Computer Networks 100 (2016) 75–89

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

An architecture for client virtualization: A case study

Syed Arefinul Haque[a], Salekul Islam[a,*], Md. Jahidul Islam[a], Jean-Charles Grégoire[b]

[a] United International University, Dhaka, Bangladesh
[b] INRS-EMT, Montréal, Canada

ARTICLE INFO

Article history:
Received 30 November 2015
Revised 17 February 2016
Accepted 18 February 2016
Available online 26 February 2016

Keywords:
Edge cloud
P2P
BitTorrent
Virtual client
Cloud-based server
Web-RTC

ABSTRACT

As edge clouds become more widespread, it is important to study their impact on traditional application architectures, most importantly the separation of the data and control planes of traditional clients. We explore such impact using the virtualization of a Peer-to-Peer (P2P) client as a case study. In this model, an end user accesses and controls the virtual P2P client application using a web browser and all P2P application-related control messages originate and terminate from the virtual P2P client deployed inside the remote server. The web browser running on the user device only manages download and upload of the P2P data packets. BitTorrent, as it is the most widely deployed P2P platform, is used to validate the feasibility and study the performance of our approach. We introduce a prototype that has been deployed in public cloud infrastructures. We present simulation results which show clear improvements in the use of user resources. Based on this experience we derive lessons on the challenges and benefits from such edge cloud-based deployments.

1. Introduction

A new trend in service deployment in the Internet, based on cloud computing and virtualization, shifts the location of applications and infrastructures from the user device to the network to reduce the costs associated with the management of hardware and software resources [1]. In such systems, service providers can provide simplified software installation, maintenance and update [2]. As cloud technology has become more popular, we have seen the emergence of edge clouds [3], that is, datacenters deployed by Internet access providers, in close proximity to customers. With such facilities comes new opportunities to shift traditional computer based applications, which could not otherwise have easily been virtualized because of their complex control plane, towards the cloud. Peer-to-peer (P2P) applications would be an example of such applications.

P2P networks are popular tools for content-sharing because they provide better scalability and fault tolerance than the traditional client-server model of computing. A P2P network can be described as a network of cooperating peers that work together to complete tasks and share resources in the Internet. Such a network is composed of numerous distributed, heterogeneous, autonomous, and highly dynamic peers with which participants share a part of their own resources such as processing power, storage capacity, software, and content [4]. P2P networks have no single point of failure and the network can grow and shrink without sacrificing the functionality of the system. Bandwidth utilization is better in such networks as the peers communicate directly with each other rather than through a hub which would present a bottleneck [5].

P2P applications are often used for file sharing. One example of a popular P2P file sharing application is

* Corresponding author. Tel.: +880 1820182777.
E-mail addresses: arefin@cse.uiu.ac.bd (S.A. Haque), salekul@cse.uiu.ac.bd (S. Islam), jahid@cse.uiu.ac.bd (Md.J. Islam), gregoire@emt.inrs.ca (J.-C. Grégoire).

http://dx.doi.org/10.1016/j.comnet.2016.02.020
1389-1286/

BitTorrent. Large corporations like the Blizzard Inc. use P2P systems to simultaneously distribute bandwidth-intensive content to thousands of users without requiring major infrastructure investments [6]. On-demand media streaming is another popular P2P application. PPTV [7] delivers video content by streaming but peers can watch and share different parts of a video at the same time thus reducing server load [8]. Distributed P2P file storage systems like Freenet [9] anonymously publish, replicate and retrieve data distributed among peers. Skype [10] is a P2P application which enables voice and video calls over the Internet to any other Skype user. P2P model can even be used to virtualize physical objects and service construction processes on smart spaces [11]. Several distributed scientific projects like Seti@Home [12] use P2P public distributed computing to share processing cycles. BitTorrent Sync [13] is a recent addition to this paradigm that synchronizes files between devices on a local network, or between remote devices over the Internet.

For running such P2P applications a user normally has to install a client application on her device. The single most important task of these applications is to exchange data between peers, but apart from that, they may also perform routing/forwarding, content validation (e.g. hash checking) and implement different mechanisms for efficient bandwidth usage. As a result these client applications consume various resources including processing power, memory and bandwidth. Also, NAT traversal is an issue in P2P applications as most of the peers usually do not have globally routable IP addresses [14]. A local application requires a prior installation and has to be regularly updated for maintenance, which can be a burden for the user.

In this paper we explore how P2P applications benefit from virtualization in an edge cloud environment and study the architectural tradeoffs. For this process, we introduce *SimpleBit*, a virtual terminal-based P2P client which follows the BitTorrent protocol but is deployed on a remote cloud server. This architecture was first introduced in [15], albeit very briefly. An end user accesses and controls SimpleBit using a standard web browser, which reduces the requirements and the load on user devices by offloading the control and session management tasks to the remote server. We study two different architectures of SimpleBit:

1. A P2P-type direct download architecture where the files are downloaded directly from the peers to the end user's device.
2. A surrogate-based proxy downloader where the files are first downloaded by the surrogate server and then transferred to the end user's device.

The rest of the paper is organized as follows: Section 2 briefly describes BitTorrent and discusses a detailed overview of how the BitTorrent client works. Section 3 introduces and discusses virtualization. In Section 4 we present the architecture of SimpleBit virtual P2P client. We have designed two orthogonal models: SimpleBit with proxy downloader is explained with its implementation in Section 5 and SimpleBit with P2P download is presented with simulation results in Section 6. In Section 7 we explain the lessons we have learnt on the challenges and benefits from the edge cloud-based deployments. In Section 8 we summarize and compare existing efforts that are related to our work. Finally, Section 9 concludes the paper.

2. Dissecting BitTorrent

BitTorrent is the most popular P2P application for distributing large size files. It is implemented as a hybrid P2P system. Most of the interactions are done directly between peers but initial and further occasional interactions with a server are required for locating peers [16]. A user gets the information about the peers using a meta-information (metainfo) file (or metafile). The architecture of BitTorrent is shown in Fig. 1. It can be summarized in the following points:

1. A peer willing to download a shared content has to download the corresponding metafile from a web server and uses it to identify a *tracker* for that content.
2. The peer contacts the tracker and requests a list of peers that are already participating in the torrent (i.e., sharing that content).
3. The tracker replies with a list of peers with their IP address and access port.
4. The peer selects a number of peers from the list provided by the tracker and establishes a connection with them.
5. When connections are established, the peer exchanges pieces of that file with the neighbors.

A set of peers using the same metafile to share a particular file are part of the same *swarm*. A tracker can introduce the newly joined peer to multiple swarms at the same time. A file is divided into fixed-size pieces and peers exchange the pieces with each other. When a piece is downloaded its SHA1 hash is computed and compared with the value in the metafile. If the values match then the piece is declared downloaded and made available for downloading to other peers.

BitTorrent uses pipelining to keep the TCP connections operating at full capacity [17]. For this reason each piece is divided into many sub-pieces (usually 16 KB-sized) which are called blocks or chunks. To reduce the load on seeders (a peer who has access to the whole shared file) a peer downloads pieces not only from the seeders, but also from other peers (which are called leechers).

In this section, we have carefully studied the components of the architecture of BitTorrent based on its specifications [18]. Then we have arranged them into different modules from a developer's perspective. The modules are identified as part of either data plane or control plane, or both. We define the control plane as the part of the architecture which is concerned with drawing up the network map, or handling state oriented messages between other peers or servers. Otherwise, the data plane is defined as the part where the actual data is transferred between the participating peers. Implementation of a full BitTorrent system can be divided into three distinct parts, as follows.

