是将网络计算分散。

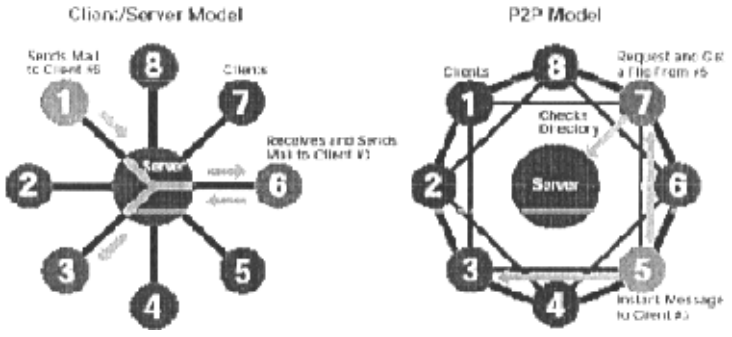

图 1-1：Client/Server 和 P2P 模式

1.2 P2P 应用领域及当前研究动态

P2P 目前在下列领域具有很好的应用前景：

- *基于 Web 的社团*。具有共同兴趣的小组通过 Web 站点创建自己的 Intranet。
- *电子商务*。P2P 可以连接供应链，提供更有效的分布信息、内容或软件，以及在原始结点上保存信息项目。
- *游戏*。P2P 基础设施为开发在线游戏提供基础并且不需要中心服务器。开发者可以把工作重心放在游戏特性上而不是通信协议接口。
- *搜索引擎*。通过在可能的地址上直接搜索取得最新的信息。
- *病毒防治*。P2P 社团结点之间允许协同进行病毒检测和预警，同时能够对进一步的攻击进行隔离。
- *分布服务*。例如，在线教学需要提供大量视频片断。如果大量的数据文件和使用者之间物理距离很近，应用就可以提供很好的服务质量。因此多个

客户提供存储空间比单个服务器有更好的灵活性和可靠性。

- *协同开发*。包括软件开发、协作文档处理和协同图像处理等多种应用。

虽然P2P计算的潜力很大而且业界的兴趣也很高，还是有几方面问题要解决。到目前为止，P2P 应用还不能容易地和其他 P2P 应用通信，每个应用有自己一套基本服务和插件。更不利的是新的应用开发者不能利用已有的 P2P 应用开发成果。结果应用开发者花费大量时间和精力开发别人已经成功开发过的服务。本来这些时间和精力可以用来为新服务增加更好的功能和特性。

这个问题可以通过业界合作使 P2P 应用之间可以互操作来解决。通过定义一组为 P2P 计算提供所需功能的公共服务实现互操作，这些公共服务可以作为中间件层。这样开发者不需要反复创建相同的基本服务。互操作意味着 P2P 应用可以在不同的软件环境下进行通信。使用不同编程语言的应用可以通过公共中间件通信和集成，并且公共中间件提供与其他非 PC 设备和服务器集成的机制和基础设施，非 PC 设备包括无线和手持式产品以及多种网络设备。

图 1-2 给出了 P2P 应用的层次结构，其中 P2P 中间件层位于对等实体本地操作系统和 P2P 应用调用的接口层之间。

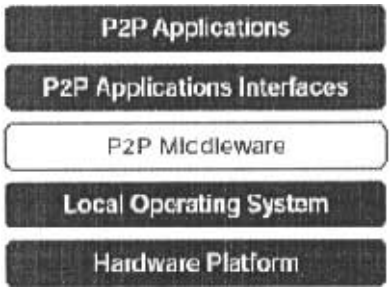

图 1-2：P2P 栈

不久前，Sun 微系统公司公开了旨在建立 P2P 通用技术基础的 JXTA 计划。“JXTA 技术是网络编程和计算的平台，用以解决现代分布计算尤其是 P2P 计算中出现的问题”。JXTA 研究项目提供给用户更便捷地访问连接在互联网上的个人电脑资源的新框架，进一步拓展互联网的空间。

JXTA 技术提供的基础机制试图解决当前分布计算应用中面临的问题，帮助实现新一代广泛(ubiquitous)、安全(secure)、可互操作(interoperable)以及异构(heterogeneous)的应用。目前它支持基于 Java 技术的平台和系统。而将来 JATX 技术将不受到内存的限制而支持更多小型移动设备。JXTA 通过 Java 技术和 XML

数据表达的结合，提供了强大的功能使得不同应用得以交互，并且可以克服目前 P2P 软件中的限制。同时，通过小型、简单、便于开发的构造模块，JXTA 将使开发者从建立各自框架的复杂工作中得以解放，可以潜心关注于构建各类新颖、具有创造性的分布式计算应用。

目前，国际上对 P2P 基础平台的研究正在积极展开，由于起步不久还没有形成固定的标准。虽然我国在这方面还缺乏相应的研究，但是构建 P2P 平台的基础技术和分布对象处理技术已经相当成熟，这给我们提供了迎头赶上的机会和技术基础。

我这次毕业设计的工作就是开发一个面向对象的 P2P 网络基础设施，完成核心资源和服务表示、基础协议以及计算环境的设计和实现。

1.3 P2P 基础平台的基本概念

1.3.1 什么是 P2P

P2P 网络中的结点不仅执行 Client/Server 模式中 Client 功能，它也包含额外的一层软件执行 Server 的功能，这样结点可以响应来自其他结点的请求。请求和响应的范围以及如何执行由具体应用指定。一般来说，请求是对属于其他结点的资源的访问，可以请求访问文件、执行计算或者传递消息。

P2P 计算中的计算作名词时，指的是计算模式或框架，提供结点直接交互的能力。直接交互能力的重要特征是计算环境分布化。

P2P 计算中的计算作动词时，指的是 P2P 框架做些什么。通过 P2P 服务，可以构造许多基于终端用户的应用，如存储、计算、消息服务、安全、分布文件共享等。这些应用的共同特点就是共享具有协作方式的资源。

一些 P2P 提倡者提出“纯 P2P 计算”和“混合 P2P 计算”的区别。“纯 P2P 计算”指所有参与的计算机都是对等实体的模式，例如 Freenet，不使用中心服务器进行控制、协同或管理对等实体间的信息交换。在“混合 P2P 计算”模式中，应用依赖中心服务器执行某些需要的功能。例如，Napster 要求用户首先要连接到控制服务器上读取文件列表。图 1-1 中的 P2P 模型是纯 P2P 计算模型，图 1-3 给出混合 P2P 计算模型。

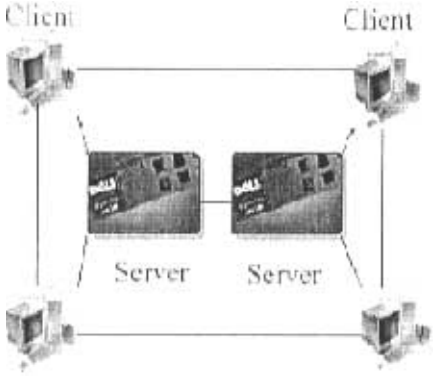

图 1-3：混合 P2P 计算模型

1.3.2 P2P 基础平台的基本设计原则

- 不存在单结点失败。
- 对等实体可以自发组织为一个自治组。
- 对等实体可以适应各种网络环境。
- 资源和服务表示与网络平台、操作系统无关。
- 统一的服务和服务访问接口。

1.3.3 P2P 基础平台的组成部分

P2P 基础平台主要由控制核心、XML 消息传输协议、分布式查找服务以及成员服务组成。在第二章中，我将介绍 P2P 基础平台的结构和资源表示；在第三章中描述 P2P 平台服务框架和控制核心的设计和实现；在第四章中讨论 XML 消息传输协议的设计和实现；在第五章中阐述分布查找服务设计；在第六章中介绍成员服务的设计；最后在第七章中介绍如何使用 P2P 平台开发 P2P 应用。

1.4 小结

本章介绍了 P2P 计算的发展和当前研究方向，解释了 P2P 计算的相关概念。同时说明建立 P2P 基础平台的必要性，并指出了建立 P2P 基础平台的基本原则和平台的组成部分。

第二章 P2P 基础平台体系和资源表示

我们的 P2P 基础平台（为了编程方便，使用 mercury 作为代号，在下面的实现代码和例子中会看到这个代号）分为三层。XML 消息传输协议负责与对等实体所在系统的传输协议进行绑定，同时为上层服务提供与具体通信协议无关的基于 XML 消息的通信协议接口。在 XML 消息传输协议之上是 P2P 基础平台核心部分，提供平台所需的基本服务。所有基本服务均由控制核心调度，并且可以方便地替换和加入新服务。统一调用接口为建立在 P2P 基础平台上的 P2P 应用提供语义一致的调用接口，屏蔽核心服务细节。系统结构见图 2-1。

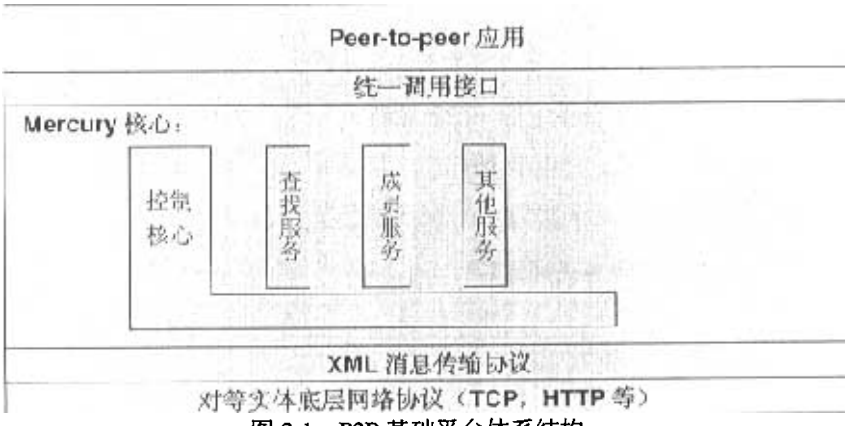

图 2-1：P2P 基础平台体系结构

图 2-2 给出了 P2P 基础平台系统 Use Case 图。

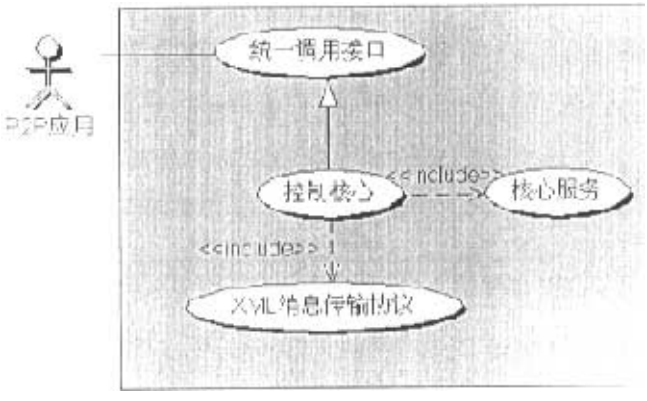

图 2-2：P2P 基础平台系统 Use Case 图

2.1 P2P 基础平台各部分概述

- **控制核心**：P2P 基础平台所有的核心服务均由控制核心管理，所有上层应用的请求都通过控制核心分派到相应的服务对象。
- **XML 传输协议**：对象传输协议分为两层，一层负责与具体网络协议进行绑定，另一层提供基于 XML 消息的通信协议。通过分层，系统实现了平台与具体网络环境的分离，针对不同的网络协议只要替换相应的绑定就可实现通信。目前我们完成对 TCP/UDP 协议的绑定，同时增加了连接管理、异步消息等功能。使用 XML 消息格式方便建立独立于具体网络环境的 P2P 基础平台，而且可以很方便地扩展不同服务需要的消息。我们希望传输协议兼容 W3C XML 协议标准，今后工作方向是使 XML 消息可以通过 SOAP，XML-RPC 等协议进行传输。
- **查找服务**：提供分布查找功能，用来查找对等实体、服务和资源。根据不同应用要求可以有不同查找算法，服务可以根据情况选择合适的算法。
- **成员服务**：负责确定对等实体所在组、加入某个组，以及查找组内其他对等实体。所有对等实体的服务请求都可以通过成员服务进行控制。

2.2 资源表示

P2P 基础平台使用公告（Placard）描述、发布对等实体资源和服务。平台定义了下列公告：

- 对等实体公告（Peer Placard）
- 对等实体组公告（PeerGroup Placard）
- 服务公告（Service Placard）
- 服务内容公告（Content Placard）
- 访问端点公告（Endpoint Placard）

2.2.1 基于 XML 的公告表示

P2P 基础平台所有公告都使用 XML 表示。XML 为分布式系统提供了强大的数据描述方法和与平台无关的数据表示。公告由一系列结构化元素组成，每个元

素包含数据或其它元素。元素可以具有属性，属性由名字/值字符串对组成。属性用来存放描述元素数据的元数据类型。用户可以根据现有的公告创建自己的公告类型，自定义公告可以增加任意元素。

平台根据公告定义的 XML Schemas 检查公告的有效性，并且依据公告操作资源或服务。这些公告也是平台最经常交换的信息。

2.2.2 对等实体公告（Peer Placard）

对等实体公告描述了对等实体资源，主要保存对等实体的信息，例如：对等实体名、可访问的服务和访问对等实体的端点。图 2-3 描述了对等实体公告的结构及对应的类。

```
<?xml version="1.0" encoding="UTF-8?>
<Mercury: PeerPlacard>
    <Name> name of the peer </Name>
    <PID> peer ID </PID>
    <Services>
        <Mercury: ServicePlacard>
            ...
        </Mercury: ServicePlacard>
    </Service>
    <Endpoints>
        <Mercury: EndpointPlacard>
            ...
        </Mercury: EndpointPlacard>
    </Endpoints>
    <InitialApp>
        <Mercury: ServicePlacard>
            ...
        </Mercury: ServicePlacard>
    </InitialApp>
</Mercury: PeerPlacard>
```

对等实体公告 XML 文档

```
package net.mercury.placard
import net.mercury.impl.util.UUID
public class peerPla
{
    String name;
    UUID pid;
    Vector srvPlaVec;
    Vector endPlaVec;
    servicePla initapp;
    ...
}
```

对等实体公告类

图 2-3：对等实体公告

对等实体公告包括下列域：

- **Name**：对等实体的名字。名字用来方便 P2P 系统按名字查找对等实体。
- **PID**：对等实体的 ID。在同一个对等实体组里，对等实体 ID 应该是唯一的。

- **Service**：对等实体可访问的每个服务的服务公告。
- **Endpoint**：对等实体可以获得的每个访问端点的 URI，例如：tcp://202.115.30.194:9000 或者 http://202.115.30.194:80。
- **InitialApp**：对等实体初始化时可选择调用特定服务的服务公告。

各元素值类型见表 2-1：

表 2-1：对等实体公告元素说明

元素名称	数量	元素值类型
Name	1	String[1]
PID	1	UUID
Services	*[2]	servicePla
Endpoints	+[3]	endpointPla
InitialApp	0/1[4]	servicePla

1. 在本文中 String 类型假定不包含任何 XML 限界符（“<”，“>”），但可以包含空格。
2. “*”指 0 个或多个元素。
3. “+”指 1 个或多个元素。
4. “0/1”指 0 个或 1 个元素，表示该元素是可选的。

2.2.3 对等实体组公告（PeerGroup Placard）

对等实体组公告描述了对等实体组资源，主要保存了对等实体所在组的信息，例如：组名、组 ID 和可供访问的组服务。图 2-4 描述了对等实体组公告的结构。

```
<?xml version="1.0" encoding="UTF-8?>
<Mercury: PeerGroupPlacard>
    <Name> name of the peer group </Name>
    <GID> Group ID </GID>
    <Services>
        <Mercury: ServicePlacard>
            ...
        </Mercury: ServicePlacard>
        <InitialApp>
            <Mercury: ServicePlacard>
                ...
            </Mercury: ServicePlacard>
        </InitialApp>
    </Service>
</Mercury: PeerGroupPlacard>
```

对等实体组公告 XML 文档

```
package net.mercury.placard
import net.mercury.impl.util.UUID
public class pGrpPla
{
    String name;
    UUID gid;
    Vector srvPlaVec;
    servicePla initapp;
    ...
}
```

对等实体组公告类

图 2-4：对等实体组公告

对等实体组公告包括下列域：

- **Name**：对等实体组的名字。用来表示对等实体构成的自治组。
- **GID**：对等实体组 ID。
- **Service**：对等实体组提供的每个服务的服务公告。当对等实体加入组时，不需要实例化所有服务，但是新对等实体加入组至少要执行成员服务。目前的 P2P 基础平台不对对等实体进行成员身份认证，也就是说任何对等实体都可以加入到组中。
- **InitialApp**：对等实体加入组时可选择调用的特定服务的服务公告。

各元素值类型见表 2-2：

表 2-2：对等实体组公告元素说明

元素名称	数量	元素值类型
Name	1	String
GID	1	UUID
Services	+	servicePla
InitialApp	0/1	servicePla

2.2.4 服务公告（Service Placard）

服务公告描述了对等实体提供的服务。通过服务公告确定如何调用和使用对等实体提供的服务。图 2-5 描述了服务公告的结构。

```
<?xml version="1.0" encoding="UTF-8?>
<Mercury: ServicePlacard>
    <Name> service name </Name>
    <Version> Version Id </Version>
    <Code> code implementing service </Code>
    <URI> service provider location </URI>
    <Provider> Service Provider </Provider>
    <Content>
        <Mercury: ContentPlacard>
            ...
        </Mercury: ContentPlacard>
    </Content>
</Mercury: ServicePlacard>
```

服务公告 XML 文档

```
package net.mercury.placard
public class servicePla
{
    String name;
    String version;
    String code;
    String uri;
    String provider;
    contentPla content;
    ...
}
```

服务公告类

图 2-5：服务公告

服务公告包括下列域：

- **Name**：服务名字。
- **Version**：服务版本。
- **Code**：实现这个服务的实现代码。
- **URI**：指定服务代码提供者的地址。
- **Provider**：服务提供者信息，通常是提供者名字。
- **Content**：服务提供的共享资源类型。通常提供共享资源的服务拥有大量的同类型资源。为了保持一定的对象粒度，我们为同一类型的资源定义一个服务对象。例如文件共享服务，我们不可能为每个文件创建一个操作对象，我们只定义了文件共享服务对象，提供文件共享操作。

各元素值类型见表 2-3：

表 2-3：服务公告元素说明

元素名称	数量	元素值类型
Name	1	String
Version	1	String
Code	1	String
URI	0/1	String
Provider	1	String
Content	0/1	ContentPla

2.2.5 服务内容公告（Content Placard）

服务内容公告描述了提供共享资源服务的资源类型，共享资源可以是文件、存储空间或者 CPU 处理资源。图 2-6 描述了服务内容公告的结构。

```
<?xml version="1.0" encoding="UTF-8?>
<Mercury: ContentPlacard>
    <Name> content name </Name>
    <MimeType> mime type </MimeType>
    <ResList> resources list </ResList>
</Mercury: ContentPlacard>
```

服务内容公告 XML 文档

```
package net.mercury.placard
import net.mercury.impl.util.mimetype
public class contentPla
{
    String name;
    mimetype mType;
    Vector resList;
    ...
}
```

服务内容公告类

图 2-6 服务内容公告

服务内容公告包括下列域：

- **Name**：服务内容类型名字。
- **MimeType**：服务内容的 mime 类型，根据 mime 类型决定如何处理服务内容。
- **ResList**：这种服务内容类型的所有共享资源列表。

各元素值类型见表 2-4：

表 2-4：服务内容公告元素说明

元素名称	数量	元素值类型
Name	0/1	String
MimeType	1	mimetype
ResList	*	String

2.2.6 访问端点公告（Endpoint Placard）

访问端点公告描述了对等实体传输协议。每个对等实体可以有多个传输协议，每个传输协议对应一个访问端点。访问端点公告是对等实体公告的元素，表示对等实体拥有的访问端点。访问端点由虚拟访问端点地址表示，这个虚拟地址包含有用来创建物理连接需要的全部信息。例如 URI 字符串 tcp://202.115.30.194:1000 或 http://202.115.30.194:80 表示端点的虚拟地址。图 2-7 描述了访问端点公告的结构。

```
<?xml version="1.0" encoding="UTF-8?>
<Mercury: EndpointPlacard>
      <Name> endpoint name </Name>
      <Address> logical address </Address>
</Mercury: EndpointPlacard>
```

访问端点公告 XML 文档

```
package net.mercury.placard
public class endpointPla
{
    String name;
    String uri;
}
```

访问端点公告类

图 2-7：访问端点公告

访问端点公告包括下列域：

- **Name**：访问端点的名字。
- **Address**：访问端点的逻辑地址，平台根据这个地址创建相应的物理连接。

各元素值类型见表 2-5：

表 2-5：访问端点公告元素说明

元素名称	数量	元素值类型
Name	0/1	String
Address	1	String

2.2.7 P2P 基础平台公告类和 ID 定义

P2P 平台的公告在系统中都有对应的类描述，定义公告的元素以及解析和生成公告的 XML 文档。图 2-8 给出了公告类图。

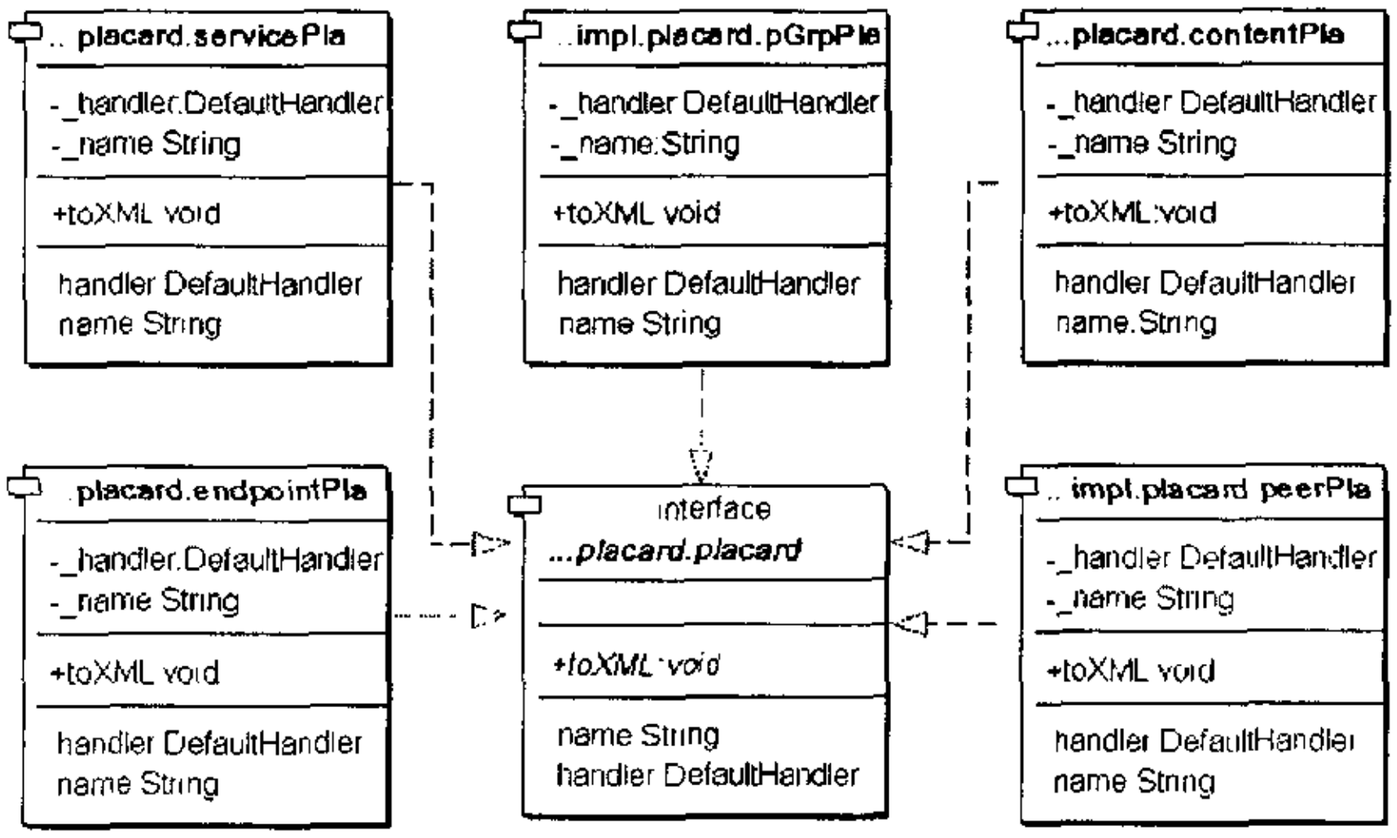

图 2-8：公告类图

接口 ***placard*** 定义了公告类需要实现的基本操作：

- ***public void toXML(OutputStream out);*** 生成公告的 XML 文档。
- ***public String getName();*** 取得公告名字。
- ***public DefaultHandler getHandler();*** 取得处理公告元素的 Handler。

