BitTorrent协议概述与实现

陈鸿峥* 冯家苇* 符尧* 傅畅*

数据科学与计算机学院 计算机类

17341015，17341035，17341037，17341038

摘 要 对等网络(P2P)具有去中心化、扩展性高、可靠性高等特点，近年来广泛应用在多媒体、比特币、区块链等领域，成为步入网络3.0时代一项不可或缺的技术。而BitTorrent作为快速的文件共享传输协议，占据了互联网上35%的流量，在对等网络中占据着举足轻重的地位。本文将对BitTorrent协议进行研究，对其概念进行深入剖析与阐述，同时对其进行代码实现，为之后的研究者提供一定的参考。

关键词 对等网络(P2P)，BitTorrent协议

1 引言

随着大数据时代的到来，人们在互联网上交互的信息越来越多，传统的客户端-服务器端(client-server)模型已逐渐体现出弊端—由于中心化的单一服务器端的特点，越来越多的客户端在连入服务器时将存在严重的带宽问题。而对等网络(peer-to-peer, P2P)具有去中心化的特点，在近年来愈发得到人们的关注 [1]。

如图1所示，传统的中心化模型在客户与服务器之间存在十分清晰的界线，客户只能从服务器端下载内容。当存在大量的客户端时，服务器端将面临巨大的负载压力。受限于网络带宽，数据在客户端与与服务器端的传输时间将大幅增加。而去中心化的对等网络中，每一台主机既可以作为客户端，也可以作为服务器端，主机与主机之间不存在任何区别，因而称为对等实体。当从其他对等实体处下载数据时，该对等实体就是客户端；反之，上传数据并为其他对等实体提供资源的即为服务器端。由于数据在对等网络中采用分布式存储，故要获取某些特定数据往往要从多个对等实体处获取，从而实现负载均衡，大大减轻了单一主机的带宽压力，获得更高的数据传输性能。同时，对等网络还具有高可扩展性、高可靠性、高安全性等无可比拟的优势，在多媒体 [2]、比特币 [3]、区块链 [4]等方面得到了广泛应用，已成为步入网络3.0时代一项不可或缺的技术 [5]。

而比特洪流(BitTorrent, BT)协议 [6]作为去中心化的快速的文件共享、传输协议，在对等网络中占据十分重要的地位。英国网络分析公司CacheLogic的调查指出，BitTorrent占据了互联网上约35%的流量，并且有数亿用户每天都在使用 [7]。大量的应用都采用了BitTorrent协议，如迅雷 [8]和μTorrent [9]运用BT协议进行文件传输，Facebook [10]和Twitter [11]都采用BT协议进行内容分发，Amazon S3则内置了BT协议进行分布式的文件存储。由此可以看出BitTorrent协议的重要性，故本文将对BT协议进行详细阐述，同时给出其实现方法供后人参考。

*按学号顺序排序，不代表贡献大小！具体分工如下：陈鸿峥负责论文撰写及文章插图制作；冯家苇负责服务器代码编写与PPT制作；符尧负责客户端代码编写与资料查找；傅畅负责哈希函数与读写种子文件代码编写与资料收集。

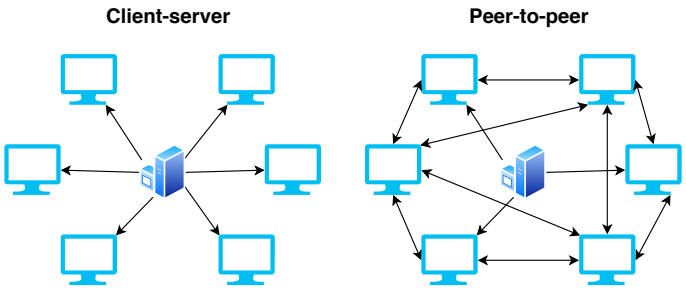


图 1: 客户端-服务器端网络和对等网络示意图

2 BitTorrent协议

BitTorrent协议是一个文件分发协议，它使用URL来定位内容并且能够与互联网进行无缝衔接。它相比纯HTTP的好处在于当同个文件的多个下载同时进行时，下载者可以通过上传自己已获得的部分文件供其他人下载，从而实现快速的大文件传输。

BitTorrent协议最早由Bram Cohen于2001年提出，最初只能连接两个实体，传输速度也十分受限 [12]。随着互联网的不断发展，BitTorrent协议也不断更新及完善，现在已能够实现大规模应用，并且充分利用网络带宽，实现快速的文件传输。本文将针对2017年的协议规范(BitTorrent Enhancement Proposals, BEPs) [13]进行阐述[1]，先介绍BitTorrent的基本概念，然后逐一阐述每一个环节的内容，再对完整的文件分享流程进行叙述，最后对具体的算法进行介绍。

2.1 概述

在BitTorrent协议中主要有三方面的内容，首先是以B编码的元信息文件（见第2.2节和第2.3节），其次是参与协议工作的各类实体。

对等实体可以分为播种者(seeder)和吸血者(leecher)。播种者即将自己的文件做成种子，然后对外进行分享。而吸血者仅仅下载别人的文件，而不自己做种，其他人将无法从它这里获得好处。

还有一种实体是追踪者(tracker)，它记录某个文件在哪些播种者中存在副本，并将这些信息告知需要下载文件的对等实体。

关于对等实体的内容见第2.4节，追踪者的内容见第2.5节，BitTorrent协议的具体流程见第2.6节，具体协议内的算法见第2.7节和第2.8节。

2.2 B编码

B编码(Bencoding)是一种数据紧密存储的方式，具体的格式如表1所示。注意任意两项之间没有任何空格，仅仅通过不同的格式区分。列表的内容可以是任意不同类型的组合，即列表内可以嵌套列表也可以包含字符串或字典。

2.3 元信息文件

元信息(metainfo)文件，即我们常见的`.torrent`格式的文件或种子文件，用B编码存储，包含了以下信息：

- 公告(announce)：追踪者的url地址，详情见第2.6节。

[1]注意这是BT协议公开的最新版本

表 1: B编码的格式

类型	格式	例子
字符串	<length>:<data>	7:network
整数	i<integer>e	i3e
列表	l<contents>e	l8:advanced7:networke
字典	d<keys><values>e	d3:onei1e3:twoi2e5:threei3e4:fouri4ee

- 信息(info)：一个字典，其中键值如下，都采用UTF-8编码。
  - 文件名称(name)：一个字符串。如果涉及多个文件或文件夹，则在信息中还包含路径名称(path)。
  - 片段长度(piece length)：每一个片段的字节数。为了传输方便，BitTorrent协议将文件分为等大的片段（除了最后一段）。每一个文件片段的长度通常是2的指数次幂，目前默认为$2^{18} = 256$字节。
  - 片段(piece)：每20个字节代表一个片段的SHA1哈希值，总长度总为20的倍数。
  - 其他可选项：如公告列表、创建日期、创建者、评论等。

如图2所示，元信息文件的格式还是比较清晰的。红色框标注的部分为键的名称，后面紧跟的为键值。

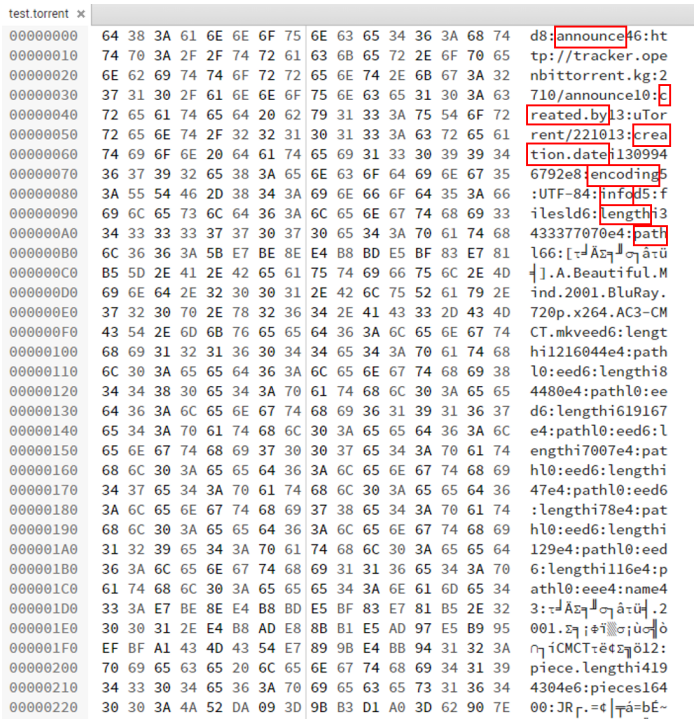


图 2: Torrent测试文件

2.4 对等实体互连协议

对等实体互连协议(peer wire protocol)构建在TCP或μTP协议 [14]上，其连接都是全双工的。

当一个对等实体下载完一个片段并查验其哈希值匹配后，它将对所有实体公告它已经拥有该片段，从而其他实体可以从它这里下载该片段。

对等实体连接两端都包含以下两种不同的状态，用两个二进制位存储：

- 阻塞(choked)：若某一客户端被远端的播种者阻塞，则该客户端无法从该播种者处获取片段。
- 感兴趣(interested)：若某一客户端对远端的播种者的片段感兴趣，则该客户端将在未阻塞时从该播种者处获取片段。