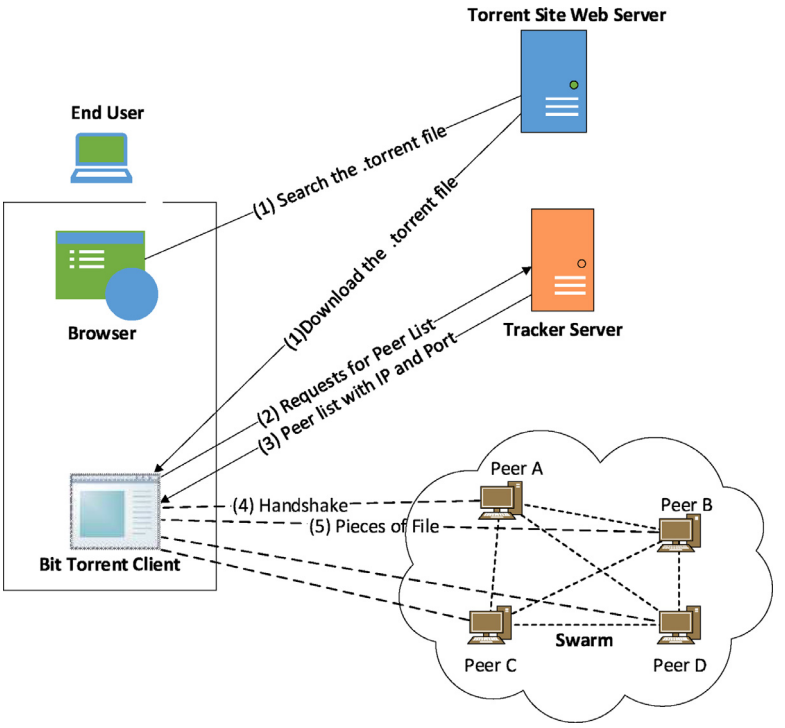


Fig. 1. BitTorrent's file sharing process.

2.1. Metafile processor (control plane)

The Metafile processor is a module which is required by both the tracker server and the client running in each peer's machine. A peer willing to publish or share some content using the BitTorrent protocol has to create a metafile holding the description of the content that is to be shared along with contact information for the tracker server. A peer willing to retrieve the same content using the BitTorrent protocol must find the metafile and decode it to start the download process. This file is encoded in a special format called *bencoding* [18]. This module decodes the bencoded file to extract useful information for other modules. The metafile is structured as a dictionary with the following keys and values:

- `announce` The URL of the torrent tracker;
- `info` An embedded dictionary with following keys:
  - `name` Suggested name to download the file;
  - `piece length` Number of bytes each piece of the shared file is split into;
  - `length` If the content is a single file it indicates the size of the file in bytes;
  - `files` If the content is composed of multiple files it contains a dictionary with the length and the ordered sequence of the directories to save the files;
  - `pieces` The SHA1 hash of the pieces for future verification of the integrity of the downloaded pieces.

2.2. Tracker communication module (control plane)

A peer willing to initiate P2P file sharing contacts the tracker server through this module using the `announce` key in the metafile. A tracker server is an HTTP server which maintains a list of connected peers as well as information about the evolution of their status. It answers to a peer's requests for the addresses and ports of other peers. To support efficient data sharing between clients, it also tracks which fragment of that file each peer possesses. To contact the tracker, this module must send a standard `HTTP GET Request` with the following parameters [18]:

- `info_hash` A URL-encoded 20-byte SHA1 hash of the value of the `info` key from the metafile.
- `peer_id` A 20-byte self-designated ID of the peer.
- `port` The port number the peer is listening to for incoming connections from other peers.
- `uploaded` The total amount of bytes that the peer has uploaded in the swarm.
- `downloaded` The total amount of bytes that the peer has downloaded from the swarm.
- `left` The amount of bytes that the peer needs in this torrent to complete the download.

After receiving the `HTTP GET Request`, the tracker responds with a document with the `text/plain` MIME type. It contains a bencoded dictionary with the following keys:

- `failure reason` A human readable string containing an error message with the reason for the failure if the peer is unfit to join the swarm.
- `complete` The number of seeders (optional).
- `incomplete` The number of leechers (optional).
- `peers` A bencoded list of dictionaries containing a list of peers. Each peer is a dictionary containing the keys `peer_id`, `ip` and `port`.

Each peer contacts the tracker typically every 30 min. When a peer leaves the torrent, it informs the tracker to have its name removed. If the peer leaves silently the tracker automatically removes the peer after a predefined period of time, following the last connection of the peer to the tracker, i.e. a timeout [19].

2.3. Peer-to-Peer communication module (data plane)

This module facilitates communications between two peers. It handles creating and listening for peer handshakes and connections. The information about the peers is gathered from the tracker using the tracker communication module. Peer connections are established using the TCP protocol to the appropriate host IP and port.

Handshaking. Before any data transfer two peers must establish a connection via a `handshake`. A handshake exchanges a string of bytes with the following structure:

- `Protocol Name` The name of the protocol in ASCII, such as `BitTorrent protocol`.
- `Reserved` Reserved for future extensions.
- `Info Hash` The SHA1 hash of the `info` value.
- `Peer ID` The self-designated `peer_id`.

Message Communication. Following the handshake both ends of the TCP channel send messages to each other in a completely asynchronous fashion. These messages have two different types:

1. State-oriented Messages. These messages inform peers of changes in the state of neighboring peers.
2. Data-oriented Messages. These messages handle requesting and sending of data.

A peer-to-peer message contains three fields: the `Message Length`, the `Message ID` and the `Payload` [18]. In the following we briefly present some important messages:

- `interested` Informs the peer that the sender peer wants some pieces that the peer has.
- `not interested` Informs the peer that the sender peer is not interested in any pieces that the peer has.
- `have` A message sent to all peers once the client has a complete piece.
- `bitfield` A message only sent after the handshake is completed to tell peers which pieces the client has.
- `request` A message sent to a peer indicating that they would like to download a given piece. The payload of the message has following fields namely `index`, `block offset` and `length`
- `Piece` The payload of this field has three fields namely `index`, `block offset` and `data` (contains the binary data).

Additionally, the client sends `keep-alive` messages to keep the connections open.

2.4. File handling module (data plane)

A file is split into smaller pieces which are of fixed size and treated as binary data. Each piece can then be assigned a hash code, which can be checked by the downloader for data integrity. The most common piece sizes are 256 KB, 512 KB and 1 MB. The file handling module manipulates raw binary data in the local storage. When the pieces of the files start arriving at the client, the file handling module temporarily stores them in the local file system. When all the pieces of a single file have arrived, they are merged into the desired file.

3. Virtualization

The idea of virtualization has evolved during the early days of computing when the virtual systems in mainframes [20] were designed. In general, the process of recreating a (virtual) hardware or software environment by emulation on top of a real system is called virtualization. The use of virtualization is quite broad including network architecture virtualization, desktop virtualization, or client program virtualization. The use of virtualization has emerged due to the increased popularity of Cloud computing. Note that virtualization of hardware and computing resources is one of the fundamental building blocks of Cloud computing [21].

Virtualization in networking is mostly observed at the network architectural level for hiding the underlying infrastructure network. For example, *Cabo* [22] presents a high–level architecture for a flexible and extensible system that supports multiple simultaneous network architectures through network virtualization. Cabo identifies two different entities: infrastructure providers (e.g. the ISPs) who manage the substrate resources and service providers who operate their own customized network inside the allocated slices.

Desktop virtualization or virtual network computing [23] can be achieved in a number of ways and with various technologies, the most popular being Microsoft's Remote Desktop Protocol [24] or Citrix's XenDesktop [25]. A virtual desktop means in essence that applications run on a machine different from the user's. The desktop is presented to the user as an image of the one running remotely and local keyboard and mouse are used to interact with it, in typical fashion. The virtual desktop server, located on a remote platform, is responsible for on-demand application provisioning, managing user settings and the running the Operating System's functions. The matching virtual desktop's image is rendered at the user's platform through a virtual desktop protocol. Since the user's platform has no permanent storage, the remote data center works as its virtual storage.

With the aid of virtualization cloud computing dynamically pools computing resources from a cluster of servers [26]. Depending on the demand cloud computing dynamically assigns and reassigns virtual resources. In the case of the Infrastructure-as-a-Service (IaaS) model, a virtualization layer, also known as the infrastructure layer uses well-known virtualization technologies such as Xen, KVM and VMware. Thus, a pool of storage and computing resources are created by partitioning the underlying physical resources.