类 ***peerPla***、***pGrpPla***、***servicePla***、***contentPla*** 和 ***endpointPla*** 分别对应对等实体公告、对等实体组公告、服务公告、服务内容公告和端点公告。

P2P 平台的对等实体和对等实体组需要有唯一 ID，ID 由 UUID 表示。UUID 是 128 位的通用身份表示符，由 ***UUID*** 类生成和处理。图 2-9 给出 ***UUID*** 类和 ***UUIDFactory*** 类图。

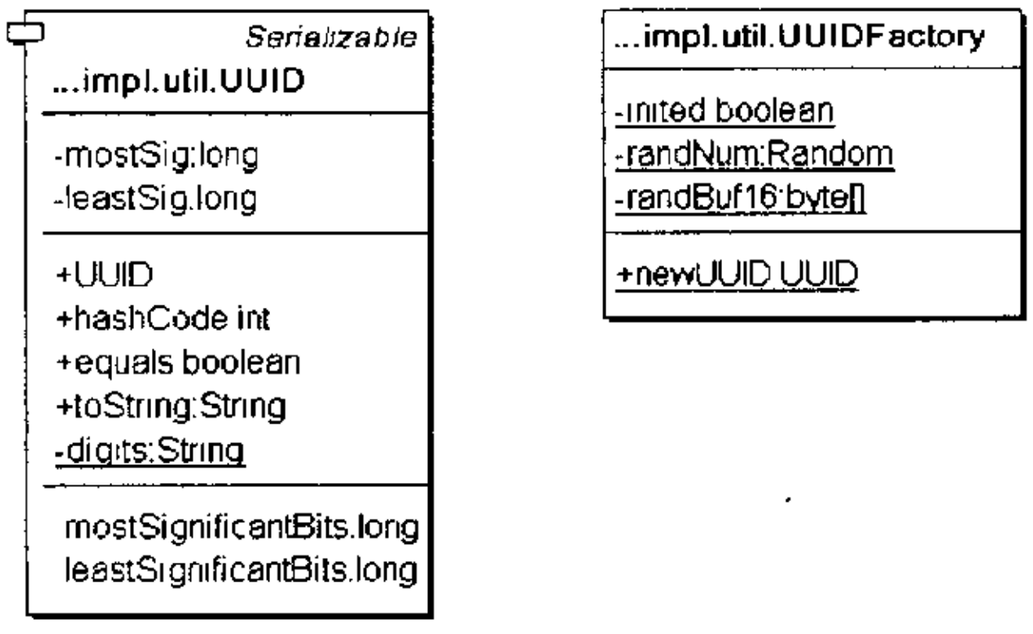

图 2-9：UUID 类及其工厂类类图

2.3 小结

本章介绍了 P2P 基础平台的框架和组成部分，以及平台的资源和服务表示。平台所有资源和服务都使用公告表示，公告是 XML 定义的描述文档。利用 XML 的数据表示能力和中性语言特性，这些表示独立于具体的实现环境。P2P 基础平台服务交换的信息中，很大一部分就是各种公告。

第三章 P2P 基础服务与控制核心

3.1 P2P 基础平台服务

服务是 P2P 基础平台的主要组成部分，决定平台提供的功能。我们的工作主要是为 P2P 应用提供基本运行环境，为此平台实现 P2P 运行环境需要的基本服务。目前我们实现了 XML 通信服务、分布式查找服务和成员服务，并提供用户建立服务的接口。

3.1.1 服务结构描述

P2P 基础平台的所有服务都要实现 ***service*** 接口。由于控制核心服务 ***kernel*** 是管理平台服务运行的基础模块，与其他的服务不同，它不需要实现这个接口。***service*** 接口规定了服务需要实现的基本方法。图 3-1 给出了服务体系的类图。

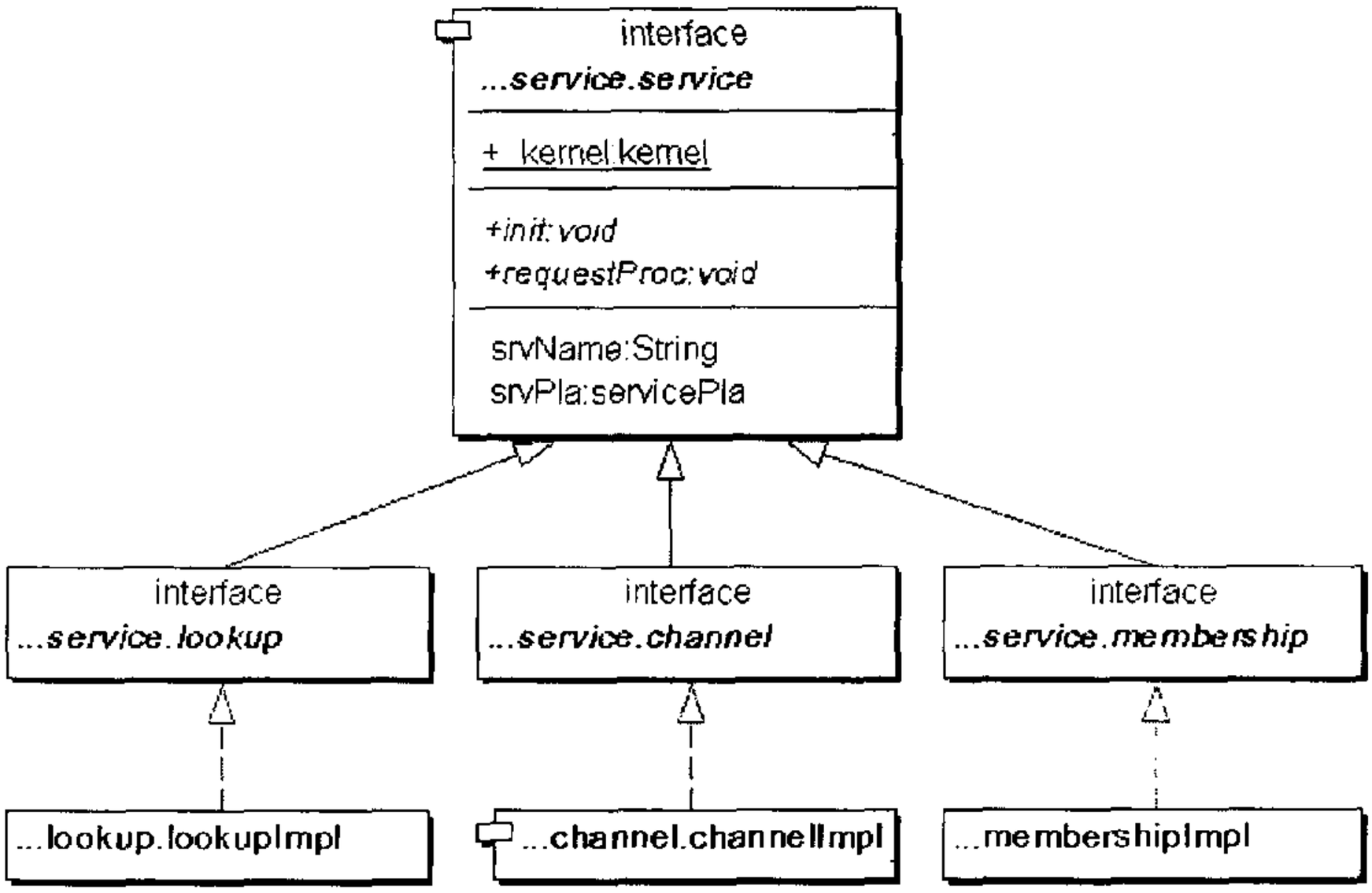

图 3-1：服务体系类图

接口 *service* 方法与属性说明如下：

- ***kernel _kernel***：P2P 基础平台控制核心服务对象的引用，为了使服务能够与控制核心交互，服务对象必须具有控制核心服务对象的引用。
- ***void init (kernel kn)***：初始化服务，并将控制核心服务对象引用传进来。
- ***String getSrvName()***：取得服务名字，控制核心服务将服务名字与服务对象相关联，方便根据消息提供的服务名字分派消息给相应服务处理。
- ***servicePla getSrvPla()***：取得服务的服务公告对象。
- ***void requestProc(message msg)***：处理接收的消息。

我们在设计基础服务时使用了两层接口，这主要是为了方便替换基础服务的实现类。

3.1.2 P2P 基础平台消息概述

P2P 平台的服务主要通过消息的交互实现其功能。每个服务一般都要定义一组自己使用的消息。但是消息服务在接收到消息时，并不知道如何生成具体的消息类型，因此我们定义了一个基本消息类描述接收到的所有消息。图 3-2 给出了基本消息类的定义。

```
import ...
public abstract class message {
        private String _msgName;
        private String _srvName;
        private SAXParser _parser;
        private InputStream msgStream;
        public String getMsgName();
        public String getSrvName();
        public void setMsgName(String msgName);
        public void setSrvName(String srvName);
        public void setParser(SAXParser parser);
        public void toXML(BufferedWriter out);
        public SAXParser getParser();
        public InputStream getMsgStream();
        public void setMsgStream(InputStream msgStream);
}
```

图 3-2：基本消息类定义

基本消息类主要属性和方法：

- ***private InputStream msgStream;*** 接收到的消息内容。
- ***public String getMsgName();*** 获取消息名字。
- ***public String getSrvName();*** 获取处理消息的服务对象名字。
- ***public Void toXML(BufferedWriter out);*** 生成消息的 XML 文档流。将表示消息内容的属性值写到一个输出流中，并且生成相应的 XML 标记。
- ***public SAXParser getParser();*** 取得 XML 解析器引用。
- ***pubic InputStream getMsgStream();*** 取得消息内容。

这个基本消息类主要是提供处理消息所需的信息。P2P 基础平台采用异步 XML 消息传输机制，当消息传输服务接收到一个消息的时候，并不会提前知道消息类型，只能从接收到的消息中解析出来消息的类型和处理消息的服务名称。所以通过这个消息的对象将接收的消息转到相应服务处理。其他特定消息均继承这个基本消息类。接收到一个基本消息后，服务可以通过解析消息中所存消息内容构造相应的消息。各种服务所需的特定消息在介绍该服务的章节中定义。

3.1.3 解析 XML 数据源

P2P 基础平台所有的消息、配置信息都是由 XML 表示的，这就经常需要根据消息对象生成相应 XML 数据，或者将 XML 数据源映射成相应对象。由消息对象生成 XML 数据不困难，一般就是将属性按 XML 数据格式输出到输出流中即可。而由 XML 数据源映射到对象就要复杂些，需要一些解析器的帮助。我们使用 SAX（Simple API for XML）解析 XML 数据。

SAX 定义了应用程序从 XML 文档中读取数据的一套接口，这是基于事件驱动处理 XML 文档的接口。SAX 接口的具体实现由 XML 解析器完成。我们使用 Sun Microsystem®提供的 JAXP（Java API for XML Processing），这套 API 实现 SAX 接口的解析器是 Apache 的 crimson 解析器。我们之所以使用 SAX 接口基于以下原因：

- **可以解析任意大小的文件**：因为 SAX 不需要把整个文件加载到内存中，所以对内存的占用比较小，而且不会随着文件大小的增加而增加。
- **适合创建自己的数据结构**：服务对象往往需要根据公告、消息这样的高

级对象而不是一些低级元素、属性和处理指令来创建数据结构。这些“高级对象”可能只是和 X M L 文件内容有一点关系，例如它们可能只是组成 X M L 文档和其他数据源的数据。因此，没有必要象 DOM 接口那样将所有 XML 元素都按对象处理。

- **适合少量信息子集**：SAX 一个非常好的特点就是可以非常容易地忽略不感兴趣的数据信息。
- **简单、快速**：S A X 非常易于使用，而且如果从文档的简单序列中获取需要的信息，S A X 几乎一定是最快的方法。

下面我们看一下使用 JAXP 处理 XML 文档的例子。图 3-3 给出使用 JAXP 解析 XML 数据源的示例代码。

```
...
//创建解析器工厂对象
SAXParser parser;
SAXParserFactory factory = SAXParserFactory.newInstance();
try {
    //创建解析器，假设已有了 XML 数据源 InputStream xmlDataSrc
    parser = factory.newSAXParser();
    //创建元素处理 Handler
    DefaultHandler handler = new myHandler();
    //解析 XML 数据源
    parser.parse(xmlDataSrc, handler);
    servicePla srvPla = handler.retrivePla();
    xmlDataSrc.close();
} catch (SAXException se) {
} catch (IOException ioe) {
} catch (ParserConfigurationException pce) {
    //异常处理
}
...
```

图 3-3：使用 JAXP 解析 XML 数据源示例

图 3-3 只是应用程序调用解析器的流程，真正对 XML 数据源的处理由 ***myHandler*** 类定义。这个类继承 ***Defaulthandler*** 类，定义如何处理 XML 数据源中的元素。下面以服务公告为例，给出如何将服务公告的 XML 表示映射成服务公告对象。图 3-4 给出服务公告 XML 文档示例和对应的服务公告对象。

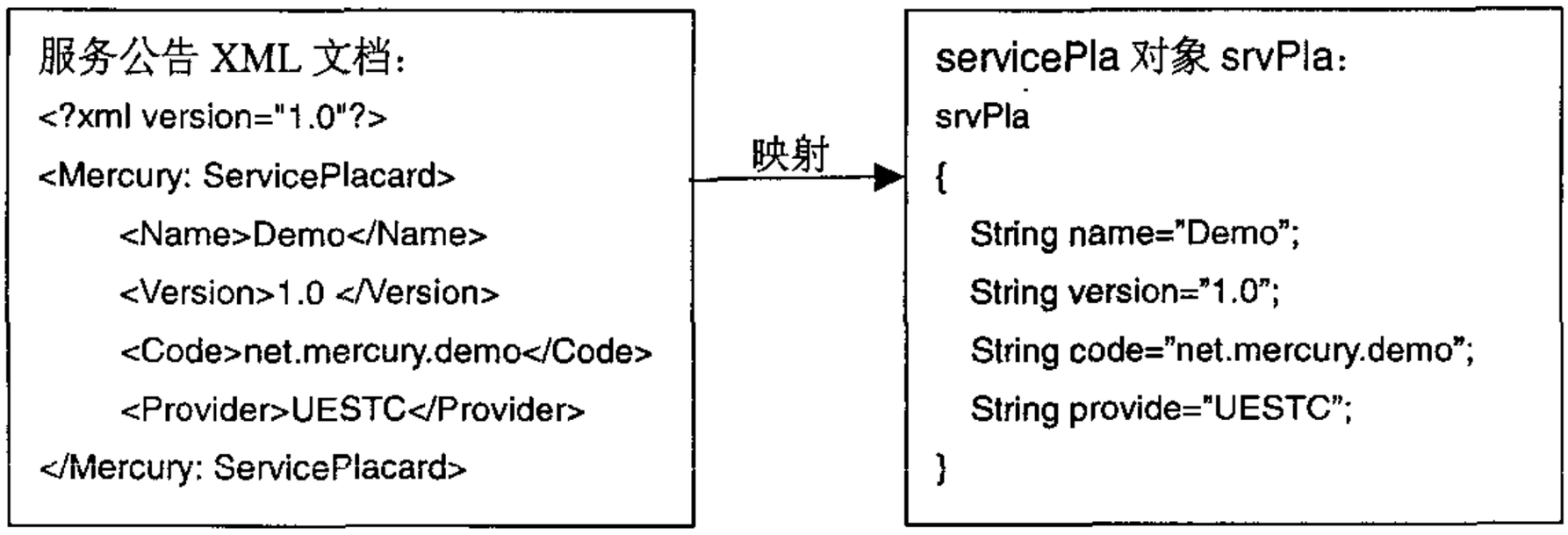

图 3-4：服务公告 XML 文档及其对应对象

图 3-5 给出处理服务公告 XML 文档元素的 Handler—*myHandler*。

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import net.mercury.impl.placard.*;
public class myHandler extends DefaultHandler {
    private servicePla srvPla = new servicePla();
    private CharArrayWriter contents = new CharArrayWriter();
    public void startElement( String namespaceURI, String localName,
                    String qName, Attributes attr) throws SAXException {
        contents.reset();
    }
    public void endElement( String namespaceURI, String localName,
                    String qName) throws SAXException {
        if(qName.equals("Name"))
            srvPla.name = contents.toString();
        if(qName.equals("Version"))
            srvPla.version = contents.toString();
        if(qName.equals("Code"))
            srvPla.code = contents.toString();
        if(qName.equals("Provider"))
            srvPla.provider = contents.toString();
    }
    public void characters( char[] ch, int start, int length ) throws SAXException {
        contents.write( ch, start, length );
    }
    public servicePla retrievePla() {
        return srvPla;
    }
}
```

图 3-5：服务公告 XML 文档元素处理 Handler

3.1.4 服务使用的消息队列

P2P 平台的服务一般都需要发送和接收消息。为了方便消息的处理，平台定义了消息队列，定义如下：

```
public class MsgQueue
{
    private Vector _inMsg;
    private message next();
    public MsgQueue();
    public synchronized void push(message msg);
    public synchronized message poll(long timeout);
}
```

向量*_inMsg* 存放需要处理的接收消息。一般都是服务的 ***requestProc*** 方法将接收的消息解析后调用 ***push*** 方法放入消息队列。服务中需要接收消息的方法调用 ***poll*** 方法提取消息，如果消息队列中没有消息，***poll*** 方法调用 ***wait*** 将线程阻塞，直到 ***push*** 方法被调用或者超时。当 ***push*** 方法被调用存储消息时，***push*** 方法调用 ***notifyAll*** 唤醒所有阻塞在这个消息队列上的线程。

3.2 P2P 基础平台控制核心服务

控制核心服务实现了对 P2P 平台上各种服务的管理和调度功能，同时为建立在平台上的 P2P 应用提供访问接口。平台通过控制核心服务初始化和注册服务，并且通过控制核心完成服务调度。

3.2.1 控制核心服务描述

控制核心由 ***kernel*** 接口和 ***kernelservice*** 实现类组成。***kernel*** 接口为其他服务和应用提供访问接口，起了统一调用接口的作用。***kernelservice*** 实现接口定义的功能，并且控制平台运行。图 3-6 给出了 ***kernel*** 和 ***kernelservice*** 定义。

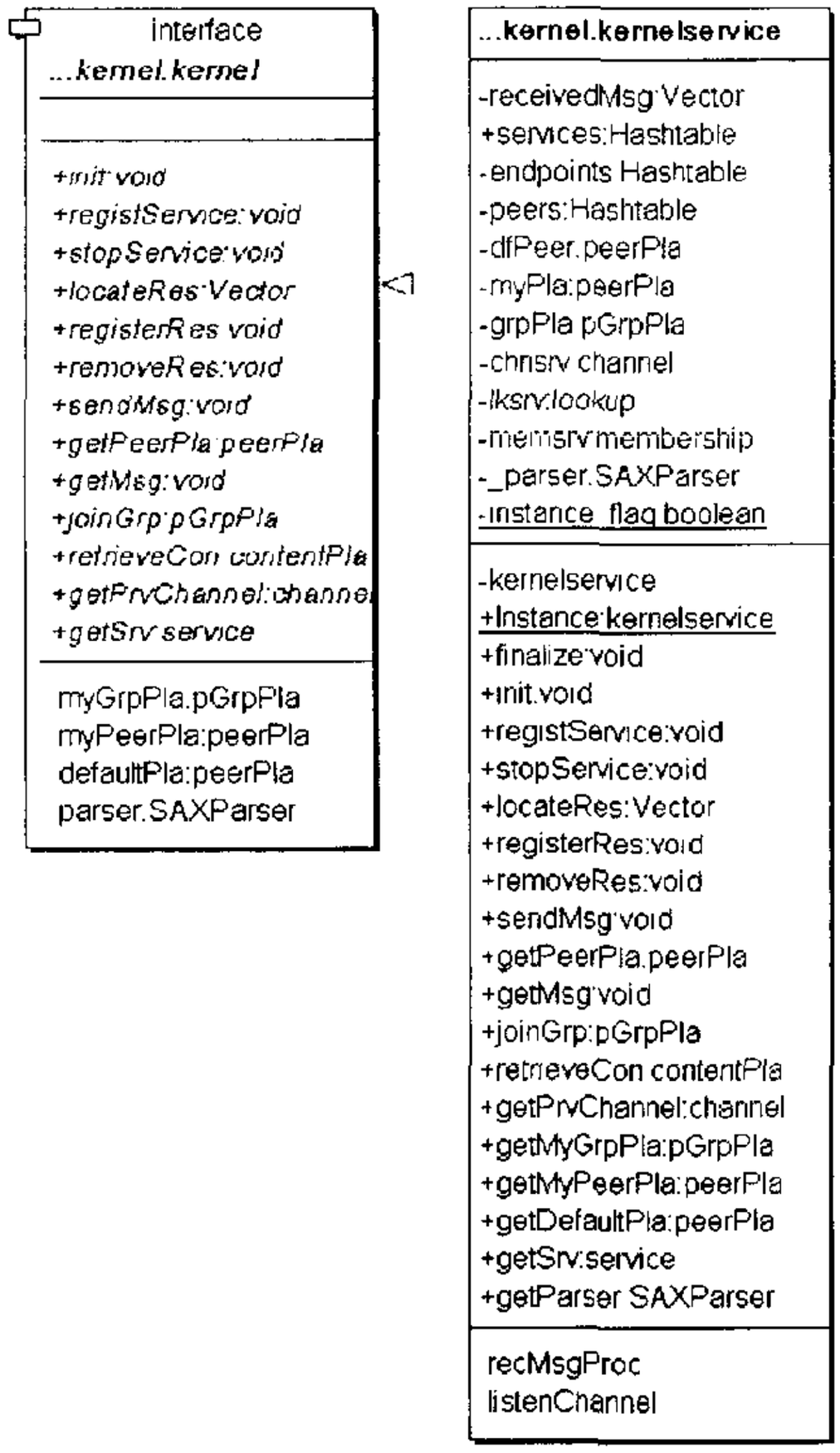

图 3-6：kernel 和 kernelservice 定义

kernel 接口定义了 P2P 平台提供的基本方法。

- ***public void init();*** 初始化控制核心。
- ***public Vector locateRes(String keyword,String resType);*** 查找资源所在对等实体。
- ***public void registerRes(contentPla content);*** 将资源注册到索引点上。
- ***public void removeRes(contentPla contenet);*** 删除资源索引。
- ***public void getMsg();*** 读取发送到本地的消息。
- ***public contentPla retrieveCon(UUID dstpeer,String srvname);*** 取得服务内容公告。
- ***public pGrpPla getMyGrpPla();*** 取得自己的对等实体组公告。
- ***public peerPla getMyPeerPla();*** 取得自己的对等实体公告。

- ***public peerPla getDefaultPla();*** 取得缺省引入点对等实体公告。
- ***public void registService(Service newsrv);*** 注册服务。
- ***public void stopService(Service srv);*** 终止服务。
- ***public void sendMsg(Message msg, UUID pid);*** 发送消息。
- ***public peerPlacard getPeerPla(UUID pid);*** 取得对等实体公告。
- ***public Vector locateRes(ContentPlacard conpla);*** 查找资源。
- ***public pGrpPlacard joinGrp(UUID pgid);*** 加入对等实体组
- ***public channel getPrvChannel(endpointPla ePla);*** 取得建立在参数给出的端点上的通信管道。
- ***public SAXParser getXMLParser();*** 取得 XML 解析器引用，解析器在 ***kernelservice*** 初始化时创建，所有服务都使用这个解析器解析 XML 文档。

P2P 平台的其他服务和应用通过 ***kernel*** 接口提供的方法访问 ***kernelservice***。

kernelservice 实现了 ***kernel*** 接口。因为系统运行时只需要一个控制核心服务，我们将 ***kernelservice*** 设计成 Singleton 模式确保 ***kernelservice*** 类仅有一个实例，并提供一个访问它的全局访问点。为 ***kernelservice*** 类定义静态成员方法 ***Instance***，用来创建它的唯一实例以及提供用户创建实例的访问点。具体实现见图 3-7。