数据传输仅仅发生在一方感兴趣，另一方没有阻塞时。

对等实体之间建立连接需要进行握手确认，其握手信号如下所示。

```
<pstrlen><pstr><reserved><info_hash><peer_id>
```

各字段含义如表2所示。

表 2: 对等实体间握手信号字段含义

字段名	描述
pstrlen	pstr的长度，目前设为19
pstr	协议标识符字符串 目前用"BitTorrent protocol"作为标识
reserved	8个保留字节
info_hash	元信息文件中信息键值的20字节SHA1哈希编码
peer_id	每个实体独立的ID编号，20字节字符串

2.5 追踪者传输协议

追踪者一般采用HTTP协议，分为请求和回复两个阶段。

请求阶段，对等实体通过HTTP GET指令向追踪者发送请求，其中GET的参数如表3所示。

表 3: 追踪者HTTP协议GET请求

参数名	描述
info_hash	元信息文件中的20字节SHA1哈希值
peer_id	20字节客户端ID字符串
port	客户端监听的端口号，默认保留端口为6881-6889
uploaded	当前已上传总量，以十进制ASCII编码
downloaded	当前已下载总量，以十进制ASCII编码
numwant	请求对等实体数目，默认为50

回复阶段，追踪者通过HTTP协议向对等实体回复信息，具体参数如表4所示。

与追踪者的握手信号如表5所示。其中Bitfield同样用来指代当前已经被成功下载的片段编号，需要在握手信号已经完成，还未进行其他消息发送之前进行告知。 X代表位域长，位域的最高位代表片段0。而Have则在建立连接后用来指代当前已经被成功下载的片段编号，同样需要发送给追踪者，方便获取新的播种者名单。

表 4: 追踪者HTTP协议回复

参数名	描述
complete	播种者的数目
incomplete	非播种者的数目
peers	一个字典，包括对等实体ID、IP地址及端口号

表 5: 追踪者握手信号含义

握手信号	<length prefix>	<message ID>	<payload>
Keep-alive	0000	0	无
Choke	0001	0	无
Unchoke	0001	1	无
Interested	0001	2	无
Not-interested	0001	3	无
Have	0005	4	片段编号
Bitfield	0001+X	5	位域
Request	0013	6	<index><begin><length>
Piece	0009+X	7	<index><begin><block>
Cancel	0013	8	<index><begin><length>
port	0003	9	<listen-port>

由于HTTP协议基于TCP协议，需要连接始终保持开启，同时导致大量数据开销，故为了减少开销，BitTorrent的追踪者协议也可以基于UDP协议进行，这在BEP15 [15]中进行了规定。

另一方面，由于单一的追踪者已无法满足这么多对等实体的需求，故现在更多采用分布式稀疏哈希表(Distributed sloppy Hash Table, DHT)用来存储对等实体的通信信息。每个对等实体都将成为一个追踪者，这在BEP5 [16]中进行了规定。

2.6 文件共享机理

BitTorrent协议完整的操作流程如图3所示。

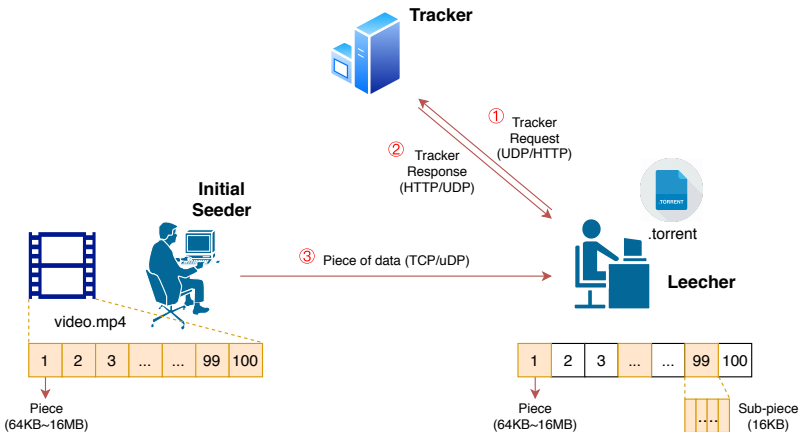


图 3: BitTorrent协议完整流程

初始的播种者将要传输的文件分割为很多片段，每个片段的长度取值可以从64KB到16MB。吸血者从元信息文件的announce键值中获取追踪者的信息，然后通过HTTP或UDP的方式与追踪者建立连接。追踪者将播种者的IP地址、端口号等信息回传给吸血者。吸血者根据对应的信息通过TCP或μDP方式，与播种者建立连接，并逐次下载对应片段。每个片段又可划分为很多个子片段，其中16KB的子片段为最小传输单位。当吸血者获取了所有片段并依序整合后，传输结束。

而吸血者在获取了部分片段后可以转变成播种者，同样共享资源给其他吸血者下载。如图4所示，新的吸血者同样从追踪者处获取播种者名单。追随者默认会返回50个随机的播种者，吸血者将并行地从这些播种者处获取缺失的片段，直到所有片段都收集齐为止。因此BitTorrent协议具有更多播种者，更多资源副本，下载速度更快的特点。

注意，由于种子文件较小，故一般可以通过搜索引擎或专门的种子服务器得到，由于不是本文的重点，故图中未给出种子传播的途径。

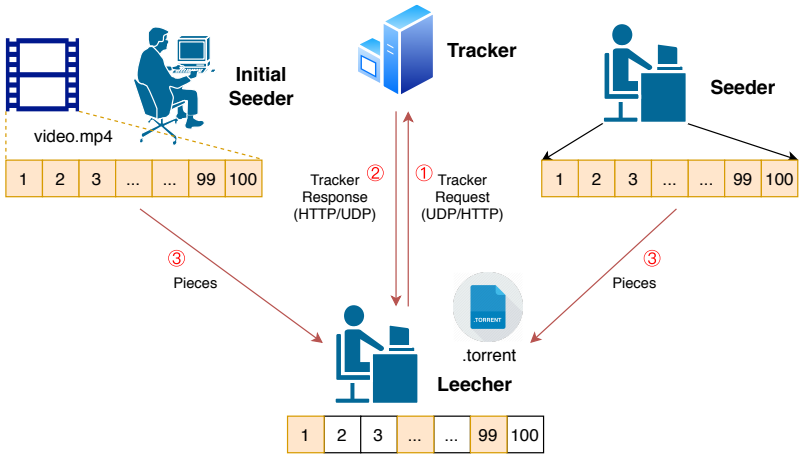


图 4: BitTorrent协议完整流程（多个播种者）

2.7 对等实体选择算法

对等实体选择策略用来决定从哪里下载对应片段，目前BitTorrent协议主要采用以下几种算法 [17]：

- 以牙还牙(Tit-for-tat, TFT)策略：一个对等实体更倾向于将其片段分享给那些提供给它更大下载速率的实体。这个策略大幅提升了BitTorrent协议的性能，有效避免搭便车的实体(free-rider)，即只下载不上传。
- 最优非阻塞：每隔30秒，一个对等实体会随机将它的邻居实体取消阻塞，而无关其上传速率。这能有效发现一些上传速度更快的对等实体。
- 反冷落(anti-snubbing)算法：如果一个实体注意到隔了一段时间它都没有从某一实体处获得片段，则该实体会认为被对方冷落了，并采取反制措施，不再向其上传片段。
- 只上传算法：一旦一个对等实体完成整个文件的下载，它将变成播种者。因为播种者没有其他东西要下载，因此无法基于下载速率选择最好的对等实体。相反地，播种者会更倾向于选择与有更优上传速率的实体进行分享。