Unlike virtualization of the whole operating system or network architecture, virtualization could be used in

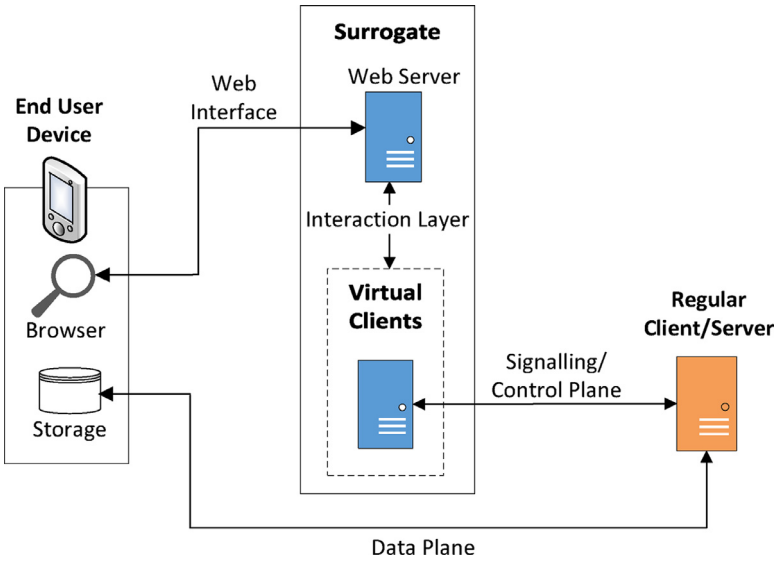


Fig. 2. A generic architecture for virtual clients.

smaller levels at the client side applications. Fig. 2 presents a generic architecture that facilitates client application virtualization. Essential in this architecture is the existence of a cloud-based server, which we shall call a *surrogate*, which is meant to take most of the control load off the terminal client. The surrogate deploys virtual client applications and thus acts as a remote server for the user to access, organize, provision and monitor their application's control plane related services.

By transferring most of the control plane's complexity to the surrogate, a simple user device with functionalities restricted to operating the GUI and data transfer could be used. Web-based GUIs are quite commonly used now for virtual clients, say for email or desktop applications, and most user-device platforms are equipped with at least one and often several web browsers. The web browser is the most commonly deployed software platform across computing devices of all sizes and format available to developers today. Present cumulative industry growth projections give us an estimation that there will be 20 billion Internet-connected devices by 2020 [27]. Web browsers are getting ever richer features every day and so are browser-based applications. The type of platform, its manufacturer, or the nature/revision of its operating system do not matter for browser-based applications.

A surrogate implements a Web server, which receives a user's input through the GUI running on the web client inside the user device. An interaction layer is needed between the Web server and the virtual client for establishing communications between them. The virtual client communicates with a regular client (e.g., in case of P2P communications) or with a server (e.g., in case of a SIP server). The existence of a surrogate is transparent to the other regular clients/servers in the network, i.e., they behave as if they were communicating with a traditional client applications. Thus, all signaling or control plane related activities will be originated and also terminated at the virtual client. However, data plane activities will be performed by the end user's device. Such examples include uploading/downloading files, video streaming, etc.

Note that the term surrogate was used previously to demonstrate the idea of *edge cloud*, where a wide number of services including virtual IMS client [28], virtual SIP client *edge cloud*, transcoding [3] and content processing [29] could be deployed. Following the same rationale, a SimpleBit P2P client would be deployed in an edge cloud.

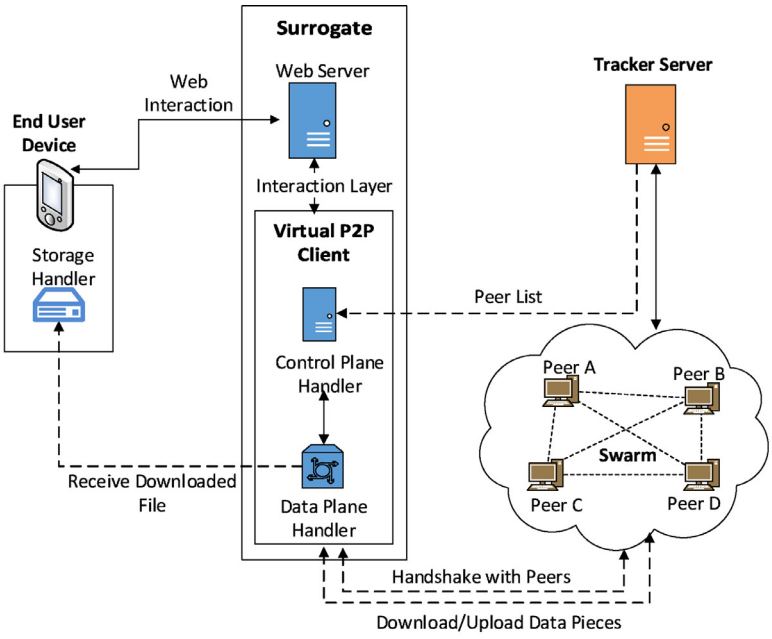
4. Virtual P2P client architecture

The functionalities of a P2P client can be broadly divided into control plane and data plane. Note that these functionalities will be implemented in two places, at the user device and inside the surrogate. Depending on the locations where these functionalities are implemented two orthogonal models may be developed, which are shown in Fig. 3. In this figure, we present two different SimpleBit P2P client architectures: a proxy downloader model and a pure P2P downloader model. These two models are illustrated in Fig. 3a and b, respectively. Comparing these architectures will allow us to explore trade-offs in the location of some functionalities.

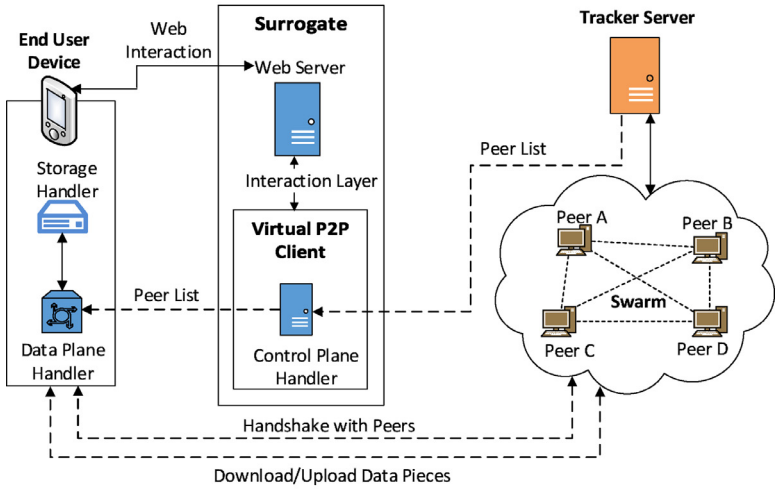
We would like to mention that the proposed architecture has been designed to be usable by any P2P protocol. Since BitTorrent is the dominant P2P protocol at this stage, we describe in the following how our model can implement a virtual P2P client. Note however that although some parts of this description are BitTorrent-specific, this model, with minor modifications could easily be adopted to other and any future P2P protocols.

4.1. Proxy downloader model

We can offload both control and data planes activities from the user device to the surrogate, with the great benefit of reducing resource usage and implementation complexities on the user side. Thus the surrogate acts as a



(a) With a Proxy Downloader



(b) With a P2P download

Fig. 3. Virtual P2P client architecture of SimpleBit.

proxy for the user's P2P client. In this model, different pieces of a file will be downloaded to the surrogate, then merged together to rebuild the file. Depending on the device, the whole file could incrementally be transferred to the user as the pieces are received, or conversely be held by the surrogate for a later, completed file transfer. We designate this model as the SimpleBit with proxy downloader. This model is illustrated in Fig. 3a. We can see in the figure that data plane is included in the surrogate server which completes all download activities and then notifies the user device when download is completed. The implementation details of this model have been described in Section 5, where the surrogate first completes the whole download then provides a direct download link to the user.

4.2. Pure P2P downloader model

In the pure P2P downloader model, the data plane handler is implemented inside the user device. The surrogate is responsible for fetching the peer list from the tracker server. This list is forwarded to the user device, which then communicates with the peers. Thus, the client in the user device directly receives the data pieces from other peers. We designate this model as the SimpleBit with P2P download. This model is illustrated in Fig. 3b, where the data plane handler is implemented inside the user device, but the control plane-related activities are offloaded to the virtual P2P client residing inside the surrogate. This model is further studied with a simulation model in Section 6.