```
import …
public class kernelservice implements kernel {
        …
        static private boolean instance_flag = false;
        private kernelservice() {}
        static public kernelservice Instance()
        {
                if (!instance_flag)
                {
                        instance_flag = true;
                        return new kernelservice();
                }
                else
                        return null;
        }
        public void finalize()
        {
                        instance_flag = false;
        }
        …
}
```

图 3-7：创建 ***kernelservice*** 单实例代码

3.2.2 控制核心初始化

应用创建 ***kernelservice*** 实例后，调用 ***init*** 方法初始化 P2P 平台。平台的配置

信息存储在 *config* 文件中，图 3-8 给出配置文件的局部。

```
...
    <PeerName>
        Caugar
    </PeerName>
    ...
    <Pid>
    Mercury://FDA5D413778147658600B63E87542146
    </Pid>
    ...
    <isRendezvous>
        false
    </isRendezvous>
    <Services>
        <Mercury: ServicePlacard>
            ...
        </Mercury: ServicePlacard>
    </Services>
    <Endpoints>
        <Mercury: EndpointPlacard>
            ...
        </Mercury: EndpointPlacard>
    </Endpoints>
    <DefaultPeer>
        <Mercury: PeerPlacard>
            ...
        </Mercury: PeerPlacard>
    </DefaultPeer>
    ...
```

图 3-8：配置文件局部

init 方法根据配置文件内容生成平台的组、对等实体、服务和端点等公告，并且初始化各种服务对象。具体过程见图 3-9。

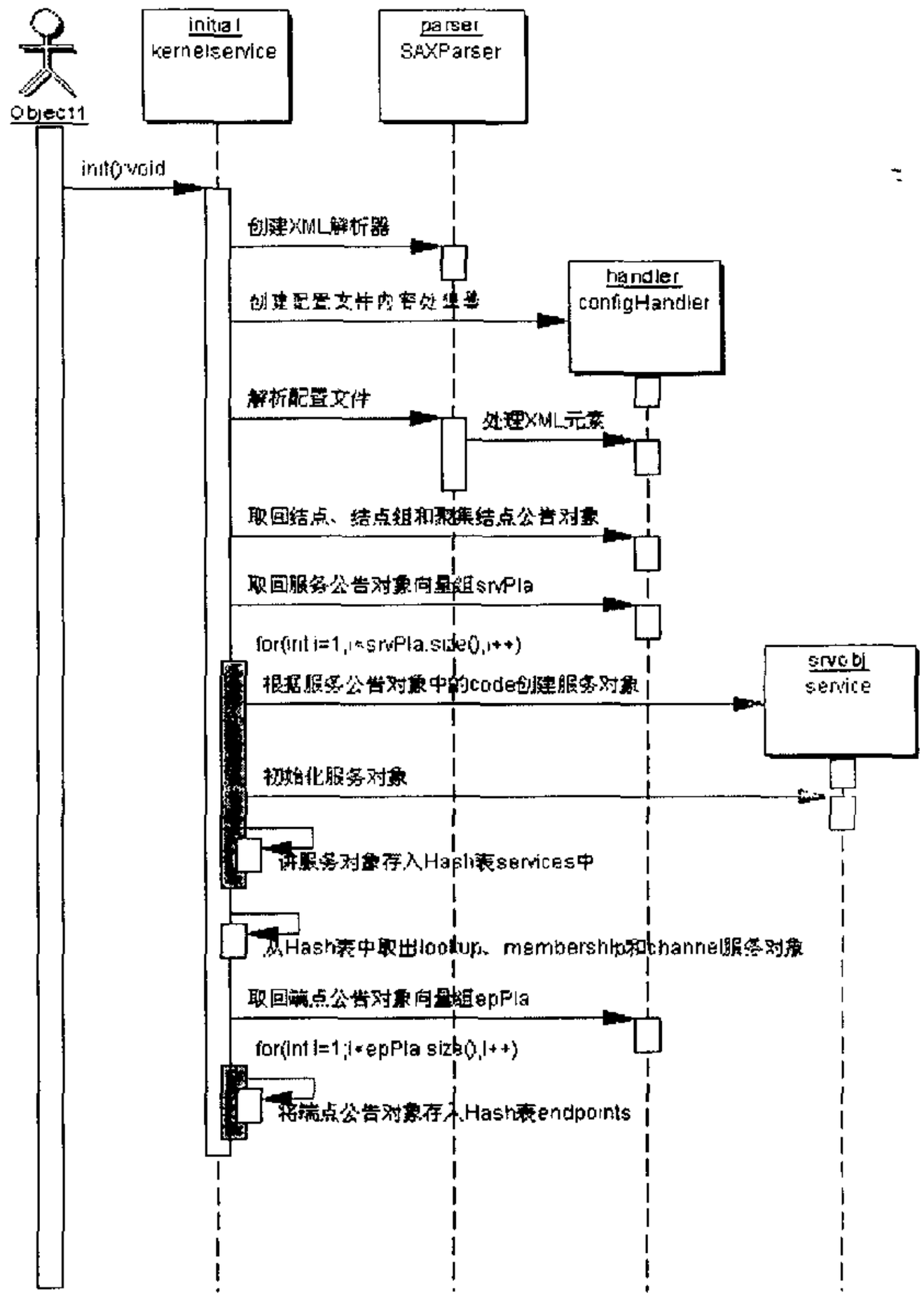

图 3-9：*kernelservice* 初始化顺序图

在前面我们已经介绍了使用 SAX 解析 XML 文档的基本流程，下面我们来看如何使用已有的 XML 元素 Handler 解析配置文件。从配置文件我们可以看到，文件由已经定义的不同元素组成，我们可以利用每种元素自己 Handler 组合起来解析配置文件。通过替换 SAX 解析器的 ***ContentHandler***，我们可以将整个解析工作分派到不同模块，那么我们就可以在局部项目范围使用已有的 Handler 实现解析工作。图 3-10 给出了如何通过替换 ***ContentHandler*** 利用不同元素的 Handler 解析 XML 文档。

```
import …
public class configHandler extends DefaultHandler
{
    //其他部分与基本的 Handler 相同。
    public void startElement( String namespaceURI, String localName, String qName,
                    Attributes attr ) throws SAXException {
        contents.reset();
        if(qName.equals("Mercury:ServicePlacard")) {
            servicPla sp = new servicePla();
            srvpla.addElement(sp);
            //创建子元素 Handler。
            servicePlaHandler srvhd = (sp.getHandler).Instance();
            //调用子元素 Handler 解析子元素
            srvhd.parsePla(parser, this, sp);
        }
        if(qName.equals("Mercury:EndpointPlacard"))
        { // 与上面类似，只不过创建 endpointPlaHandler 的实例。 }
    }
    …
}
```

图 3-10：利用已有 XML 元素 Handler 解析复合 XML 文档

子元素 Handler 的定义与前面讲的基本一样，不过增加了 parsePla 方法，这个方法主要就是为了实现元素 Handler 之间的切换。下面是 parsePla 方法的实现代码。

```
public void parsePla (SAXParser parser, DefaultHandler parent, servicePla newsrv)
{
    //保存上一级的元素 Handler
    this.parent = parent;
    //将解析器 XML 的元素 Handler 设定为子元素的 Handler
    XMLReader reader=parser.getXMLReader();
    reader.setContentHandler(this);
    …
}
```

此外，在 endElement 方法中增加如下语句。

```
//子元素解析完毕
if (qName.equals("Mercury:ServicePlacard")
{
        //恢复上一级元素 Handler
        XMLReader reader=parser.getXMLReader();
        reader.setContentHandler(parent);
}
```

3.2.3 控制核心运行

P2P 平台服务与应用都是通过交换消息完成功能，控制核心通过将接收的消息分发给相应服务、服务消息发送至相应目的地驱动平台运行。应用在完成 ***kernelservice*** 的初始化后调用 ***getMessage*** 方法。这个方法创建一个线程 ***listenChannel*** 监听接收通道，一旦接到消息创建一个处理线程 ***recMsgProc***。在这个消息处理线程中，接收的消息分派给相应的服务处理。前面讲过，每个消息都包含有处理消息的服务名字，根据服务名字可以从 Hash 表 ***services*** 中查到相应服务对象。图 3-11 给出了调用 ***getMessage*** 方法的顺序图。

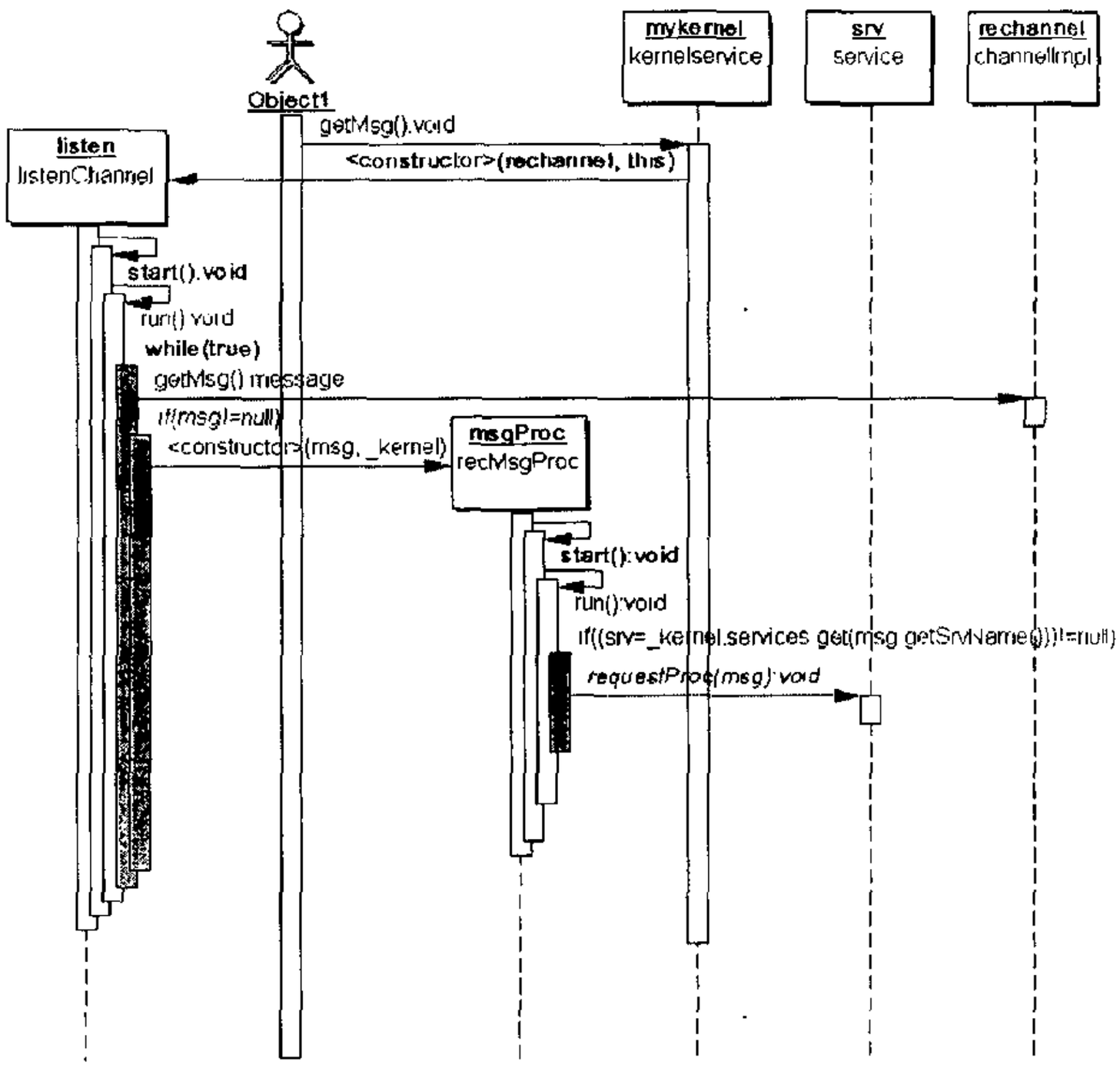

图 3-11：调用 getMessage 方法顺序图

3.2.4 控制核心与其他服务的交互

由于整个 P2P 平台由多个服务组成，不同的功能分布在各个服务对象中。这种分布导致对象间产生很多连接。在最坏情况下，每个对象都知道其他所有对象。

对象间相互连接的增加势必降低系统的可复用性，任何一个对象都不太可能在没有其他对象的支持下工作，系统表现为一个不可分割的整体。而且由于行为被分布在不同对象中，对系统行为进行任何较大的改动都十分困难。结果会使开发者不得不定义很多子类来定制系统的行为。因此，我们通过将集体行为封装在一个单独的中介者对象（mediator）以避免这种问题，这就是平台控制核心服务对象。控制核心负责控制和协调一组服务对象间的交互，充当一个中介使服务对象不再相互显式引用。服务对象仅知道中介者，从而减少了相互连接的数目。这就是设计模式中典型的 Mediator 模式。图 3-12 给出了 ***kernelservice*** 和其他服务关系的类图。

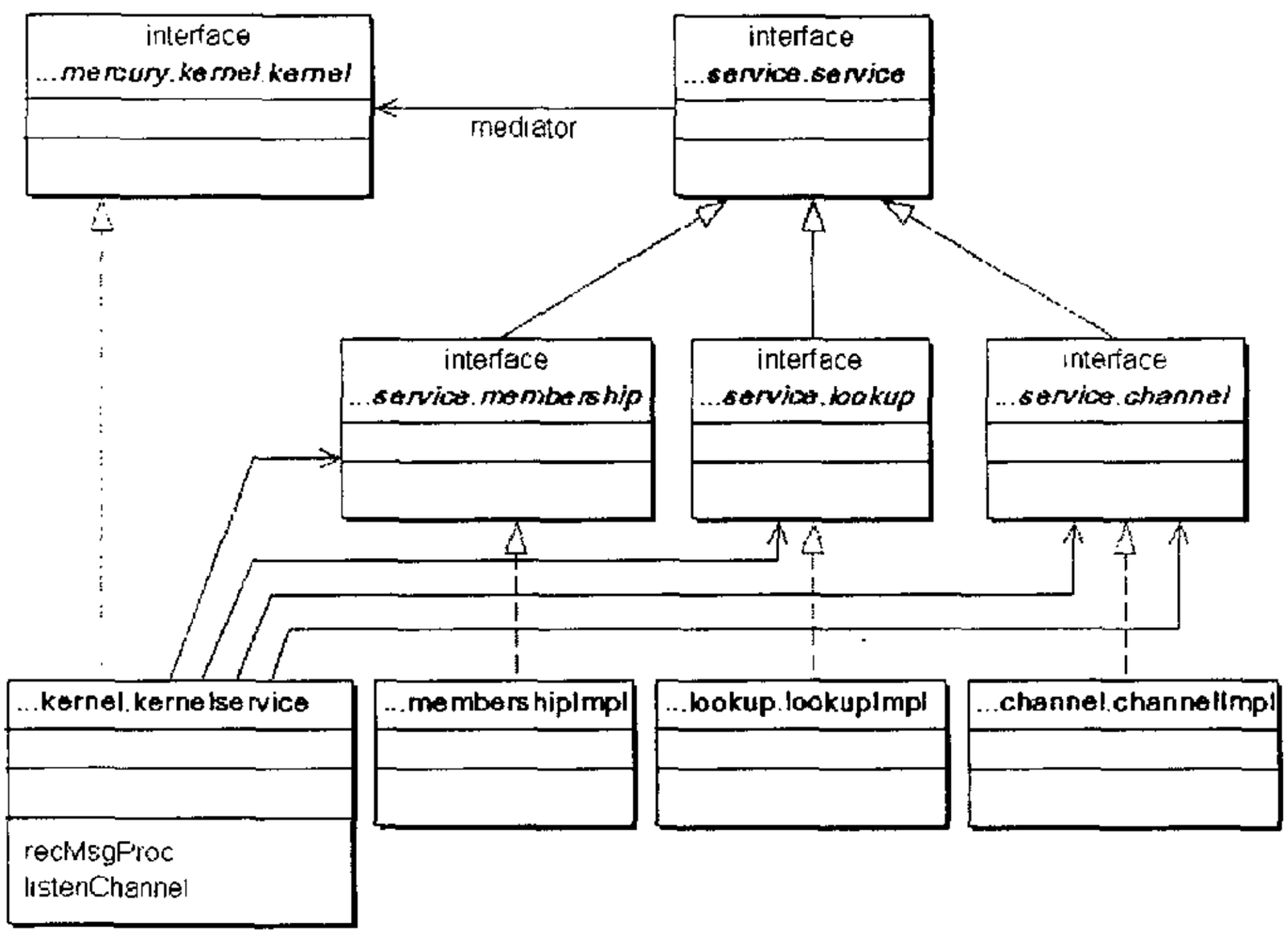

图 3-12：***kernelservie*** 与其他服务关系

从图 3-12 中我们可以看出 ***kernel*** 作为中介者接口，实现 ***kernel*** 接口的 ***kernelservice*** 负责协调其他服务对象来实现协作行为。这里每个服务对象都知道它的中介者对象，当它需要与其他服务通信的时候，与它的中介者通信。每个服务对象向中介者 ***kernelservice*** 发送，并从 ***kernelservice*** 接收请求，***kernelservice*** 在各个服务对象间适当地转发请求以实现协作。图 3-13 给出调用 ***kernelservice*** 的 ***locateRes*** 方法时，控制核心如何与其他服务协作的例子。

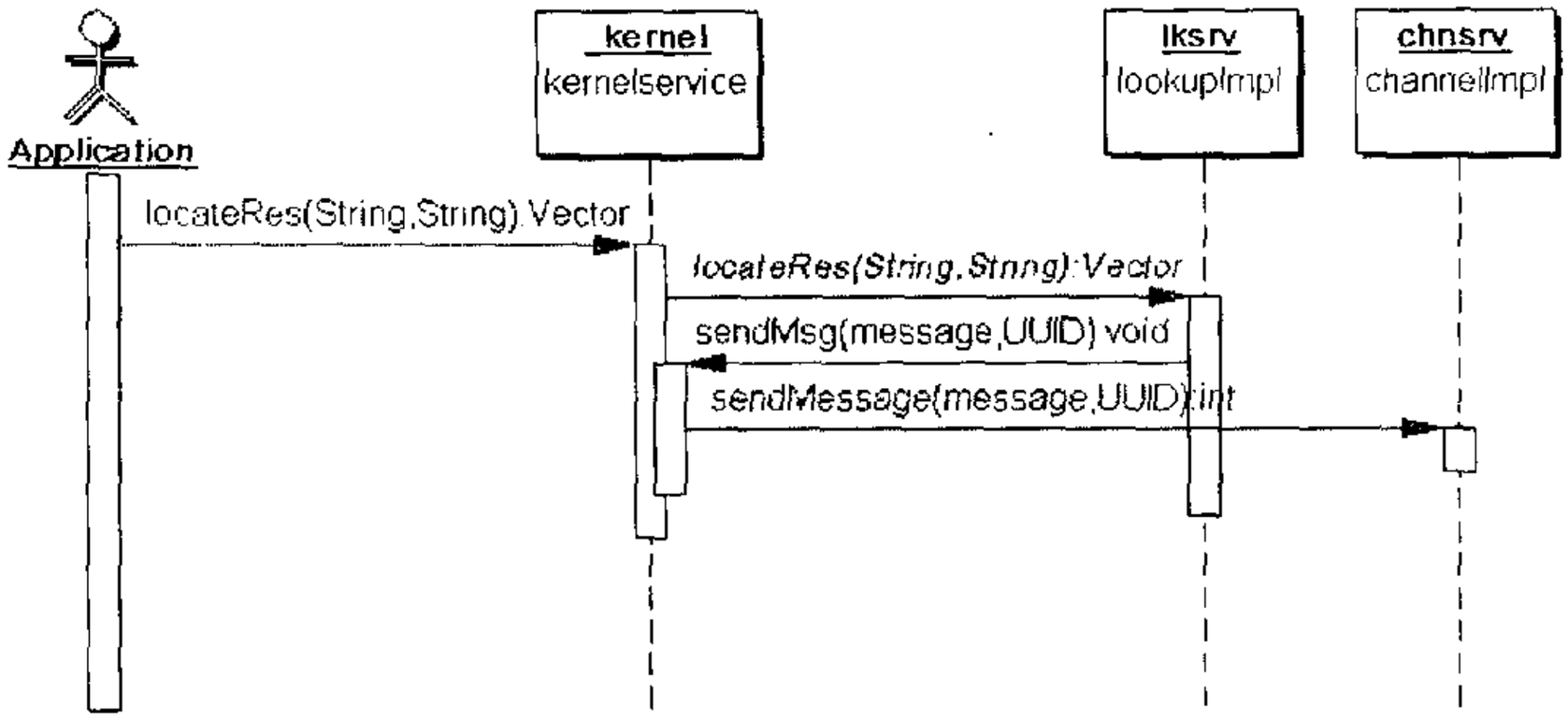

图 3-13：*kernelservice* 中介者模式交互顺序图

由图 3-13 可以我们看出应用调用 ***kernel*** 的 ***locateRes*** 方法时，***kernel*** 其实调用的是查找服务 ***lookupImpl*** 对象的 ***locateRes*** 方法。而查找服务在执行操作过程中需要发送消息，它并不是直接调用通道服务 ***channelImpl*** 对象的发送消息方法，而是通过 ***kernel*** 的 ***sendMessage*** 方法发送消息。控制核心 ***kernel*** 调用消息服务 ***channelImpl*** 对象的 ***sendMessage*** 方法完成消息发送。因此，控制核心作为中介者将查找服务和消息服务解耦，避免两者间的互相引用。

3.3 小结

本章主要介绍了 P2P 基础平台服务的框架和控制核心的设计。平台服务使用 XML 消息交换信息，基本消息类定义了服务使用的 XML 消息的基本结构和处理方法。其他特定消息都是从基本消息扩展而来，定义自己需要的特定元素和解析方法。控制核心服务管理 P2P 平台所有服务，通过采用中介者模式减少这些服务之间的耦合性，方便了服务的设计和复用。但是中介者模式的最大弊端在于控制核心需要了解所有服务的功能和接口，增加了控制核心的复杂性，也使它难以被复用。

第四章 XML 消息传输服务

XML 消息传输服务为 P2P 平台的服务和应用提供网络通信平台，服务使用 XML 消息作为信息交换载体。为了使平台能建立在多种网络环境上，传输服务分为两层，一层为其他服务提供 XML 消息传输语义和接口，我们称之为通道（channel）；另外一层是由端点（endpoint）对应的网络传输层，这一层是通道与底层网络协议层的接口。端点通过绑定不同的底层网络协议适应多种网络环境，我们目前实现端点对 TCP/UDP 协议的绑定。

4.1 XML 消息传输服务结构

图 4-1 给出了 XML 消息传输服务类图。

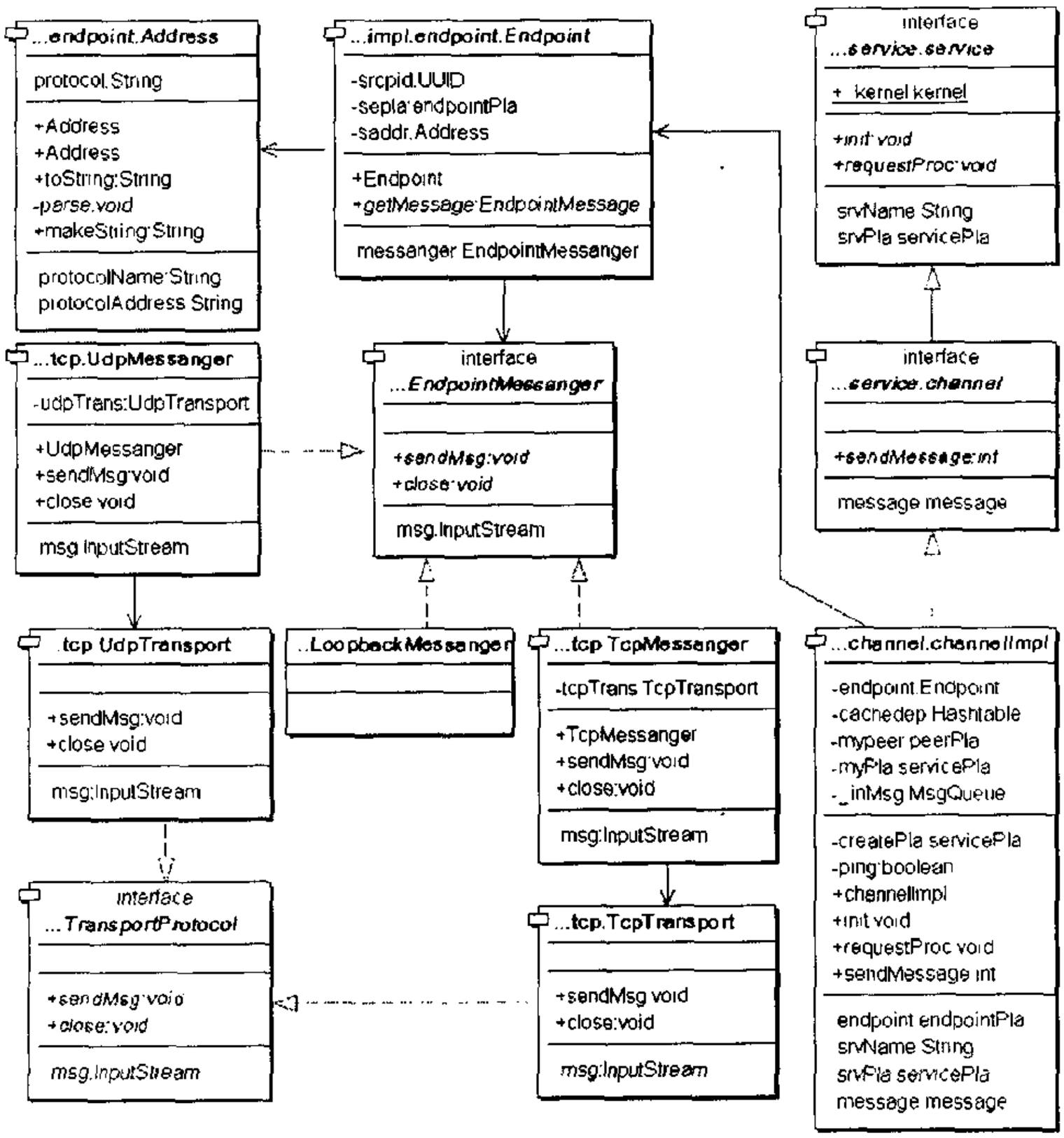

图 4-1：XML 消息传输服务类图

根据平台的设计，服务通过异步消息进行通信，这样做是由 P2P 应用的特点决定的。P2P 应用不同于传统的 Client/Server 应用，对等实体一般不会只和某个固定对等实体通信。一个对等实体在与其他对等实体通信前，往往要通过某些对等实体找到通信目的地，这个过程可能是发散的而且耗时也较长，所以不适合采用同步消息通信。当然找到目的对等实体之后，根据具体应用语义可能会需要同步消息传输机制。因此，控制核心服务通过 ***getPrvChannel*** 方法为某些需要同步消息的应用提供一个独享通道。在第七章中介绍的文件共享应用就会用到这种独享通道。对于传输异步消息，控制核心复用一个通道发送和接收消息。从第二章的对等实体公告定义我们可以看出每个对等实体可以有多个通信端点，这样对等实体可以通过多个网络协议进行通信。在图 4-1 中，我们可以看到每个通道对应一个端点。所以，如果需要使用多个网络协议进行通信，就要建立多个使用不同端点的通道。每个端点是一个网络协议的逻辑抽象。

4.1.1 XML 消息服务通道层

XML 消息服务接口 ***channel*** 为控制核心 ***kernelservice*** 提供访问消息服务的接口。这个接口定义了访问服务的方法：***getMessage*** 和 ***sendMessage***。通过这两个方法，控制核心实现消息的接收和发送。在 ***service*** 接口和消息服务实现类之间增加这个接口是为了方便替换消息服务的实现。

XML 消息服务实现类 ***channelImpl*** 实现了 ***service*** 和 ***channel*** 接口定义的方法，并管理对等实体拥有的通信端点。***channelImpl*** 类中重要的属性和方法：

- ***private endpointPla endpoint;*** 通道拥有的端点。
- ***private Hashtable cachedep;*** 缓存与通道通信的其他通道的端点公告，在发送消息时，如果目的对等实体的端点公告在缓存中，就可以用来直接发送消息而不必通过查找服务获得目的对等实体公告。
- ***private MsgQueue _inMsg;*** 存放接收到的 ***Pong*** 消息的消息队列。
- ***private boolean ping(UUID dstpid);*** 用来验证目的对等实体是否活跃。这个方法向目的对等实体发送 ***Ping*** 消息，如果在规定时间内没有收到 ***Pong*** 消息就认为目的对等实体不活跃。因为 P2P 环境中对等实体状态变化很快，在发送消息前确定目的对等实体是否活跃很有必要。

- ***public void requestProc(message msg);*** 处理接收到的消息服务的消息。消息服务定义了两个自己使用的特定消息，***Ping*** 和 ***Pong***。***Ping*** 消息用来询问目的对等实体是否活跃，目的对等实体接到 ***Ping*** 消息后发送 ***Pong*** 消息表示自己处于活跃状态。如果接收的是 ***Ping*** 消息，构造并发送 ***Pong*** 消息。如果接收的是 Pong 消息，将消息存入***_inMsg*** 消息队列中。
- ***public message getMessage();*** 从端点接收消息。
- ***public void setEndpoint(endpointPla ePla);*** 设定通道使用的端点。
- ***public int sendMessage(message msg,UUID dstpid);*** 向目的对等实体发送消息。根据对等实体 ID 从 ***cachedep*** 中查找对等实体的端点公告，如果得到缓存的端点，调用 ***ping*** 方法检查目的对等实体是否活跃。如果目的对等实体活跃，使用端点信使（***EndpointMessanger***）发送消息。如果没有缓存目的对等实体的端点公告或者目的对等实体不活跃，返回出错值。

4.1.2 XML 消息服务端点层

端点绑定特定的网络协议，作为该协议的逻辑抽象提供给 XML 消息服务通道层访问。端点主要功能就是绑定网络协议，构造提供给网络协议的底层 XML 消息。

端点由 ***Endpoint*** 类定义，主要属性和方法如下：

- ***private EndpointMessanger messanger;*** 绑定网络协议的端点信使。
- ***private endpointPla sepla;*** 这个端点的端点公告。
- ***private Address saddr;*** 端点的逻辑地址，这个地址提供具体协议地址信息给端点信使用来初始化特定网络协议，如建立 TCP 或 UDP Socket。
- ***public EndpointMessanger getMessanger();*** 取得端点信使，消息服务通道使用端点信使发送接受消息。
- ***public EndpointMessage getMessage(message msg);*** 生成交给网络协议发送的底层 XML 消息。

端点通过端点信使发送和接收消息，端点信使接口由 ***EndpointMessanger*** 定义。接口主要方法：

- ***public void sendMsg(EndpointMessage msg,endpointPla dstePla);*** 向目的端点发送消息。
- ***public InputStream getMsg();*** 获取发送到这个端点的消息。
- ***public void close();*** 关闭端点信使。为了释放某些资源，不再使用端点信使的时候应该关闭它。

UdpMessanger 类是绑定在 UDP 协议上的端点信使，实现了 ***EndpointMessanger*** 接口的方法。

LoopbackMessanger 类定义了本地发送和接收信使，消息并不发送到网上。这主要是为了开发服务时做测试用，类似 TCP/IP 协议中的 127.0.0.1 地址。

端点对特定网络协议的绑定由 ***TransportProtocol*** 接口定义，接口封装了具体网络协议的操作。接口的主要方法：

- ***public void sendMsg(InputStream in,Address dstaddr);*** 向目的地址发送数据，数据在输入流 ***in*** 中。
- ***public InputStream getMsg();*** 取得网络协议接收的数据流。
- ***public void close();*** 关闭网络协议，如关闭 Socket。

目前我们实现了对 TCP/UDP 协议的绑定，由于 P2P 基础平台主要使用异步消息方式，端点绑定在 UDP 协议上。***UdpTransport*** 类定义了对 UDP 协议的绑定，是 ***TransportProtocol*** 接口的实现。

对于需要使用同步消息的服务或应用，可以使用绑定在 TCP 协议上的端点。协议绑定由 ***TcpTransport*** 类定义。

端点在发送和接收消息的时候使用端点公告作为参数。在第二章中我们讲过端点公告中包含有表示端点逻辑地址的 URI，但是网络协议不能直接使用 URI 传送消息。因此使用 ***Address*** 类将 URI 转为网络协议可以识别的地址。

4.2 XML 消息服务定义的消息

4.2.1 XML 消息服务通道层消息

前面已经提到消息服务定义了两个消息，***Ping*** 和 ***Pong*** 消息。

图 4-2 给出了 ***Ping*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: Ping>
    <SourcePid>Source Peer Id</SourcePid>
    <TargetPid>Target Peer Id</TargetPid>
    <SerialNum>serial numbler</SerialNum>
</Mercury: Ping>
```