2.8 片段选择算法

由于吸血者需要从播种者处获取文件片段(piece)，最后再一并整合，故获取哪一个片段是需要考虑的问题，常见的有以下几种算法：

- 严格优先：对等实体只关注于当前下载的片段，先将当前片段的所有子片段下载完，才下载其他片段。这可以保证自己有完整的片段可以与其他实体进行交易。
- 稀少优先：这是一般的情况，对于在所有对等实体中最缺失的片段，应该最先下载。这可以保证最常见的片段留到最后才进行下载。
- 随机优先：同字面意思，随机选取片段进行下载。
- 游戏终止模式(End game mode)：为避免获取最后几个片段造成的延迟，对等实体将向其他所有实体发送请求。而一旦片段获得，则应发送`cancel`消息给不必要的对等实体，防止带宽浪费。

3 实现细节

在本次研究过程中，我们设计实现了一个简化的BT协议。并且独立地使用C++语言，基于前面几次套接字实验的经验，编写了服务器和客户端程序。程序实现的基本功能有：创建、解析种子文件与下载、校验对应文件。接下来分协议(3.1节)，服务器(3.2节)，客户端(3.3节)三个部分介绍。

3.1 BT协议

我们的BT协议是基于标准的BT协议简化而来，保留了BT协议中的必选字段7，删除了部分目前没有实现的字段，目前支持多个用户在局域网内进行单文件分享。

一个标准的BT种子文件(.torrent)格式如图**??**，实现结果如图5。其中服务器地址和片段大小由用户在制作种子时输入，文件长度以及每个片段的SHA1哈希值由程序计算得来。

一个典型的，基于BT协议的，分享文件的过程为：

1. 文件拥有者启动客户端，用客户端制作种子文件，然后向服务器发出通知，告诉服务器文件名、IP地址和监听的端口号。
2. 其他用户通过各种手段(种子分享网站、云盘或U盘)得到种子之后运行客户端，解析种子文件信息，根据种子文件中存储的信息(announce和name)向服务器索要文件拥有者的信息(IP地址和端口号)。
3. 下载者和文件拥有者(可能有多个)建立连接，下载文件
4. 下载完所有块之后，对每一块进行SHA1哈希校验，如果都成功则接受，否则报告错误并退出下载
5. 下载完成之后，转变为文件的拥有者，并通知服务器。

这里，我们将服务器与客户端间的通信机制设置为如下格式。

```
<peer_num> <IP1> <Port1> <IP2> <Port2> ...
```

客户端可发布三种命令。0号命令用于发布文件信息，1号命令用于获取文件拥有者列表，2号命令表示不再对外分享文件并退出连接。服务器则只会对1号命令回复有效信息，即一个文件拥有者列表。

表 6: Tracker与node间通信机制

指令号	指令格式	描述
0	<filename> <port>	用于接受连接的数据端口
1	<filename>	需要下载的文件名
2	none	退出连接

种子文件结构如下所示。

```
{
    announce: 服务器地址(字符串)
    info:{
        length: 文件长度(整数)
        name: 文件名(字符串)
        piece_length: 每块字节数(整数)
        pieces: 块校验码(字符串)
    }
}
```

表 7: torrent文件各个字段含义

字段名	描述
announce	服务器地址
info	目标文件信息，包含以下内容
length	目标文件长度
name	目标文件名
piece_length	片段大小
pieces	每个片段的20个字节的SHA1哈希值

3.2 服务器

这里实现了一个简化版的Tracker服务器，用于维护与发布当前网络中的种子信息。服务器与客户端间的通信使用TCP协议，而客户端会设置一个特殊的控制端口专用于传达指令。

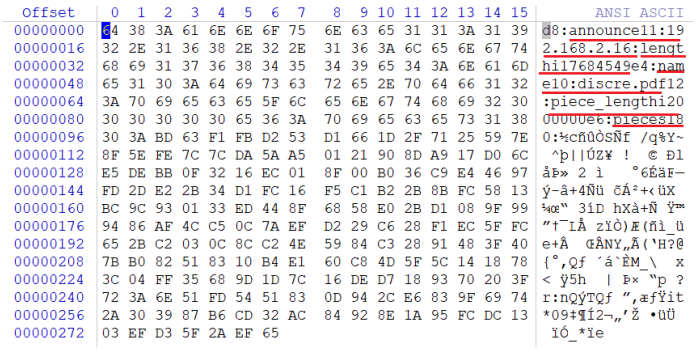
一个完整的通信流程如下：

1. 第一个文件持有客户端会与服务器建立连接并告知服务器自己拥有的文件名，IP地址与数据端口号，服务器会将信息登记存储。
2. 当一个客户端需要下载这个文件时，会根据种子文件获取服务器的IP地址，与服务器建立连接，并告知服务器自己需要的文件；服务器将依据文件名返回一个文件拥有者的列表。
3. 当一个文件拥有者退出连接时，服务器会将记录中与其IP地址匹配的表项全部清除。

3.3 客户端

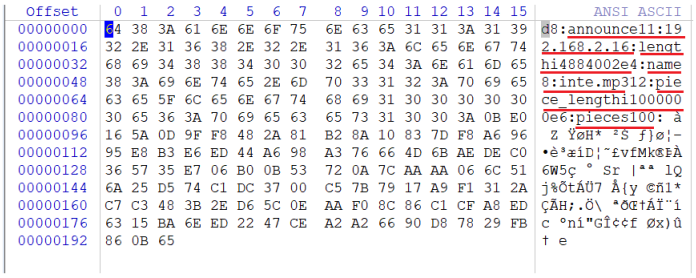
客户端具有并行处理功能，启动之后客户端同时并行执行两个线程：主线程负责读取并执行用户命令；一个upload线程等待其他客户端的连接，准备传输文件。

客户端目前支持制作种子与下载文件功能。



Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00000000	d4	38	3A	61	6E	6E	6F	75	6E	63	65	31	31	3A	31	39	d8:announce11:19
00000016	32	2E	31	36	38	2E	32	2E	31	36	3A	6C	65	6E	67	74	2.168.2.16:lengt
00000032	68	69	31	37	36	38	34	35	34	39	65	34	3A	6E	61	6D	hi17684549e4:nam
00000048	65	31	30	3A	64	69	73	63	72	65	2E	70	64	66	31	32	e10:discre.pdf12
00000064	3A	70	69	65	63	65	5F	6C	65	6E	67	74	68	69	32	30	:piece_lengthi20
00000080	30	30	30	30	30	65	36	3A	70	69	65	63	65	73	31	38	00000e6:pieces18
00000096	30	3A	BD	63	F1	FB	D2	53	D1	66	1D	2F	71	25	59	7E	0:½cñûÒSÑf /q%Y~
00000112	8F	5E	FE	7C	7C	DA	5A	A5	01	21	90	8D	A9	17	D0	6C	^þ||ÚZ¥ ! © Ðl
00000128	E5	DE	BB	0F	32	16	EC	01	8F	00	B0	36	C9	E4	46	97	åÞ» 2 ì °6ÉäF—
00000144	FD	2D	E2	2B	34	D1	FC	16	F5	C1	B2	2B	8B	FC	58	13	ý-â+4Ñü õÁ²+‹üX
00000160	BC	9C	93	01	33	ED	44	8F	68	58	E0	2B	D1	08	9F	99	¼œ" 3íD hXà+Ñ Ÿ™
00000176	94	86	AF	4C	C5	0C	7A	EF	D2	29	C6	28	F1	EC	5F	FC	"†¯LÅ zïÒ)Æ(ñì_ü
00000192	65	2B	C2	03	0C	8C	C2	4E	59	84	C3	28	91	48	3F	40	e+Â ŒÂNY„Ã('H?@
00000208	7B	B0	82	51	83	10	B4	E1	60	C8	4D	5F	5C	14	18	78	{°‚Qƒ ´á`ÈM_\ x
00000224	3C	04	FF	35	68	9D	1D	7C	16	DE	D7	18	93	70	20	3F	< ÿ5h | Þ× "p ?
00000240	72	3A	6E	51	FD	54	51	83	0D	94	2C	E6	83	9F	69	74	r:nQýTQƒ ",æƒŸit
00000256	2A	30	39	87	B6	CD	32	AC	84	92	8E	1A	95	FC	DC	13	*09‡¶Í2¬„'Ž •üÜ
00000272	03	EF	D3	5F	2A	EF	65										ïÓ_*ïe