There are clear trade-offs in terms of server and client extra complexity and resource consumption to handle these different scenarios, but they do not pose any specific challenge.

4.3. Handling the functions of control and data planes

In the following we describe the different handlers, which implement all functionalities of the control and data planes. In case of BitTorrent type P2P applications, collecting chunks of data from different peers and then merging them to produce the whole file is challenging. Therefore, we introduce here a storage handler, which is in general a part of the data plane handler.

Control plane handler. The control plane handler is responsible for registration, processing HTTP request/response messages and interaction with the surrogate, the tracker server and other peers. It consists of a metafile processor and the virtual P2P client running in the surrogate. It replicates all the control plane functions of the existing architecture, which include metafile translation, finding the address and state of available peers from the tracker and sending them to the peer device. Peers using the virtual client also send their state oriented messages to the surrogate and the surrogate in turn sends them to the tracker server.

Data plane handler. The data plane handler may act differently based on the model chosen for the virtualized P2P client. In the case of the Proxy Downloader model, the data plane handler will be deployed inside the surrogate. In the P2P download model, the data plane handler would be deployed in the web browser of the user device. The data plane handler receives the list of active peers from the control plane handler of the application. When it finds a suitable peer with the required piece of data, it requests that piece from the peer. It opens and maintains multiple data connections from the peers. This is the most challenging module to be implemented in the end user device. We will discuss this in the next sections.

Storage handler. The storage handler is a modified, enhanced downloader, which can download the data from multiple peers and keep track of them. It can keep the incremental chunks of the file in the local storage or in the RAM. In the case that the accumulated size of the file chunks exceeds the maximum allowable size in the local storage or RAM (e.g., if the user device has very limited RAM), it can start writing them to a secondary storage. When to write data to a secondary storage rather than holding it in the local storage or RAM is a dynamic decision, typically based on user profile. An implementation should carefully address the memory capability of a wide range of devices from PC to tiny cell phones.

5. SimpleBit with proxy downloader

In this model the surrogate server acts as a proxy downloader. The message sequence for this model is shown in Fig. 4. The different steps of this figure run as follows:

1. A user would connect to the web server running inside the surrogate through a web browser. He provides the metafile or the magnet URI of the desired file to the user interface.
2. The web server transfers the metafile to the virtual P2P client running inside the same or a different server. The virtual P2P client handles the control plane activities on behalf of the user and creates a session with the user. It communicates with the tracker server requesting the peer list from the active swarm.
3. The tracker responds to the virtual P2P client with the peer list.
4. The virtual P2P client completes the handshakes with the peers and finds the active ones.
5. The virtual P2P client requests the file pieces from the active peers, and thus different pieces of a file will be downloaded in the virtual P2P client.
6. The virtual P2P client periodically exchanges state-oriented messages with the peers to keep them aware of the availability of different pieces of the file.
7. The virtual P2P client also sends keep-alive messages to the active peers to check if the opened peer connections are alive.
8. The virtual P2P client sends state oriented messages to the tracker server to keep the tracker updated of the state of the user participating in the swarm.
9. After the last piece of the file has been downloaded correctly, the surrogate merges them all to produce the file, which will be written to the disk of the server and the storage handler of the user device is notified of the availability of the file.
10. When the file download is completed the user's browser window is notified of the completed download and the link to the downloaded file is available as long as the user's browser window is open. It can even forward and store the downloaded file to a user-designated cloud storage (e.g. Dropbox [30]) using a suitable API .

5.1. Cloud implementation

We have implemented a SimpleBit proxy surrogate as a NodeJS [31] web application. We have used ExpressJS [32], which is a minimal and flexible NodeJS web application framework that provides a robust set of features for web and mobile applications and socket.io [33] which enables real-time bidirectional event-based communication from the server to the client. They provide a thin layer of fundamental web application features, without obscuring NodeJS features that are being used in the back-end. We have deployed the instances of this surrogate server in three different environments: in a desktop computer connected to the Internet through a residential cable connection with 50 Mbps bandwidth, a Linode cloud platform [34] and an Amazon EC2 cloud platform [35].

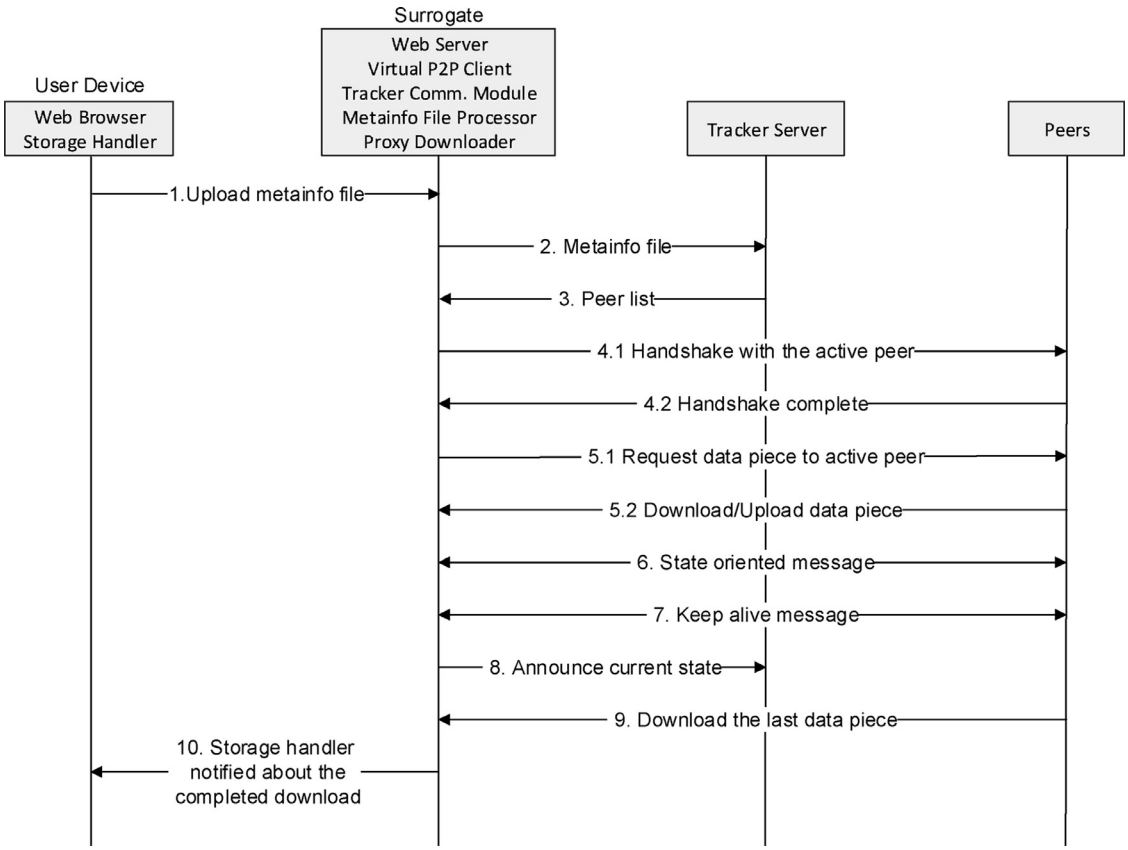


Fig. 4. Message Sequence of SimpleBit with the proxy downloader.

The surrogate server hosts an ExpressJS application including the web-based user interface. The web server receives the user input through the GUI running on the web client which is built using the Bootstrap [36] framework. The user requests to download a file by entering a magnet link in a form provided in the GUI through the web interfaces. When the form is submitted a `post` request will be submitted to the same route. After the post request is validated a new page will be rendered where the download links would appear. In the meanwhile, the virtual P2P client running behind takes the magnet link as an input and starts the proxy download.

We have borrowed the APIs provided by the WebTorrent [37] library of NodeJS to build the virtual P2P client. It is currently in active development by open source communities. It provides a simple torrent client module, using TCP and UDP to talk to other normal torrent clients such as BitTorrent, μTorrent, transmission or deluge. After a file has been found in any swarm, it exposes it as a stream and fetches pieces from the network on-demand. As soon as the surrogate server finishes downloading a file using the APIs provided by WebTorrent, a download link is sent to the user who can download it directly to his device.