图 4-2：Ping 消息 XML 文档

Ping 消息包括下列域：

- **SourcePid**：发送 Ping 消息的对等实体 ID。
- **TargetPid**：Ping 消息目的对等实体 ID。
- **SerialNum**：Ping 消息的序列号，用来匹配接收到的 Pong 消息。

各元素值类型见表 4-1：

表 4-1：Ping 消息元素说明

元素名称	数量	元素值类型
SourcePid	1	UUID
TargetPid	1	UUID
SerialNum	1	long

图 4-3 给出了 Pong 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: Pong>
    <SourcePid>Source Peer Id</SourcePid>
    <TargetPid>Target Peer Id</TargetPid>
    <SerialNum>serial numbler</SerialNum>
</Mercury: Pong>
```

图 4-3：Pong 消息 XML 文档

Pong 消息包括下列域：

- **SourcePid**：发送 Pong 消息的对等实体 ID。
- **TargetPid**：Pong 消息目的对等实体 ID。
- **SerialNum**：Pong 消息的序列号，拷贝 Ping 消息中的序列号。

各元素值类型见表 4-2：

表 4-2：Pong 消息元素说明

元素名称	数量	元素值类型
SourcePid	1	UUID
TargetPid	1	UUID
SerialNum	1	long

4.2.2 端点消息

最终通过网络协议发送的是端点消息。端点消息将服务提供的消息编码后封装在一个 XML 文档中，图 4-4 给出端点消息结构

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: Message>
    <MessageVersion> version</ MessageVersion>
    <MessageSourcePid>source peer ID</ MessageSourcePid>
    <Mercury: EndpointPlacard>
        ...
    </ Mercury: EndpointPlacard >
    <MessageTargetPid>target peer ID</ MessageTargetPid>
    <MessageName>embedded message name</ MessageTagName>
    <MessageServer>name of processing server</ MessageServer>
    <MessageBody> encoded message body</MessageBody>
</Mercury: Message>
```

图 4-4：端点消息 XML 文档

端点消息包括下列域：

- **MessageVersion**：端点消息版本号。
- **MessageSourcePid**：发送消息对等实体的 ID。
- **EndpointPlacard**：发送消息对等实体的端点公告。
- **MessageTargetPid**：目的对等实体 ID。
- **MessageName**：端点消息内封装的消息名字。
- **MessageServer**：处理消息的服务名字。
- **MessageBody**：将封装的消息编码后的结果，这里使用 Base64 编码方法。

各元素值类型见表 4-3：

表 4-3：端点消息元素说明

元素名称	数量	元素值类型
MessageVersion	1	String
MessageSourcePid	1	UUID
EndpointPlacard	1	endpointPla
MessageTargetPid	1	UUID
MessageName	1	String
MessageServer	1	String
MessageBody	1	char[]

4.2.3 端点消息处理

端点消息的创建和解析由 ***EndpointMessage*** 类定义。图 4-5 给出端点消息处理类图。

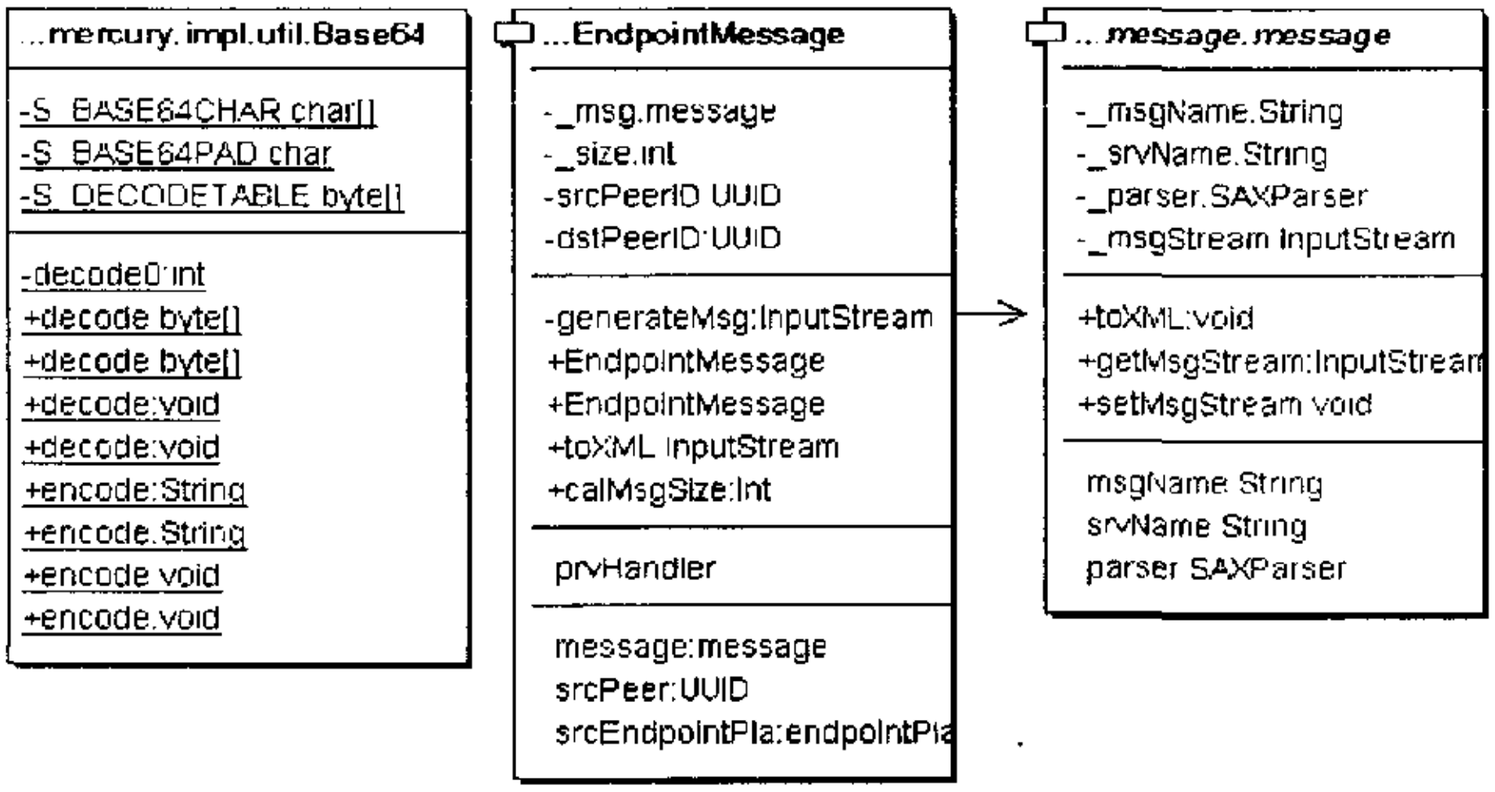

图 4-5：端点消息处理类图

EndpointMessage 类中的重要属性和方法如下：

- ***class prvHandler extends DefaultHandler;*** 内部类定义了解析端点消息 XML 元素的 Handler。
- ***private message _msg;*** 端点消息封装的消息。
- ***private int _size;*** 端点消息大小，因为通过 UDP 发送消息需要知道发送的消息的大小。
- ***private UUID srcPeerID;*** 发送消息的对等实体 ID。
- ***private endpointPla srcEndpointPla;*** 发送消息对等实体的端点公告。
- ***private UUID dstPeerID;*** 发送消息的目的对等实体 ID；
- ***public EndpointMessage(message msg, UUID srcpid, endpointPla srcePla, UUID dstpid);*** 创建端点消息的构造方法。
- ***public EndpointMessage(InputStream in, SAXParser parser);*** 根据接收的数据流构造端点消息。在这个过程中需要使用 ***Base64*** 类提供的解码方法对 ***MessageBody*** 元素内容解码生成基本消息的消息内容。
- ***public InputStream toXML();*** 生成端点消息的 XML 文档字节流。在这个方法中需要使用 ***Base64*** 类提供的编码方法对基本消息的消息内容编码。

- ***public int calMsgSize();*** 计算生成的端点消息的大小。

Base64 类提供了 Base64 编解码方法。我们使用 Base64 编码方法是为了将消息编码成字节流。由于消息本身就是 XML 文档，如果不编码，XML 文档中的标记就会破坏端点消息的结构。关于 Base64 的编码方法见参考文献[17]。

4.3 XML 消息服务调用流程

图 4-6 给出了调用 ***channelImpl.getmessage*** 方法的顺序图。

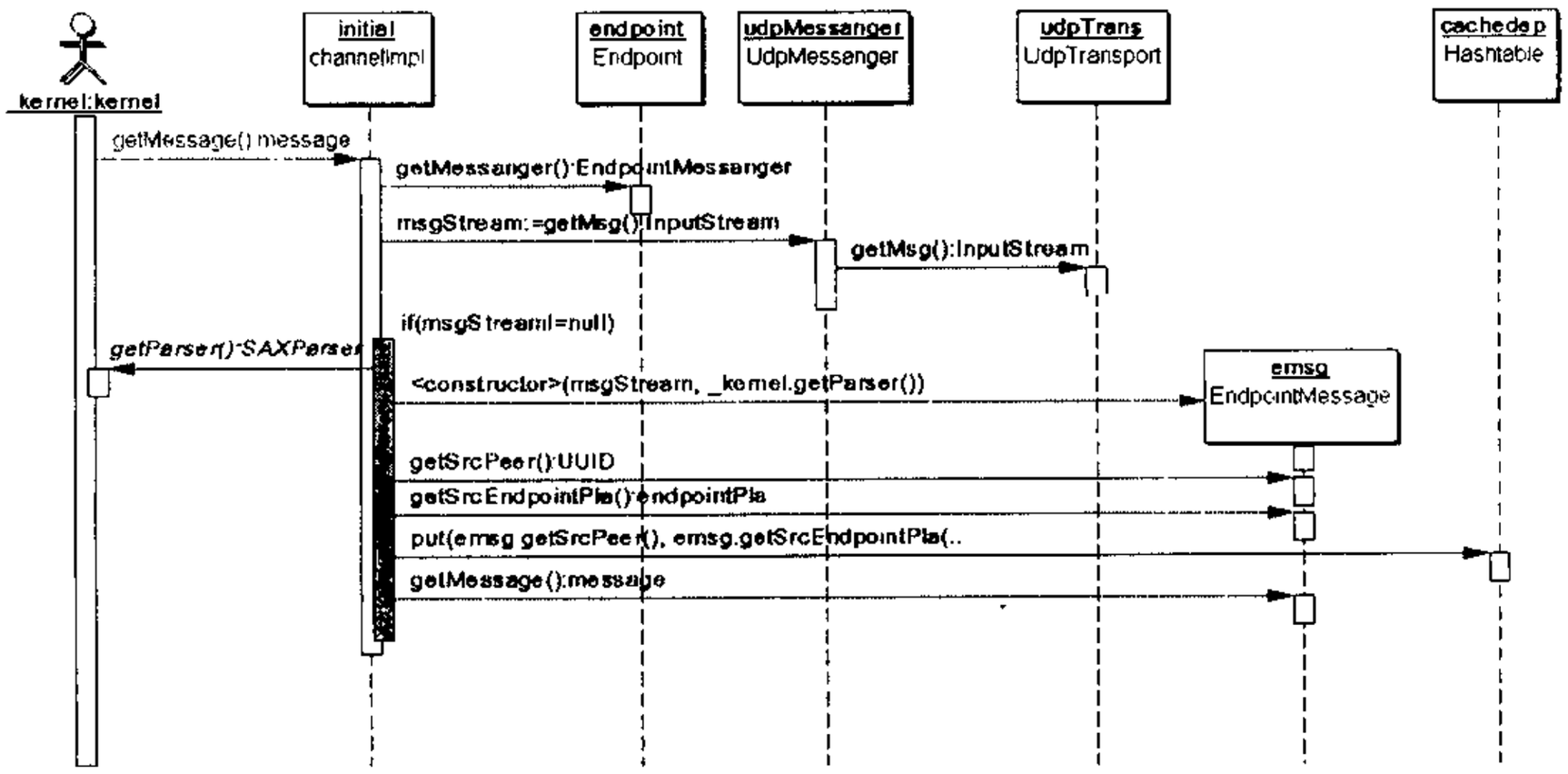

图 4-6：调用 ***channelImpl.getMessage*** 方法顺序图。

图 4-7 给出了调用 ***channelImpl.sendMessage*** 方法的顺序图。

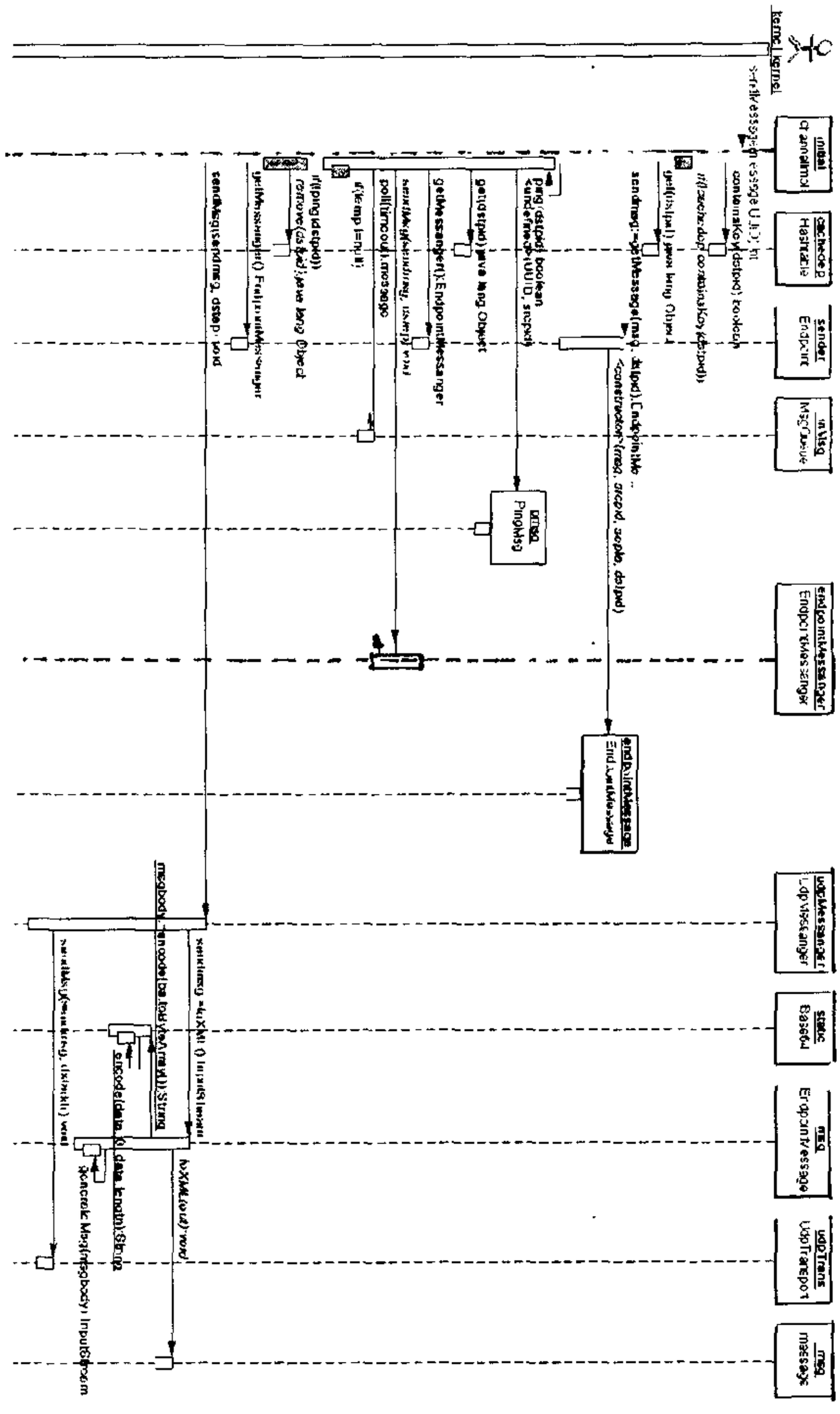

图 4-7：调用 ***channelImpl.sendMessage*** 方法顺序图。

图 4-6 和 4-7 图中没有显示 ***UdpTransport*** 类如何绑定 UDP 协议，下面我们简单阐述一下。

1. 在创建 ***UdpMessanger*** 的对象时，***UdpMessanger*** 构造方法通过 ***UdpTransport(Address addr)***构造方法创建 ***UdpTransport*** 对象。
2. ***UdpTransport(Address addr)***构造方法根据 ***addr*** 中的地址创建 UDP Socket，***DatagramSocket socket = new (addr.port);***

