PEER-ASSISTED SECURE PATCH DISTRIBUTION

Paskuwal Handi Manoj Piyumal De Silva

Registration No.: 138207H

Degree of Master Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March 2016

PEER-ASSISTED SECURE PATCH DISTRIBUTION

Paskuwal Handi Manoj Piyumal De Silva

Registration No.: 138207H

Dissertation submitted in partial fulfilment of the requirements for the Degree Master of Science in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

May 2016

Declaration

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

Candidate

....

Manoj Piyumal De Silva Date

The above candidate has carried out research for the Masters Dissertation under my supervision.

Supervisor

....

Dr. H.M.N. Dilum Bandara Date

ABSTRACT

Traditional software update and patch distribution mechanisms use a centralized approach where clients pull updates from central servers or a content distribution network. Peer-assisted patch distribution provides a scalable alternative to the centralized approach. It has great potential for providing fast patch delivery and while reducing the load on central servers. Due to speedy patch distribution, Peer-to-Peer (P2P) approach is effective in combating against fast spreading worms. However, during P2P patch sharing, peers are exposed to each other and the patch interest of individual peers is revealed. Therefore, P2P approach is not effective in combating against a topological worm as vulnerable hosts are exposed to other nodes in the P2P overlay. A topological worm spreads by attacking on known or connected hosts to the infected node. Therefore, a vulnerable peer will certainly be compromised, if it is connected with an infected peer. Furthermore, as the patch interest of individual peers is revealed peer privacy is also compromised.

We propose a BitTorrent based, peer-assisted patch distribution system that prevents the exposure of peers to each other. The proposed solution utilizes a set of non-vulnerable peers already available in the P2P network to mediate the connections between patch-sharing peers. These peers are called pseudo-downloaders as they enable patch downloading without exposing the existence or interests of patch-sharing peers. Pseudo-downloaders can be accommodated by introducing minor modifications to the BitTorrent protocol. Through simulations, we show that our solution can significantly reduce the infections due to address exposure while having a minimal effect on the patch download time. When the pseudo-downloader population exceeds 50% of the patch downloaders, our solution achieves the same download time as a typical BitTorrent network. Moreover, when the number of peers participating in the system is large, proposed solution download patches even faster than a typical BitTorrent network.

ACKNOWLEDGMENT

First and foremost, I would like to express my gratitude to my supervisor, Dr. Dilum Bandara for his guidance, inspiring ideas, precious time in reviewing my work and always expecting high standards. Secondly, I would like to express my gratitude to Dr. Narasimha Shashidhar, Assistant Professor in the Department of Computer Science at Sam Houston State University, for his feedback. I am thankful to our MSc research project coordinator, Dr. Malaka Walpola and all the lecturers at the Department of Computer Science and Engineering, University of Moratuwa for their guidance and valuable advice. Finally, I am thankful to my parents and my family for their continuous support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABREVIATIONS

CDN	Content Delivery Network
CDS	Connected Dominating Set
DDoS	Distributed Denial of Service
HTTP	Hyper Text Transfer Protocol
IP	Internet Protocol
ISP	Internet Service Provider
NAT	Network Address Translation
P2P	Peer-to-Peer
PAPD	Peer-Assisted Patch Distribution
RTT	Round Trip Time
TCP	Transmission Control Protocol
TTL	time-to-live
UDP	User Datagram Protocol

Chapter 1

Introduction

1.1 Software Patch Distribution

A computer worm is a malicious program, which replicates itself and spreads to other computers without any human intervention [1]. Typically, worms use a known vulnerability of the operating system or the installed software of victim hosts to propagate through the network. After compromising a host, worms typically look for the next target to propagate by scanning IP addresses. Recent worm attacks indicate that thousands of computers on Internet could be compromised within minutes. Few of these well-known worms are CodeRed, Slammer and Blaster [2].

Software vendors distribute software patches and updates to fix vulnerabilities in their software. Typically, software vendors use a set of centralised servers to serve these software patches to client computers. Alternatively, there could be Content Distribution Network (CDN) based approach which is centralised in nature as well. However, such a set of servers has a finite capacity and can only serve a limited set of clients at any given time. Moreover, some clients might have to wait until their turn arrives. When a large number of clients need to be patched it takes long time to patch all the vulnerable hosts with a fixed number of patch servers [2]. Nevertheless, it is essential to distribute these software patches to hosts as quickly as possible, because of the potential for a worm to spread rapidly. Furthermore, centralized approach is costly for software vendors as they need to keep scaling their servers and bandwidth as their user base grows.

1.2 Peer-Assisted Software Patch Distribution

A Peer-to-Peer (P2P) network is a distributed system, in which nodes in the network exchange data without any centralized control. There are no dedicated servers in a P2P network and all nodes act as both clients and servers. P2P network creates a self-organizing virtual topology on top of the Internet, which is referred to as the *P2P overlay network* or P2P logical network.

To address the limitations of centralized patch distribution, research community has considered the possibility of patching one node by obtaining the security patch from previously patched nodes, rather than obtaining the security patch directly from central servers for each node [3], [4], [5], [6]. This kind of patch distribution processes are classified as *peer-assisted patch distribution* (PAPD) systems because peers assist

each other to download the patch instead of solely depend on the central servers. As only a small subset of the nodes initially obtains the patch from the central servers, peer assisted solution is much more scalable and time efficient than the centralized approach. Furthermore, Gkantsidis et al. [6] and Shakkottai and Srikant [2] demonstrated that P2P approach has a great potential for providing a fast, scalable, and effective patch delivery compared to the centralized server based solution.

1.3 Research Problem

Although peer-assisted patch distribution appears to be a better candidate than the centralized approach, there is an address exposure problem in peer-assisted patch distribution. The address exposure problem can be described as follows. Consider two peers *A* and *B*. When Peer *A* requests the patch from Peer *B*, *B* gets to know the existence and the address of *A*. Note that the patch requesting peer is always vulnerable as it does not have the patch. Thus, when sharing the patch in a P2P style, vulnerable peers get exposed to potential attackers. If the worm in question is capable of attacking on the origin of incoming connections and *B* is already infected, then *A* will become a victim of the worm as well. Such a worm that uses local communication topology to find new victims is known as a *topological worm* [1]. With a topological worm in effect, peer-assisted patch distribution mechanism will facilitate the propagation of the worm outweighing the propagation prevention intentions. Wu et al. [7] showed that a large majority of the vulnerable hosts will become compromised with such an attack targeting a peer-assisted patch distribution system.

Another problem arising from the address exposure is revealing the content interests of patch sharing peers or inability to preserve the privacy of patch sharing peers. For instance, a peer sharing a patch targeted for particular software implies the fact that the peer is most probably installed with that particular software. Therefore, in a peer-assisted patch distribution system, an attacker can discover which software is installed on which nodes. This is known as the *privacy problem* in peer-assisted patch distribution. This information can be used for a later attack when some vulnerability of some software is disclosed.

Therefore, the problem addressed in this research is that:

How to develop a peer-assisted software patch distribution system to share the patch or update in P2P style without being exposed to the potential attackers?

One might think that, one obvious solution to prevent the address exposure is to use an anonymizing network such as TOR [8]. TOR sends traffic through a series of volunteered relays to hide the origin of the communication and data is encrypted such that no one can determine both origin and content of the communication. Such a solution has already been explored by Wu et al. [7]. Nevertheless, requesting patch through an anonymizing network does not completely solve the privacy problem. This is because, in such a scenario only the privacy of patch requesting peer is preserved and the patch provider's privacy is not preserved as the patch requester is well aware that the patch provider is sharing the patch. Patch provider's privacy is also important to protect against attackers who try to discover patch interest of peers. In addition, these networks send data through a series of relays and in turn it introduces delays to the patch distribution process. Therefore, peer-assisted patch distribution over TOR is not the most optimum solution to prevent exposure of peers to each other.

Several solutions have been suggested to hide the content interests of users or to preserve privacy in BitTorrent networks [9], [10]. OneSwarm [11] proposes a novel P2P protocol which preserves the privacy of its users. Peers in most of these solutions use cover traffic (i.e., downloading uninterested files), to obscure their content interests. A topological worm would attack on any known node regardless of its content interests; hence, preserving privacy of peers is insufficient. In other words, vulnerable peers will become targets even if they managed to conceal their content interests once they interacted with an infected peer.

Moreover, prior work that requests patches through an anonymous network or preserving privacy while sharing the patches, do not propose a self-contained, peer-assisted patch distribution system that prevents the exposure of patch sharing peers to potential attackers.

1.4 Objectives

The high-level objective of this research is to design a self-contained, peer-assisted software patch and software update distribution system which does not expose patch sharing peers to each other. The specific objectives of the research are:

1. How to prevent the address exposure of vulnerable peers to defend against attacks similar to topological worms?
2. How to preserve the privacy of patch sharing peers or in other words, how to hide content interests of patch sharing peers?
3. How to design a peer-assisted patch distribution system to provide an infrastructure to distribute software patches and updates from multiple software vendors or multiple sources by utilizing a single P2P system, preferably an existing public P2P network.

1.5 Contributions

We propose a peer-assisted patch distribution system which is based on the most popular P2P file sharing protocol, BitTorrent [12]. We demonstrate the utility of the proposed solution using an OMNet++ [13] based simulator.

Thus main contributions of this research are as follows:

1. Design of a peer-assisted patch distribution system that conceals identities of patch sharing peers from possible attackers. Mainly these attackers are topological worms and attackers who try to find out installed software on patch sharing nodes. Furthermore, the proposed system is capable of distributing patches from multiple software vendors by utilizing the same P2P network, which may also be used to share other contents such as songs and movies.
2. Validation of the effectiveness of the proposed system through OMNet++ [13] simulation framework based BitTorrent simulation module [14]. Simulation results show that our solution can completely eliminate infections due to the address exposure problem. Furthermore, we show that the patch download time would not get affected due to the intermediate hop introduced by the pseudo-downloaders, if sufficient number of pseudo-downloaders are available in the P2P network.

In addition to those contributions this research made significant improvements to the OMNet++ based BitTorrent simulation module which was initially developed by Katsaros et al. [14]. These improvements include porting BitTorrent simulation module from OMNet++ version 3 to OMNet++ version 4, and improving extensibility of BitTorrent simulation module. The improved BitTorrent OMNet++ module is publically available at Github [15].

1.6 Outline

The rest of the thesis is organised as follows. Chapter 2 presents the background of the research and related work. It discusses about computer worms, P2P systems, BitTorrent P2P protocol, and related work on peer-assisted patch distribution. Chapter 3 presents the details of the proposed patch distribution solution. Simulation setup and performance evaluation are presented in Chapter 4. Chapter 5 summarizes the findings and suggest future works.

Chapter 2

Literature Review

This chapter provides the background related to the research. The chapter starts by presenting about computer worms (Section 2.1), P2P networks and BitTorrent protocol (Section 2.2) and solutions that have been proposed to preserve the privacy in P2P networks (Section 2.3). It also discusses the limitations of the traditional centralized patch distribution approach and how the peer-assisted patch distribution provides solutions to those limitations. Several existing peer-assisted patch distribution solutions are presented in Section 2.4. Finally the in Section 2.6 chapter concludes by presenting the OMNet++ BitTorrent module [14] which is used as the simulation platform in this research.