5.2. Performance analysis of the proxy downloader

As mentioned earlier, we have deployed instances of our implementation in three different environments. Table 1 shows different configuration parameters of these environments.

To study the performance of these instances, we measure the average download time for files of different sizes. In our implementation, we are saving two timestamps: the start time of the download and the arrival time of the final piece of the file. The download time for a file has been calculated in seconds by taking the difference between these two timestamps. The average download time is measured by repeating the download ten times in a row. We use the same torrent file for different instances and the downloading of a file has been started at the same time in three different instances. To observe the variations of these download times, for a specific file download, we have calculated the standard deviation over the ten download times we recorded. Table 2 shows the average download time with standard deviations for files of different sizes in three environments. Note the standard deviations have been presented in brackets.

Except for the case of the 19.5 MB file, the instances running in the cloud outperform the local instance running in a desktop. Many factors including upload/download bandwidth, aggregate bandwidth at the seed(s), number of peers and seed(s), degree of nodes, finding rare pieces, presence of high bandwidth peers, etc. influence the download time of a file [38]. Since we started downloading in different environments at the same time, many factors remained unchanged. However, available download

Table 1
Configuration of different instances.

Instance	Local desktop	Linode	Amazon EC2
Instance Type	–	Linode 1GB	Tiny (t2.micro)
Processor	2.7 GHz Intel Core i5	Intel Xeon CPU E5-2680 v3 @ 2.50GHz	Intel Xeon 2.5GHz
RAM	16 GB	1 GB	512 MB
Download Bandwidth	50 Mbps	125 Mbps (Approx.)	500 Mbps (Approx.)
Operating System	Ubuntu	Ubuntu	Ubuntu
Location	Boston, USA	Newark, USA	N. Virginia, USA

Table 2
Average download time with standard deviations in second.

File Size	Local Desktop (s)	Linode (s)	Amazon EC2 (s)
8.9 MB	11.24 (3.24)	6.41 (2.57)	6.17 (0.93)
19.5 MB	16.53 (5.5)	17.68 (6.87)	19.85 (6.49)
43.5 MB	22.68 (5.23)	13.99 (4.99)	12.25 (3.9)
71.0 MB	29.36 (12.69)	17.82 (13.13)	18.37 (16.02)
86.1 MB	29.77 (2.73)	10.83 (1.57)	10.98 (0.77)

bandwidths were higher in the cloud platforms and thus reduced the overall average download time. However, it should be noted that several factors can influence the download time. More importantly, the factors we have mentioned are dynamic in nature and may change within a short period of time. As a result, we may experience significant variations in download times if we consecutively download the same file several times. If we observe the standard deviations of average download times, we find large variations in all three environments, as reflected in Table 2.

6. SimpleBit with P2P download

In the SimpleBit with P2P download model, most of the control plane activities are offloaded to the surrogate yet the data plane related activities are handled in the user's device in a P2P fashion. The message sequence of this model is shown in Fig. 5. The different steps of this figure run as follow:

1. A user connects to the web server residing in the surrogate through a web browser. There might be a provision for user authentication. On successful authentication, the end user uploads the metafile or provides the magnet URI of the desired file to the user interface running in the surrogate.
2. The surrogate transfers the metafile to the virtual P2P client running inside the same or a different server. The virtual P2P client communicates with the tracker server requesting the peer list from the active swarm.
3. The tracker responds to the surrogate with the peer list.
4. The surrogate immediately returns the peer list (IP addresses and port numbers of the active peers) to the user device.
5. The data plane handler running in the user device completes the initial handshaking with the active peers.
6. The data plane handler requests the file pieces from the active peers, and thus different pieces of a file will be downloaded to the user device. The data plane handler can simultaneously download and upload file pieces in a P2P fashion. The storage handler in the user device keeps track of the pieces downloaded so far.
7. The data plane handler periodically exchanges state-oriented messages with the peers to keep them aware of the availability of different pieces of the file.
8. The data plane handler also sends keep-alive messages to the active peers to check if the opened peer connections are alive.
9. The data plane handler sends state oriented messages to the tracker server to keep the tracker updated of the state of the user participating in the swarm.
10. Finally, after the last piece has been downloaded correctly, the storage handler will merge them all to produce the whole file, which will be written to the local disk of the user device. When the file download is completed the user's machine will seed the file as long as the user's browser window is open.

6.1. Performance evaluation using discrete event simulation

The implementation of this model in a traditional web browser is not possible using existing web standards. Possible modification in the browser or web standards required for the implementation of this model are discussed in Section 7.2. Thus, in the absence of such web standards we evaluate and compare the performance of a regular BitTorrent model and a modified tracker-free BitTorrent model using discrete event simulation.

We use the ubiquitous discrete event simulator `ns-2` for our evaluation. Note that the goal of this simulation is to compare the performance and resource utilization of a regular BitTorrent client with a BitTorrent client that performs only data plane activities. Therefore, in this study, we compare the performance of a regular BitTorrent model with a tracker free BitTorrent model, using the widely referenced `ns-2`-based model presented in [39].

In the basic model, tracker functionalities are limited to keeping track of the registered peers' addresses and availability. The peers communicate with the tracker periodically to get the list of active peers for downloading the file pieces. Besides, the tracker monitors the peer-to-peer

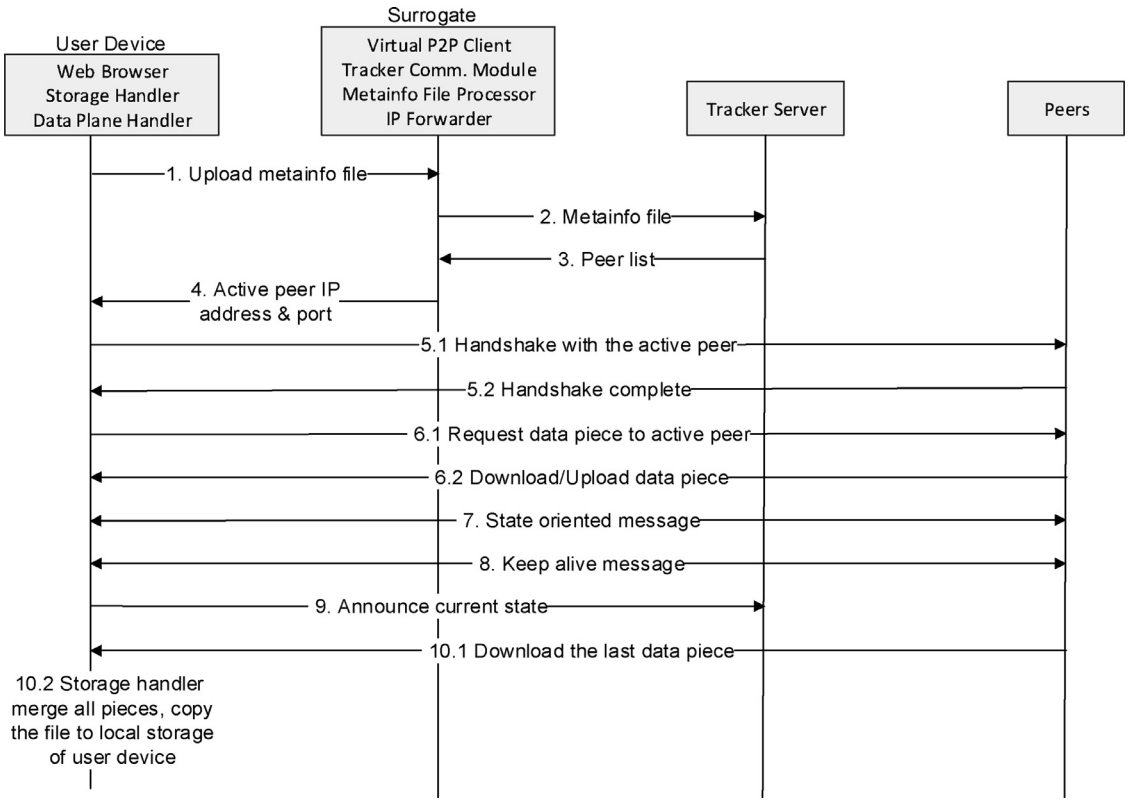


Fig. 5. Message sequence of SimpleBit with P2P download.