(a) discre.torrent



Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00000000	d4	38	3A	61	6E	6E	6F	75	6E	63	65	31	31	3A	31	39	d8:announce11:19
00000016	32	2E	31	36	38	2E	32	2E	31	36	3A	6C	65	6E	67	74	2.168.2.16:lengt
00000032	68	69	34	38	38	34	30	30	32	65	34	3A	6E	61	6D	65	hi4884002e4:name
00000048	38	3A	69	6E	74	65	2E	6D	70	33	31	32	3A	70	69	65	8:inte.mp312:pie
00000064	63	65	5F	6C	65	6E	67	74	68	69	31	30	30	30	30	30	ce_lengthi100000
00000080	30	65	36	3A	70	69	65	63	65	73	31	30	30	3A	0B	E0	0e6:pieces100: à
00000096	16	5A	0D	9F	F8	48	2A	81	B2	8A	10	83	7D	F8	A6	96	Z ŸøH* ²Š ƒ}ø¦–
00000112	95	E8	B3	E6	ED	44	A6	98	A3	76	66	4D	6B	AE	DE	C0	•è³æíD¦˜£vfMk®ÞÀ
00000128	36	57	35	E7	06	B0	0B	53	72	0A	7C	AA	AA	06	6C	51	6W5ç ° Sr |ªª lQ
00000144	6A	25	D5	74	C1	DC	37	00	C5	7B	79	17	A9	F1	31	2A	j%ÕtÁÜ7 Å{y ©ñ1*
00000160	C7	C3	48	3B	2E	D6	5C	0E	AA	F0	8C	86	C1	CF	A8	ED	ÇÃH;.Ö\ ªðŒ†ÁÏ¨í
00000176	63	15	BA	6E	ED	22	47	CE	A2	A2	66	90	D8	78	29	FB	c °ní"GÎ¢¢f Øx)û
00000192	86	0B	65														† e

(b) inte.torrent

图 5: 客户端做的种子

制作种子时，由用户输入目标文件名、tracker服务器地址和种子文件分块大小。客户端将目标文件分块读取并为每一块生成20字节的SHA1哈希校验码存在种子文件中。创建完种子之后，客户端向服务器发布消息，包括文件名和upload线程监听的端口号(默认为50520)。

下载文件时，客户端首先打开种子文件并解析其中的信息。然后根据announce字段连接服务器，根据name字段向服务器询问文件拥有者的信息。服务器返回一个文件拥有者的列表给客户端。如果有n个拥有者，客户端就会尽量将文件分为n份(每一份都是块大小的整数倍)，然后启动n个download_t子线程向每个拥有者索要对应的文件块。主线程等待每个子线程下载完毕之后，按顺序读取各个临时文件，存储到最终下载的目标文件中，每读一块数据进行一次SHA1哈希校验，如果和种子文件中存储的哈希值相同则接受，否则报错并退出下载。将临时文件合并完之后，删除临时文件。下载完毕之后，客户端就转变为了文件的拥有者，向服务器发出通知，告知服务器文件名和监听的端口号。之后就可以向其他客户端发送该文件。

下面的内容详细介绍了客户端的各个模块具体工作原理。

3.3.1 文件下载与上传

文件的下载与上传是客户端的核心功能，客户端在启动之后就立即开启upload线程，随时准备接受其他客户端的连接，向其他客户端发送文件。而每当用户输入下载文件的命令时，客户端就进入download函数，下载对应的文件。

upload 类似于聊天程序客户端，每收到一个连接请求，upload线程就启动一个upload_t子线程负责通信。

```
1 void upload()
```

```
{
    SOCKET msock, sock;
    struct sockaddr_in sin, fsin;
    int alen = sizeof(struct sockaddr_in);

    msock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons((u_short)stoi(lis_port));
    bind(msock, (struct sockaddr *)&sin, sizeof(sin));

    listen(msock, 5);

    while (true)
    {
        sock = accept(msock, (struct sockaddr *)&fsin, &alen);
        mu.lock();
        cout << "收到" << inet_ntoa(fsin.sin_addr) << "的连接请求! " << endl;
        mu.unlock();
        thread temp(upload_t, sock);
        temp.detach();
    }

    return;
}
```

upload_t upload_t函数实现如下。upload函数从下载者那里接受要上传的文件名、文件起始位置和字节数。然后打开对应的文件，将文件指针移动到指定位置，读取指定数量的字节并发送。

```
void upload_t(SOCKET sock)
{
    char buff[bufflen];
    //文件名
    int cc = recv(sock, buff, bufflen, 0);
    buff[cc] = 0;
    ifstream myifs(string(buff), ios::binary);
    //send只起同步作用
    send(sock, "y", 1, 0);
    //起始位置
    cc = recv(sock, buff, bufflen, 0);
    buff[cc] = 0;
    int start = stoi(string(buff));
    send(sock, "y", 1, 0);
    //字节数
    cc = recv(sock, buff, bufflen, 0);
```

```
    buff[cc] = 0;
    int num = stoi(string(buff));
    //移动文件指针
    myifs.seekg(start, ios::beg);
    mu.lock();
    cout << "正在发送文件! " << endl;
    mu.unlock();
    while (!myifs.eof() && num > 0)
    {
        myifs.read(buff, min(bufflen, num));
        send(sock, buff, myifs.gcount(), 0);
        num -= myifs.gcount();
    }
    myifs.close();
    mu.lock();
    cout << "文件发送成功! " << endl;
    mu.unlock();

    return;
}
```

download download函数首先根据种子文件的信息(存储在结构体tf中)连接服务器(tf.announce)，向服务器询问指定文件(tf.name)的拥有者。服务器返回文件拥有者的列表，其中包含IP地址和端口号。这部分的代码如下。

```
struct sockaddr_in sin;
SOCKET sock;

char buff[bufflen];

sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = inet_addr(tf.announce.c_str());
sin.sin_port = htons((u_short)stoi(server_port));
int ret = connect(sock, (struct sockaddr*)&sin, sizeof(sin));

if (ret == 0)
{
    send(sock, "1", 1, 0);
    recv(sock, buff, bufflen, 0);
    send(sock, tf.name.c_str(), tf.name.length(), 0);
    int cc = recv(sock, buff, bufflen, 0);
    buff[cc] = 0;
    map<string, string> target;
    //解析返回的文件拥有者列表
    istringstream ifs(buff);
```

```
    int n;
    ifs >> n;
    mu.lock();
    cout << "找到" << n << "个拥有者! " << endl;
    mu.unlock();
    // ... continued
```

客户端解析服务器返回的列表之后，会根据目前文件拥有者的数目决定开启几个线程下载、每个线程需要下载的块数目。

假如目前有n个文件拥有者，文件有m块，那么客户端最多会启动n个线程下载，每个线程最多下载$\lceil \frac{m}{n} \rceil$块。

- 假设目标文件有8块，目前有3个拥有者。那么客户端启动3个线程，分别下载3，3，2块。
- 假设目标文件有2块，目前有3个拥有者。那么客户端会启动2个线程，分别下载1，1块。

对于每个文件拥有者，客户端启动一个download_t子线程与之通信，download_t函数需要文件名，IP地址，端口号，起始下载位置，下载字节数和序号作为参数。

```
for (int i = 0; i < n; i++)
{
    string ip, port;
    ifs >> ip >> port;
    target[ip] = port;
}
//计算文件的总块数
int piece_num = (tf.length + tf.piece_length - 1) / tf.piece_length;
//计算每个线程应该下载的块数
int task_num = (piece_num + n - 1) / n;
//一个线程向一个拥有者索要
int index = 0;
int num = tf.length;
vector<thread*> t_pool;
for (auto item : target)
{
    /*最后一个可能会有不足*/
    t_pool.push_back(new thread(download_t, item.first, item.second, tf.name,
        ↪ index*task_num*tf.piece_length,
        min(task_num*tf.piece_length, num), index));
    ++index;
    num -= task_num * tf.piece_length;
    if (num <= 0)
        break;
}

for (auto item : t_pool)
{
    item->join();
    delete item;
}
```

之后，主线程等待所有子线程执行完毕后，将所有子线程下载的临时文件(*index.temp)按顺序读取。每读取一块数据，就进行一次SHA1哈希校验，和种子文件中存储的哈希值进行比较，如果校验失败则报错并退出下载；否则将数据写入到最终的目标文件中。