2.1 Computer Worms

A computer worm is a malicious program that self-propagates across a network by exploiting security flaws in widely used services [1]. Worms gained wide attention after the Morris worm was released in 1988, which is considered as one of the first identified computer worms [1]. Computer worms are different from computer viruses because, computer viruses infect non-mobile files and they require user actions to propagate [1], whereas computer worms do not require such user intervention for propagation. Worms spread faster than viruses because human intervention is not required for the worm propagation. Most worms attack by exploiting vulnerabilities once the vulnerability of a service has been disclosed [2], [16]. In addition, some worms are reverse engineered from patches that are released to fix vulnerabilities [17]. Therefore, in most cases a software patch is available to fix the vulnerability to prevent the propagation of a potential worm. Nevertheless, time between the patch release and the worm appearance is decreasing, and some worms appear even before the patches are released, e.g., ANI worm [16]. Furthermore, worms that exploit unidentified vulnerabilities are known as *zero-day worms* [2]. While such a worm could cause significant damages before a suitable patch is released, such worms are typically rare. Typically a worm does not destroy the host during the propagation phase because it may disrupt worm's propagation [1].

A worm can be used to capture sensitive information of users of compromised hosts or can be used for other attacks such as Distributed Denial of Service (DDoS). Worms have already demonstrated that they can be quite destructive. CodeRed infected more

than 35,900 hosts within 14 hours in 2001 and caused a damage of about a $2.6 billion [18]. Slammer infected more than 90% of the vulnerable hosts in less than ten minutes [2], [18]. Measurement studies indicate that the number of infected hosts by a worm usually follows sigmoid curve as shown in Figure 2.1. It can be seen that the number of infected hosts due to CodeRed (v2) worm increased in an exponential manner. Shakkottai and Srikant [2] showed this fact analytically.

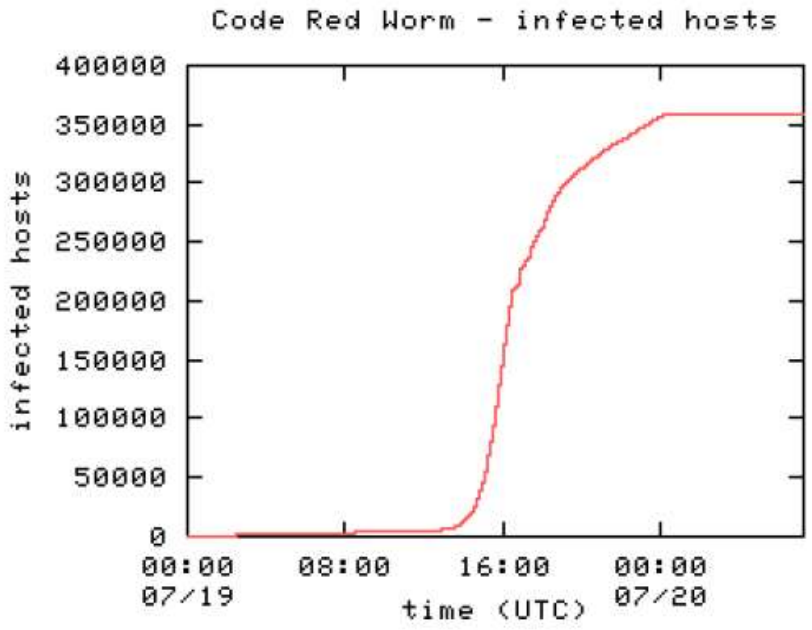

Figure 2.1: Propagation of CodeRed (v2) worm [2].

To locate new targets most worms use IP address scanning or probing. However, these IP address scanning can be easily detected from intrusion detection systems because IP address scanning traffic is very different from normal traffic. Some other worms use target lists for propagation. This kind of a list is called a *hit list* and it contains IP addresses of hosts, to which worm is intended to propagate. Some applications contain connectivity information about other hosts. This information can be used to locate new targets by a worm. These kinds of worms, where worm uses the local information in the infected host to locate new targets are known as *topological worms* [1]. For example, the original Morris worm used IP addresses in the */etc/hosts* file to spread to other hosts. Topological worms can spread very fast, because they do not need to waste time on IP address scanning. In addition, a topological worm would not create any abnormal network traffic, because it connects to already known computers to the current infected hosts [18].

Worms that are considered in this research are topological worms. In this research it is assumed that, the worm is smart enough to obtain the IP address of the remote end of established network connections and attack on those IP addresses.

2.2 P2P Overlay Networks

A P2P overlay network is an application layer, or virtual network, which is constructed on top of the Internet Protocol (IP) networks. Logical topology of the P2P overlay network does not necessarily represent the underlying physical topology of the physical network. A host or node participating in the overlay network is referred to as a peer.

P2P systems are distributed systems and they are different from traditional client-server architecture because every peer can act as both a server and a client. Therefore, in most circumstances, role of every peer in a P2P network is symmetric. Because of this symmetry in roles, P2P systems can offer services beyond the client-server systems. These services include building self-organizing overlay network, resource sharing and searching between peers, redundant storage, load balancing, selection of nearby peers, robust routing architecture, trust between peers, and massive scalability [19]. While the most popular practical application of P2P computing is being the file sharing, there are many other application areas such as application-level multicast, distributed file systems, and web caches.

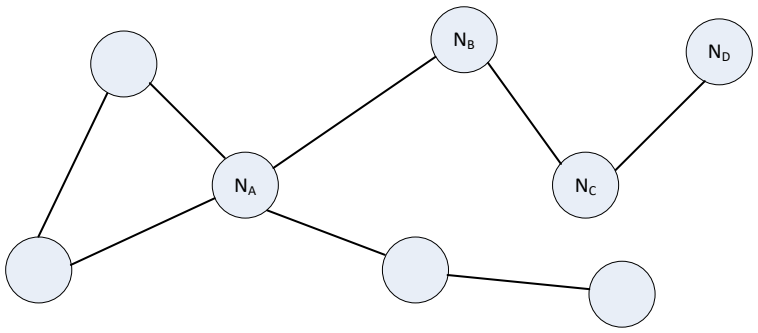

Figure 2.2: Topology of a P2P network.

Figure 2.2 shows connections between peers in a P2P network. We use this diagram to explain some common terms in P2P networks. In Figure 2.2 circles are the peers or nodes and lines between circles are open connection between peers. Two peers who

maintain an open connection is referred as *neighbours* [20]. In Figure 2.2 N_A and N_B are neighbours. An open connection between two peers is known as an *edge* [20]. A graph can be drawn, taking these open connections as edges and peers as vertices. This graph is known as the topology of the P2P network. Figure 2.2 shows topology of an unstructured P2P network. The number of neighbours of a peer is called as its out-degree [20]. The out-degree of peer N_A is four. If a message needs to be sent from one peer to another, who are not neighbours, message has to be sent over multiple edges. The length of the path or the number of edges passed by the message is known as the *hop count* between two peers those who have exchanged the message [20]. Therefore, hop count between peer N_A and N_D is three.

To find a particular data item in the P2P network, different P2P overlay networks use different schemes. Typically, a peer sends a lookup message to other peers to find out a particular data item in the P2P network. This kind of a message is known as a *query*. When a peer receives a query from another peer it needs to decide what to do with such a query message. This decision making process is known as *query processing* in P2P networks. Typically a peer chooses to respond to the query message, if it possesses the particular data item, and it forwards the query to one or more peers it is does not possesses the particular data item.

There are two main classes of P2P overlay networks, which are called as structured P2P networks and unstructured P2P networks. The main difference between these two types is how the P2P logical topology is created and maintained.

2.2.1 Structured P2P Networks

In structured P2P networks, P2P overlay topology is tightly controlled and content shared in the P2P network is placed at deterministic locations, which makes query processing for data items more efficient and guaranteed [19]. In structured P2P networks, each node has a unique ID. These structured P2P networks are built on the concept of Distributed Hash Table (DHT), where each data object has a unique ID (chosen from the same space of node IDs) and each of these data objects are assigned to a particular node based on the ID. Hence, every peer in the network knows where (at which peer) to store a particular data object and where to look exactly for a particular data object. For an instance, Figure 2.3 shows the topology of Chord [21],

which is a structured P2P network. In Chord each peer has a node ID obtained by hashing the IP address of the node and each data object has a key, obtained by hashing the identifier of the data object. Both node ID and key of a data object have *m* bits which makes node ID and object keys in the same space. Figure 2.3 shows an instance of Chord where $m = 6$. Therefore, both node IDs and object keys are in the range of 0 - 63 in this particular instance. Nodes are ordered in a circle ordered by node ID. Key *k* is assigned to the first node whose ID is equal to the key or follows key. For example, key 38 is assigned to node 38 and key 30 is assigned to node 32. Widely known examples of structured P2P networks are Content Addressable Network (CAN), Pastry, Tapestry, Chord, Kademlia, and Viceroy [19].

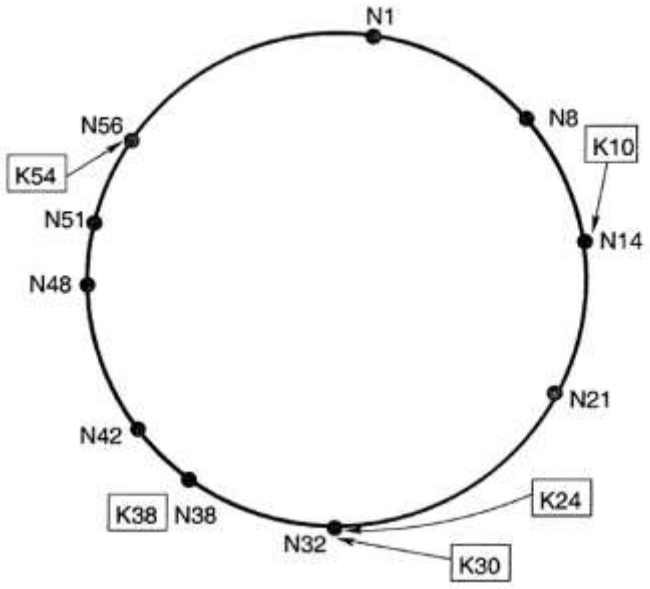

Figure 2.3: Topology of Chord P2P network [21].

2.2.2 Unstructured P2P Networks

Unstructured P2P networks are the most common P2P systems in the today's Internet. Best known example of unstructured P2P networks is P2P file sharing applications such as Gnutella and BitTorrent [19]. Unstructured P2P networks are composed of peers without any constraints on the topology and a peer can join at any location of the P2P topology. As the topology of the network is unknown, queries in unstructured P2P networks are resolved using flooding of messages though the network. When one peer is looking for some data item, it forwards the query to all its neighbours and every peer who receives this request, checks whether it has the particular data item and if it does, it replies to the query. If it does not have the particular data item, it

forwards the query to all of its neighbours (in some cases query may be forwarded to neighbours, even if the result is locally available). In order to prevent the network overwhelming from this kind of queries, there is a Time-To-Live (TTL) value for each query. The TTL is the number of overlay hops query should be discarded. So TTL is reduced by one at each peer and once TTL reaches zero, peers drop the query instead of forwarding it. For example, Figure 2.4 shows a scenario of query processing in an unstructured P2P network. Node N_q generates the search query and sends it to all the neighbours. This query is flooded in the network until hop count reaches two as TTL is set to two. When TTL reaches zero (or in other words hop count reaches two) the query is discarded. As N_r has the particular data item, it responds to the query by sending a message back to N_q.