Table 3
Simulation parameters and their values in our simulation.

Parameter	Description	Values(s)
file_size	Size of the file being shared	{50, 100, 200} MB
chunk_size	Size of each chunk of the file being shared	256 KB
Q_size	Size of the queue at each access link	25
C_up	Upload bandwidth of each peer	2000 Kbps
C_down_fac	Ratio of download capacity over upload capacity	8
num_of_seeds	Number of initial seeds	1
num_of_peers	Number of peers	100–1500

connections and availability of the peers. On the other hand, in the tracker-free BitTorrent model, a peer does not require any tracker communication or control plane related activities; rather the peers' addresses of the swarm are embedded in the peer's client. This model most closely mimics the activities of a peer that only handles the data plane related activities and leaves the control plane activities to the surrogate.

Table 3 shows the important simulation parameters and the values used in our simulation. For other parameters and detailed information about the implementation procedure, please refer to [39].

6.2. Simulation procedure and results

We have performed our simulation on a 4 GB RAM, Core-*i*5, 2.7 GHz machine running Windows 7. As shown in Table 3, we vary the number of peers in the network (from 100 to 1500) and the file size (50 MB, 100 MB, and 200 MB) to generate different scenarios. For each of these scenarios, the other parameters such as the size of a single file chunk (256 KB), number of initial seeds (1), upload bandwidth of each peer (2000 KBps), etc., remain constant. We model each of these random scenarios 30 times and take the average value. Over the simulation period, we record the download start and finish times of each peer, and the download finish time of the entire swarm. Based on these data, we evaluate and compare the performance of a regular BitTorrent model with the tracker-free BitTorrent model in terms of *average download finish time*, *upload link utilization* and *download link utilization*.

Average download finish time refers to the average time needed by a peer to download the required file. We have utilized the following formula for calculating the *average download finish time*:

$$Average\ download\ finish\ time = \frac{\sum_{i=1}^{num\_of\_peers} T_i}{num\_of\_peers}$$

Here, T_i is the download finish time for the ith peer. Next, using the following equations we compute the *upload link utilization* and the *download link utilization* considering all

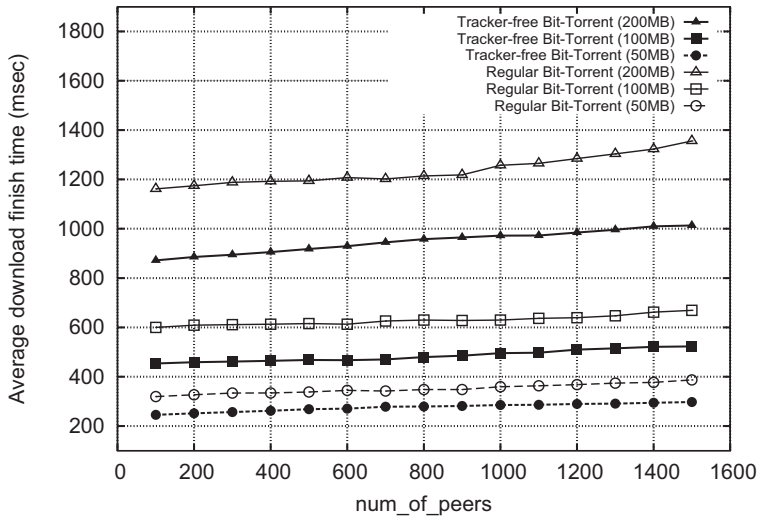


Fig. 6. Average download finish time of the swarm.

peers participated in the swarm:

$$Upload\ link\ utilization = \frac{num\_of\_peers \times file\_size \times 1024 \times 8}{Swarm\ end\ time \times C\_up \times num\_of\_peers}$$

$$Download\ link\ utilization = \frac{Upload\ link\ utilization}{C\_down\_fac}$$

Here, *swarm end time* is the time when all peers of the swarm finish downloading the file. We multiply the *file_size* by 1024 × 8 to convert the file size from MB to Kb, since *C_up* has been given in Kbps. While calculating the *upload link utilization*, we divide how much data has been actually transferred during the *swarm end time* by the aggregate bandwidths of the links. The *download link utilization* has been calculated by dividing the *upload link utilization* with the *C_down_fac* while all other parameters remain unchanged.

We present comparative simulation results for regular BitTorrent and tracker-free BitTorrent by varying the number of peers. The *average download finish time* shown in Fig. 6 demonstrates that the modified tracker-free BitTorrent model completes downloading quickly compared to the regular BitTorrent model. The figure also demonstrates that, predictably, the average download time increases with the growth of the file size. Moreover, for a specific file size, the average download time slightly increases more or less linearly with the number of peers, which reflects the increased latency before a transfer can be initiated when a copy becomes available on a peer whose link is not saturated.

We present the comparison of regular BitTorrent and tracker-free BitTorrent for *upload link utilization* and *download link utilization* in Figs. 7 and 8, respectively. These two figures illustrate that the bandwidth utilization is better in the tracker-free BitTorrent model compared to the regular

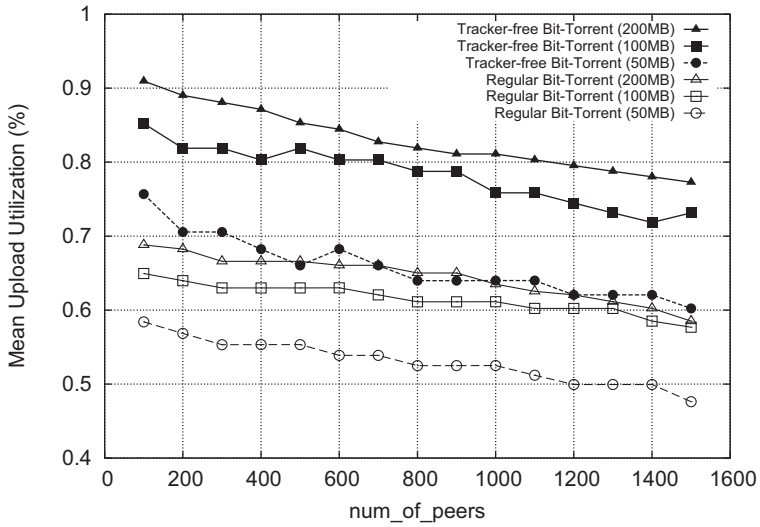


Fig. 7. Upload link utilization.

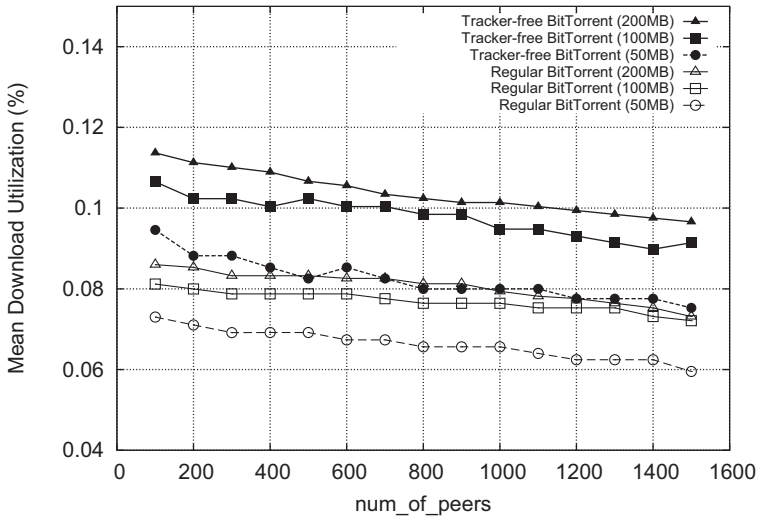


Fig. 8. Download link utilization.

BitTorrent model. Moreover, it can be observed that the mean upload/download utilization decreases with the increase of the number of peers, which is the dual of the behavior observed in Fig. 6. Observing the equation of the *upload link utilization*, it can be found that all parameters except the *swarm end time* remain unchanged. Therefore, the *upload link utilization* is inversely proportional to the *swarm end time*. Although we are not presenting the *swarm end time*, during our simulation we have recorded this parameter and observed that it increases with the number of peers. For example, for a 200 MB file size, in regular Bittorent client, for 100 and 1500 nodes, the *swarm end times* are 1190 and 1400 msec, respectively. Note that this same trend is observed in Fig. 6 where the *average download finish time* increases with the number of peers.