所有校验完毕之后，文件就下载成功了。客户端会通知服务器，告诉服务器自己拥有的文件、IP地址和监听的端口号。

```
for (int i = 0, j = 0; i < index; i++)
{
    string tempname = tf.name + to_string(i) + ".temp";
    ifstream tempifs(tempname, ios::binary);
    cout << "正在合并临时文件" << i << "! " << endl;
    while (!tempifs.eof())
    {
        tempifs.read(tempbuff, tf.piece_length);
        int a = tempifs.gcount();
        if (a <= 0)
            break;
        tempbuff[a] = 0;
        string tem(tempbuff);
        //对收到的文件块进行hash校验
        if (tf.pieces[j++] != SHA1(tempbuff, tempifs.gcount()))
        {
            cerr << "校验失败! \n";
            tempifs.close();
            myofs.clear();
            myofs.close();
            return;
        }
        myofs.write(tempbuff, tempifs.gcount());
    }
    tempifs.close();
    remove(tempname.c_str());
    cout << "临时文件" << i << "合并成功! " << endl;
}
myofs.close();
mu.lock();
cout << tf.name << "下载成功! " << endl;
mu.unlock();
//通知服务器
send(sock, "0", 1, 0);
recv(sock, buff, bufflen, 0);
send(sock, tf.name.c_str(), tf.name.size(), 0);
recv(sock, buff, bufflen, 0);
send(sock, lis_port.c_str(), lis_port.size(), 0);
```

3.3.2 SHA1哈希函数

在SHA1算法中，我们必须把原始消息转化成位串。消息必须进行补位，以使其长度在对512取模以后的余数是448。也就是说，（补位后的消息长度）%512 = 448。即使长度已经满足对512取

模后余数是448，补位也必须要进行。接着要将原始数据的长度补到已经进行了补位操作的消息后面。通常用一个64位的数据来表示原始消息的长度。

在SHA1中我们使用一系列的函数。对于每组的512字节，分成16子组80次轮换计算摘要，具体细节这里不予赘述。

```
std::string SHA1class::final()
{
    /* Total number of hashed bits */
    uint64_t total_bits = (transforms*BLOCK_BYTES + buffer.size()) * 8;

    /* Padding */
    buffer += (char)0x80;
    size_t orig_size = buffer.size();
    while (buffer.size() < BLOCK_BYTES)
    {
        buffer += (char)0x00;
    }

    uint32_t block[BLOCK_INTS];
    buffer_to_block(buffer, block);

    if (orig_size > BLOCK_BYTES - 8)
    {
        transform(digest, block, transforms);
        for (size_t i = 0; i < BLOCK_INTS - 2; i++)
        {
            block[i] = 0;
        }
    }

    /* Append total_bits, split this uint64_t into two uint32_t */
    block[BLOCK_INTS - 1] = (uint32_t)total_bits;
    block[BLOCK_INTS - 2] = (uint32_t)(total_bits >> 32);
    transform(digest, block, transforms);

    /* Hex std::string */
//   std::ostringstream result;
    std::string result;
    for (size_t i = 0; i < sizeof(digest) / sizeof(digest[0]); i++)
    {
        //       result << std::hex << std::setfill('0') << std::setw(8);
        //       result << digest[i];
        for (uint32_t j = 0, dig = digest[i]; j < 4; ++j, dig >>= 8)
            result += uint8_t(dig & 255);
    }

    /* Reset for next run */
    reset(digest, buffer, transforms);

```

```
    return result;
}
```

3.3.3 种子文件创建与读取

实验中的 BT种子结构比较简浅，只有一个字典嵌套若干字符串和一个列表。种子的解析和读取只需要额外书写几个读写字符串、整数、和列表的子函数

```
string read_torrent_str(const char t[], int sz, int &i) {
    int strl = 0;
    assert(isdigit(t[i]));
    for (; i < sz && isdigit(t[i]); ++i) strl = strl * 10 + t[i] - 48;
    assert(t[i++] == ':');
    assert(i + strl <= sz);
    //  string ret=t.substr(i, strl); i+=strl;
    string ret = "";
    for (int j = 0; j < strl; ++j, ++i) ret += t[i];
    return ret;
}

int read_torrent_int(const char t[], int sz, int &i) {
    assert(t[i++] == 'i');
    int num = 0, sgn = 1;
    if (t[i] == '-') sgn = -1, ++i;
    assert(isdigit(t[i]));
    for (; i < sz && isdigit(t[i]); ++i)num = num * 10 + t[i] - 48;
    assert(t[i++] == 'e');
    return num * sgn;
}

string make_torrent_int(int num) {
    string ret = "";
    int sgn = 1;
    if (num < 0)num = -num, sgn = -1;
    if (num == 0) ret = "0";
    else for (; num; num /= 10) ret += char(num % 10 + 48);
    reverse(ret.begin(), ret.end());
    if (sgn == -1) ret = "-" + ret;
    return "i" + ret + "e";
}

string make_torrent_str(string t) {
    return to_string(t.size()) + ":" + t;
}
```

读取的函数如下

```
torrent_file read_torrent(string filename) {
    //  ifstream fin(filename);
    //  ostringstream finbuf; finbuf<<fin.rdbuf();
    //  string s(finbuf.str());fin.close();
```

```
    //  torrent_file res;
    //  assert(s.front()=='d' && s.back()=='e');
    //  s=s.substr(1, s.size()-2);
    FILE *fin;
    fopen_s(&fin, filename.c_str(), "rb"); int filesz = 0;
    for (char tmpc; fread(&tmpc, 1, 1, fin); ++filesz);
    fclose(fin); fopen_s(&fin, filename.c_str(), "rb");
    char *s = new char[filesz + 1];
    fread(s, 1, filesz, fin); fclose(fin);
    assert(s[0] == 'd' && s[--filesz] == 'e');

    torrent_file res;
    for (int i = 1; i < filesz;) {
        string itname = read_torrent_str(s, filesz, i);

        if (itname == "announce") {
            res.announce = read_torrent_str(s, filesz, i);
        }
        else if (itname == "length") {
            res.length = read_torrent_int(s, filesz, i);
        }
        else if (itname == "name") {
            res.name = read_torrent_str(s, filesz, i);
        }
        else if (itname == "piece_length") {
            res.piece_length = read_torrent_int(s, filesz, i);
        }
        else if (itname == "pieces") {
            string totpiece = read_torrent_str(s, filesz, i);
            for (int j = 0; j < totpiece.size(); j += 20)
                res.pieces.push_back(totpiece.substr(j, 20));
        }
        else {
            fprintf(stderr, "Wrong on item name");
        }
    }

    delete[]s;
    return res;
}
```

4 实验结果

我们使用我们的BT程序做了多组文件传输实验，在实验中，我们利用一台路由器和五台电脑搭建了局域网，所有参与实验的电脑连接到局域网中并配置IP如表8；实验过程中，我们使用了U盘分享种子文件，然后在不同的PC电脑上解析种子文件，下载对应的目标文件。最后的实验结果验证了协议与程序的正确性。

表 8: 实验设备

设备名称	IP地址	功能
PC1	192.168.2.1	服务器
PC2	192.168.2.2	客户端
PC3	192.168.2.3	客户端
PC4	192.168.2.4	客户端
PC5	192.168.2.5	客户端

4.1 1-1实验

我们首先进行了最简单的1个下载者和1个文件发布者之间的传输实验。PC1做服务器(图7)。PC2作为发布者(图8)，发布两份文件：discre.pdf和inte.mp3。制作的种子文件为：discre.torrent和inte.torrent。PC4解析两份种子文件，向PC2请求下载(图9)。1-1实验网络拓扑结构图如6所示。最终下载的结果如图10，后面的几组实验同样是在下载这两份文件，因此不再做展示。