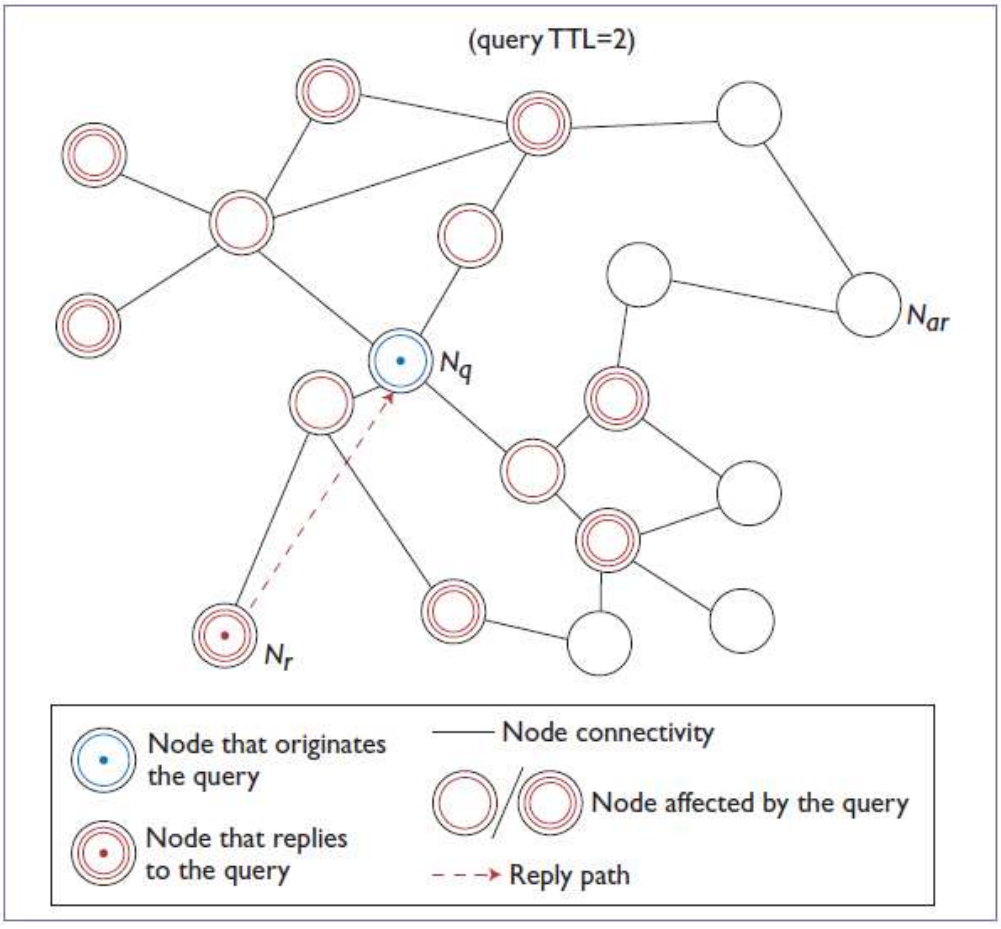

Figure 2.4: TTL-based query processing in unstructured P2P network for *TTL*=2 [22].

Even though this kind of flooding based technique is effective for locating replicated data items while peers join and leave the network frequently, it is not effective for locating rare data items. In such a case, a query may end up with no results, even

though the data item exists in the P2P network, because the TTL constraint prevents flooding the query to all nodes in the network. For example, in Figure 2.4, the query generated by N_q does not reach the node N_{ar} because of the TTL limit. If only the node N_{ar} had the particular data item searched by node N_q, the query will end up with no results, even though the network has the particular data item.

Unstructured P2P networks described above as known as pure P2P systems, which are not scalable for several reasons. First, the query-processing load on the each peer increases with the number of queries and the number of peers in the network. Secondly, load on the network increase rapidly with the number of queries because of flooding. Third, peers with limited capabilities can cause bottlenecks in pure unstructured P2P networks when processing queries [20]. Finally, search mechanism is not efficient because of the flooding of queries and TTL limitation [20]. Because of the TTL limitation, queries for rare items may always end up with no results. For example, the Gnutella network experienced degraded performance, such as slower response time when the size of the network increased in August 2000 [20]. In those days, Gnutella was a pure unstructured P2P network without super-peers.

2.2.2.1 BitTorrent

BitTorrent [12] is a P2P file sharing protocol. BitTorrent is an unstructured P2P network. However, BitTorrent does not depend on flooding to locate peers who share a particular content. Instead, BitTorrent uses a central server known as a *tracker* which facilitates the discovery of peers who holds a particular content. A single entity shared in BitTorrent is known as a *torrent*. A torrent could be a single file or collection of files. Every peer who shares a torrent registers with the tracker and tracker maintains a list of peers associated with each torrent.

The original file is logically sub-divided into smaller data units called *pieces*. Piece size is selected based on the size of the file and generally it is 512KB or less [23]. Pieces are sub-divided into smaller data units called *blocks*, typically 16KB in size. Blocks are the units that are transmitted between peers. Each file or a collection of files that is shared in the P2P network is described by a file called a *meta-info file* or *torrent file*. Torrent file contains metadata about the original file. Any peer who wants to download the file first needs to obtain the torrent file from an out of band method

such as a web site. Among other things main attributes of the torrent file are info dictionary and the URL of the tracker server. Info dictionary contains the file name, file length, md5sum of file, piece length and SHA-1 hashes of all pieces. The 20-byte SHA-1 hash of the info dictionary is known as the info-hash. Therefore, info-hash is unique to a single entity shared in the BitTorrent network and in BitTorrent protocol, torrents are uniquely identified by their info-hash.

The complete message transfer sequence between a peer and the tracker, as well as between peers is showed in Figure 2.5. Once a peer obtained the torrent file, it registers with the tracker for the particular torrent by sending the *tracker-request* message. Then the tracker adds the peer to the set of peers who share the requested torrent. Set of peers who share a particular torrent known as a *swarm*. On the reception of the tracker-request from a peer, tracker replies to the peer with the *tracker-response* message. Among several other things, tracker-response message contains a *peer list* which contains a sub-set of peers from the swarm.

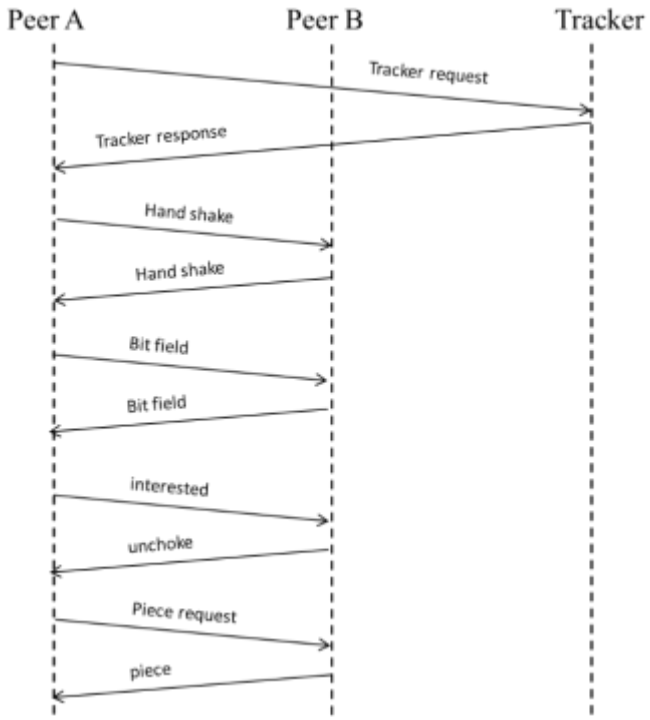

Figure 2.5: Message exchange sequence in BitTorrent.

Peers who have the complete file are known as *seeders*. Peers who do not have the complete file are known as *leechers*. Upon the reception of the tracker-response the peer uses information in the peer list to contact other peers. After connection establishment peers exchange handshake messages. *Handshake message* is the first

message in the communication between peers and connection initiator should send the handshake message first. Handshake message contains protocol identifier, info-hash and peer-id. Therefore, upon reception of the handshake message, recipient of the connection can safely identify the details about the torrent using the info-hash. If the recipient of the connection does not serve the specified torrent, recipient will drop the connection. After exchanging handshake messages peers exchange bit-field messages. *Bit-field* message indicates which pieces are available at the sender of the message. The bit field message may only be sent immediately after the handshaking sequence is complete, and before any other messages are sent. The bit field message is an optional message.

After exchanging bit fields, each peer knows which pieces the other peer can offer, and proceeds to request specific blocks of the file. A peer sends an interested message to notify the other peer that it would like to download pieces. The other peer responds with the *unchoke* message only if it is willing to share the pieces with sender of the interested message. Upon receiving the unchoke message, the interested peer asks for specific blocks of the file by sending *piece-request* messages. Then the other peer responds with the piece message which contains the requested block.

2.3 Preserving Privacy in P2P Networks

Peers are exposed to each other when sharing a file in P2P manner and a peer is able to know other peers who are interested in the file. Furthermore, in BitTorrent anyone can obtain a list of peers who share a particular file by just querying the tracker. Thus in a typical P2P network, one cannot download a file while hiding his or her interest about downloading the file. Anonymization networks, such as TOR [8] or Freenet [24] can offer complete anonymity in the cost of hindered performance. As P2P networks do not require such a complete anonymity and performance penalty is significant in anonymous networks, many solutions have been proposed to preserve privacy in P2P network without significantly degrading the performance.