Thus, it is clearly shown that the removal of tracker communication from the client reduces the time to finish and enhances the bandwidth utilization in the swarm. Therefore, we can conclude that segregating the control plane and placing it in a surrogate server would increase overall performance of the P2P client from both the users' and swarm's perspective.

7. Lessons learned

We draw here some lessons from our work, on the benefits of the architecture, and the challenges of its implementation.

7.1. Benefits of the virtual P2P model

In the following we summarize some of the key benefits of the proposed model.

1. **No installation hassle**: To use a traditional BitTorrent client the user needs to install some software. A wide selection of client software is available now, with some good enough and quite simple but the user needs to understand the basic working principles of BitTorrent and have some skills in network configuration (routers, firewalls etc.) to make it work effectively. However, a virtual P2P client does not require any installation.
2. **Operability in restricted environments**: A P2P client program could not be installable on the user's terminal due to various reasons such as organizational policy, licensing issues or platform support. In such a restricted environment, the proposed model works without any difficulty through web-based clients.
3. **Location independence**: As cloud infrastructures are off-site (typically provided by a third-party) and accessed via the Internet, users can connect to the virtual P2P client from anywhere. Moreover, it will be possible to access P2P services from behind the firewall. Still, we need to further study in the future how virtual P2P clients could be accessed behind their firewall or the case of NAT traversal and its impact on end-to-end data transfer.
4. **Simpler maintenance**: P2P applications and protocols are still evolving. To implement the most efficient method of content sharing a P2P client application requires frequent updates. In the case of a virtual P2P client implementation, any update in the protocol or application can be easily maintained on the surrogate. The client does not have to update or modify anything locally.
5. **Network access mobility**: The virtual P2P client architecture decouples a swarm from the user terminal while the surrogate maintains swarm-related information. Hence, a terminal could be switched off without disrupting or restarting the swarm. A user could change her access network seamlessly, say due to the availability of a faster or cheaper access network.
6. **NAT traversal**: In traditional BitTorrent applications, two peers behind the NAT cannot directly communicate with each other. In our proposed model, the virtual P2P client is running in a public server and

thus can facilitate hole-punching [40] for NAT traversal between the peers. To do so it can first ask for the public and private endpoints of the peers and provide the user's end points to them. The peers can then negotiate with each other for hole-punching. As a result the user device can reach more peers, resulting in a higher download speed.

7.2. Implementation challenges

Implementing a BitTorrent application in a web-browser requires a few modifications in the BitTorrent protocol and the web browser standards as well. Traditional browsers have some limitations to deploy peer-to-peer applications. We discuss the possible modifications to web browser standard and BitTorrent protocol in the following.

Bidirectional connections in a browser. Existing BitTorrent applications use the BitTorrent protocol in the application layer and open a bidirectional TCP connection with each of the peers. However, a browser does not natively "speak" the BitTorrent protocol. The WebSocket protocol [41] allows two-way communications which can be initiated from the browser, with the server [27]. It can be used to tunnel BitTorrent messages, as well as the keep-alive messages.

Data exchanges between peers can further be facilitated with WebRTC, which is now available in mainstream browsers like Chrome, Firefox and Opera [42].

BitTorrent messages over HTTP/WebRTC. The BitTorrent protocol has its own defined headers and payloads. A browser-based implementation can only communicate using the HTTP/WebRTC protocol suite. However, a browser based client can be designed using the HTTP/WebRTC protocol where the payloads will be the BitTorrent messages (both state-oriented and data-oriented). These types of clients can only communicate with peers that are using similar browser-based clients. To communicate with both browser based and traditional BitTorrent clients, hybrid clients could be designed. The hybrid clients may have a module that would be able to parse HTTP/WebRTC requests from browser based clients with BitTorrent protocol oriented messages in the payload; along with the ability to communicate with traditional BitTorrent clients using the BitTorrent protocol. It will then be able to seed and leech to either type of implementations.

Downloading pieces from forwarded IP addresses. Because of the same-origin policy, the web browser enforces constraints on which requests can be initiated by the application and to which content originator. As a result, when the IP forwarder returns with the IP addresses of the peers to connect to, it cannot initiate a connection to the peer which has an origin outside of the domain of the initial remote server. However, implementation of peer-to-peer applications requires bidirectional communication between multiple peers. It can be solved using Cross-Origin Resource Sharing (CORS) [27], which provides a secure opt-in mechanism for client-side cross-origin requests.

8. Existing work

In the following, existing bodies of work related to our goal are summarized, as well as their limitations. It is to be noted that not a single one of them could fully implement our architecture.

8.1. Java plugin based implementations

BitLet [43] is a browser-based BitTorrent client that runs on any browser that supports the Java Plugin (version 1.5 or newer). Note that using a Java based plugin has its pitfalls. The Java Runtime Environment (JRE) has to be installed on the system. If a plugin requires a newer than, or a version of the JRE more specific than the one available on the system, the user may need for an update to complete, or be left unable to run the plugin. Most importantly, a significant portion of mobile devices e.g. PDAs (Blackberry, Palm), tablets (iPad, Android, Windows Surface RT), smart phones (iPhone, Android), gaming consoles (Nintendo Wii) do not support java plugins at all [44], thus this model has become obsolete.

8.2. Cloud-based, pre-downloaded P2P file retrieval system

Some on-demand cloud-based P2P direct downloader applications such as ZbigZ [45], put.io [46] and CloudTorrent [47] first complete the download of the whole file on their server. Then the latter provides the end user with a download link or keeps the file in the user's cloud storage from where the user can directly download the file in a client-server method. Such systems keep the user identity and activity for downloading the file anonymous to other peers as the user only downloads the pre-downloaded file from the cloud. However, these systems violate an important dimension of P2P systems through unnecessary redundancy and wastage of resource by downloading a single file twice and also storing it in the cloud.

Still, a platform such as ZbigZ is close to our concerns and its existence shows the benefits in our work. Unfortunately, they have not disclosed their architecture and we cannot directly compare our approach to theirs.

8.3. WebTorrent

WebTorrent [37], a work in progress project, is focused on developing a browser-based BitTorrent client. The objective of WebTorrent is to build a browser-based BitTorrent client which requires no installation (no plugin or extension) and has the ability to communicate with the traditional BitTorrent network. It is based on WebRTC data channels [42] for peer-to-peer transport. The web-based clients using the JS file supplied by WebTorrent can only download from other Web-based clients until mainstream BitTorrent clients support WebTorrent. Our proposed model separates the control plane and the data plane of the BitTorrent client and only the data plane related activities are delegated to the end host to reduce

complexity. On the other hand, in WebTorrent, all the activities of a BitTorrent client are mimicked in the user device using WebRTC and there is no notion of a virtual P2P client.

9. Conclusion

We have presented a scheme to virtualize P2P clients, using BitTorrent as a case study. The main point of this scheme is to reduce the load on user devices, but it also supports mobility and facilitates inter-networking. We use the ubiquitous browser to implement the user interface of the virtual client. It uses a combination of JavaScript programming combined with Web Services based communications between browser and server, as well as between peers.

In future work, we would like to explore the full potential of using WebRTC to implement this model. We also plan to better study the trade-off between execution of functions between the browser and the server, as well as investigate better ways to facilitate inter-operations with "traditional" P2P clients.