3. 在 *UdpTransport* 类的 *getMsg()*方法中，首先创建一个数据报对象 *DatagramPacket dp = new DatagramPacket(buf, buf.length)*；这里数据缓存 *buf* 是字节数组。为了防止溢出，其大小至少要大于平台中最大消息尺寸。然后调用 *socket.receive(dp);* 接收消息。接收到消息后返回接收到的消息流，*return new ByteArrayInputStream(dp.getdata(), 0, dp.getLength());*

4. 在 *UdpTransport* 类的 *sendMsg(InputStream in, Address dstaddr)*方法中，首先创建数据缓存 *byte[] buf; buf* 从输入流 *in* 中获得。然后创建数据报 *DatagramPacket dpocket = new DatagramPacket(buf, buf.length, dstaddr.address, dstaddr.port);* 最后调用 *socket.send(dpocket);* 发送消息。

TcpTransport 类是为那些需要进行同步消息交换，或者传输大数据的服务和应用提供的，由 *TcpMessanger* 调用。

4.4 小结

本章介绍了 XML 消息服务的结构和实现方法。通过将服务分为两层，系统可以灵活地运行在不同的网络协议上，只需要替换相应的端点绑定。对于其他服务，通道提供了独立于特定网络协议的 XML 通信协议，服务不需要关心底层通信协议。通过复用异步消息通道，平台不需要维护多个通信通道，减少了控制核心工作和资源消耗。对于需要同步消息交互或者传输大量数据的服务或应用，平台提供建立在绑定 TCP 协议的端点上的通道。

第五章 分布式查找服务

不管是什么样的 P2P 系统和应用都离不开数据查找。由于 P2P 系统是分布式的，不存在集中控制或者层次结构，高效的数据查找就成为了 P2P 系统最重要的服务。不能有效的找到数据所在对等实体，P2P 对等实体间直接通信的方式就无从谈起。分布式查找提供的功能很简单但很重要，就是根据提供的数据查找到数据所在对等实体。

P2P 基础平台的分布查找服务提供两个子服务，查找和索引。查找服务显而易见就是提供分布式查找。索引服务为数据建立方便查找的索引。

5.1 分布查找服务体系

现有的各种 P2P 应用采用了多种查找方法，各有特点。例如，Napster 使用固定服务器进行查找，但这会导致单结点失败风险。Gnutella 采用了类似路由算法通过广播方式发出查找请求，这种方法缺乏分布性而且扩散慢，不能对大的网络进行全面查找。目前基于 Consistent Hashing 的分布式查找方法成为研究热点。我们设计 P2P 基础平台分布查找服务的一个原则就是可以容纳多种查找算法，应用可以自行选择需要的算法。

为了实现这一目标，我们在设计时采用了策略模式（Strategy）。这样做的目的是将一系列查找算法封装起来，并且使它们可相互替换，使得算法可以独立于它的客户而变化。图 5-1 给出了查找服务的类图。

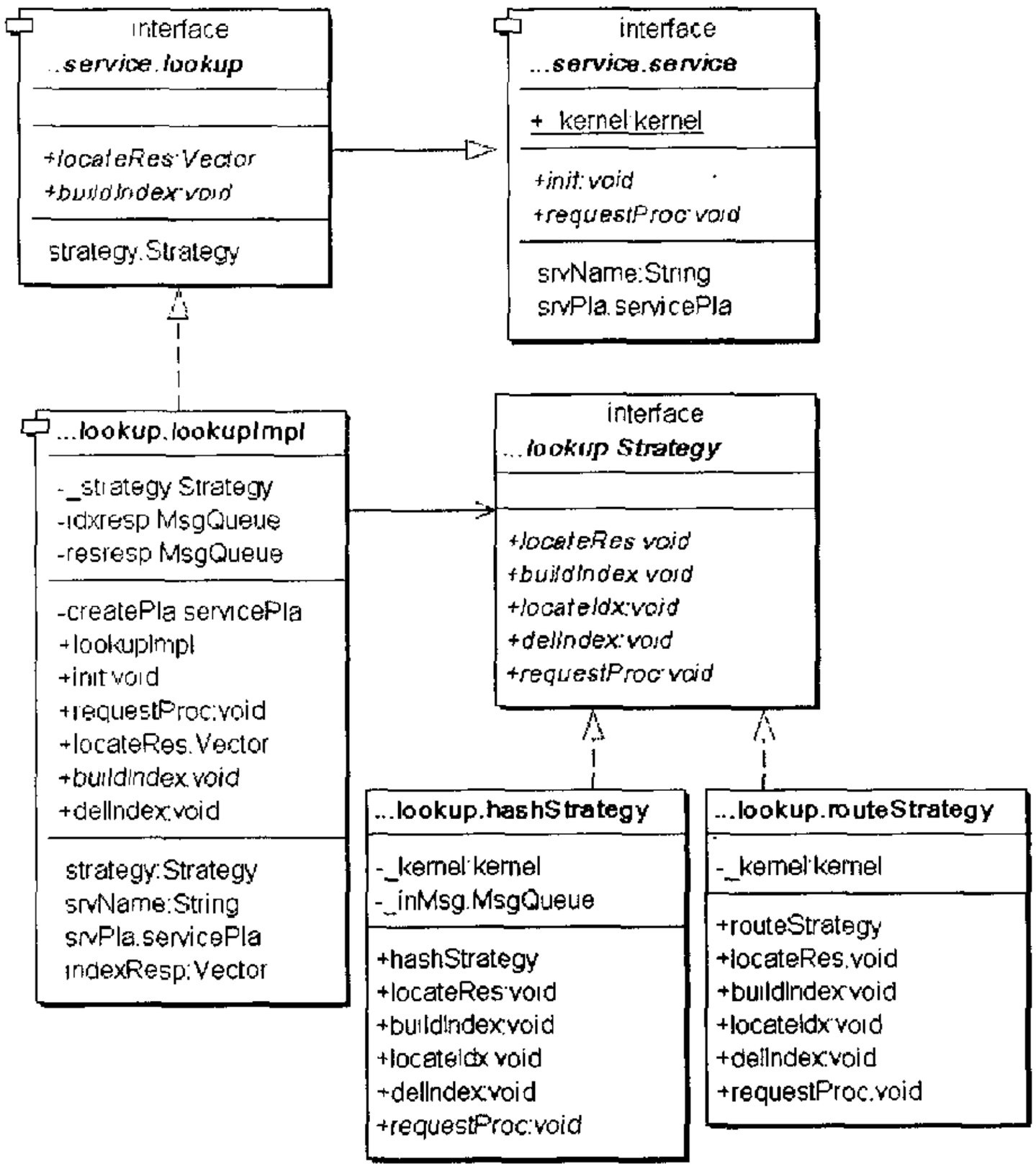

图 5-1：查找服务类图

5.1.1 查找服务使用的接口和类定义

接口 ***Strategy*** 定义了所有支持的算法的公共接口。***lookupImpl*** 使用这个接口调用特定策略的算法。具体策略使用特定算法实现 ***Strategy*** 接口，我们实现了基于一致性 Hash 算法的策略 ***hashStrategy***。平台虽然定义了基于路由算法的策略 ***routeStrategy***。但是由于路由算法在效率方面的缺陷，我们暂时没有实现这个策略。***lookupImpl*** 一方面实现了 ***lookup*** 服务接口，另外是调用不同策略的上下文（**context**）。作为调用某个策略的上下文，***lookupImpl*** 将客户的请求转发给 ***Strategy***。客户通常创建并传递一个具体策略给 ***lookupImpl***，这样客户只和 ***lookupImpl*** 交互。如果可能的话，我们可以提供一系列的具体策略供用户选择。

下面我们分别来看一下各个类或接口的主要属性和方法。

接口 ***Strategy***，支持所有算法的公共接口：

- ***public void locateIdx(String keyword, peerPla originPeer);*** 查找存放给定关键字的对等实体。***originPeer*** 是发起查找的对等实体公告，因为查找可能需要通过中间对等实体完成，所以保存最初的对等实体很重要。找到关键字所在的对等实体后，就可以直接发送消息通知最初发起查找的对等实体。
- ***public void buildIndex(String keyword,String resType, peerPla peer);*** 为 ***keyword*** 关键字表示的资源建立索引。这里的 ***peer*** 是关键字将要存放的目标对等实体。
- ***public void requestProc(message msg);*** 查找服务主要操作都是通过策略完成的，每种策略可能需要定义不同的消息，所以除了两种响应消息外，其他消息都转交给具体策略处理。

类 ***hashStrategy***，实现了一致性 hash 查找算法的策略实现类：

- ***private void locateIdx(String keyword, peerPla originPeer);*** 找到存放关键字为 ***keyword*** 的对等实体。这是分布式查找算法的核心：根据一个数据值，找到数据所在对等实体。将这个查找算法独立出来，是由算法本身特点决定的，为了方便查找和建立索引。如果找到，就向最初发起查找的对等实体发送 ***IndexResponse*** 消息，将关键字所在的对等实体信息发送给最初的对等实体。如果本地没有关键字所在对等实体信息，就向其他对等实体发送 ***IndexRequery*** 消息继续查找。
- ***public void locateRes(String keyword, String resType, peerPla dstPeer);*** 向 ***dstPeer*** 对等实体发送 ***ResourceRequery*** 消息，查找关键字为 ***keyword*** 的类型为 ***resType*** 的资源所在的对等实体信息。
- ***public void buildIndex(String keyword,String resType,peerPla dstPeer);*** 向 ***dstPeer*** 对等实体发送 ***ResourceRegister*** 消息，为关键字为 ***keyword*** 的 ***resType*** 类型的资源在 ***dstPeer*** 对等实体上建立索引。
- ***public void delIndex(String keyword,String resType,peerPla dstpeer);*** 向 ***dstpeer*** 对等实体发送 ***ResourceDelete*** 消息，删除关键字为 ***keyword*** 的 ***resType*** 类型的资源在 ***dstPeer*** 对等实体上建立的索引。

接口 ***lookup***，定义查找服务的提供的方法的接口：

- ***public Vector locateRes(String keyword,String resType);*** 根据关键字 ***keyword*** 查找类型为 ***resType*** 的资源所在对等实体。
- ***public void buildIndex(String keyword,String resType,peerPla peer);*** 为关键字为 ***keyword*** 的类型为 ***resType*** 的资源建立索引。
- ***public void delIndex(String keyword,String resType,peerPla peer);*** 删除关键字为 ***keyword*** 的类型为 ***resType*** 的资源已经建立的索引
- ***public void setStrategy(Strategy stg);*** 设置查找服务所需的策略对象。
- ***public void requestProc(message msg);*** 处理接收到的消息。
- ***private MsgQueue _inMsg;*** 接收到的消息的消息队列。

类 ***lookupImpl***，实现了定义查找服务的 ***lookup*** 接口，同时作为调用特定查找策略的上下文：

- ***private Strategy _strategy;*** 当前使用的查找策略对象。
- ***private MsgQueue idxresp;*** 供其他方法存取 ***IndexResponse*** 消息的消息队列。
- ***private MsgQueue resresp;*** 供其他方法存取 ***ResourceResponse*** 消息的消息队列。
- ***public void setStrategy(Strategy stg);*** 用户设定自己希望使用的查找策略对象。如果用户不通过这个方法设定查找策略，就使用 ***lookupImpl*** 在构造方法中创建的缺省查找策略对象。
- ***public void requestProc(message msg);*** 处理查找服务接收的消息。
- ***public Vector locateRes(String keyword,String resType);*** 将客户的查找请求转交给相应的策略对象处理。
- ***public void buildIndex(String keyword,String resType);*** 将客户的建立索引的请求转交给相应策略对象处理。
- ***public void delIndex(String keyword,String resType,peerPla peer);*** 将客户删除索引的请求转交给相应策略对象处理。

5.1.2 查找服务使用的消息

查找服务定义了六个消息：***IndexRequery***，***IndexResponse***，***ResourceRequery***，***ResourceResponse***，***ResourceRegister***，***ResourceDelete***。

查找服务使用 ***IndexRequery*** 消息查询关键字存放哪个对等实体上。图 5-2 给出了 ***IndexRequery*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: IndexRequery>
    <Mercury: PeerPlacard>
        ...
    </Mercury: PeerPlacard >
    <SourcePeerID>source peer ID</SourcePeerID>
    <Keyword>keyword</Keyword>
</Mercury: IndexRequery>
```

图 5-2：IndexRequery 消息 XML 文档

IndexRequery 消息包括下列域：

- **PeerPlacard**：最初发送查询的对等实体公告。
- **SourcePeerID**：发送这个消息的对等实体 ID
- **Keyword**：要查询的关键字。

各元素值类型见表 5-1：

表 5-1：IndexRequery 消息元素说明

元素名称	数量	元素值类型
PeerPlacard	1	peerPla
SourcePeerID	1	UUID
Keyword	1	String

找到关键字所在的对等实体后，查找服务使用 ***IndexResponse*** 消息将对等实体公告发送给发起查询的对等实体。图 5-3 给出了 ***IndexResponse*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: IndexResponse>
    <Mercury: PeerPlacard>
        ...
    </Mercury: PeerPlacard >
    <Keyword>keyword</Keyword>
</Mercury: IndexResponse>
```

图 5-3：IndexResponse 消息 XML 文档

IndexResponse 消息包括下列域：

- **PeerPlacard**：存放关键字的对等实体公告。
- **Keyword**：查询的关键字。

各元素值类型见表 5-2：

表 5-2：IndexResponse 消息元素说明

元素名称	数量	元素值类型
PeerPlacard	1	peerPla
Keyword	1	String

查找服务使用 ***ResourceRequery*** 消息要求获取具有相应类型和关键字的对等实体向量组。图 5-4 给出了 ***ResourceRequery*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury:ResourceRequery>
    <SourcePid>source peer ID</SourcePid>
    <Keyword>keyword</Keyword>
    <ResType>resource type</ResType>
</Mercury: ResourceRequery>
```

图 5-4：ResourceRequery 消息 XML 文档

ResourceRequery 消息包括下列域：

- **SourcePid**：发起请求的对等实体 ID。
- **Keyword**：查询的关键字。
- **ResType**：关键字代表的资源的类型。

各元素值类型见表 5-3：

表 5-3：ResourceRequery 消息元素说明

元素名称	数量	元素值类型
SourcePid	1	UUID
Keyword	1	String
ResType	1	String

查找服务使用 ***ResourceResponse*** 消息将被查找的资源所在的对等实体公告发给发起查询的对等实体。图 5-5 给出了 ***ResourceResponse*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury:ResourceResponse>
    <Keyword>keyword</Keyword>
    <ResType>resource type</ResType>
    <Mercury: PeerPlacard>
        ...
    </Mercury: PeerPlacard >
</Mercury: ResourceResponse>
```

图 5-5：ResourceResponse 消息 XML 文档

ResourceResponse 消息包括下列域：

- **Keyword**：查询的关键字。
- **ResType**：关键字代表的资源的类型。
- **PeerPlacard**：被查询资源所在的对等实体公告。

各元素值类型见表 5-4：

表 5-4：ResourceResponse 消息元素说明

元素名称	数量	元素值类型
Keyword	1	String
ResType	1	String
PeerPlacard	*	peerPla ·

查找服务使用 ***ResourceRegister*** 消息将资源的关键字放到相应对等实体上。图 5-6 给出了 ***ResourceRegister*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury:ResourceRegister>
    <Keyword>keyword</Keyword>
    <ResType>resource type</ResType>
    <Mercury: PeerPlacard>
        ...
    </Mercury: PeerPlacard >
</Mercury: ResourceRegister>
```

图 5-6：ResourceRegister 消息 XML 文档

ResourceRegister 消息包括下列域：

- **Keyword**：资源的关键字。

- **ResType**：资源的类型。
- **PeerPlacard**：资源所在的对等实体公告。

各元素值类型见表 5-5：

表 5-5：ResourceRegister 消息元素说明

元素名称	数量	元素值类型
Keyword	1	String
ResType	1	String
PeerPlacard	1	peerPla

查找服务使用 ***ResourceDelete*** 消息通知将资源的关键字从存放它的对等实体上删除。图 5-7 给出了 ***ResourceDelete*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury:ResourceDelete>
    <Keyword>keyword</Keyword>
    <ResType>resource type</ResType>
    <Mercury: PeerPlacard>
        ...
    </Mercury: PeerPlacard >
</Mercury: ResourceDelete>
```

图 5-7：ResourceDelete 消息 XML 文档

ResourceDelete 消息包括下列域：

- **Keyword**：资源的关键字。
- **ResType**：资源的类型。
- **PeerPlacard**：资源所在的对等实体公告。

各元素值类型见表 5-6：

表 5-6：ResourceDelete 消息元素说明

元素名称	数量	元素值类型
Keyword	1	String
ResType	1	String
PeerPlacard	1	peerPla

5.1.3 查找服务调用流程

查找服务的提供的 ***locatRes*** 和 ***buildIndex*** 方法的调用过程基本一致，首先通过策略提供的 ***locatIdx*** 方法找到存放关键字的对等实体，然后再调用策略的 ***locateRes*** 或 ***buildIndex*** 方法从这个对等实体取到关键字表示的资源的信息或者

将关键字表示的资源存放在该对等实体上。图 5-8 给出了控制核心调用查找服务的 ***locateRes*** 方法的顺序图。

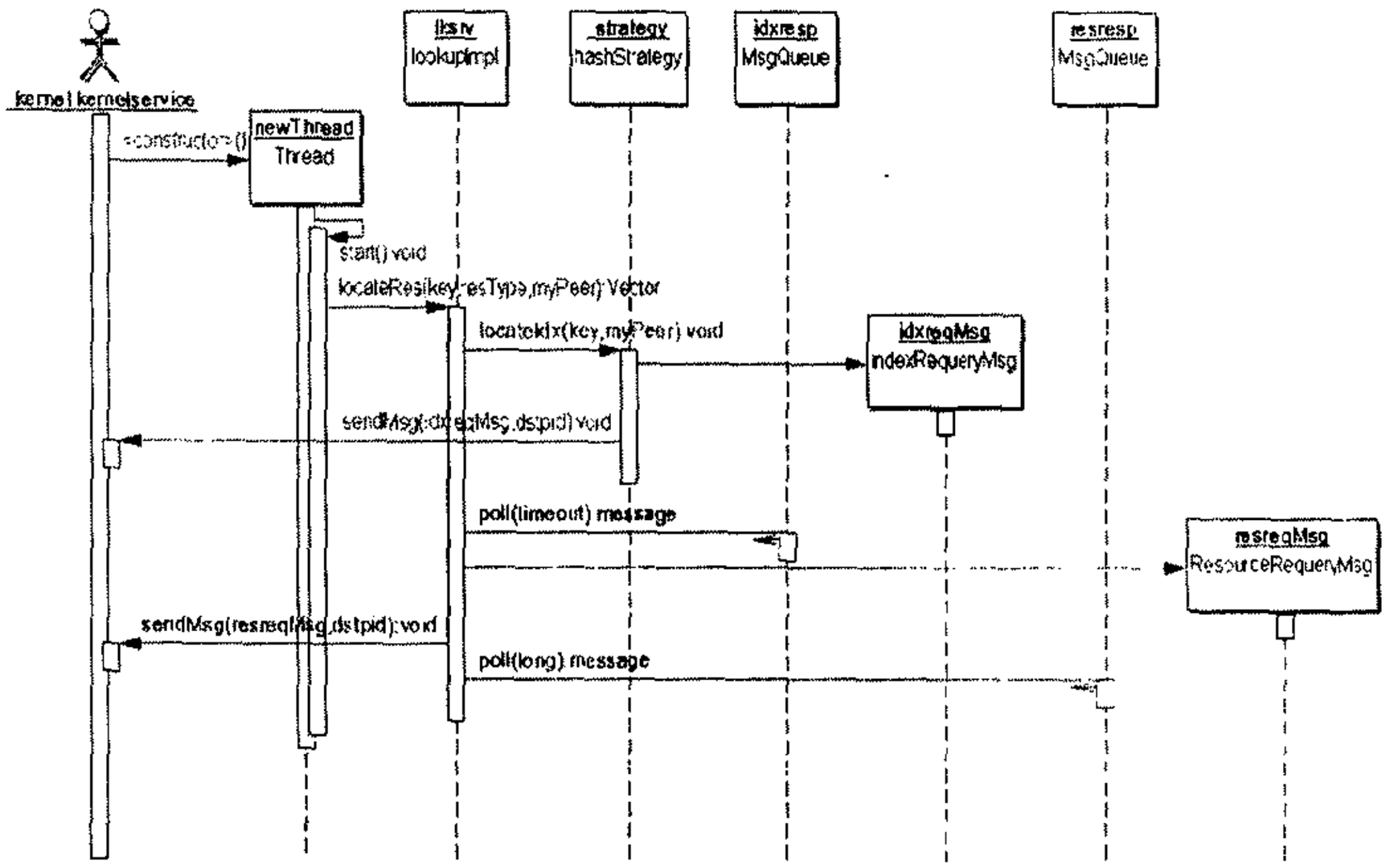

图 5-8：调用查找服务 ***locatRes*** 方法的顺序图

由于查找服务的各项操作由具体策略负责，除了两种响应消息，服务在接到消息后直接将消息转交具体策略处理，图 5-9 给出处理消息的顺序图。

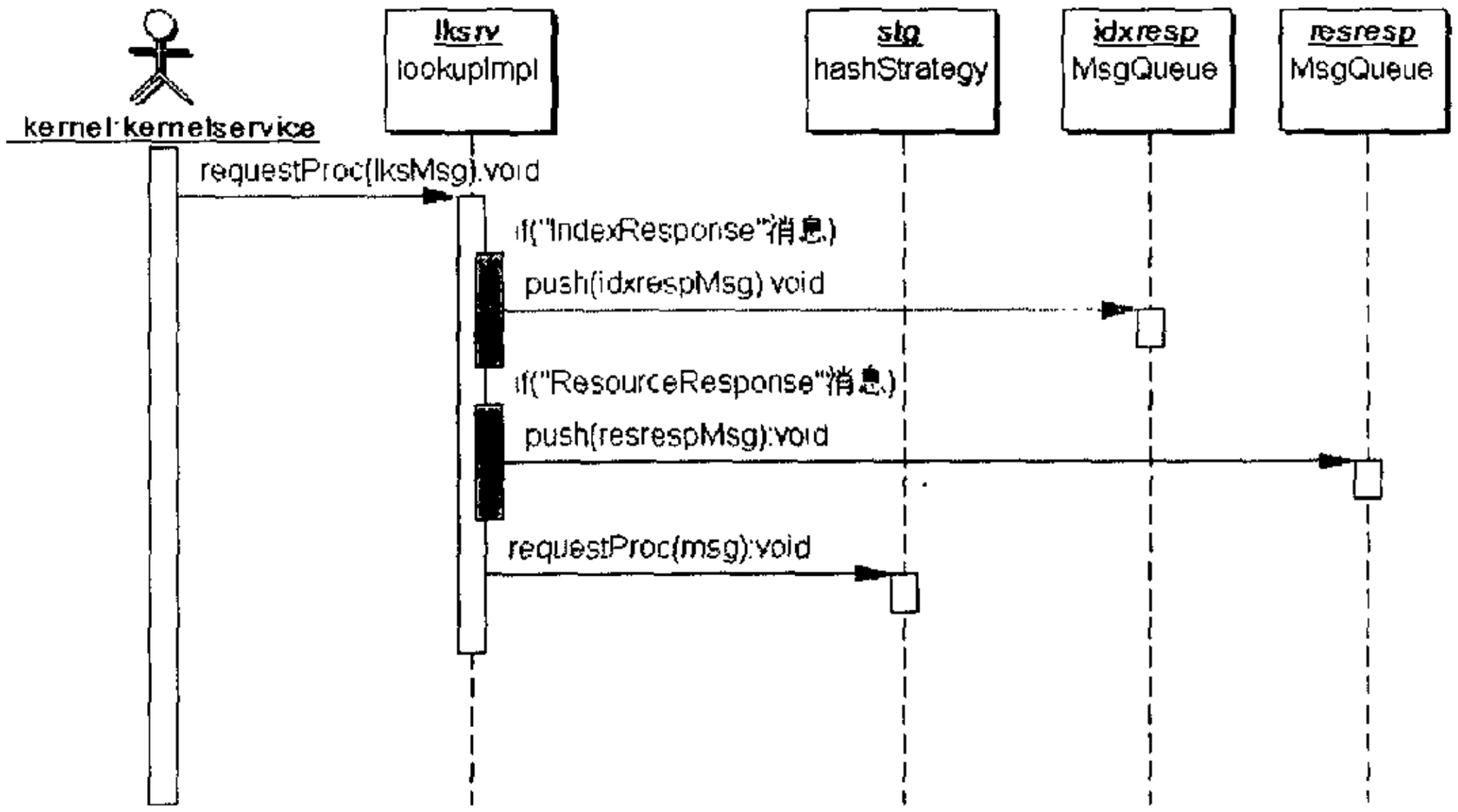

图 5-9：查找服务处理接收的消息的顺序图。

5.2 基于一致性 Hash 和 B+树的分布式查找算法

前面我们已经提到在 P2P 网络环境中，高效分布式查找算法的重要性。为了解决这一问题，我们设计了一个在动态 P2P 网络中（对等实体变化较快，经常加入、离开网络）进行查找的分布式协议。