BitBlender [9] is an anonymous network protocol which is specifically tailored for BitTorrent. BitBlender does not provide complete anonymity but satisfies requirements to achieve plausible deniability. Plausible deniability in a BitTorrent network ensures that every peer listed in the tracker or participating in the swarm is

not actively downloading the file. Thus, an adversary cannot discover which peers actually share the file. BitBlender achieves plausible deniability by utilizing a set of peers which are called *relay peers* who forwards messages on behalf of other peers who are actively participating in the swarm. In order to attract relay peers for an anonymous torrent, the tracker contacts the blender server and requests that some number of relay peers to join the torrent. The *blender* is a directory server in which relay peers are listed. The number of relay peers is a percentage of *true peers* participating in the swarm and this percentage can be set per torrent. Upon reception of such a message from a tracker, the blender server probabilistically asks relay peers to join the particular swarm such that requested number of relay peers will join the swarm. As the blender server asks relay peers probabilistically to join the swarm, blender server itself is unaware about which peer joined the swarm. As a result, a peer participating in the swarm might establish connections with other peers in the swarm and relay peers as well. Relay peers establishes connections with other peers in same manner by querying the tracker. When a relay peer receives a piece request, it forwards the request to another peer that might be a relay peer or a peer who is actively sharing the file. Ultimately, the piece request will receive by a peer who actively shares the file and that peer will responds with the piece and the response message will be routed along the same path as the request. Therefore, when a peer receives a piece request from another peer, receiving peer cannot determine exactly whether the request arrived from a peer who is actively engaged in the file sharing or from a relay peer. Furthermore, an adversary cannot get a list of peers by querying the tracker as peer list from the tracker contains relay peers as well. Authors show that expected path length of a piece request is $1/(1-r)$ where r is the fraction of relay peers to total peers participating in the swarm. Note that as r increases degree of anonymity is increased while degrading the performance. Authors claim that BitBlender does not require any protocol modification to BitTorrent. Using a modified BitTorrent client and 20 peers deployed in PlanetLab [25] authors show that when relay peer to true peer ratio is 1.0, download time only increase by 31% with respect to the download time without relay peers. Nevertheless, authors have not specified clearly how relay peers are listed in the Blender server and how a particular peer is encouraged to participate in a swarm as a relay peer.

OneSwarm [11] is a novel P2P protocol which is designed to preserve the privacy of its users. OneSwarm provides much better privacy than BitTorrent while providing much better performance than P2P data sharing over an anonymous network. OneSwarm provides flexible privacy where each individual user can control the trade-off between performance and privacy per file by managing trust placed on other peers. OneSwarm offers three sharing scenarios which are public distribution, sharing with permissions and sharing without attribution. In public distribution, data that need not to be private is shared and OneSwarm works similar to BitTorrent in this mode. Therefore, OneSwarm client is fully backward compatible with BitTorrent protocol to share data publically. When sharing with permissions, a user can specify with which other users, he or she is willing to share a specific file. Identifying other users in the network is possible as OneSwarm uses persistent identities to identify its peers. Sharing without attribution is used to share sensitive data by obscuring the source and the destination of the data transfer. OneSwarm peer limits direct communication to a small set of persistent contacts and these contacts provide connectivity to the rest of the overlay indirectly. These persistent contacts define the overlay links in OneSwarm P2P overlay. Two peers initiate such a link by exchanging their public keys. These public keys serve as identities of OneSwarm peers and these identities are persistent in order to manage trust relationships between peers.

To connect to a particular peer, its public key should be known. Therefore, OneSwarm automates these key exchanges by three ways; that are, 1) automatic peer discovery and key exchange in a Local Area Network (LAN), 2) using existing social network relationships, and 3) email invitations. These persistent peer identities are mapped to IP addresses using a DHT based on Kademlia [26]. Although peer connectivity information is published publically in a DHT, contact information of a particular peer can only be decrypted by peers who have initiated a persistent contact with that particular peer. This is because when a peer inserts its contact information in to the DHT, it inserts connectivity information multiple times for each of its persistent contacts by encrypting data by their public keys individually. Each of these entries is indexed in the DHT using a 20-byte shared secret which was generated at the initiation of the persistent contact between the two peers. Instead of using a tracker as in BitTorrent, OneSwarm locate data by flooding search messages in the overlay. Search messages do not contain a time-to-live field to hide the source of the message.

A peer forwards a search message only if it has not forwarded the message previously. Each peer adds 150ms delay to a search message. When the source peer found enough results, it generates a search cancel message and which will be forwarded in the same path as the search message without any delay. Therefore, the search cancel message reaches the search frontier quickly and cancels the search effectively.

In OneSwarm the path taken by the search message determine the path of the data transfer. Therefore, it is important not to forward search messages along the overloaded links. OneSwarm achieves this by adding additional delays to search message at nodes where load is high which effectively route the search message around the overloaded links and peers. If a peer can fulfil the search, it stops forwarding the message and generate a search reply by introducing a random delay to hide the source of the search reply message. When the search reply message reaches the search initiator, search reply message contains a path identifier. The path identifier is a unique identifier to identify the circuit between the data source and data receiver. Every hop in the search reply path has contributed to generate the path identifier and each hop in the path knows where to forward the data given the path identifier. OneSwarm uses the wire-level protocol from BitTorrent and search initiator treats each overlay path identified by one path identifier as a virtual BitTorrent peer. A peer in OneSwarm can improve its rating by contributing to the system whenever possible and during periods of contention peers give priority of service to peers that have higher rating.

OneSwarm clients have been implemented for Windows, Mac OS X and Linux. OneSwarm is publically available and according to authors OneSwarm has been downloaded by hundreds of thousands of users. Authors validate OneSwarm by instrumented clients run on PlanetLab [25] and simulations. For simulations overlay is built based on social relationships from a web site called last.fm. Authors show that OneSwarm performs well against adversaries who attempt to infer the data source by measuring the response time of a search. Furthermore, authors discuss about how various attacks are challenging on OneSwarm due to its protocol design. Authors show that when 50% of users use TOR to tunnel their BitTorrent traffic the median download time increases by a factor of 1.9 relative to OneSwarm. Authors claim that OneSwarm's average download time is competitive with BitTorrent as well; nevertheless exact numbers are not specified. We note that even though OneSwarm

offers good privacy benefits to its users, OneSwarm users have to spend significant amount of their computational resources to be a peer in the OneSwarm Overlay. For an instance to facilitate data transfer between two peers several peers have to invest their processing power and network bandwidth as the data transfer is indirect between peers. Moreover, we note that flooding-based search might fail for rare data objects and download speed for rare data objects will be significantly low due to longer path lengths. Moreover, users do not benefit from network effect, as they have to use a completely new P2P solution which does not have a relationship or connectivity to a network/community like BitTorrent.

Petrocco et al. [10] presented a P2Pfile sharing system that hides content interests of its users. Peers of this system use cover traffic to hide their content interests. Similar to BitBlender [9], this system only achieves plausible deniability and the solution is based on BitTorrent. A peer creates several deception downloads to obscure its real interest from rest of the swarm and these deception downloads are indistinguishable from actual downloads. As peers create several deception downloads, it results in more collaboration in the swarm, which in turn increases the overall performance of the swarm. Peers who participate in a swarm to generate cover traffic known as helpers. Seeders, leechers and helpers cannot be differentiated from each other by examining their traffic externally. In contrast to BitTorrent, peers in this solution do not advertise their complete set of available pieces to other peers in order to hide seeders. However, a peer can request unadvertised pieces from another peer. When a peer receives a request for such un-advertised piece, and the piece is locally unavailable it relays the request to another peer in the swarm. Even the requested piece is locally available, but the piece is unadvertised, the receiving peer of the request will respond to the request after introducing some random delay to make it appear as the piece has been obtained from relaying. However, we note that introducing additional delays to locally available pieces might results in higher overall download times. While relaying, peers always send the piece request to an advertised peer to avoid relay loops. All these relayed requests look similar to original requests. Seeders also relay some fraction of piece requests to hide the identification of seeders. When a peer participates in a swarm as helper, it keeps the pieces it has downloaded in the memory, which is known as caching of pieces. Cache size of a peer is limited and when a helper removes a piece from cache it removes the least recently used

piece. Furthermore, cache of a helper is filled with rarest pieces in the perspective of the helper peer to improve the piece availability in the swarm. Authors analytically show that their system performs better than OneSwarm [11] in face of a collusion attack. In a collusion attack an adversary controls multiple nodes in the swarm that work in collaboration to reveal content interest of a peer by connecting with the particular peer. Authors evaluate performance of their system using experiments. Although authors claim that their system is efficient in terms of overall average download time, it is difficult to compare the presented results with BitTorrent.

2.4 Peer-assisted patch Distribution

2.4.1 P2P Networks for Patch Distribution

In order to address limitations of centralized patch distribution, P2P-based patch distribution has gained the interest of both the industry and research community. This is also known as peer-assisted patch distribution. In peer-assisted patch distribution, a node that requires the patch can request the patch from other nodes that have already downloaded the patch. Thus, only a subset of vulnerable nodes will obtain the patch from central servers and nodes that have already downloaded the patch, will share the patch with other nodes in P2P style. As only a subset of nodes obtains the patch from the central servers, load on the central servers reduce significantly.

Gkantsidis et al. [6] explored the possibility of distribution of software patches in P2P style. They characterized "Windows Update", one of the largest update services in the world. Their goal was to come up with a set of guidelines on how to design and architect a fast and effective large-scale patch dissemination mechanism. They considered two standard content distribution architectures, caching at the ISP level and peer-to-peer, and evaluated their applicability to patch dissemination. They demonstrated that the P2P approach has a great potential for providing fast and effective patch delivery. Although they did not propose any specific P2P architecture or mechanisms to distribute patches, using empirical observations and analytical results they showed that P2P patching is highly effective in reducing the load on the central servers. Alternatively, they also showed that P2P can generate significant inter ISP traffic (backbone traffic), if the patch distribution algorithms do not give preference for downloading the patch from peers which are located in same ISP.

Shakkottai and Srikant [2] analytically showed that with fixed number of patch servers and with a well-designed worm, maximum number of infections is $\theta(N)$, where N is the number of hosts in the vulnerable population. That is, almost all the hosts in the population get infected with a fixed number of the patch servers and a well-designed worm. Authors also showed that, the worm only takes $\theta(\ln N)$ time to infect a significant fraction of the hosts in the vulnerable population. Provided that hosts can be recovered by providing the patch even after the worm infection, authors showed that it takes $\theta(N)$ time to disinfect the system with a fixed number of patch servers. This is because number of sources for the worm propagation increases at every time unit by worm replication, while the number of patch sources/servers remains fixed at each time unit. Therefore, a fixed number of patch servers cannot outrun a well-designed worm. Shakkottai and Srikant [2] also showed that a P2P patch distribution approach can defend against worms effectively. They analytically showed that with a P2P patch distribution approach, the maximum number of infected hosts is $\theta(N^{1/\gamma})$ and the time taken to disinfect the system is $\theta(\ln N)$. Here γ is the ratio of the maximum rate of patch propagation to worm's virulence. The *virulence* of the worm is infections per unit time or in other words maximum rate at which worm can spread.

Therefore, it is clear from the above work that P2P patch distribution mechanism provides a fast, scalable and cost effective alternative to the centralized server based patch distribution mechanism.

2.4.2 P2P Patch Distribution Solutions

There are many peer-assisted patch distribution approaches proposed in literature. This section presents a subset of such peer-assisted patch distribution solutions that are important in the content of our research problem.