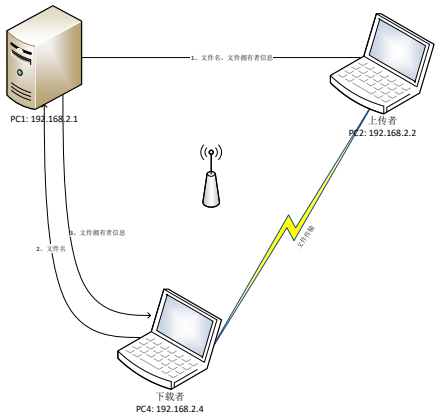


图 6: 1-1实验　网络拓扑结构图

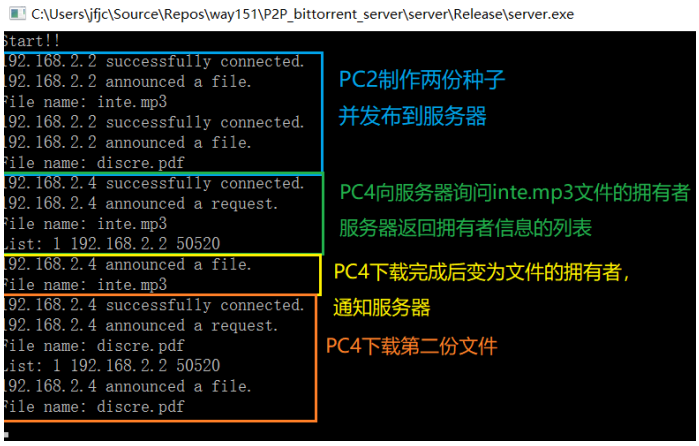


图 7: 1-1实验　PC1作为服务器

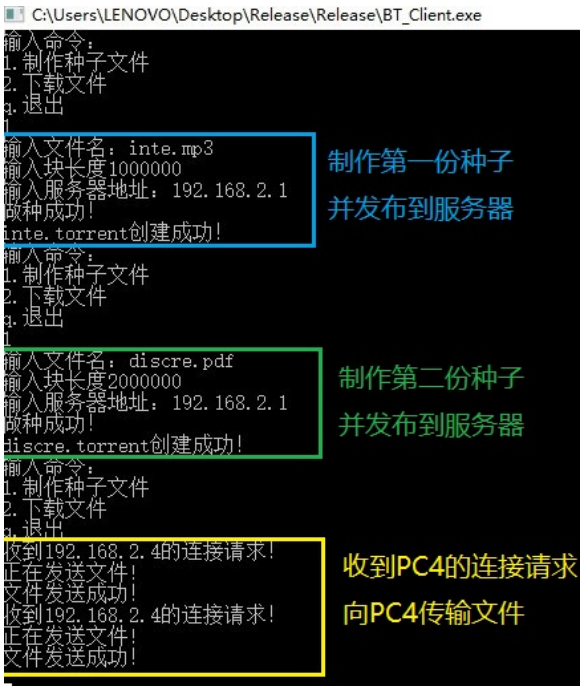


图 8: 1-1实验　PC2作为文件发布者

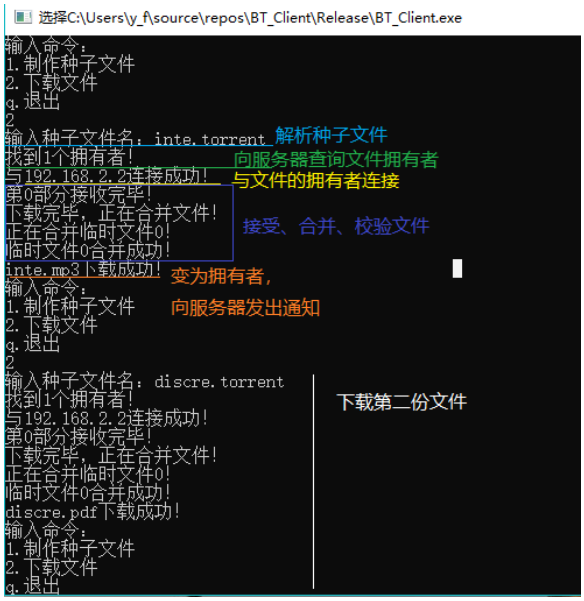
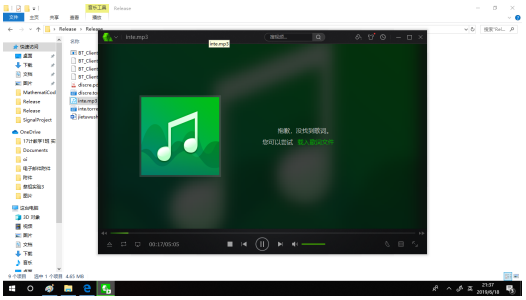


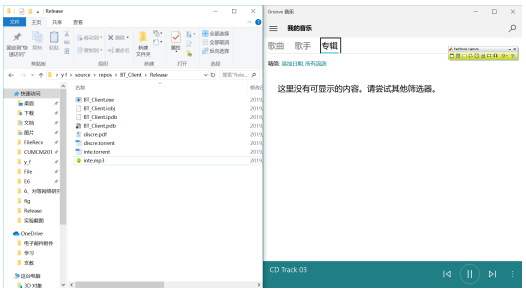
图 9: 1-1实验　PC4作为下载者

(a) PC2上的discre.pdf



(b) PC2上的inte.mp3

(c) PC4接受的discre.pdf



(d) PC4接受的inte.mp3

图 10: PC2发布的文件和PC

4.2 1-3实验

接下来我们尝试了1个下载者向3个拥有者下载文件。PC1作为服务器(图12)。PC2(图14)、3(图15)、5(图16)作为文件发布者。PC4从三个文件拥有者处下载文件(图13)。1-3实验网络拓扑结构图如11所示。