协议提供两个功能：将关键字映射到相应对等实体；根据给定关键字找到所在对等实体。如果某个对等实体可以提供数据供其他对等实体访问，协议将数据的关键字映射到相应的对等实体上，其他对等实体就可以根据关键字访问这些数据。协议利用一致性 Hash（consistent hashing）技术的特性将关键字分派到相应对等实体。因为使用一致性 Hash 技术可以使每个对等实体保存的关键字数量基本相同，而且当对等实体加入和离开系统时只需要移动很少一部分关键字，所以在负载均衡方面这种技术具有一定优势。

有关一致性 Hash 的最初研究假定每个对等实体都保存有系统中大部分对等实体的信息，这对于拥有大量状态可变对等实体的动态网络环境来说是不可行的。因此，我们将协议中的所有提供关键字索引的对等实体组织成 m 阶 B+树的逻辑视图。在稳定的状态下，协议中每个对等实体只保存 $O(\log_{\lceil m/2 \rceil} N)$数量级的其他对等实体信息，执行查找也只需要 $O(\log_{\lceil m/2 \rceil} N)$数量级的消息数。当对等实体加入或离开系统时，协议负责维护每个对等实体的“路由”信息，这项操作需要发送 $O(\log_{\lceil m/2 \rceil}{}^2 N)$数量级的消息。

5.2.1 基本协议

我们在本节中描述协议的基本原理，如何找到关键字、新对等实体如何加入系统以及当对等实体离开系统时如何保证逻辑视图的正确性。

协议的核心是将对等实体的逻辑地址（也就是端点地址）通过 hash 函数映射到一个值空间，然后将数据的关键字也映射到相同的值空间。那么根据关键字的 hash 值，协议就可以找到存放关键字的相应对等实体。查找一个关键字的时候，协议使用关键字的 hash 值作为索引找到对等实体，再从这个对等实体取得关键字表示的数据信息。

5.2.1.1 一致性 Hash 原理

一致性 Hash 使用基本的 hash 函数（例如 SHA-1）为每个对等实体和关键字分配一个 i 位的*标示*值。对等实体的*标示*通过 Hash 对等实体的逻辑地址获得，关键字的*标示*通过相同的 hash 函数产生。*标示*的长度 i 必须足够大以保证两个对等实体或关键字 Hash 结果相同的可能性忽略不计。

一致性 Hash 按照下面的方法将关键字分配到相应对等实体上：*标示*构成一个 m 阶的 B+树。关键字 k 的*标示*分配到标示值空间中第一个大于等于 k 的*标示*代表的对等实体上，这个对等实体称为关键字 k 的后继。

一致性 Hash 减少了对等实体加入或离开是造成的破坏。当对等实体 n 加入系统时，以前分配到 n 的后继上的某些关键字就要分配到 n 上。如果对等实体 n 离开，它上面分配的关键字全部重新分配到它的后继上。然而分配到其他对等实体上的关键字不需要做任何改动。

下面关于一致性 Hash 的结论在介绍一致性 Hash 的论文中证明过，见参考文献[13]，[14]。

定理一：*对于任意 N 个对等实体的集合和 K 个关键值的集合，有很大的可能可以得到：*

1. *每个对等实体只存放最多（1+ ε）K/N 个关键值。*
2. *当第$(N+1)^{st}$ 对等实体加入或离开系统时，只有 O(K/N)的关键字需要改变存放的位置。*

如果一致性 Hash 按照上面所述实现，那么 $\varepsilon = O(\log_{\lceil m/2 \rceil} N)$

由于在 P2P 环境下，对等实体和关键字都是具有很大的随机性，只要不存在故意破坏者，上面的结论是成立的。因此对等实体和关键字的分布不会出现很大的不平衡。不过也许会有故意破坏者试图选择一些关键字 Hash 成相同的*标示*来破坏平衡。在一致性 Hash 的有关论文中使用“k-universal hash function”使得不随机的关键字也能映射成随机的*标示*。

我们在这里选择标准的 SHA-1 算法作为基本 hash 算法。这样我们的协议就具有可确定性，Hash 结果随机的可能性变得不再重要。虽然可能找到一些关键字会在 SHA-1 算法下产生冲突，但是这是很难做到的，这由 SHA-1 算法本身保证。

5.2.1.2 Hash 值空间的组织

为了让每个对等实体都只保存少量的其他对等实体信息并且可以高效的找到所需要的对等实体，我们将对等实体*标示*（逻辑地址的 hash 值）组织成 B+树的逻辑视图。m 阶 B+树有如下特点：

- 具有 n 棵子树的节点有 n 个关键字。
- 每个节点至多有 m 棵子树。
- 若根节点不是叶子节点，则至少有两颗子树。
- 除根节点之外的所有非终节点至少有⌈m/2⌉棵子树。
- 叶节点包含所有了对等实体，所有关键字都保存在叶节点中的对等实体上。
- 所有中间节点作为索引，只保存其子树根节点中具有最大*标示*值的对等实体。

图 5-10 给出了 B+树的逻辑视图。

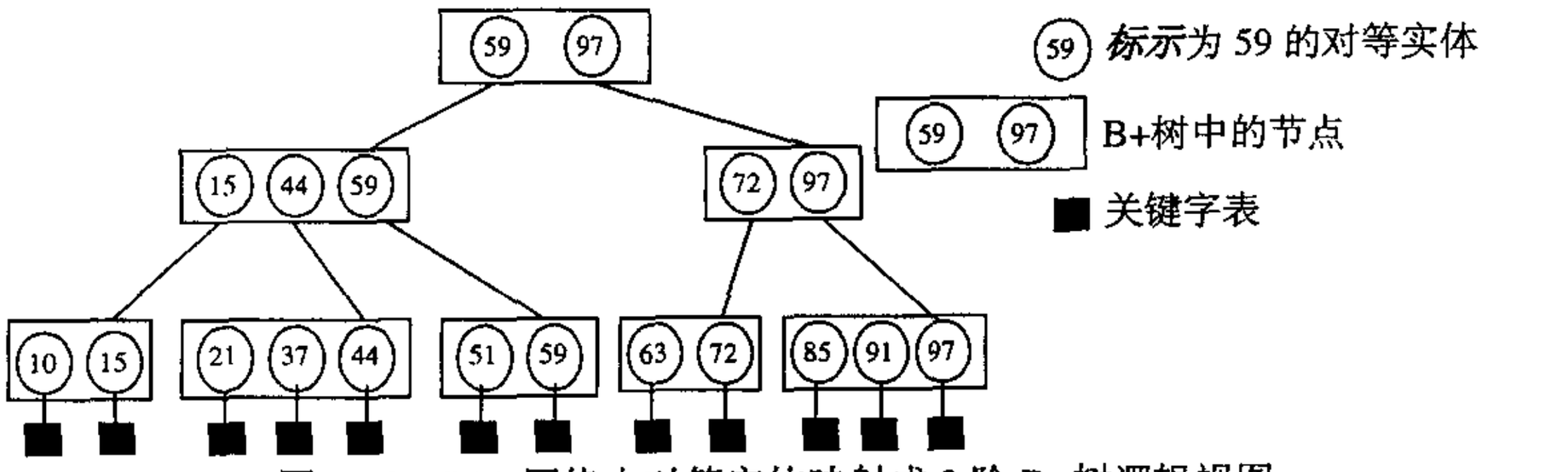

图 5-10：P2P 网络中对等实体映射成 3 阶 B+树逻辑视图

协议中的每个对等实体都要保存同属于 B+树一个叶节点的其他对等实体信息、子树根节点（根据 B+树的定义，就是这个对等实体本身。对于叶节点中的对等实体就为 null）以及父节点（由 B+树特点可知，对等实体的父节点指向它所在节点中*标示*最大的对等实体）。每个对等实体保存的关键字表存放在叶节点中的对等实体上。根据对等实体保存的信息，协议可以构造和维护 B+树的逻辑视图。图 5-11 描述了 B+树对等实体的类定义。

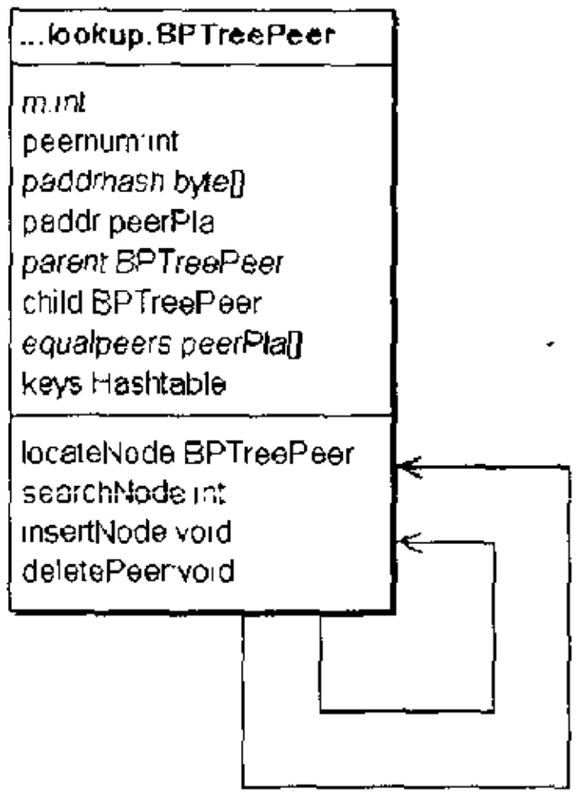

图 5-11：B+树中对等实体定义

从对等实体的定义中可以看出这是一个递归类，如果对等实体同时作为 B+树的中间节点，那么父节点就是它自己，直到父节点不是自己或者为空（这时它处于根节点中）。图 5-12 给出图 5-10 中对等实体 59 和 44 的定义的例子。

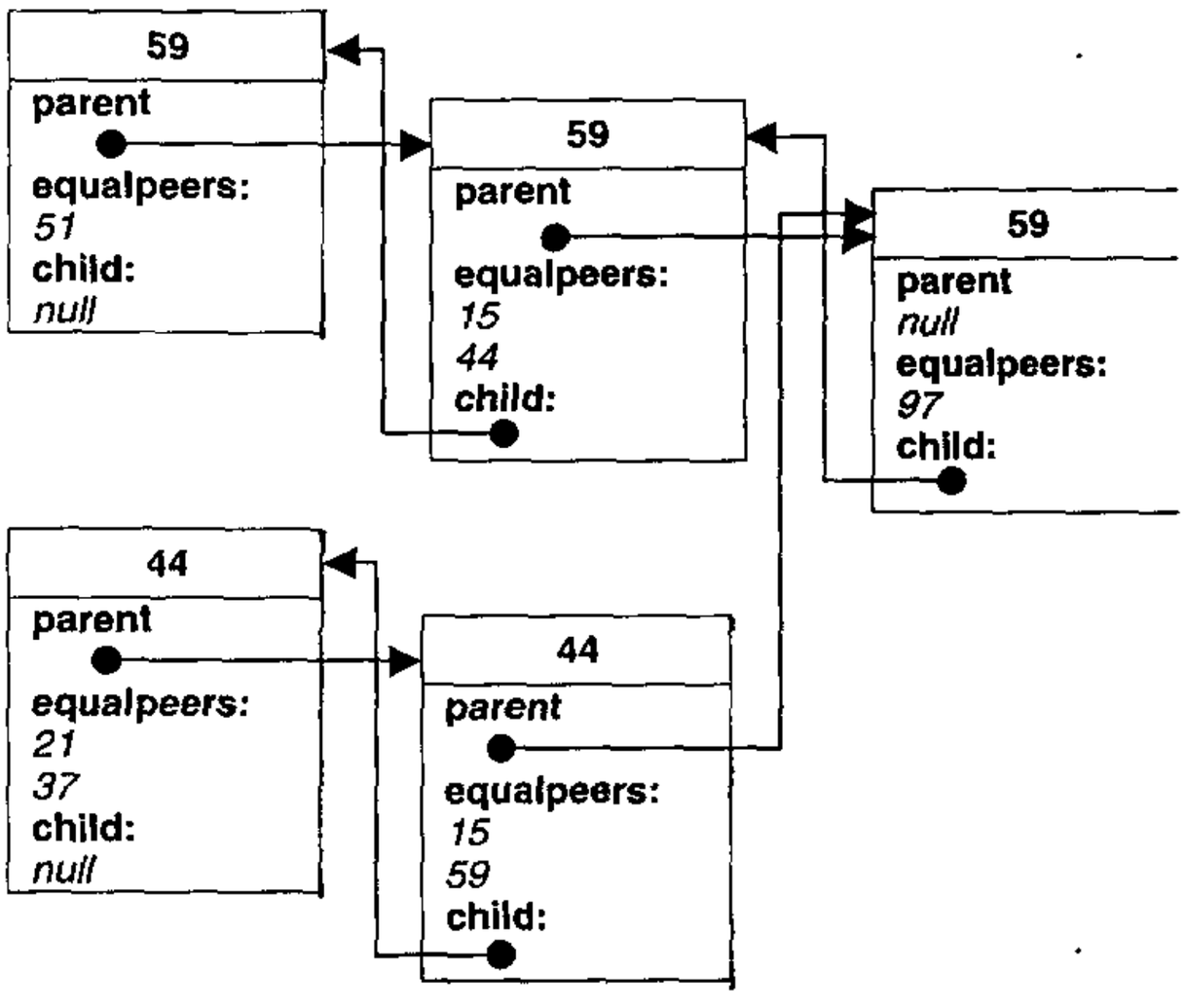

图 5-12：对等实体嵌套实例

根据对等实体保存的信息，我们来看协议如何实现下列 B+树的基本算法。

- ***BPTreePeer locateNode (peeraddr peer);*** 找到自己与*peer*同在的节点，这个方法是同时属于B+树中间节点的对等实体需要的，因为对等实体的定义是嵌套的，接到一个对等实体查寻请求消息后，首先要确定哪个中间节点应该接到这个消息，这样才能保证将消息再转发到正确的父节点或者子节点去。实现示例如下：

```
p=this;
while((addr∉p.equalpeer)&&(p.parent.paddr==p.paddr))
   p=p.parent;
if(addr∈p.equalpeer)
   return p;
return NULL;
```

- ***int searchNode (byte[] queryhash);*** 根据参数给定的hash值，在自己所在的节点中找一个可能存放这个值的节点。如果返回正数表示参数所给的hash值在***equalpeers[i]***节点的子树中，如果返回负数则提交父节点进行查找。实现代码如下：

```
If (hashing(equalpeer[i-1])<equryhash< = hashing(equalpeer[i]))
      return i;
return –i;
```

- ***void insertNode (peerPla peer);*** 插入新对等实体后，对 B+树的节点进行调整。如果加入新对等实体后，节点包含的对等实体数小于等于 m 则不需要对节点进行调整。否则，将节点分割为两个新节点，每个节点含有⌈(m+1)/2⌉和⌊(m+1)/2⌋个对等实体，同时将新分裂出来的左节点中*标示*最大的对等实体插入到父节点，这是一个递归过程，有可能直到创建新的根节点为止。
- ***void deletePeer (peerPla peer);*** 将参数给定的对等实体从 B+树中删除。首先将对等实体从其所在叶节点中删除，如果这个对等实体同时还位于树的中间节点内，则继续从中间节点删除它。如果删除后对等实体所在的节点中对等实体数目小于⌈m/2⌉，进行相邻节点合并。

5.2.1.3 关键字的查找与分配

我们这里讲的 B+树是对等实体*标示*的逻辑视图，每个对等实体不知道这棵树的整体。因此，协议执行操作的时候可能需要其他多个中间对等实体帮助传递消息。协议通过 ***lookup*** 方法查找关键字所在对等实体，图 5-13 给出 ***lookup*** 方法实现代码。

```
void lookup ((IndexRequeryMsg) idxreqMsg)
{
   int dstpeer = searchNode(sha.update(idxreqMsg.getKey()));
   if (dstpeer>0)
      _kernel.sendMessage(new IndexResponseMsg(equalpeers[dstpeer]));
}
void receivedQuery (byte[] queryhash, peerPla originpeer, peerPla saddr)
{
   BPTreePeer np=this.locateNode(saddr);
   if (np.child==NULL)
      while (np.parent!=NULL)
            np=np.parent;
      if ((int i=np.searchNode(queryhash))>0)){
            np.sendResult(originpeer, np.equalpeer[i]);
            return;}
      if ((-i)!=np.peernum)
         np.sendQuery(equalpeer[-i].paddr, queryhash, originpeer);
      while(np.child!=NULL){
            np=np.child;
            if ((i=np.searchNode(queryhash))>0)
                  np.sendResult(originpeer, np.equalpeer[i]);
            if (-i!=np.peernum)
                  np.sendQuery(equalpeer[-i].paddr, queryhash, originpeer);
      }
      np.sendResult(originpeer, np.equalpeer[-i]);
}
```

图 5-13：***lookup*** 方法实现代码

5.2.1.4 对等实体加入

随着新的对等实体加入网络，需要对 B+树进行调整。同时，分布在对等实体上的关键字也需要调整，保证关键字分布在正确的对等实体上。对等实体的插入也是构造 B+树的过程，当一个新对等实体加入网络时，我们在配置文件中指定缺省的对等实体作为加入网络的引入点。新对等实体通过引入点发起对自己*标示*的查询，确定插入位置。B+树的插入总是从叶节点开始的。根据插入位置将自己插入到 B+树的相应节点中，如果节点中的对等实体数大于 m，节点分裂成两个节点，每个节点包含的对等实体个数分别是$\lceil (m+1)/2 \rceil$和$\lfloor (m+1)/2 \rfloor$，并且它们的父亲节点中应该同时包含这两个节点中的最大*标示*对等实体。图 5-14 给出了插入节点的代码，限于篇幅代码中很简单的和标准的 B+树操作就不一一列出了，

主要集中在节点内部属性的调整上。

```
void insertPeer ( BPTreePeer intropeer )
{
    ... 找到插入位置对等实体 destpeer;

    //将对等实体插入到插入位置对等实体所在 B+树的叶节点中。
    insertNode ( destpeer.pid, this.pid );

    //获取插入对等实体中其他对等实体信息，这些信息用来配置对等实体属性。
    getNodeInfo ();
    return;
}

void insertNode ( UUID destpid, UUID newpid )
{
    BPTreePeer destpeer = getPeerInfo ( destpid );
    if ( destpeer.keynum < m )
        将自己插入 equalpeer 向量中并保持向量顺序;

    /*根据我们的查找算法除非新对等实体的标示比原来对等实体的标示都大，其他情况下插入的对等实体不是它所在 B+树节点内的标示最大的对等实体，不需要调整树的非终节点。*/

    if ( this.pidhash < destpeer.equalpeer[destpeer.keynum] )
        通知 destpeer.equalpeers 向量中所有对等实体，加入了新对等实体;
        return;

    /*如果插入的对等实体是网络中标示最大的对等实体，那么插入的位置就是原来 B+树中最大标示的对等实体右边。*/
    通知 destpeer.equalpeers 向量中所有对等实体加入了新对等实体;

    /*这些对等实体接到通知后会自动替换父节点中对等实体向量。*/
    逐层调整 destpeer 的父节点，替换新对等实体，通知所在 B+树节点内的其他对等实体;

    if ( destpeer.keynum >= m )
        通知 destpeer.equalpeers 向量中所有对等实体;

    如果插入新对等实体后，节点中对等实体数目大于 m，每个收到通知的对等实体会自动将自己的 equalpeers 向量截断并抛弃其中一半。位于第⌊(m+1)/2⌋位置的对等实体比较特殊，这个对等实体由于 B+树节点分割的原因而成为所在节点中最大的对等实体，所以它又成为非终节点的成员。

    /*将这个对等实体 插入到父节点中。*/
    insertNode ( equalpeer[⌊(m+1)/2⌋]. parent, equalpeer[⌊(m+1)/2⌋] );

    /*如果出现新对等实体标示最大这种情况，就同刚才讲过的情况一样，需要调整到 B+树的根节点。*/
}
```

图 5-14：插入对等实体描述

插入新的对等实体后，已分配到它的后继上的关键字就可能需要调整。根据我们的关键字分配方法可以看到关键字总是分配到标示比它的标示大且差距最

小的对等实体上，而且只分配在叶节点上。如果关键字的*标示*大于节点中的最大*标示*，关键字分配到*标示*最大的对等实体上。当插入一个新对等实体的时候，可能它后继上的关键字*标示*更接近新加入的对等实体*标示*，这时就要把这个关键字分配到新的对等实体上。我们在插入对等实体的时候首先得到的就是插入位置，这个位置恰好就是有可能要调整的关键字所在对等实体的位置。所以在插入对等实体的时候可以先通知插入位置对等实体新对等实体的*标示*，这样插入位置对等实体就可以判断哪些关键字需要分配到新对等实体上。关键字的分配不会影响B+树的调整，因为所有关键字都分配在叶节点中的对等实体上。

5.2.1.5 对等实体离开

当一个 P2P 对等实体离开网络时，它要调用对等实体删除方法。在本节中我们先不讨论对等实体没有调用删除方法而意外失败的情况，这种情况留在下节讨论。B+树的删除操作也是从叶节点开始的，如果删除后 B+树节点中对等实体的数量$\geq\lceil m/2\rceil$，直接将对等实体从 B+树节点中删除，不需要对树进行调整。如果删除后节点中对等实体的数量$<\lceil m/2\rceil$，需要将 B+树节点与相邻节点合并。具体方法，参照标准 B+树算法和上小节描述的算法不难得到。同理，对等实体离开时也要对分配给它的所有的关键字重新分派，这些关键字分派到对等实体的右邻居上。