Serenyi and Witten presented RapidUpdate [4], a peer-assisted distribution system to distribute security content. Their goal was to save vendor's bandwidth, while meeting distribution deadlines and allowing peers to participate fully in the system even if they are located behind a firewall or NAT device. The meaning of distribution deadline in their work is that every client in the community should have a file by an administratively controlled time. RapidUpdate is geared towards small files that are 200KB or less in size. RapidUpdate system consists of a *topology server* that

coordinates file distribution by routing clients that need a file to clients that already have it. RapidUpdate use specialized UDP (User Datagram Protocol) based protocol for client-server and client-client communication. This protocol facilitates UDP NAT hole punching such that clients can ultimately communicate directly with each other even if they are located behind a NAT device. Use of UDP makes their communication and topology server more scalable. RapidUpdate does not divide file into pieces as in BitTorrent because it is only tailored for small files. As in many other P2P patch distribution systems, RapidUpdate assumes that clients will discover availability of a patch by out-of-band polling mechanism. When a client wants to download the patch it first contacts the topology server. Then the server sends a reply message which instructs the client to obtain the patch from the HTTP server or from another client. If it is to download the patch from another client, topology server sends the contact details of the other client in reply message. Which action the server instructs the client to take is a function of its goal to minimize vendor bandwidth while meeting the specified deadline. If the deadline is a tight one, server will direct more clients to the HTTP server, to increase the number of initial seeds. If the deadline can be met by fetching the patch from another client and if there are any seeder clients available to fulfil the request, patch requesting client will be directed those seeding clients. Once a client obtains the patch, it will notify the server such that server can track which clients have the patch. Experimental results showed that RapidUpdate was able to meet the specified deadline while saving more than 70% of vendor's bandwidth in all cases. As RapidUpdate is tailored for small files, it can only be used for software patch distribution but not for software update distribution as software updates could be few or more Megabytes in size.

Rahman et al. proposed iDispatcher [5], a P2P information dissemination platform that is targeted to disseminate security patches quickly to client computers. The uniqueness of iDispatcher is that it provides a unified platform to support information dissemination from multiple sources in a seamless manner. iDispatcher uses a hybrid approach with both push and pull based information dissemination to achieve low dissemination latency. In iDispatcher, when a node initially join the network, it consults another component in the system called community server to obtain the bootstrapping information such as the IP address and port of another node already connected with the network. Information sources are known as dissemination centres

that are also a part of the P2P network. When some information is available the dissemination centre starts to push it to subset of its neighbours. Those nodes push that information to subset of their neighbours and so on. This is called a *push campaign*. Before pushing the received data to a neighbour, sending node verifies that the neighbour has not received the particular data previously by querying the neighbour. How many nodes a node pushes the received data is tuneable and this parameter it is called *quota*. Larger quota is used for time critical information. Thus, lower distribution latency is achieved in the expense of higher bandwidth usage as larger quota implies that nodes receive push requests for the same data from several neighbours. Furthermore, lower quota means a large number of nodes will not receive data from a push campaign as some nodes might leave out while other nodes pushing data to only a few of their neighbours. Size of the quota is selected dynamically depending on the time criticalness of the received data. Nodes that have missed information from a push campaign use pull method to retrieve the missed information. To identify whether it has missed any information, a node sends the list of all information IDs it received in the last few hours to its neighbours in a certain interval. After receiving such a list, the neighbour can verify whether it has missed any information. If it finds any information missing, it requests the sender to send that specific information. Authors have not clearly specified what the sender node is, in the above mentioned pull-based scenario. Dissemination centres sign the information they want to disseminate with their private keys and every node validates a dissemination centre's identity using its public key upon receiving that information. iDispatcher stores public keys of dissemination centres in a distributed hash table. As data from many sources is distributed in the same network, a node might receive data it does not need to consume. Nevertheless, the node should validate the received information and forward that information to its neighbours. This might be a waste of network resources. Authors have implemented a prototype of iDispatcher and deployed in PlanetLab [25] which had around 1,400 nodes at the time of iDispatcher experiments. Their results show that, for 2KB file, it took 20 seconds to reach around 90% nodes and took 70 seconds to reach all the nodes with quota size of 15. In this case, in average each node received around nine push requests to the same data. For a file of 2MB, it took around 300 seconds to reach around 90% nodes and took around 2,000 seconds to reach data to all the nodes with quota size of 15.

Xie and Zhu [3] studied the feasibility and effectiveness of using the existing P2P overlay structure to distribute security patches automatically to vulnerable nodes. Authors examined two approaches; partition-based approach and a Connected Dominating Set (CDS) based approach. In partition-based approach, a set of selected nodes is protected initially by applying the patch. These selected nodes are known as *guardian nodes*. Authors assume that those guardian nodes would block the spread of the worm through them after patching. Therefore, the P2P network is partitioned by these worm-immune guardian nodes and worm will not propagate from one partition to another. Hence, worm containment is possible in a small area in the P2P network. In CDS-based approach, a dominating set of nodes is chosen from the P2P overlay, which are known as the *key nodes*. Security patches are first delivered to these key nodes and those nodes will deliver the patch to the rest of the nodes. In the CDS-based approach, if P2P network do not have rich node connectivity, then the number of key nodes may be unreasonably large. If that is the case, authors propose to push security patches to a randomly selected subset of nodes from key nodes. However, if only a subset of key nodes receive the patch initially, some nodes may experience high latency for patch delivery, because they are located far away from a key node who received the patch. In both of their approaches to select a subset of nodes (guardian nodes or key nodes) security servers needs to be deployed in the P2P network. These security servers crawl the P2P network, construct the P2P topology, and then select a subset of nodes from the topology as guardian nodes or key nodes. If network is very large, crawling the P2P network will not be feasible and then their methods may not be scalable. Moreover, topology information in the security servers may be stale compared to actual topology due to churn in the P2P network.

2.4.3 Privacy Issues

Wu et al. [7] presented privacy issues that arise in P2P patch dissemination and proposed solutions for the same. The privacy problem in peer-assisted patch distribution is that, when peer *A* requests a patch from another peer *B*, it announces its vulnerability to *B*, which *B* can exploit instead of providing the patch [7]. Using analytical modelling and simulation authors show that large fraction of hosts gets compromised with a basic model (which does not consider privacy issues) for peer-assisted patch distribution. They proposed two solutions for the privacy problem in

peer-assisted patch distribution, that are honeypot-based approach and anonymizing network based approach. In honeypot-based approach, honeypots detect and blacklist malicious peers who listen for patch requests and attack vulnerable hosts. In the anonymizing network based approach, patch requests are sent through an anonymizing network. In an anonymizing network, receiver of the message cannot find the identity of the sender, and this is usually achieved by forwarding the message through multiple intermediate hops. Anonymizing network hides the identities of susceptible hosts from malicious peers. Using simulations authors showed that honeypot-based approach is not a good candidate and anonymizing network is more suitable for secure patch dissemination. However, anonymizing networks are slow as the messages are forwarded through many hosts to hide the identity of the sender. Moreover, anonymizing network based approach is not very effective for large patch sizes. Moreover, when using an anonymizing network to request patches only the patch requester's identity is hidden from other peers while patch provider is always known to the patch requester. Therefore, patch provider's privacy is not preserved with an anonymizing network. In their simulations, the simulation network ended up with about 80% of infected peers at network scanning rate of 0.01 when patch size is 675KB. Therefore, both approaches are not effective, if patch size is large.

2.5 Defending Against Worms

A new trend of defending against P2P worms is to use benign worms [27]. These worms are also known as anti-worms [16]. A benign worm is a worm, which utilizes the spreading mechanism of a malicious worm. Benign worms are used to disinfect the infected nodes and to patch the vulnerability [27]. A benign worm use the same security vulnerability as the malicious worm to spread. Benign worms have some drawbacks like generating large traffic on the network when spreading [27], [16]. Some of the practical benign worms are Nachi, Code green, CRClean and Welchia [16]. In many countries, it is illegal to intrude a system without the user's permission [27]. Therefore, benign worms are not legal as they spread by intruding.

Chen et al. [27] presented two benign worms known as SWORM and RWORM, which are supposed to work together to quarantine malicious worms. Both of these worms spread in an active manner that is these worms spread actively by searching

targets. SWORM is aimed at disinfecting and repairing the infected hosts. SWORM does not contain the patch for the vulnerability hence hosts which are affected by SWORM still susceptible to malicious worms. Disinfecting and repairing made by SWORM includes killing the worm processes, recovering the damaged files and registry keys, deleting files and registry keys created by malicious worms. After some time when the patch is available, propagation of RWORM will be started. RWORM contains the patch for vulnerability and it will patch the vulnerability when it spreads to susceptible hosts. If the host is infected (not affected by SWORM) RWORM will patch the security hole. If the host is affected by SWORM (but still contains the vulnerability and susceptible), RWORM will remove the SWORM and patch the security hole. If the host is susceptible (not affected by malicious worm or the SWROM) RWORM will fix the security hole. The reason to have two kinds of worms is benign worms like SWORM can be generated quickly while benign worms like RWORM which contain the patch requires time, to generate and test. Authors do not specify about how to detect malicious worms or how to generate these benign worms. They derive discrete differential equations to describe the propagation and the interplay of malicious and their benign worms. Authors also implemented a simulation to validate their approach. Using simulations authors show that their approach is about two times faster and protects about 35% more hosts compared to sheer manual reactions and their method protects about 34.4% hosts with lower consumption of bandwidth when compared to the random scanning benign worms.

Jia et al. [16] also proposed a benign worm to patch the infected and vulnerable hosts. Their benign worm propagates by using a hit list, which consists of the IP addresses of the peers who are vulnerable to the malicious worm. This hit list is generated at some server, which initiates the propagation of benign worm. Their benign worm propagates through the same vulnerability, which has been used by the malicious worm. Authors assume that benign worm is stronger than the malicious worm so that if benign worm comes across the malicious worm it can destroy and clean the malicious worm. The functions of their benign worm include invading the vulnerable peer, cleaning the malicious worm, dividing the hit list to propagate to next targets, patching, recording peer information, reporting to the server and self-destructing. When propagating from one host to another hosts, benign worm divide the existing hit list among the next hosts and propagation stops when hit list becomes null. To

decrease the payload of the benign worm the patch has been made external to the benign worm. Therefore, the patch is downloaded from the central server or some peer who has been patched previously by the propagation of the benign worm. The benign worm should report to the server which peers it passed by before it self-destructs. The server uses the reporting information from the benign worm to update the vulnerability information of the network. In our perspective, approach proposed by Jia et al. is not practical because in practical networks, it is very difficult to know which hosts are vulnerable to a particular vulnerability. Also generating a hit list by examining these vulnerabilities of a larger population at a single location is not feasible. Hence, generating a hit list for the benign worm is not practical. In addition, if an intelligent worm closes the backdoor or the vulnerability after infection, propagation of their benign worm is not possible.

2.6 OMNet++ Based BitTorrent Module

Evaluation of the proposed patch distribution system in this research is achieved through simulations. To simulate the proposed patch distribution system, a BitTorrent module [14] developed for the OMNet++ [13] simulation framework is used. This specific BitTorrent module was selected due to following reasons:

1. It provides packet-level simulation enabling more realistic simulation environment.
2. It simulates a more realistic network which is based on INET [28]. INET simulation framework implements several network protocols including complete TCP/IP stack for the OMNet++ platform.
3. Its underlying framework, OMNet++ is a very powerful simulation framework, which provides modularity by design and simulation modules in OMNet++ are highly customizable and extensible.
4. It provides more realistic BitTorrent node arrival process which is proposed in [29].