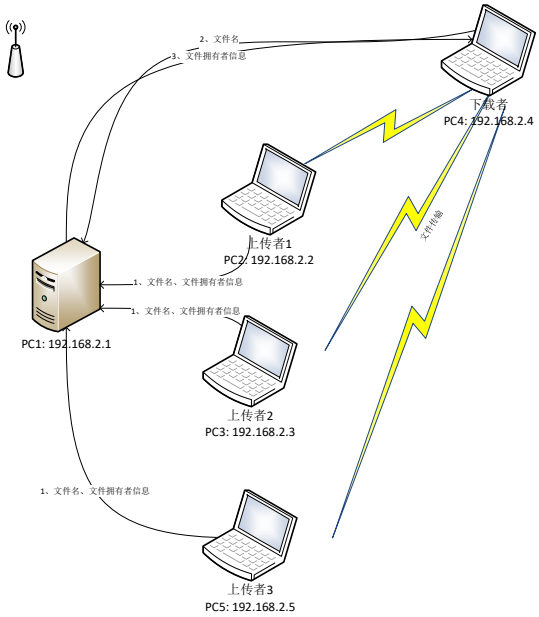


图 11: 1-3实验　网络拓扑结构图

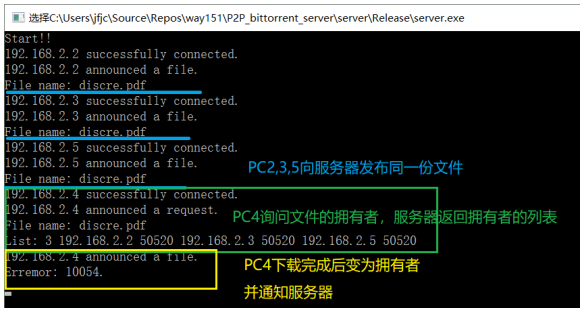


图 12: 1-3实验　PC1作为服务器

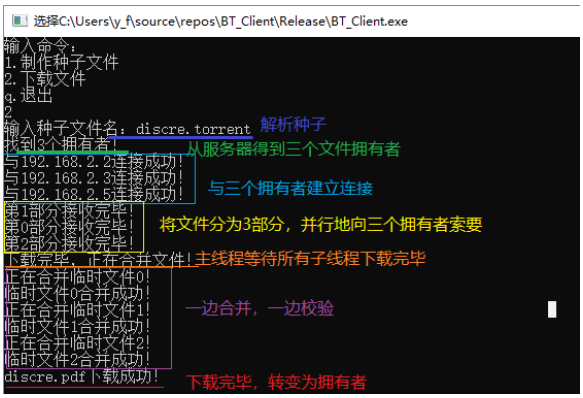


图 13: 1-3实验　PC4作为下载者

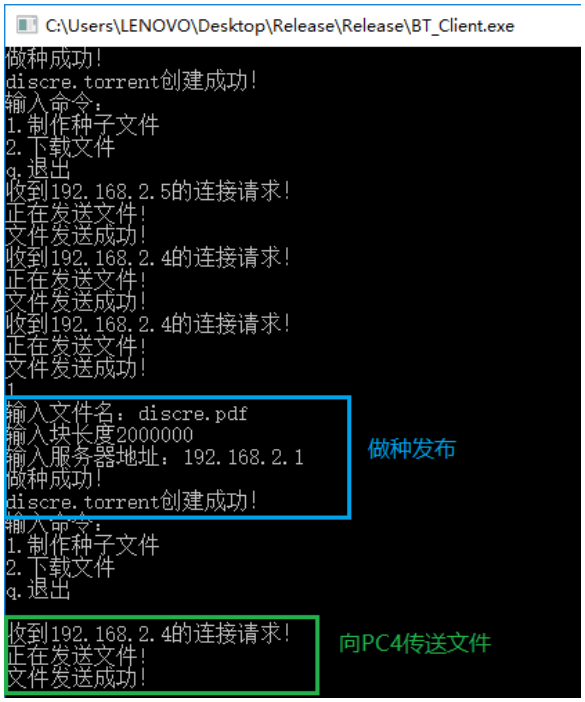


图 14: 1-3实验　PC2作为文件发布者

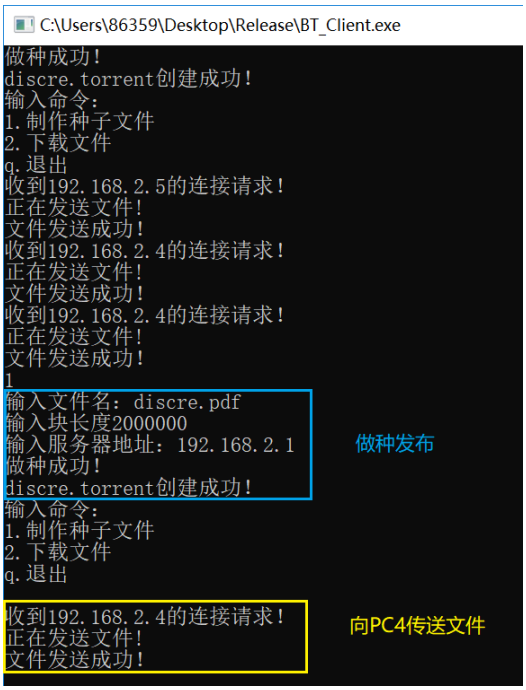


图 15: 1-3实验　PC3作为文件发布者

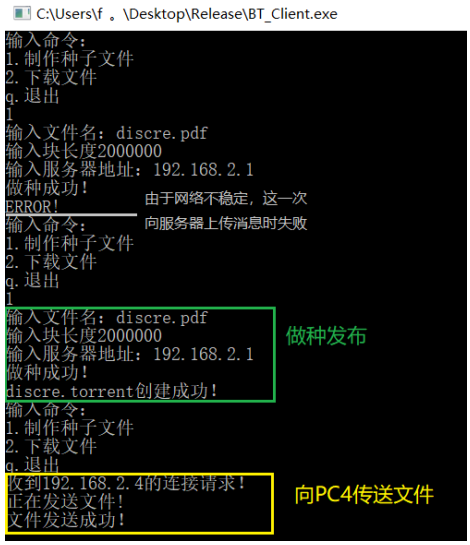


图 16: 1-3实验　PC5作为文件发布者

4.3 2-1实验

这一步尝试2个下载者同时从1个拥有者处下载文件。PC1作为服务器(图18)。PC2作为文件发布者(图19)。PC3(图20)，4(图21)从PC2处下载文件。2-1实验网络拓扑结构图如图17所示

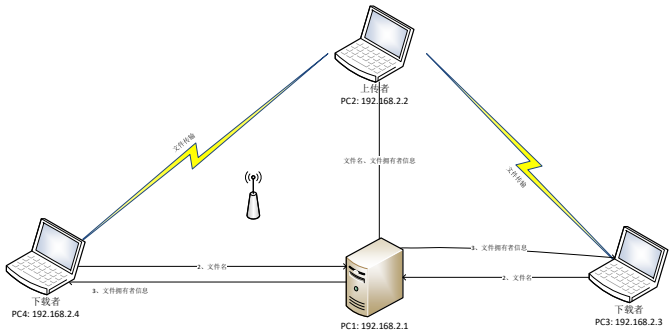


图 17: 2-1实验　网络拓扑结构图

图 18: 2-1实验　PC1作为服务器

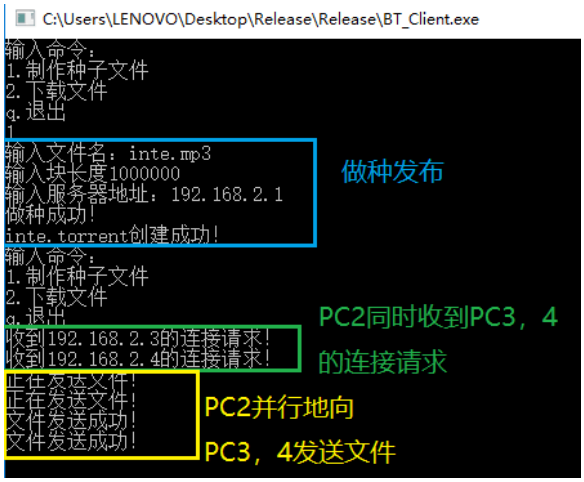


图 19: 2-1实验　PC2作为文件发布者

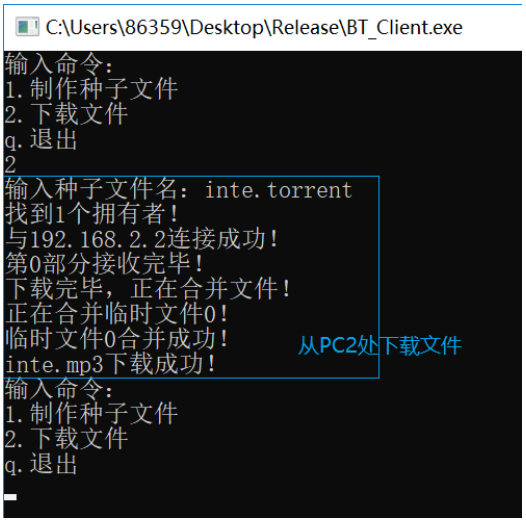


图 20: 2-1实验　PC3作为下载者

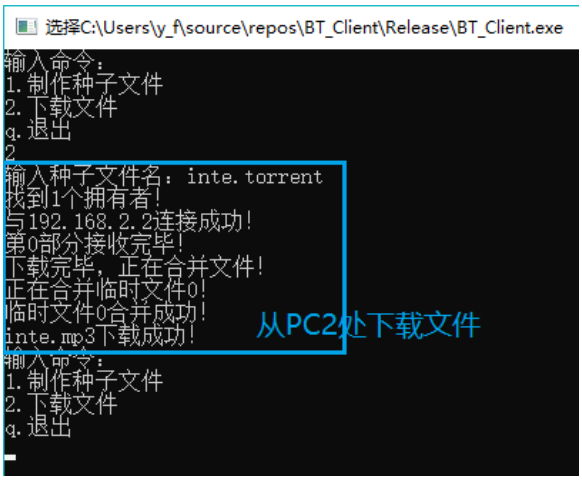


图 21: 2-1实验　PC4作为下载者

4.4 2-2实验

最后我们尝试了多个下载者，多个文件拥有者之间传输文件。PC1作为服务器(图23)。PC2首先发布一份文件(图24)，然后PC3从PC2处下载。PC3下载完之后就变为了文件拥有者(图25)。之后PC4(图26)，5(图27)同时从PC2，3处下载文件。2-2实验网络拓扑结构图如图22所示