5.2.2 分布式查找协议扩展

上一节讨论了在单个对等实体加入网络和正常离开情况下，协议的处理过程。但是在实际应用中多个对等实体同时加入网络和突然意外失败而没能执行对等实体删除操作的情况有可能发生。在这一节里，我们扩展基本协议来处理这些问题。

5.2.2.1 多对等实体同时插入网络

在 5.2.1.4 节中我们提到了新的对等实体插入首先要找到合适的插入位置，如果对等实体插入后，B+树节点中对等实体个数>m，还要对 B+树节点进行分割并调整其父节点。如果两个或多个对等实体同时插入一个 B+树节点，这些新对等

实体通知节点中已有的对等实体，很容易造成新对等实体中 ***equalpeers*** 向量内容不一致，而且问题会随着调整非终节点而扩散。为了解决这个问题，可以规定每个 B+树节点同一时间只能执行一个对等实体插入。但是在网络环境下很难保证能够实施这个规定。我们在查找协议中采用冲突检测与解除策略实现多对等实体同时插入。

根据 5.2.1.4 节中的对等实体插入算法，新对等实体首先找到插入位置，然后通知在同一节点中的所有对等实体。当这些对等实体接到通知，他们就将自己锁定，然后发回 ***locked*** 消息。如果这些对等实体已经被锁定则发回 ***colliding*** 消息，这个消息中带有锁定这些对等实体的新对等实体的信息。当对等实体接收到 ***colliding*** 消息，它可以使用消息中的信息刷新自己的 ***equalpeers*** 向量，然后发回 ***refreshed*** 消息。接到 ***refreshed*** 消息的对等实体发回 ***ack*** 消息。一旦新对等实体收到全部 ***ack*** 和 ***locked*** 消息，它就可以调用 ***insertNode*** 方法插入 B+树节点中。最后，新对等实体发送 ***unlock*** 消息给被它锁定的对等实体。图 5-15 给出了这一过程的 Petri 网。

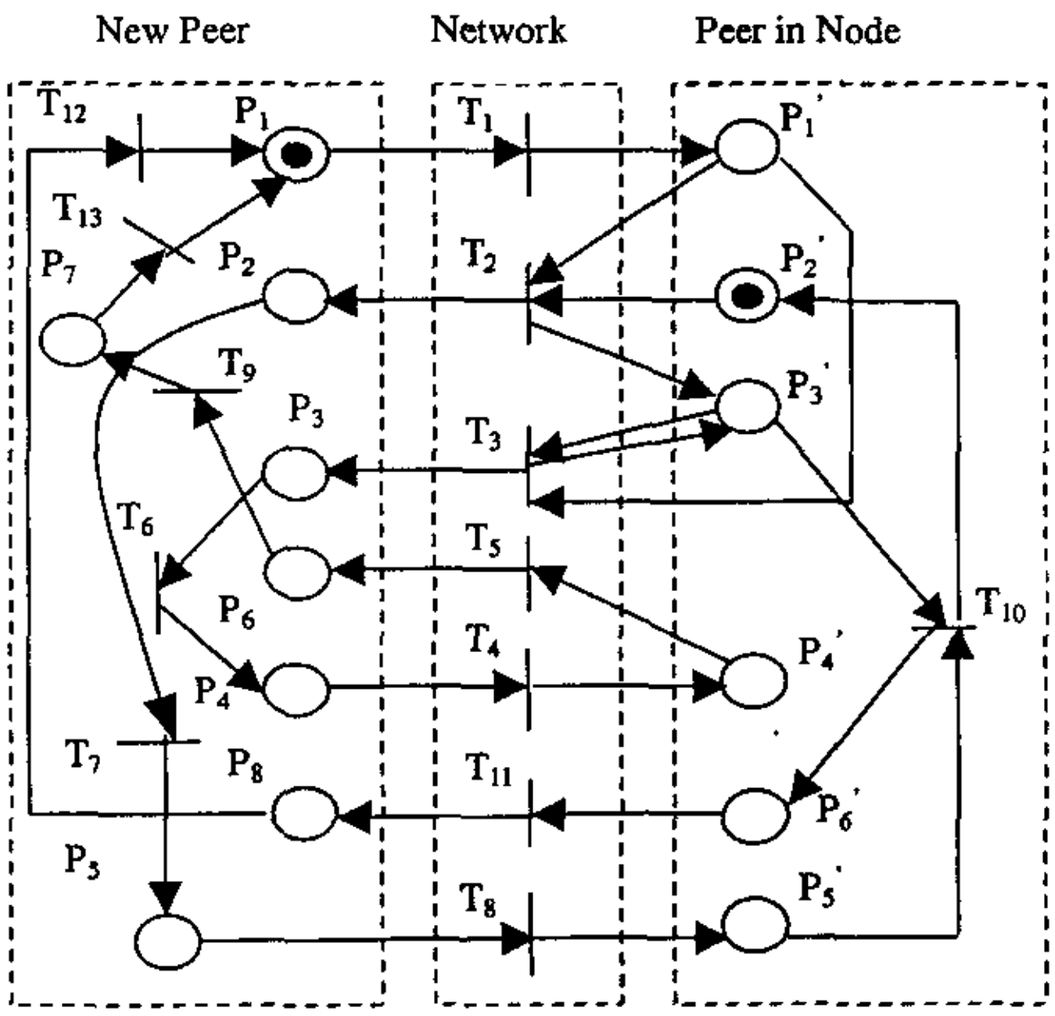

图 5-15：处理多对等实体同时插入的 Petri 网

表 5-7：Petri 网中库所和变迁的定义：

库所	说明	变迁	说明
P_1	初始状态	T_1	发送 ***notify*** 消息
P_2	插入就绪	T_2	发送 ***locked*** 消息
P_3	刷新就绪	T_3	发送 ***colliding*** 消息
P_4	刷新完毕	T_4	发送 ***refreshed*** 消息
P_5	插入完毕	T_5	发送 ***ack*** 消息
P_6	节点信息一致	T_6	刷新 ***equalpeers*** 向量
P_7	插入完毕	T_7	插入新对等实体
P_8	解除锁定完成	T_8	发送 ***unlock*** 消息
P_1'	等待通知	T_9	插入新对等实体
P_2'	空闲	T_{10}	解除锁定
P_3'	锁定	T_{11}	发送 ***ack*** 消息
P_4'	节点消息一致	T_{12}	所有操作完成
P_5'	锁定完成	T_{13}	所有操作完成
P_6'	解除锁定完成		

下面我们证明这个协议不会死锁。

证明：

图 5-15Petri 网定义：Petri 网 $M=(P, T, I, O, \mu)$, $P=\{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_1', P_2', P_3', P_4', P_5', P_6'\}$; $T=\{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}\}$; I: $T->P^\infty$，库所到变迁的输入映射；O: $T->P^\infty$，变迁到库所的输出映射；M 的记号向量，μ: P->N, N={0, 1}, $\mu=(\mu_1, \mu_2 \ldots \mu_n)$, $n=|P|$, $\mu_i \in N$。如果库所 P_i 内没有 token 时，$\mu_i=0$，否则$\mu_i=1$。图 5-14 中的 Petri 网初识记号μ^0是 (1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0) or (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0)。从μ^0开始我们可以得到 Petri 网 M 的可达树，见图 5-16。

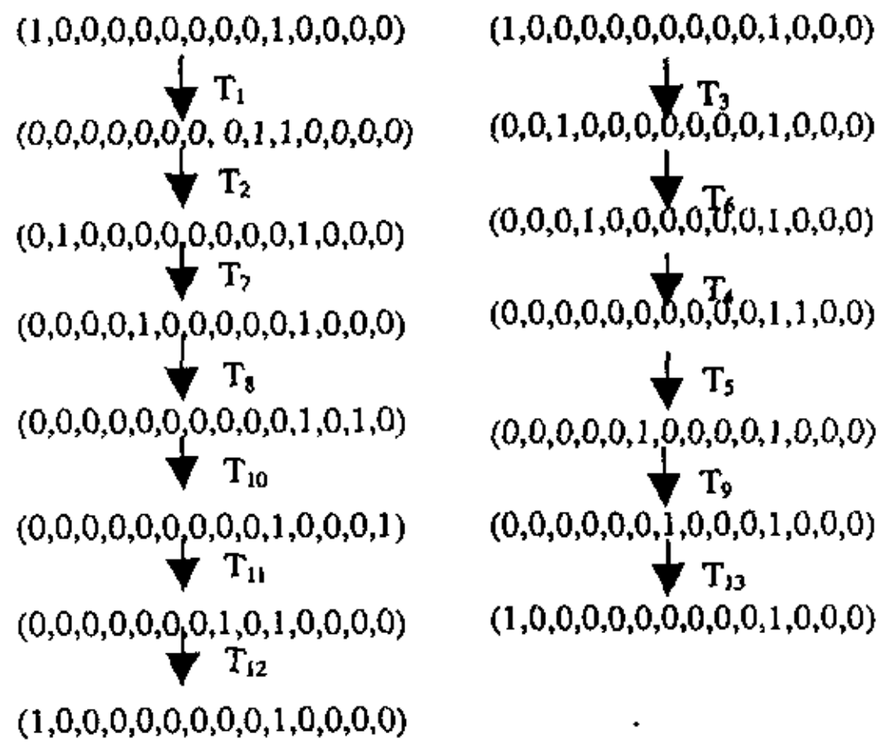

图 5-16：Petri 网 M 的可达树

从图 5-16 的可达树可以看出，Petri 网 M 从初始记号出发经过所有变迁，最终回到原来的初始状态。可达树的两个分支的变迁序列没有交集，且覆盖所有变迁。所有变迁都是可触发的，不存在因竞争而使某个变迁不能触发，所以不会出现死锁。

5.2.2.2 对等实体失败处理

对等实体失败分为两种情况，如果对等实体只处于 B+树的叶节点中，则该对等实体的失败不会导致 B+树节点的丢失。同一节点中的其他对等实体在查找时就可以发现问题并可以很容易的从 ***equalpeers*** 向量中删除失败对等实体，如果所在 B+树节点内对等实体个数小于$\lceil m/2 \rceil$就合并相邻节点，这与对等实体的删除操作完全一样。不过这种情况下丢失的对等实体上的关键字不可恢复，只有关键字代表的资源所在对等实体重新分派关键字。当然，这个对等实体可以周期性刷新关键字的分派，这只是个策略问题。

如果失败对等实体也位于 B+树的非终节点中，这个对等实体的失败会导致其所在的 B+树的叶节点丢失。为了解决这个问题，我们扩展对等实体类的定义，增加 ***BPTreePeer nextNode[]***向量组，其中每个元素都是右邻居叶节点中的对等实体，B+树右边最后一个叶节点中对等实体的 ***nextNode*** 向量组指向 B+树左边第一个叶节点。在对等实体插入的时候很容易得到下一个叶节点中对等实体。在叶节点分裂的时候左边的叶节点中对等实体通知自己的左邻居刷新 ***nextNode*** 向量组，分裂出来的右边叶节点中的对等实体不变。相邻叶节点合并时，将左边节点内对等实体的 ***nextNode*** 向量组修改为右边节点内对等实体的 ***nextNode*** 向量组即可。若一个位于非终节点的对等实体失败了，在查找时会发现这个对等实体指向的叶节点丢失。通过 ***nextNode*** 向量组可以找到丢失的叶节点，然后对这个丢失对等实体做删除操作即可。与第一种情况一样，丢失的对等实体上分配的关键字也是不能恢复的。同样可以采用周期刷新解决。

5.2.2.3 协议分析

对于由 N 个对等实体构成的 M 阶 B+树，协议中每个对等实体所需存储量 K 小于对等实体所在叶节点中最多对等实体数+相邻叶节点中最多对等实体数+ B+树的高度与非终节点中最多对等实体个数 M 之积：$K<2M+M*(\log_{\lceil m/2 \rceil} N)$

查找一个对等实体需要发送的消息数 T_S：最坏情况下查找一个对等实体需要回溯到 B+树的根节点，再向下遍历到叶节点，这样所需发送的最多查找消息为：

$T_S<2*\log_{\lceil m/2 \rceil} N$

插入对等实体需要发送的消息数 T_I：查找插入点 T_S+冲突检测+调整每一层

父节点直到顶层的最坏情况+叶节点分裂时通知左邻居修改 *nextNode* 向量

$$T_I < T_S + 5*M + (\log_{\lceil m/2 \rceil} N)*(\log_{\lceil m/2 \rceil} N) + \log_{\lceil m/2 \rceil} N$$

删除一个对等实体需要发送的消息数 T_E：通知同一节点中其他对等实体+合并相邻 B+树节点的最坏情况+通知左邻居叶节点修改 *nextNode* 向量

$$T_E < M + (\log_{\lceil m/2 \rceil} N)*(\log_{\lceil m/2 \rceil} N) + \log_{\lceil m/2 \rceil} N$$

处理 B+树某个叶节点丢失与删除一个对等实体需要相同数量级的消息数。

关键字分派需要发送的消息数 T_D：$= T_S$。

查找关键字需要发送的消息数 T_R：$= T_S$。

5.2.3 一致性 Hash 算法在分布查找服务中的应用

我们在实现查找服务的时候，没有让每个对等实体都存放关键字，这主要考虑到一般对等实体加入和离开网络比较频繁，会造成经常性的更新和调整关键字的分布。为了减少在这方面的开销，只有那些表明自己可以作为索引点的对等实体才加入到 B+树中，这由对等实体配置文件中的 *isRendezvous* 属性确定。只有 *isRendezvous* 属性为 *true*，才将其作为索引点加入到 B+树中。

5.3 小结

在本章中，我们讨论了分布式查找服务的结构和实现，重点介绍了基于一致性 Hash 和 B+树技术的分布式查找方法。这种查找方法以较小的存储和通信代价实现了整个 P2P 空间范围的查找，在性能方面比基于路由广播算法的查找方法优越的多。但是应该指出这种方法的容错能力不如路由广播算法，多个对等实体的失败很可能造成 B+树被分割成森林，而且恢复起来难度较大。我们在设计时考虑到了这方面的因素，通过采用策略模式，可以很方便地加入新的查找算法。

第六章 成员服务

每一个对等实体作为 P2P 系统的成员，需要一些共同的基本操作。我们在成员服务中为对等实体提供了加入对等实体组，获取对等实体信息功能。

6.1 成员服务接口和实现类的定义

成员服务由 ***membership*** 接口和 ***membershipImpl*** 实现类定义，图 6-1 给出了成员服务的类图。

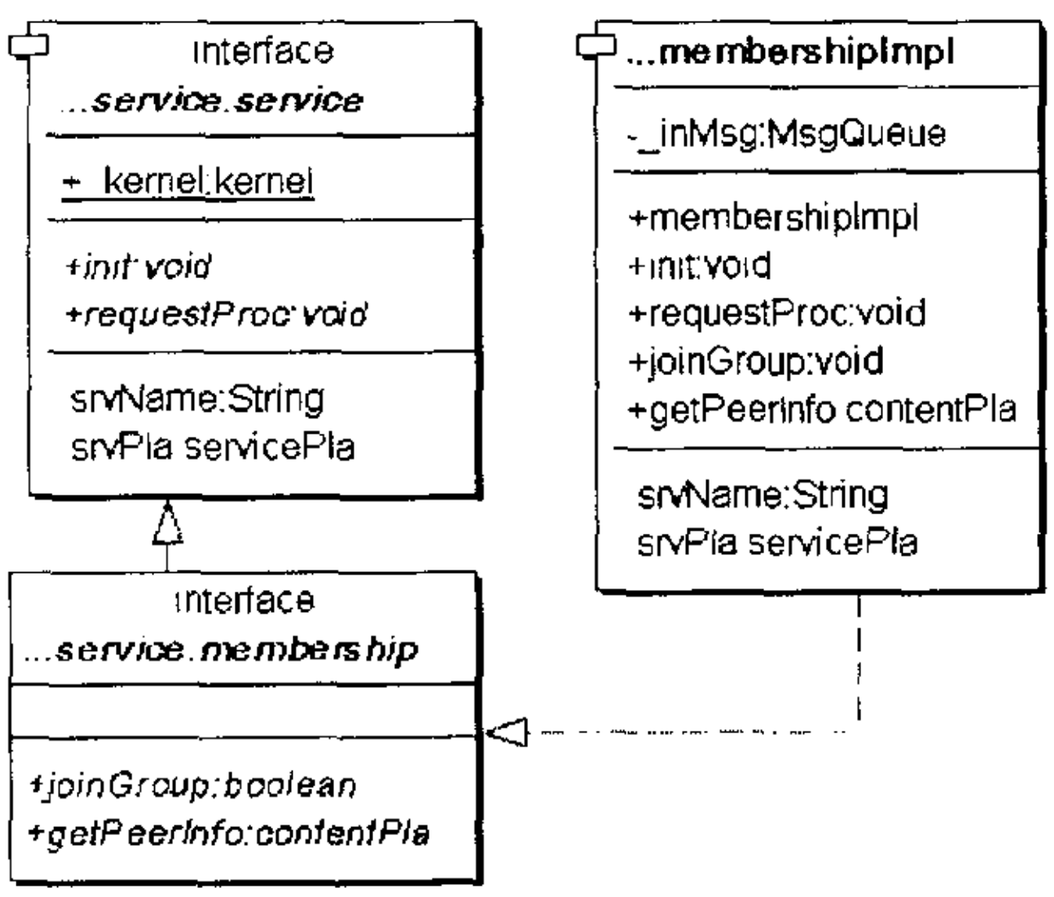

图 6-1：成员服务类图

接口 ***membership*** 定义了成员服务的基本操作，主要方法如下：

- ***public boolean joinGroup(UUID grpID,peerPla myPeer);*** 将对等实体加入组 ID 为 ***grpID*** 的组。
- ***public contentPla getPeerInfo(UUID dstPeer, String srvname);*** 获取目标对等实体 ***dstPeer*** 中相应服务的服务内容公告。

类 ***membershipImpl*** 实现了接口 ***membership*** 和 ***service*** 中的方法，主要属性和方法如下：

- ***private MsgQueue _inMsg;*** 存放成员服务接收的消息的消息队列。
- ***public void requestProc(message msg);*** 处理接收到的消息。

- ***public void joinGroup(UUID grpID, peerPla myPeer);*** 加入对等实体组方法的实现。
- ***public contentPla getPeerInfo(UUID dstPeer, String srvname);*** 取得目标对等实体相应服务服务内容公告方法的实现

6.2 成员服务中定义的消息

成员服务定义了四种消息：***MembershipJoin***、***MembershipAck***、***PeerInfoRequery*** 和 ***PeerInfoResponse***。

想要加入对等实体组的对等实体向执行成员认证的对等实体（对等实体组公告中给出执行认证的对等实体的端点公告）发送 ***MembershipJoin*** 消息，请求加入对等实体组。图 6-2 给出了 ***MembershipJoin*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: MembershipJoin>
    <Mercury: PeerPlacard>
        ...
    </Mercury: PeerPlacard >
    <Credential>reserved</Credential>
</Mercury: MembershipJoin>
```

图 6-2：MembershipJoin 消息 XML 文档

MembershipJoin 消息包括下列域：

- **PeerPlacard**：想要加入对等实体组的对等实体公告。
- **Credential**：对等实体拥有的信任证，根据信任证决定对等实体是否允许加入或者加入后具有的权限。目前我们还没有对对等实体加入组进行任何限制，信任证作为预留字段，供今后扩展使用，所以在其他服务定义的消息中都没有出现这个字段。

各元素值类型见表 6-1：

表 6-1：*MembershipJoin* 消息元素说明

元素名称	数量	元素值类型
PeerPlacard	1	peerPla
Credential	1	String

执行成员认证的对等实体接到某个对等实体的 ***MembershipJoin*** 消息后，发

回 ***MembershipAck*** 消息，通知对等实体是否加入，新的信用证以及能够使用的组服务。图 6-3 给出了 ***MembershipAck*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: MembershipAck>
    <isAccepted>true or false</isAccepted>
    <Credential>reserved</Credential>
    <Mercury: PeerGrpPlacard>
        ...
    </Mercury: PeerGrpPlacard >
</Mercury: MembershipAck>
```

图 6-3：MembershipAck 消息 XML 文档

MembershipAck 消息包括下列域：

- **isAccepted**：是否接受对等实体加入请求。
- **PeerGrpPlacard**：加入对等实体组的对等实体可以使用的对等实体组公告，公告规定了对等实体能够使用的对等实体组服务。对等实体不一定能够使用所有的组服务，这由对等实体拥有的信任证决定。目前我们还没有对对等实体能够使用的服务进行限制，这个功能留待今后实现。
- **Credential**：对等实体拥有的信任证，根据信任证决定对等实体是否允许加入或者加入后具有的权限。

各元素值类型见表 6-2：

表 6-2：MembershipAck 消息元素说明

元素名称	数量	元素值类型
isAccepted	1	boolean
PeerGrpPlacard	1	peerGrpPla
Credential	1	String

对等实体发送 ***PeerInfoRequery*** 消息给某个对等实体获取该对等实体的共享资源列表。图 6-4 给出了 ***PeerInfoRequery*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: PeerInfoRequery>
    <SourcePeerID>source peer ID</SourcePeerID>
    <ServiceName>
        service providing resources sharing
    </ServiceName>
</Mercury: PeerInfoRequery>
```

图 6-4：PeerInfoRequery 消息 XML 文档

PeerInfoRequery 消息包括下列域：

- **SourcePeerID**：发出请求的对等实体 ID。
- **ServiceName**：对等实体希望取得的特定服务共享资源的服务名字。对等实体可能有多种服务提供不同类型的资源共享，根据服务名字确定需要哪类共享资源列表。

各元素值类型见表 6-3：

表 6-3：PeerInfoRequery 消息元素说明

元素名称	数量	元素值类型
SourcePeerID	1	UUID
ServiceName	1	String

接到 ***PeerInfoRequery*** 消息的对等实体发回包含所请求服务的服务内容公告 ***PeerInfoResponse*** 消息。图 6-5 给出了 ***PeerInfoResponse*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: PeerInfoResponse>
    <SourcePeerID>source peer ID</SourcePeerID>
    <ServiceName>
        service providing resources sharing
    </ServiceName>
    <Mercury: ContentPlacard>
        ...
    </Mercury: ContentPlacard>
</Mercury: PeerInfoResponse>
```

图 6-5：PeerInfoResponse 消息 XML 文档

PeerInfoResponse 消息包括下列域：

- **SourcePeerID**：发送消息的对等实体 ID。
- **ServiceName**：服务名字。
- **ContentPlacard**：服务提供的服务内容公告

各元素值类型见表 6-4：

表 6-4：PeerInfoResponse 消息元素说明

元素名称	数量	元素值类型
SourcePeerID	1	UUID
ServiceName	1	String
ContentPlacard	1	contentPla

6.3 成员服务调用流程

成员服务的调用流程比较简单，基本上都是创建消息、发送消息、挂在消息队列上，接收消息和从挂起状态恢复。图 6-6 给出了发送消息的流程，图 6-7 给出了消息处理流程。