An OMNet++ simulation model consists of modules which communicate with each other by message passing. There are two types of modules in OMNet++, which are simple modules and compound modules. Simple modules are the basic building

blocks of a simulation model. Behaviour of a simple module is defined by writing C++ code and behaviour of one simple module is encapsulated in one C++ class. A compound module consists of several simple or compound modules. The behaviour of a compound module is defined by its sub-modules and how those sub-modules are connected together.

The BitTorrent module consists of three main modules which are Tracker, TrackerClient and Peer-Wire modules as shown in Figure 2.6. One BitTorrent node is known as BTHost. BTHost is a compound module which consists of simple modules TrackerClient and Peer-Wire. Tracker module provides the functionality of the tracker. TrackerClient module handles the communication between the peer and the tracker. Peer-Wire module provides the functionality of the peer-wire protocol. Peer-wire protocol is the protocol between two BitTorrent peers, which is described in Section 2.2.2.1. Communication between a peer and the tracker or between two peers happens through a simulated TCP/IP network which is provided by INET.

Tracker module consists of two C++ classes that are BTTrackerBase and BTTrackerClientHandlerBase. BTTrackerBase class handles the server functionality of the tracker and BTTrackerClientHandlerBase class handles the communication with the client. For each client BTTrackerClientHandlerBase instance is created. Similarly Peer-Wire consists of two C++ classes that are BTPeerWireBase and BTPeerWireClientHandlerBase. All the functionality of a BitTorrent peer is implemented in the BTPeerWireBase class including the BitTorrent choking algorithm. BTPeerWireClientHandlerBase class handles the communication between peers and instance of this class is created for each connection between peers at both ends of the connection.

For dynamic BitTorrent node deployment in the network, the OMNet++ BitTorrent module make use of the OverSim [30], which is P2P simulator implemented on OMNet++. For the node arrival process, BitTorrent node arrival process proposed in [29] is used.

It should be noted that a BitTorrent seeder is implemented as a separate node called BTHostSeeder and only one seeder is supported, which is initially deployed in the network along with the tracker. Having only one seeder is a limitation of the OMNet++ BitTorrent module.

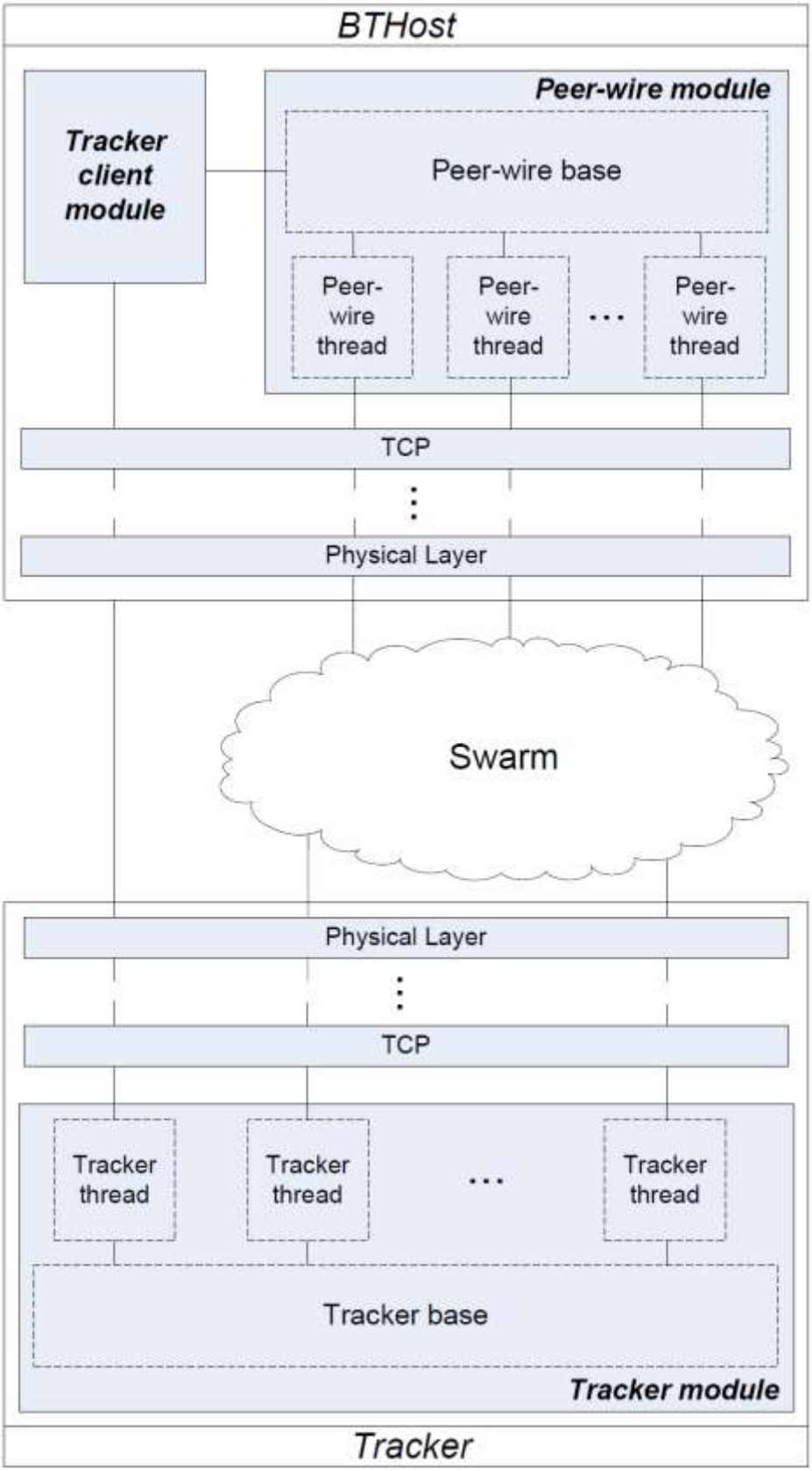

Figure 2.6: BitTorrent module architecture [14].

2.7 Summary

This chapter explained computer worms, P2P overlay networks, existing solutions for preserving privacy in P2P networks, existing peer-assisted patch distribution solutions and existing solutions for preserving privacy of patch sharing peers in peer-assisted patch distribution. It was highlighted that only existing solution for preserving privacy in peer-assisted patch distribution, is not sufficiently robust in terms of performance. The OMNet++ based BitTorrent module, which is used carry out simulations in this work, was described as well. The following chapters utilize the theory presented in this chapter to build up a novel peer-assisted patch distribution system.

Chapter 3

Methodology

This chapter describes the implementation of the proposed peer-assisted patch distribution system. The chapter starts by presenting the solution overview (Section 3.1) and system overview (Section 3.2). Section 3.3 presents pseudo-downloaders, their selection and their behaviour. Section 3.4 describes modifications that are required at the tracker to implement the proposed patch distribution system. Finally, chapter concludes by presenting suggested modifications to the BitTorrent protocol in Section 3.5 and operational constraints of the proposed system in multi-tracker environment in Section 3.6.

3.1 Solution Overview

We propose a peer-assisted patch distribution system which is based on the most popular P2P file sharing protocol, BitTorrent [12]. BitTorrent peers who shares a particular file is collectively known as a *swarm*. We adopt the BitTorrent protocol, by introducing minor modifications, such that it would not expose patch sharing peers to each other and it would prevent interaction between vulnerable peers and attackers. *Attackers* are the peers who are infected with the topological worm. The proposed patch distribution system is supposed to be utilized by multiple software vendors at the same time. A set of non-vulnerable peers, that do not have the vulnerability addressed by the patch, are inserted into the swarm to avoid the direct contact between patch sharing peers. We term these peers as *pseudo-downloaders*. Such none vulnerable peers anyway exist in the patch distribution system as the proposed patch distribution system is utilized by multiple software vendors. For example, for a Windows patch, peers who share a Linux patch or update could be selected as pseudo-downloaders. Instead of connecting directly, peers are connected with each other through pseudo-downloaders. Therefore, pseudo-downloaders act as intermediate hops while facilitating the file transfer between patch sharing peers and as such, peers who actually share the patch are not exposed to each other. As these pseudo-downloaders are not vulnerable, the worm cannot exploit pseudo-downloaders. Therefore, pseudo-downloaders would not become infected by the worm, and as a result pseudo-downloaders would not become attackers. Thus, vulnerable peers are protected from the topological worm as vulnerable peers would not connect with an infected peer.

These pseudo-downloaders are selected randomly by the tracker and a peer cannot deliberately elect itself as a pseudo-downloader. Tracker plays the role of placing the pseudo-downloaders as intermediate hops between patch sharing peers. When a patch sharing peer requests a peer list from the tracker, tracker only sends a set of pseudo-downloaders and when a pseudo-downloader requests a peer list from the tracker, tracker sends a mix of pseudo-downloaders and patch sharing peers. Therefore, a patch sharing peer would never know about other patch sharing peers and even randomly selected pseudo-downloaders cannot differentiate peers who share the patch actually from other pseudo-downloaders. Thus, the privacy of patch sharing peers is preserved.

3.2 System Overview

3.2.1 Selection of P2P File Sharing Protocol

As stated in the problem statement the goal of this research is to design a peer-assisted patch distribution system which alleviates the address exposure problem. File sharing protocol among peers plays a key role in a peer-assisted patch distribution system. We had two design options when selecting P2P file sharing protocol. First option was to design a brand new P2P file sharing protocol and second option was to use an existing well-established P2P file sharing protocol. If new P2P file sharing protocol was introduced, its effectiveness and practical usefulness should be validated thoroughly, and it is not a trivial task. Moreover, building a sufficiently large user community with both the peers interested in a particular software and of the pseudo-downloaders would be difficult due to network effect where users tend to stay in a large well-known network than adopting a new unknown network. As there are several well established file sharing protocols available at present we decided to select an existing P2P file sharing protocol.

BitTorrent [12] is the most popular and successful P2P file sharing protocol to the date. Organizing peers who share the same file into a single P2P overlay, providing fast download rates by allowing peers to download multiple chunks of the file concurrently from multiple sources, rarest first chunk scheduling and discouraging free-riding by introducing an tit-for-tat incentive mechanism were the main reasons for success of BitTorrent over early P2P file sharing protocols [29]. Therefore, in the

proposed peer-assisted patch distribution system BitTorrent is used as the file sharing protocol between peers. However, we adopt the BitTorrent protocol by introducing minor changes to the BitTorrent protocol to prevent the address exposure problem. The modifications introduced to the BitTorrent protocol is described in Section 3.5. We note that, even though our work is based on BitTorrent, techniques we have introduced in this research can also be used with other P2P file sharing protocols. In that case, the role of the BitTorrent tracker needs to be fulfilled by some other central component which assists peers to discover each other.

3.2.2 High-Level Architecture

A key feature of the proposed peer-assisted patch distribution system is its ability to download patches of all kinds of software from various vendors through a single system. Figure 3.1 shows the high-level architecture of the proposed patch distribution system. The patch distribution system contains three main components, which are patch distribution servers, a tracker, and client computers. Central *patch distribution servers* assume the same role as in a traditional patch distribution system. In addition to that, as in other peer-assisted patch distribution systems [4], [7], central server might direct clients to download the patch from the P2P network if it is busy. *Tracker* assumes the same role as in an ordinary BitTorrent P2P network. Tracker hosts torrents of software patches and updates from many vendors and there could be multiple trackers, if one tracker cannot handle the load. Figure 3.1 shows an example setup with a tracker, two patch distribution servers for software *X* and *Y*, and clients who installed with software *X*, *Y* and both.