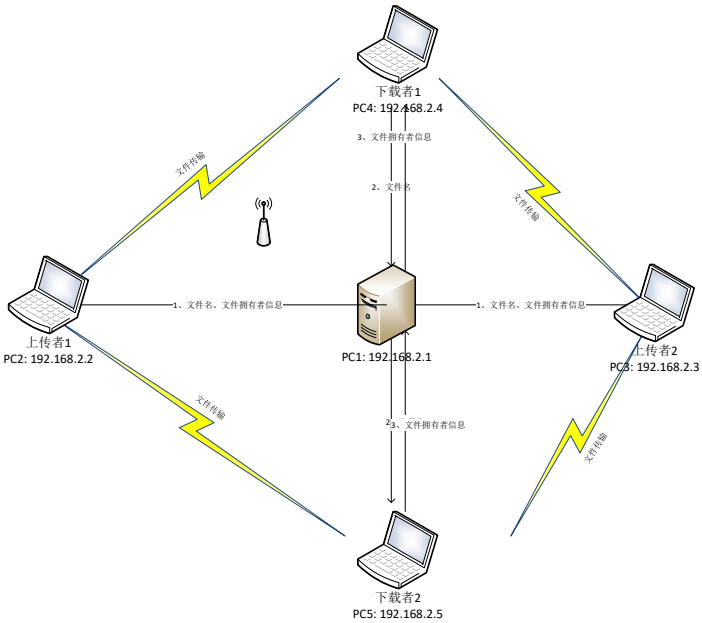


图 22: 2-2实验　网络拓扑结构图

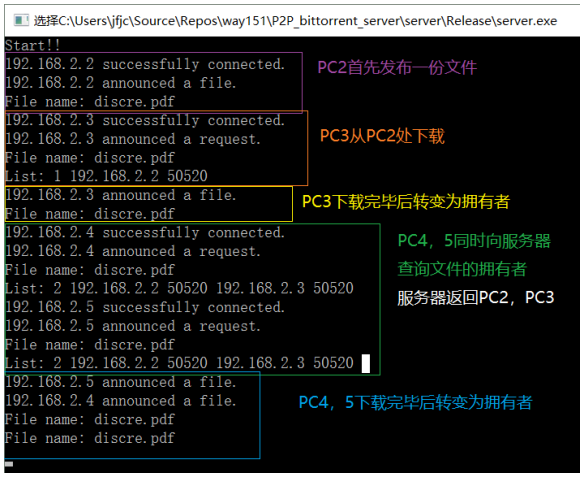


图 23: 2-2实验　PC1作为服务器

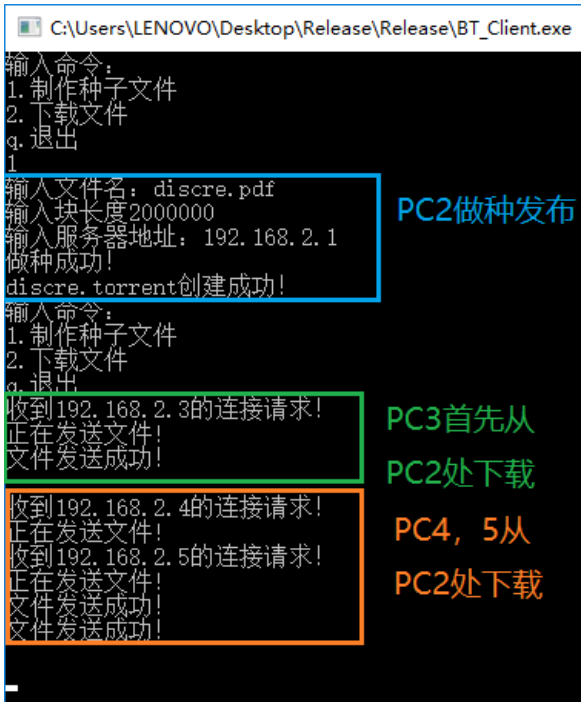


图 24: 2-2实验　PC2作为第一个文件发布者

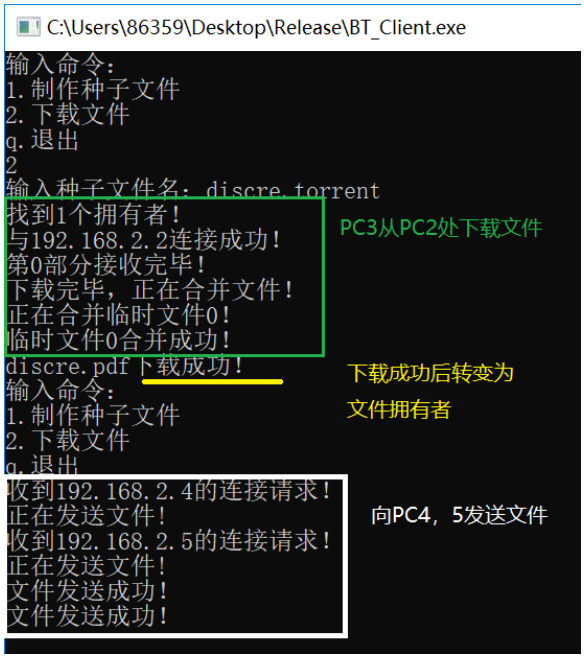


图 25: 2-2实验　PC3首先作为下载者，PC2处下载文件；下载成功后变为文件拥有者向PC4，5发送文件

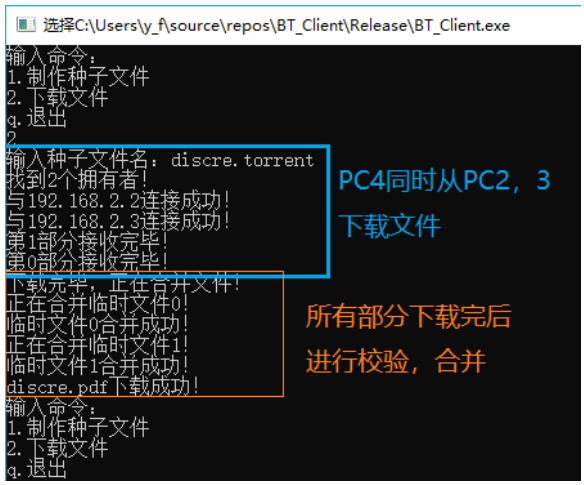


图 26: 2-2实验　PC4作为下载者

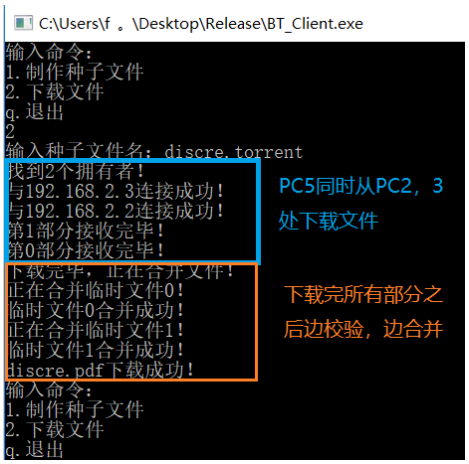


图 27: 2-2实验　P52作为下载者

5 总结

本文简要介绍了对等网络的特点，并以BitTorrent协议为例进行深入研究。通过对BitTorrent协议文档的整理，阐述该协议的一些基本概念以及工作原理。同时给出具体的实现方法，并进行了详尽的实验分析，给后面的研究者提供了一定的参考。

参考文献

[1] Wikipedia. Peer-to-peer. https://en.wikipedia.org/wiki/Peer-to-peer.

[2] Tencent. QQ live. http://live.qq.com/.

[3] Bitcoin. Bitcoin. https://bitcoin.org/en/.

[4] Blockchain. Blockchain. https://www.blockchain.com/.

[5] Matteo Gianpietro Zago. Why the web 3.0 matters and you should know about it. https://medium.com/@matteozago/why-the-web-3-0-matters-and-you-should-know-about-it-a5851d63c949, 2018.

[6] Wikipedia. Bittorrent. https://en.wikipedia.org/wiki/BitTorrent.

[7] ZDNet. Cachelogic says 35 https://www.zdnet.com/article/cachelogic-says-35-of-all-internet-traffic-is-now-bittorrent/.

[8] Ernesto. Thunder blasts utorrent's market share away. https://web.archive.org/web/20160220105711/https://torrentfreak.com/thunder-blasts-utorrents-market-share-away-091204/, 2009.

[9] uTorrent. utorrent. https://www.utorrent.com/intl/zh/.

[10] Facebook. Facebook. https://www.facebook.com/.

[11] Twitter. Twitter. https://twitter.com/?lang=en.

[12] Digital Forensics Community in Korea. Understanding of bittorrent protocol. http://dandylife.net/docs/BitTorrent-Protocol.pdf.

[13] Bram Cohen. Bep 3 - the bittorrent protocol specification. http://bittorrent.org/beps/bep_0003.html, 2017.

[14] Arvid Norberg. Bep 29 - utorrent transport protocol. http://bittorrent.org/beps/bep_0029.html, 2017.

[15] Olaf van der Spek. Bep 15 - udp tracker protocol for bittorrent. http://bittorrent.org/beps/bep_0015.html, 2017.

[16] Andrew Loewenstern and Arvid Norberg. Bep 5 - dht protocol. http://bittorrent.org/beps/bep_0005.html, 2017.

[17] Raymond Lei Xia and Jogesh K. Muppala. A survey of bittorrent performance, 2010.