References

[1] L.M. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner, A break in the clouds: towards a cloud definition, ACM SIGCOMM Comput. Commun. Rev. 39 (1) (2008) 50–55.
[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al., A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58.
[3] S. Islam, J.-C. Grégoire, Giving users an edge: A flexible cloud model and its application for multimedia, Future Gener. Comput. Syst. 28 (6) (2012) 823–832.
[4] B. Pourebrahimi, K. Bertels, S. Vassiliadis, A survey of peer-to-peer networks, in: Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc, 2005, Citeseer, 2005.
[5] C. Muoh, A Tutorial on Gnutella, Bittorrent and Freenet Protocol, 2006. http://medianet.kent.edu/surveys/IAD06S-P2PArchitectures-chibuike/P2P%20App.%20Survey%20Paper.htm (accessed 01.02.16).
[6] T. Karagiannis, P. Rodriguez, K. Papagiannaki, Should internet service providers fear peer-assisted content distribution? in: Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement, 2005, p. 6.
[7] PPTV, 2015. URL http://www.pptv.com/ (accessed 01.02.16).
[8] Y. Huang, T.Z. Fu, D.-M. Chiu, J. Lui, C. Huang, Challenges, design and analysis of a large-scale p2p-vod system, in: Proceedings of ACM SIGCOMM Computer Communication Review, 38, 2008, pp. 375–388.
[9] Freenet, 2015. URL https://freenetproject.org/ (accessed 01.02.16).
[10] Skype, Free Calls to Friends and Family, 2015. last accessed: July 2015. URL http://www.skype.com/.
[11] D. Korzun, S. Balandin, A peer-to-peer model for virtualization and knowledge sharing in smart spaces, in: Proceedings of 8th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 2014, pp. 87–92.
[12] SETI@Home, 2015. URL http://setiathome.ssl.berkeley.edu/ (accessed 01.02.16).
[13] BitTorrent Sync, 2015. URL http://getsync.com/ (accessed 01.02.16).
[14] B. Ford, P. Srisuresh, D. Kegel, Peer-to-peer communication across network address translators, in: USENIX Annual Technical Conference, 2005, pp. 179–192.
[15] S. Haque, S. Islam, J.-C. Grégoire, Short paper: 'Virtual P2P client: accessing P2P applications using virtual terminals', in: Proceedings of 2015 18th International Conference on Intelligence in Next Generation Networks (ICIN), 2015, pp. 142–144.
[16] R.L. Xia, J.K. Muppala, A survey of BitTorrent performance, Commun. Surv. Tutorials, IEEE 12 (2) (2010) 140–158.
[17] B. Cohen, Incentives build robustness in BitTorrent, in: Proceedings of Workshop on Economics of Peer-to-Peer systems, 6, 2003, pp. 68–72.
[18] B. Cohen, The BitTorrent Protocol Specification, 2008,
[19] S.L. Blond, A. Legout, W. Dabbous, Pushing BitTorrent locality to the limit, Comput. Netw. 55 (3) (2011) 541–557.
[20] S. Nanda, T.c. Chiueh, A Survey on Virtualization Technologies, Technical report, 2005. Available: URL http://www.ecsl.cs.sunysb.edu/tr/TR179.pdf(accessed 01.02.16).
[21] L.M. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner, A break in the clouds: towards a cloud definition, SIGCOMM Comput. Commun. Rev. 39 (1) (2008) 50–55.
[22] N. Feamster, L. Gao, J. Rexford, How to lease the Internet in your spare time, ACM SIGCOMM Comput. Commun. Rev. 37 (1) (2007) 61–64.
[23] T. Richardson, Q. Stafford-Fraser, K.R. Wood, A. Hopper, Virtual network computing, IEEE Int. Comput. 2 (1) (1998) 33–38.
[24] Remote Desktop Protocol, 2015, Available: URL http://msdn.microsoft.com/en-us/library/aa383015(VS.85).aspx (accessed 01.02.16).
[25] Citrix System, 2015. Available: URL http://www.citrix.com/ (accessed 01.02.16).
[26] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, J. Int. Serv. Appl. 1 (1) (2010) 7–18.
[27] I. Grigorik, High Performance Browser Networking: What Every Web Developer Should Know about Networking and Browser Performance, O'Reilly Media, Incorporated, 2013.
[28] S. Islam, J.-C. Grégoire, Converged access of IMS and web services: a virtual client model, Netw., IEEE 27 (1) (2013) 37–44.
[29] S. Islam, J.-C. Grégoire, Beyond CDN: Content Processing at the Edge of the Cloud, John Wiley & Sons, Inc., 2014, pp. pp.243–258.
[30] Dropbox, 2015. URL https://www.dropbox.com/ (accessed 01.02.16).
[31] NodeJS, 2015. URL https://nodejs.org/ (accessed 01.02.16).
[32] ExpressJS, 2015. URL http://expressjs.com/ (accessed 01.02.16).
[33] socket.io, 2015. URL http://socket.io/ (accessed 01.02.16).
[34] Linode, 2016. URL https://www.linode.com/ (accessed 01.02.16).
[35] Amazon EC2, 2015. URL http://aws.amazon.com/ec2/ (accessed 01.02.16).
[36] Twitter Bootstrap, 2015. URL http://getbootstrap.com/ (accessed 01.02.16).
[37] Webtorrent, Streaming Torrent Client for Node and The Browser, 2015. URL https://github.com/feross/webtorrent/ (accessed 01.02.16).
[38] A. Bharambe, C. Herley, V. Padmanabhan, Analyzing and improving a bittorrent networks performance mechanisms, in: Proceedings of 25th IEEE International Conference on Computer Communications, INFOCOM 2006, 2006, pp. 1–12.
[39] K. Eger, T. Hoßfeld, A. Binzenhöfer, G. Kunzmann, Efficient simulation of large-scale P2P networks: packet-level vs. flow-level simulations, in: Proceedings of the Second Workshop on Use of P2P, GRID and Agents for the Development of Content Networks, ACM, 2007, pp. 9–16.
[40] D. Maier, O. Haase, J. Wasch, M. Waldvogel, NAT hole punching revisited, in: Proceedings of 36th IEEE Conference on Local Computer Networks (LCN), 2011, pp. 147–150.
[41] I. Fette, A. Melnikov, The Websocket Protocol, RFC 6455, 2011,
[42] WebRTC, 2015. URL http://www.webrtc.org/ (accessed 01.02.16).
[43] Bitlet, 2015. URL http://www.bitlet.org/ (accessed 01.02.16).
[44] Availability of Java Plugin in Mobile Devices, 2015. URL http://www.java.com/en/download/faq/java_mobile.xml (accessed 01.02.16).
[45] ZbigZ, 2015. URL http://www.zbigz.com/ (accessed 01.02.16).
[46] put.io, Hassle-Free Torrenting in The Cloud, 2015. URL https://put.io/ (accessed 01.02.16).
[47] I. Kelenyi, A. Ludanyi, J.K. Nurminen, BitTorrent on mobile phones-energy efficiency of a distributed proxy solution, in: Proceedings of IEEE Green Computing Conference, 2010, pp. 451–458.

Syed Arefinul Haque completed Bachelors in Business Administration from the Institute of Business Administration (IBA), University of Dhaka. He completed his Master's in Computer Science and Engineering from United International University (UIU) in 2015. His thesis was in peer-to-peer (P2P) networks. He is currently employed in a software company where he is establishing a liaison between business processes and intelligent software.

Salekul Islam is an Associate Professor and Head of the Computer Science and Engineering (CSE) Department of United International University (UIU), Bangladesh. Earlier, from 2008 to 2011, he worked as an FQRNT Postdoctoral Fellow at the Énergie, Matériaux et Télécommunications (EMT) center of Institut national de la recherche scientifique (INRS), Montréal. He completed his PhD in 2008 from Computer Science and Software Engineering Department of Concordia University, Canada. His present research interests are in Cloud computing, virtualization, future Internet and analysis of security protocols. He also carried out research in IP multicast especially in the area of security issues of multicasting.

Md. Jahidul Islam a PhD student at the department of CSE, University of Minnesota (UMN)-Twin Cities. He has completed undergraduate (B.Sc.) and postgraduate (M.Sc.) studies from Bangladesh University of Engineering and Technology (BUET) in Computer Science and Engineering (CSE), on April-2012 and February-2015, respectively. Then he worked (currently on leave) as an assistant professor of department of CSE, United International University (UIU) until Fall 2015. His fields of interest for research are Artificial Intelligence and Machine Learning.

Jean-Charles Grégoire is an associate professor at INRS, a constituent of the Université du Québec with a focus on research and education at the Masters and Ph.D. levels. His research interests cover all aspects of telecommunication systems engineering, including protocols, distributed systems, network design and performance analysis, and more recently, security. He also has made significant contributions in the area of formal methods.