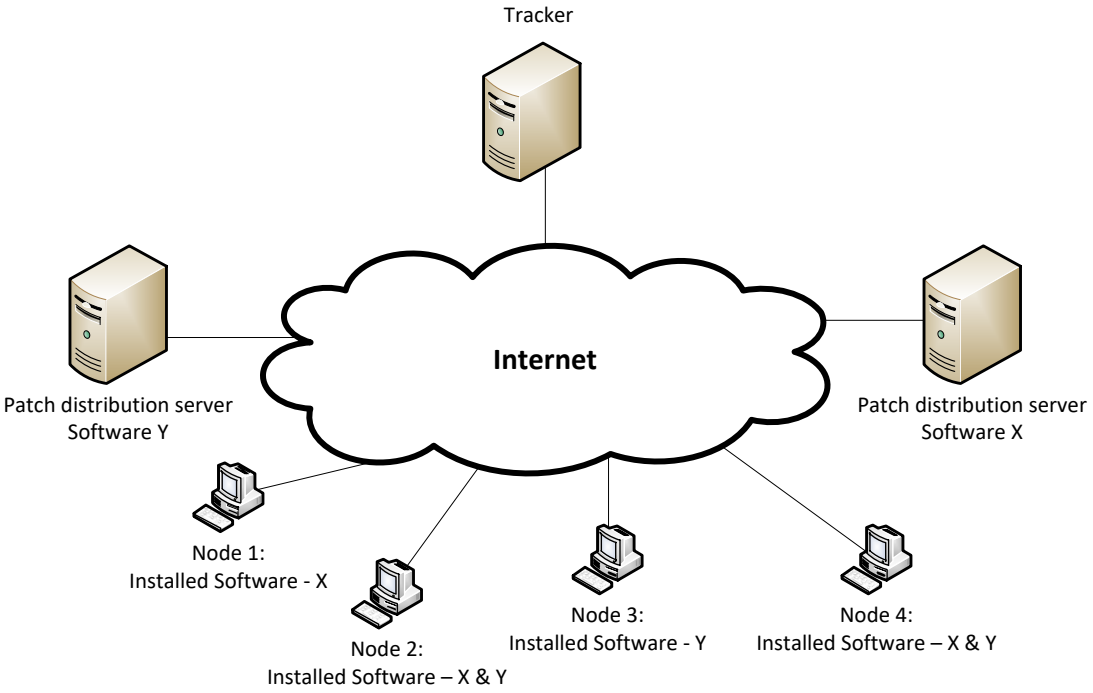

Figure 3.1: Basic components of the patch distribution system.

3.2.3 P2P Client Application

Client computers are installed with a P2P client application to download the software patches and updates in P2P style. Both the tracker and the P2P client application run the modified version of the BitTorrent protocol. The P2P client application installed in client computers is used to download software patches for multiple software installed on the client machine as proposed patch distribution system is intended to download software patches from many software vendors. The P2P client application provides a service to other software installed on the client machine to download software patches and updates through the P2P network. Other software installed on the client machine register themselves with the P2P client application as it needs to know the details of software and their versions installed on the client machine. Why these details are necessary will be described in Section 3.3.3. Once software installed on client machine needs to download a patch from the P2P network, it notifies the P2P client application with corresponding torrent file which it obtained from the central patch server after polling the central server. Once the patch or update file download is complete P2P client application will notify back to the relevant software to install the patch.

3.2.4 Node States

When particular software vulnerability is considered, client machines in the Internet have several states with respect to the particular vulnerability. We define four such states. Those states are:

1. Vulnerable – Any node that has the vulnerability and has not installed the patch is in vulnerable state.
2. Infected – If a vulnerable node infected with the worm before applying the patch, it goes to the infected state. We assume that the infection cannot be cleaned by applying the patch. This is a reasonable assumption because in most cases patches only fix the vulnerability and do not targeted for cleaning adversaries which have exploited the vulnerability. Thus, once a node is infected it will remain infected even if the vulnerability has been fixed by applying the patch.
3. Immune – Once a vulnerable node is installed with the patch before it gets infected it goes to the immune state.
4. Not-vulnerable – There are nodes that do not have the vulnerability addressed by the patch because they are not installed with the particular software which has the vulnerability. Therefore, those nodes are in not-vulnerable state.

3.2.5 Attack Model

The main attacker we consider in this work is a topological worm which exploits the vulnerability fixed by the patch which is being distributed in P2P fashion. Therefore, we treat any peer that has been infected by the worm as an attacker. Infected peers would try to attack every peer known to them. That means an infected peer will attack the remote peer of every established connection. This could be the worst-case attack model, however by assuming this model we show that our solution is resilient against such an attack model as well. Thus, in our experiments, we assume that a node would become infected if it connected with an already infected node. Note that as previously stated, we assume that node infections cannot be cleaned by applying the patch. This implies the fact that there could be attacking peers who have fixed their vulnerability and also have the complete patch.

In addition to topological worms, proposed peer-assisted patch distribution system should provide the protection against attackers who attempt to discover which peers share which patches. By doing so, those attackers can determine which software are installed on which nodes. If an attacker can build such an association between peers and installed software on those peers, that will be a threat to the privacy of patch sharing peers. Note that, to defend against topological worms, only vulnerable peers, i.e., peers without the complete patch, need to be protected. However, to protect from attackers who attempt to build an association between peers and installed software on those peers, privacy of all peers in the system, including peers who have fixed their vulnerability, should be preserved. Thus, another type of attacker we consider in this research is an attacker who attempts to determine patch interest of peers. These attackers might attempt to achieve their goal either by participating in the swarm and establishing connections with other peers as a regular peer or they might query the tracker to get a list of peers who shares a particular patch.

3.2.6 Node Classification

For analysis purposes and to build up the solution we attempt to identify the possible attackers who are infected with the topological worm. Therefore, we present following arguments:

1. Attackers are the peers who are in the infected state.
2. Any peer who has the vulnerability could be attacker because it might be already infected with the topological worm.
3. Peers who downloaded the complete patch and fixed their vulnerability might be attackers as well, due to the fact that infection cannot be cleaned by applying the patch.
4. From point 2 and 3, it is clear that any peer that has or had the vulnerability could be an attacker because it might be already infected with the topological worm.
5. Any peer who has showed an interest on downloading the patch could be vulnerable because it is looking for patch to fix its vulnerability.
6. From point 4 and 5, it can be derived that, any peer who has showed an interest on downloading the patch could be an attacker.

7. Furthermore, if some peer has not showed any interest on the patch, implies the fact that probably it is not installed with the vulnerable software and could not be infected with the worm.

Thus, by considering the past interests of peers we analytically classify them into two categories as:

1. Probable attackers – Probable attackers are the peers who have showed an interest about the patch. i.e., any peer participating in the swarm in which the patch is shared could be an attacker.
2. Probable benevolent peers – Probable benevolent peers are the peers who have not showed any interest about the patch until now. Therefore, peers who are not sharing the particular patch are most probably benevolent. However, there could be peers who have the vulnerability, but not showed any interest on downloading the patch yet. Therefore, when designing the solution we consider the fact that there could be attackers in the probable benevolent peer population as well.

3.3 Pseudo-Downloaders

In the proposed patch distribution system, exposure of peers to each other is prevented by introducing a set of peers to the swarm, which are called pseudo-downloaders. A pseudo-downloader (P_s) act as an intermediate hop between the patch sharing peers to prevent direct interaction between patch sharing peers while facilitating file transfer. Vulnerable peers only connect with pseudo-downloaders to protect themselves from probable attackers and pseudo-downloaders freely connect with other peers in the swarm. Figure 3.2 illustrates connections between two peers with and without pseudo-downloaders. We denote normal downloaders as *true-downloaders* or *true-peers* to distinguish from them pseudo-downloaders.

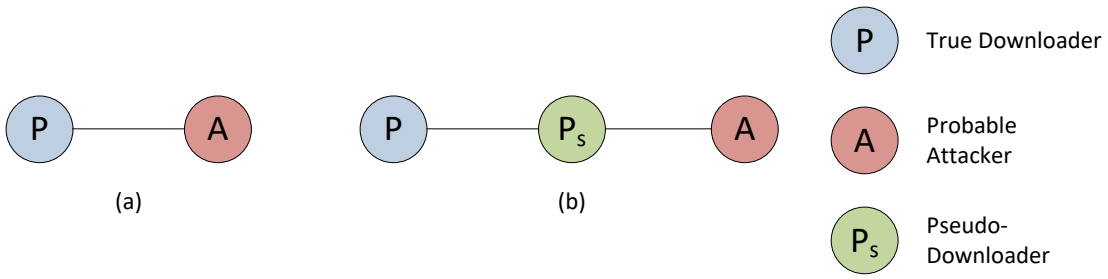

Figure 3.2: Connection between peers; (a) without pseudo-downloaders (b) with pseudo-downloaders.

Note that, pseudo-downloaders do not act as relays to facilitate file transfer between patch sharing peers. However, they download the file independently as any other peer in the swarm even though they are not interested in the file; hence named pseudo-downloader. As pseudo-downloaders also download the file, vulnerable peers who are connected to pseudo-downloaders, can request required pieces from pseudo-downloaders and complete their file download process.

3.3.1 Pseudo-Downloader Selection

Pseudo-downloaders are selected from probable benevolent peers, i.e., pseudo-downloaders are selected from peers that do not share the particular patch in the P2P network. Therefore, for the patch P_X, which is targeted for software X, pseudo-downloaders are selected from peers who share some other patch or update which is not targeted for software X. For instance, as a pseudo-downloader for a Windows patch, peers who share a Linux patch or update could be selected. This is possible because the proposed peer-assisted patch distribution system is used to distribute patches of many software vendors. Pseudo-downloaders are selected from probable benevolent peers, to ensure that pseudo-downloaders are not attackers and pseudo-downloaders are not vulnerable as well. Pseudo-downloaders' benevolence and security are assessed more thoroughly in Section 3.3.4.

Tracker selects pseudo-downloaders by examining which peers are registered for which torrents. In the proposed patch distribution system, tracker has information about targeted software of each patch or update hosted by the tracker. Therefore, tracker can select pseudo-downloaders for a particular patch simply by examining its

local data. How tracker's data structures are arranged will be described in Section 3.4.1.

Size of the peer population that could be selected as pseudo-downloaders could be huge. Thus tracker randomly selects only a subset from this population as pseudo-downloaders. We term this as *pseudo-downloader pooling* at tracker. As tracker selects small set of peers as pseudo-downloaders from a large population, no peer will be able to deliberately elect itself as a pseudo-downloader for a particular patch. This act as a good security measure against attackers, who are not necessarily peers infected with the topological worm, but who try to enter into the pseudo-downloader population of a particular patch deliberately, to carry out malicious activities.