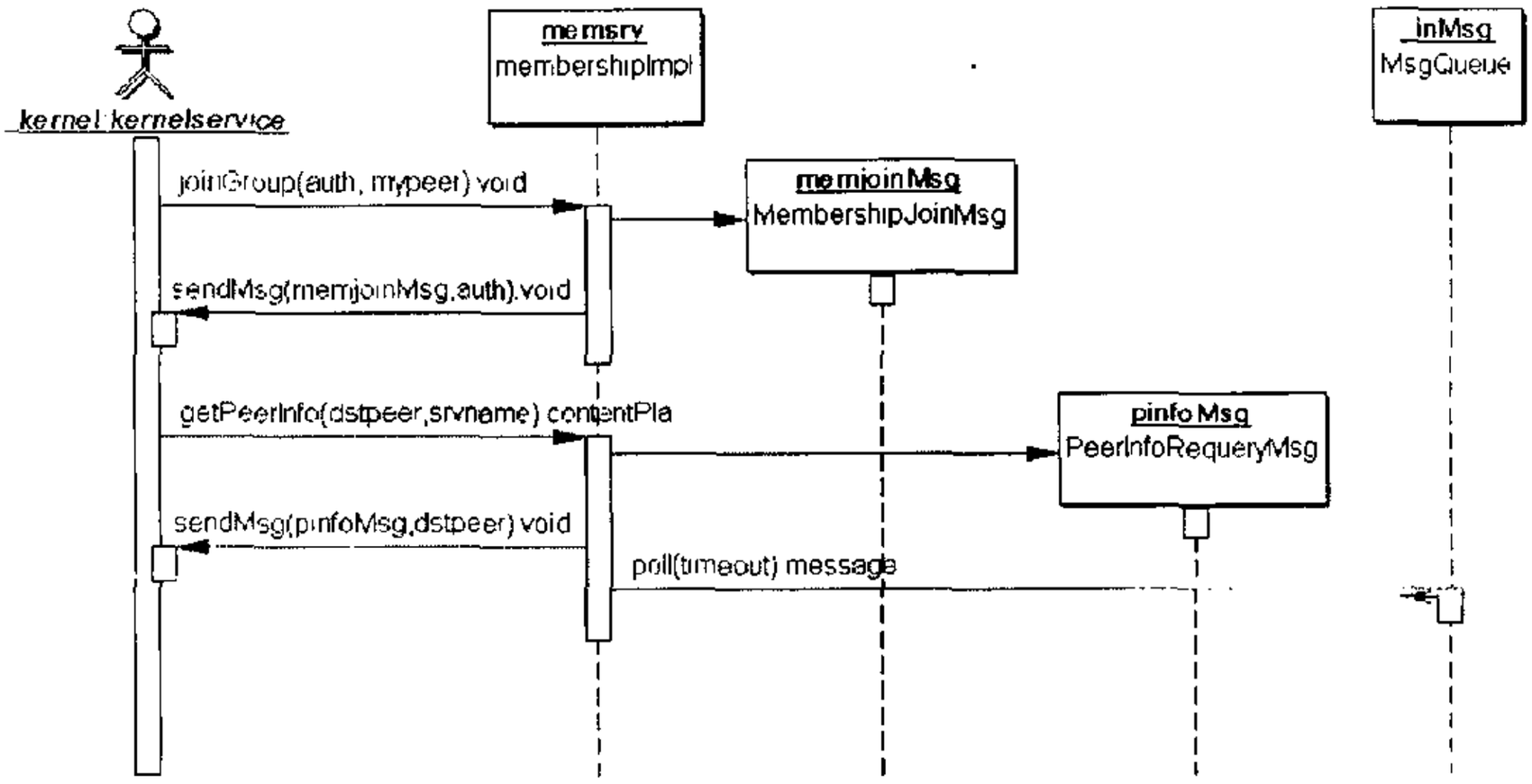

图 6-6：成员服务发送消息流程

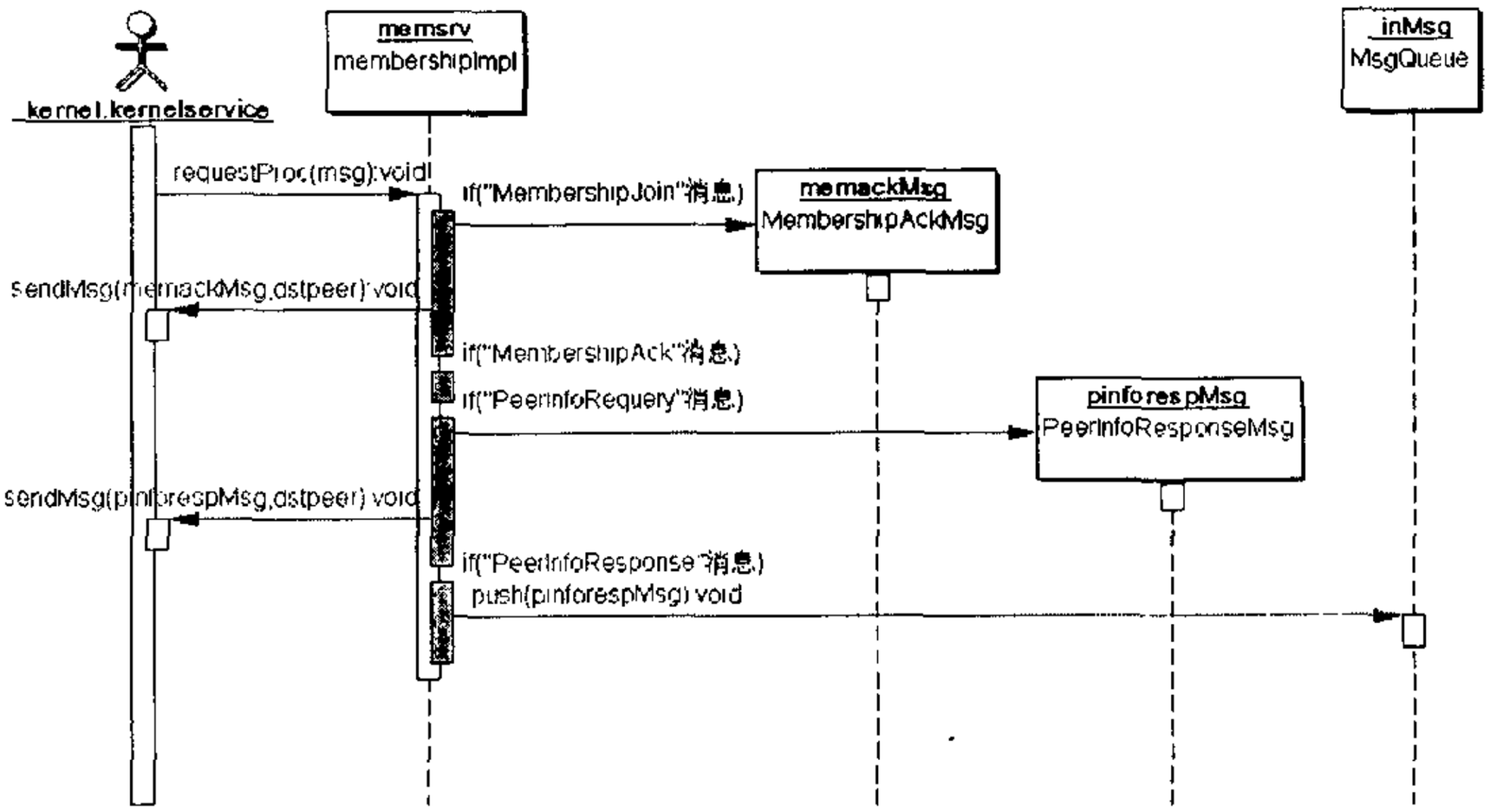

图 6-7：成员服务处理接收消息流程

6.4 小结

本章介绍了成员服务的组成、使用的消息以及调用流程。在目前的项目实施阶段，我们还没有对成员加入进行任何限制，只是预留了接口和消息定义。在下一步工作中，我们会对成员的认证进行更全面的控制，同时会根据实际情况增加新的成员服务。还要指出的是，由于我们没有足够的资源进行整个广域网络对等实体组管理（这需要长期在线的组管理服务器），现在我们的平台不支持多个组并存。不过这不会构成技术上的障碍。从应用的角度来看，单个组已经能满足大部分应用场景。

第七章 基于 P2P 基础平台的文件共享应用

前面几章介绍了 P2P 基础平台和提供的服务，下面我们通过实现一个基于 P2P 基础平台的文件共享应用，介绍如何使用平台开发 P2P 应用。

7.1 文件共享应用框架

文件共享应用包括应用界面和文件共享服务两部分。应用界面为用户提供使用环境，文件共享服务实现 P2P 基础平台的服务接口 ***service***，为应用提供文件共享功能。图 7-1 给出了文件共享服务类的定义：

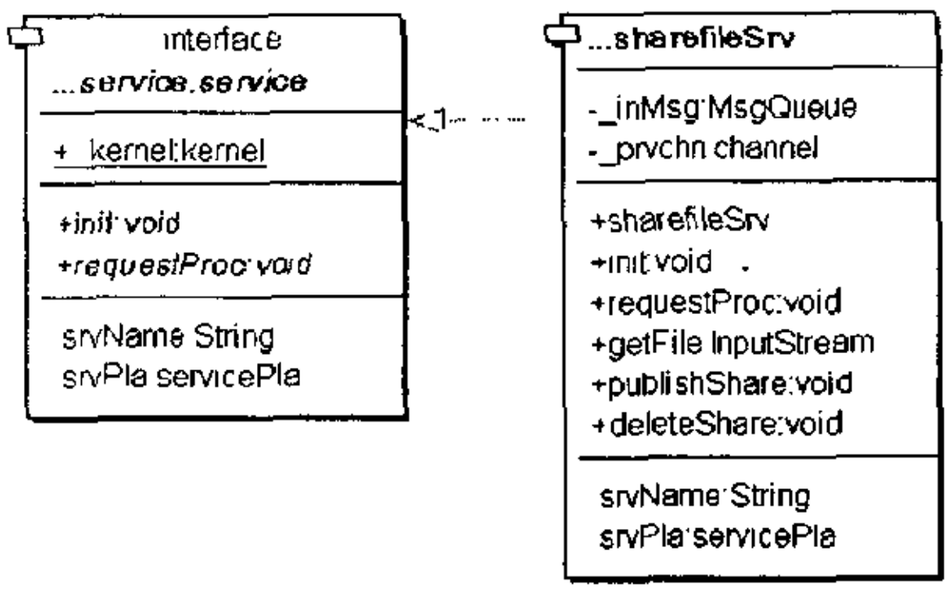

图 7-1：文件共享服务类定义

类 ***sharefileSrv*** 定义了文件共享服务的功能，主要属性和方法如下：

- ***private MsgQueue _inMsg***：存放接收的消息的消息队列。
- ***private channel _prvchn***：文件共享服务使用的专用通道。因为文件共享服务主要是进行文件传输，使用 UDP 协议显然不合适，为此通过控制核心建立专用的面向连接的通道。
- ***public void requestProc(message msg)***：处理接收到的消息。
- ***public InputStream getFile(UUID dstpeer,contentPla fileshared)***：从目标对等实体取得指定文件。
- ***public void publishShare(contentPla fileshared)***：公布自己的共享文件，以便其他对等实体访问。
- ***public void deleteShare(contentPla fileshared)***：删除某个文件的共享。

7.2 文件共享服务使用的消息

文件共享服务定义了两种消息：***FileRetrieve*** 和 ***FileTransReady***。

服务发送 ***FileRetrieve*** 消息请求传输文件。图 7-2 给出了 ***FileRetrieve*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: FileRetrieve>
    <FileName>transfer file name</FileName>
    <Mercury: EndpointPlacard>
        ...
    </Mercury: EndpointPlacard>
</Mercury: FileRetrieve >
```

图 7-2：FileRetrieve 消息 XML 文档

FileRetrieve 消息包括下列域：

- **FileName**：请求传输的文件名字。
- **EndpointPlacard**：用于文件传输的通道端点公告。

各元素值类型见表 7-1：

表 7-1：FileRetrieve 消息元素说明

元素名称	数量	元素值类型
FileName	1	String
EndpointPlacard	1	endpointPla

接收到 ***FileRetrieve*** 消息的服务准备好传输文件后，发送 ***FileTransReady*** 消息给对方，表示可以开始传输。图 7-3 给出了 ***FileTransReady*** 消息的 XML 文档定义。

```
<?xml version="1.0" encoding="UTF-8"?>
<Mercury: FileTransReady>
    <FileName>transfer file name</FileName>
    <FileLength>file length</FileLength>
</Mercury: FileTransReady >
```

图 7-3：FileTransReady 消息 XML 文档

FileTransReady 消息包括下列域：

- **FileName**：请求传输的文件名字。
- **FileLength**：传输的文件长度。

各元素值类型见表 7-2：

表 7-2：FileTransReady 消息元素说明

元素名称	数量	元素值类型
FileName	1	String
FileLength	1	long

7.3 文件共享应用执行流程

文件共享应用运行时，首先初始化控制核心，创建文件共享服务对象，然后将文件共享服务对象注册到控制核心上。具体流程见图 7-4。

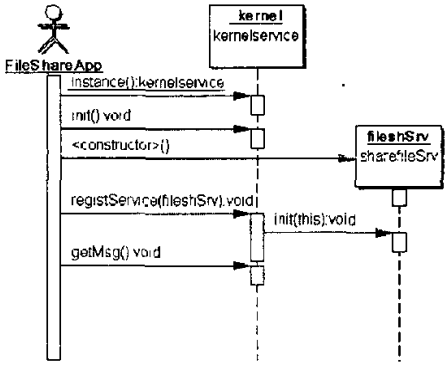

图 7-4：文件共享应用初始化顺序图

初始化完成后，应用创建主窗口，如图 7-5 所示。

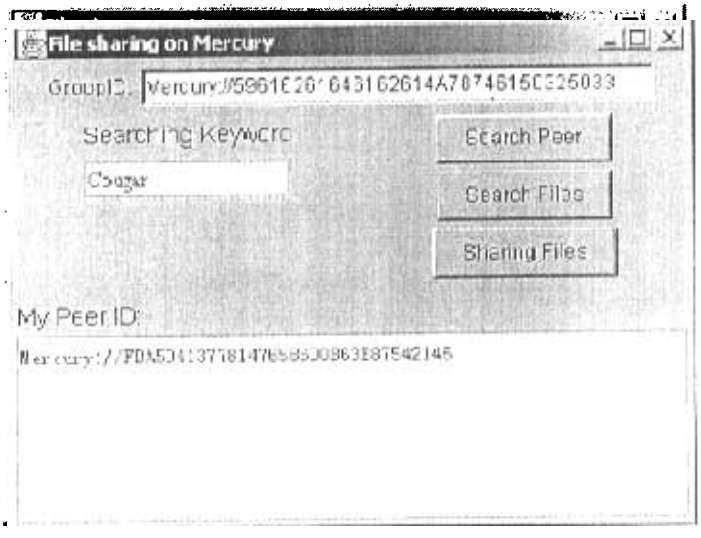

图 7-5：文件共享应用主窗口

主窗口显示了文件共享应用提供的主要功能：根据对等实体名字查找对等实体提供的共享文件；根据文件名字查找提供文件的对等实体；指定对等实体提供的共享文件。图 7-6 给出根据对等实体名字查找对等实体提供的共享文件的流程。

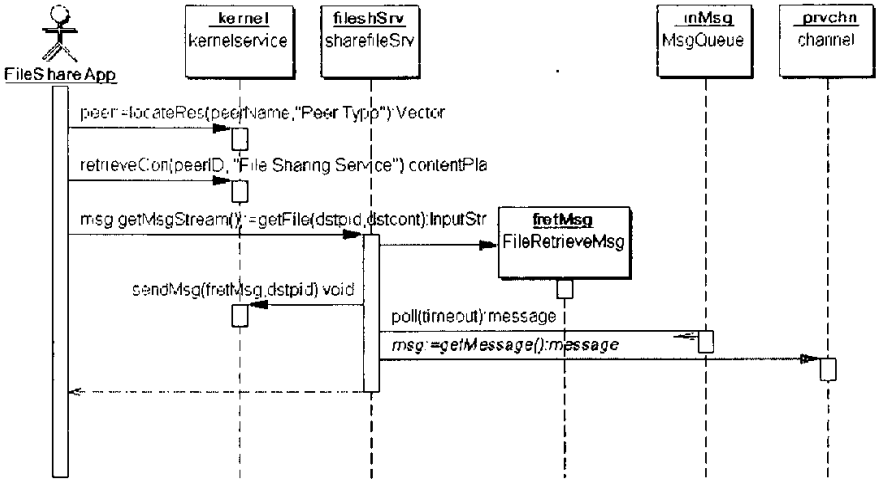

图 7-6：根据对等实体名字查找共享文件并取回指定文件的顺序图

图 7-7 显示了根据对等实体名字查找对等实体共享文件的结果。对等实体 ID 和共享文件显示在 ***Peer Information*** 窗口中，用户可以从中选择传输的文件。

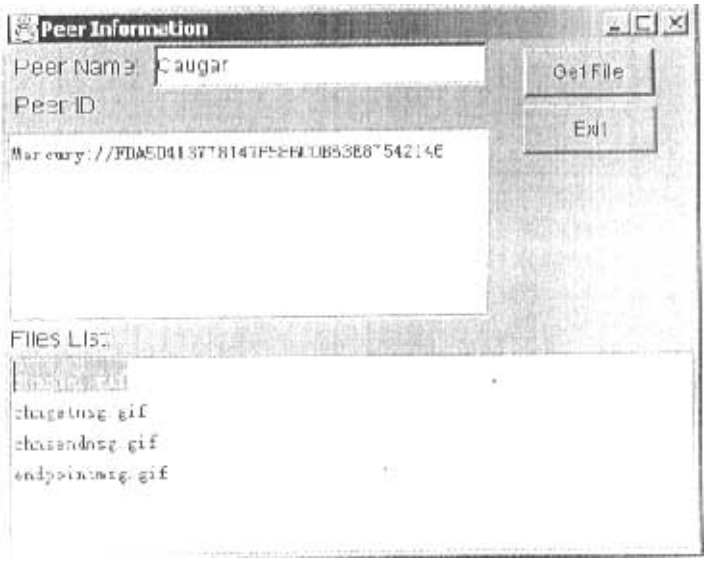

图 7-7：根据对等实体名字查找对等实体共享文件的结果窗口

根据文件名字查找文件所在对等实体的操作与根据对等实体名字查找共享文件操作类似。首先调用控制核心的 ***locateRes*** 方法取得文件所在对等实体的向量组，然后调用控制核心的 ***retrieveCon*** 方法取得文件所属的服务内容公告，接下来使用文件共享服务提供 ***getFile*** 方法取得文件。根据文件名字查找所在对等实体的结果显示在 ***File Found*** 窗口内，见图 7-8。

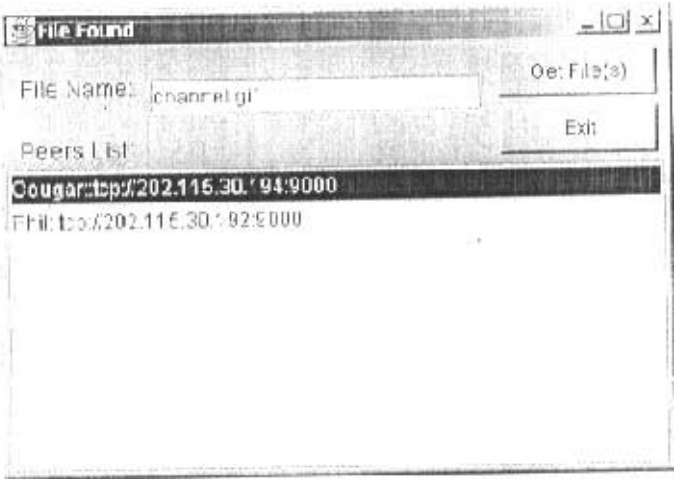

图 7-8：根据文件名字查找所在对等实体的结果窗口

文件共享应用如果要提供共享文件，在文件对话框中选择需要共享的文件。共享文件选择窗口见图 7-9。

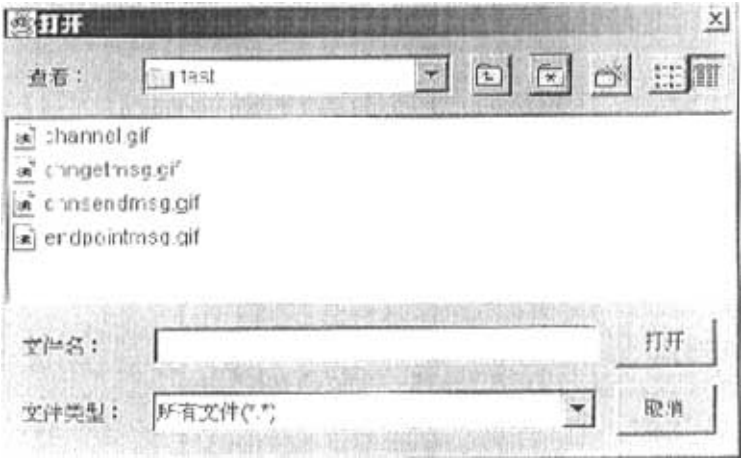

图 7-9：共享文件选择窗口

选择好共享文件后，应用调用控制核心提供的 ***registerRes*** 方法将共享文件注册到索引点上。

7.4 小结

在这章中，我们介绍了使用 P2P 基础平台实现一个简单 P2P 应用的方法。从中可以看出，通过使用 P2P 基础平台提供的服务，我们可以用很小的工作量实现 P2P 应用。这样 P2P 应用开发者就可以把主要精力放在应用逻辑本身。随着 P2P 基础平台上开发的服务越来越多，应用者可以更加容易地构建复杂的 P2P 应用。

第八章 总结

在本文中，我们阐述了一个 P2P 网络基础设施平台的体系结构、控制核心、基本服务以及基于这个平台建立的简单 P2P 应用实例。我们在设计中主要着眼于平台与具体操作系统以及网络环境的独立性。为此我们设计了基于 XML 的消息传输协议将平台与具体网络环境隔离开，使用 Java 语言保证平台可移植到多种操作系统上。我们在平台设计过程中广泛采用了面向对象技术，结合 UML 系统建模和设计模式确保平台各部分较低的耦合性和较好的复用性。针对 P2P 应用环境的特点，我们设计了基于一致性 Hash 和 B+树技术的分布式查找协议。以上工作增强了平台的实用性和先进性。当然由于开发时间比较紧张，我们在很多方面没有开展工作，这些方面将是我们下一步工作的重点：

1. 建立移动计算对象容器，使 P2P 网络中的对等实体可以进行协同计算，这是 P2P 应用的重要领域。
2. 增加对远程对等实体服务的直接调用，考虑使用 WSDL 和 SOAP 协议，为应用提供更加自然的方式调用服务。
3. 增加自适应能力。由于 P2P 环境中对等实体拥有的硬件、软件以及网络资源差异很大，对于不同的对等实体，平台应该能够自动调整运行参数，避免对等实体负载过大。
4. P2P 网络中的查找在很长一段时间内仍将是研究的重点，在对等实体失败恢复，信息更新等方面我们还有很多工作要做。

参考文献

[1] The Viant Innovation Center. The Human Side of Peer to Peer. February. 2001

[2] Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. University of Chicago Technical Report TR-2001-26, 24 July, 2001.

[3] Sun® Microsystems, Inc. Project JXTA: Technical Specification Version 1.0. April 25, 2001

[4] Sun® Microsystems, Inc. JXTA Search: Distributed Search for Distributed Networks. May 29, 2001

[5] Erich Gamma, et al. 设计模式：可复用面向对象软件的基础. 机械工业出版社，2000, 9

[6] James W. Cooper. The Design Patterns: Java Companion. Addison-Wesley Longman, Inc. 1998

[7] Object Management Group. Unified Modeling Language Specification Version 1.3 . March, 2000

[8] Mark Preistley. Practical Object-Oriented Design with UML. McGraw-Hill Companies, Inc. 2000

[9] Bruce Eckel. Java 编程思想. 机械工业出版社，1999

[10] 严蔚敏，吴伟民. 数据结构（C 语言版）. 清华大学出版社，1997

[11] Didier Martin. XML 高级编程. 机械工业出版社，2001

[12] Sun® Microsystems, Inc. Java API for XML Processing. February, 2001

[13] KARGER, D., et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663.

[14] Ion Stoica, et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, August 2001.

[15] R. Moats. RFC 2141: URN Syntax. May, 1997

[16] L. Daigle. RFC 2611: URN Namespace Definition Mechanisms. May, 2001

[17] N. Freed. RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. November, 1996

致谢

在此，我谨向我的导师周明天教授致以衷心的感谢和崇高敬意！在学习、科研和撰写论文期间，周老师给予我悉心的指导和严格的督促。周老师渊博的知识、严谨的治学态度、一丝不苟的工作作风给我留下了无法磨灭的印象，成为我今后学习和工作的目标和动力。周老师不仅为我创造优越的科研和学习环境，同时在思想上和生活上给予我多方面的关心和指导。

衷心感谢我的同窗好友吴怀谷、陈波、李亚伟，与他们的合作和交流使我得到很大的帮助和启迪。

最后，感谢曾经教育和指导过我的所有老师。衷心感谢为评阅本论文付出辛劳的老师。