3.3.2 Pseudo-Downloader Insertion into the Swarm

Tracker inserts pseudo-downloaders to the swarm by including them in the peer list of the tracker response. When a vulnerable peer requests a peer list from the tracker, tracker only includes pseudo-downloaders into the peer list. Then instead of connecting with true downloaders in the swarm, vulnerable peers will only connect with pseudo-downloaders to obtain the patch. Once a peer receives such connection for a torrent, for which it does not share, the peer will start to participate in the swarm as a pseudo-downloader. Therefore, by inserting a set of peers who do not share the patch in to the peer list, they will start to participate in the swarm as pseudo-downloaders. Once a pseudo-downloader has joined the swarm, it will query the tracker to obtain a list of peers to download file pieces just as any other peer. However, when sending a peer list to a pseudo-downloader, tracker sends mix of seeders and pseudo-downloaders in the peer list. Therefore, pseudo-downloaders will connect with seeders and other pseudo-downloaders and act as a bridge between seeders and vulnerable peers who does not have the complete patch. When, responding to a pseudo-downloader, tracker includes a mix of pseudo-downloaders and seeders in order to preserve the privacy of seeders. If tracker only sent details of seeders to a pseudo- downloader, the pseudo-downloader would be able to accurately determine peers participating in the swarm. Leechers do not have the complete patch and they are guaranteed to be vulnerable. As such, leechers are never included in the peer list by the tracker. As result of above actions, an overlay with connections between peers as shown in the Figure 3.3 is formed when sharing a patch.

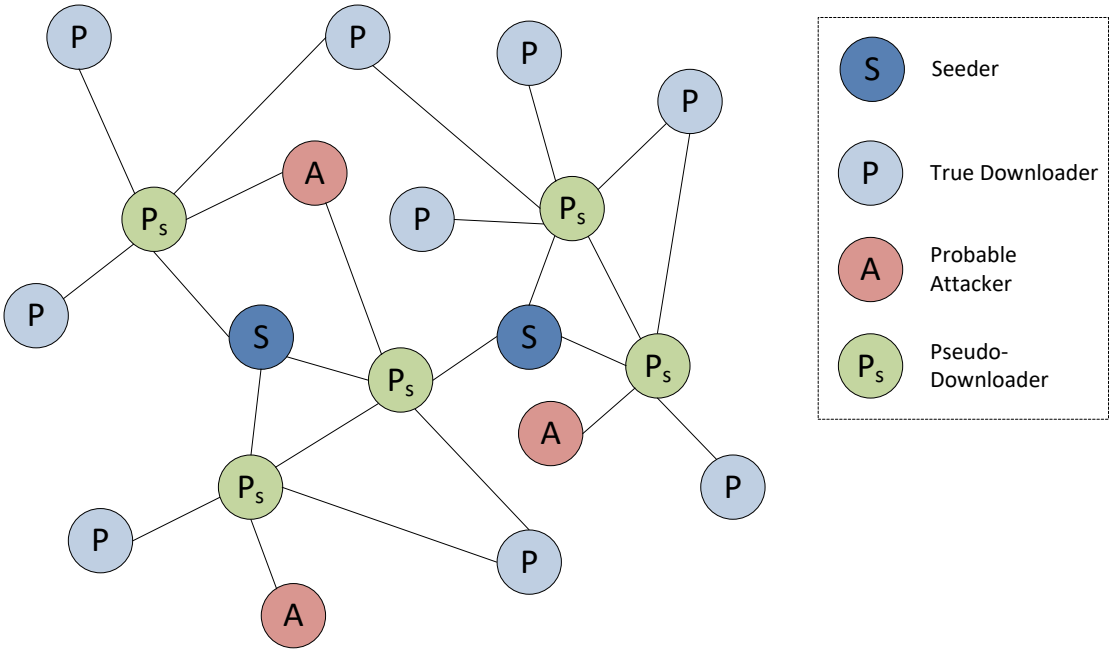

Figure 3.3: Connections between peers while using pseudo-downloaders.

3.3.3 Behaviour of Pseudo-Downloaders

Consider a Peer *A*, which is interested in Patch *X*. When *A* requests a peer list from the tracker, tracker will include pseudo-downloaders into the peer list. Suppose Peer *B* which is sharing another torrent but not Patch *X*, has been inserted into the peer list as a pseudo-downloader. After receiving the peer list from the tracker, *A* will start to connect with peers in the peer list. As a result *A* will also initiate a connection to *B*, and then *A* will send a hand-shake message to *B*. When a peer receives a connection from another peer, how that connection is handled by the connection reception peer, in this particular case Peer *B*, is shown in Algorithm 3.1.

Algorithm 3.1 Handling a new peer connection

```
1: function HandleNewPeerConnection(connection, handShakeMessage)
2:
3:     if IsSharingTorrent(handShakeMessage.infoHash) = false then
4:         SendTorrentFileRequestMsg(connection)
5:         torrentFile ← ReceiveTorrentFileResponseMsg(connection).
6:
7:         if IsPatchRequired(torrentFile.softwareInfo) = true then
8:             ParticipateInSwarmAsTrueDownloader(torrentFile)
9:         else
10:            ParticipateInSwarmAsPseudoDownloader(torrentFile, connection, handShakeMessage)
11:        end if
12:
13:    else
14:        AddConnectionToTorrent( handShakeMessage.infoHash, connection)
15:        SendHandShake(connection, handShakeMessage.infoHash)
16:        ExchangePieces(connection)
17:    end if
18: end function
```

In BitTorrent, the *handshake* message contains the info-hash of the torrent. Upon reception of the handshake message Peer *B* first checks whether it is sharing the particular torrent, using the info-hash of the received handshake message. This action is shown line 3 in Algorithm 3.1. If *B* is sharing the torrent already as a pseudo-downloaders or a true downloader, it will add the new connection to the torrent and then proceed to exchange pieces with the new peer after sending a reply handshake message. Those steps are shown in line 14-16. In this particular instance, peer *B* is not sharing the requested info-hash and that info-hash is unknown for *B*. Therefore, condition in line 3 evaluates to false. The default behaviour of BitTorrent is to drop the connection when a connection for unknown info-hash (i.e., unknown torrent) is received. However, in our solution pseudo-downloaders always receive connections for unknown torrents and they require accepting those connections to assist other peers to hide themselves from attackers. Therefore, we modified this default BitTorrent behaviour such that when a peer receives a connection for unknown torrent, instead of dropping the connection, the peer will start to share (i.e., start to download) the specified torrent as a pseudo-downloader.

Thus, when Peer *B* receives the handshake message from Peer *A*, *B* starts to download the Patch *X*, even if *X* is not relevant to *B*. This scenario is called as, Peer B is acting as a pseudo-downloader. However, in order to participate in the swarm for Patch *X*, Peer *B* needs to know, the torrent file of the patch, as at this point *B* only knows the info-hash. Therefore, *B* sends a message to *A* asking the torrent file of the requested info-hash on the same connection. This is shown in line 4. Then in response Peer A sends the torrent file for the *X* which is shown in line 5. Now Peer *B* has all the

information to participate in the swarm as a pseudo-downloader for Patch *X*. This approach reduces the load on the central patch distribution server(s) as a pseudo-downloader does not need to obtain the torrent file again. Moreover, this does not reveal any extra information about the presence of peers as *A* is already aware of pseudo-downloader *B*.

Even though Peer *B* was not sharing Patch *X*, there is a possibility that *B* also has the vulnerability addressed by *X*. For an instance, *B* might be unaware of the release of Patch *X* yet. If that is the case, if *B* start to share the patch as a pseudo-downloader, it might also get attacked, because pseudo-downloaders are always exposed to probable attackers. Therefore, before participating as a pseudo-downloader in the swarm for Patch *X*, *B* must make sure that it does not contain the vulnerability addressed by *X*. In the proposed patch distribution system, torrent file contains the targeted software name and the version of the patch. Furthermore, P2P client software has details (i.e., software names and their versions) of installed software on the system. Thus, after receiving the torrent file from *A*, *B* can safely determine whether it requires Patch *X*, by comparing the target software details of the patch with its current software details. This is shown in line 7 in Algorithm 3.1. If patch is required for *B*, instead of participating in the swarm as a pseudo-downloader *B* will start to download the patch as a true-downloader. This is shown in line 8.

If Patch *X*, is not required by *B*, it starts to participate in the swarm as a pseudo-downloader. As the name implies, pseudo-downloaders participate in the swarm just as any other peer, even though the patch is not required for them. Pseudo-downloaders send piece requests to other peers and supply pieces to other peers as they request. If pseudo-downloader stayed in the swarm for a sufficient time it might download the whole patch as well. There are two differences between a true downloader and a pseudo-downloader. First, pseudo-downloaders and true downloaders use different parameters in tracker request message. In our solution we have introduced a new field to tracker request message, which is "*isPseudoDownloader*" field. Pseudo-downloader flag indicates to the tracker that, this particular peer is a pseudo-downloader. Pseudo-downloaders set *isPseudoDownloader* flag and true downloaders do not. It helps tracker to identify who are pseudo-downloaders and who are true peers. Second difference between a true peer and a pseudo-downloader is that once there are no connections initiated by other peers for the particular torrent (i.e., Patch *X* in this

example) pseudo-downloader will stop sharing the particular patch. That means pseudo-downloader will tear down all the connections it has initiated for that particular torrent and stop sending tracker request messages to the tracker for that particular torrent. This is because, it will be a waste of resources to participate in a not interested swarm once there are no true peers to serve.

How a peer behaves when it is acting as a pseudo-downloader is shown in Algorithm 3.2. First, the peer will initialize the torrent as shown in line 3. This step includes creating necessary memory and data structures for the torrent. Then the peer handles the connection it received, which triggered the process of participation as a pseudo-downloader. This is shown in lines 4-6. Then the peer enters a loop where it sends tracker requests, initiate connections with the peers in the peer list and exchange pieces with those peers. This is shown in lines 8-25. Note that the peer sets the *isPseudoDownloader* flag to true, in the tracker request as shown in the line 10 as it is a pseudo-downloader. Furthermore, once there are no connections initiated by other peers for the particular torrent, it will stop the participation in the torrent as shown in lines 20-23.

How a peer behaves when it is acting as a true-downloader is shown in Algorithm 3.3. First, the peer will initialize the torrent as shown in line 2. Then the peer enters a loop where it sends tracker requests, initiate connections with the peers in the peer list and exchange pieces with those peers. This is shown in lines 3-14. As shown in line 4 the peer sets the *isPseudoDownloader* flag to false because it is a true-peers. Note that peer will continue to participate in the swarm even after the download is complete, as it can act as a seeder after the download is complete.

Algorithm 3.2 Participate In Swarm as a Pseudo-downloader

```
1: function ParticipateInSwarmAsPseudoDownloader(torrentFile, connection, handShakeMessage)
2:
3:     InitializeTorrent(torrentFile)
4:     AddConnectionToTorrent( handShakeMessage.infoHash, connection)
5:     SendHandShake(connection, handShakeMessage.infoHash)
6:     ExchangePieces(connection)
7:
8:     while true do
9:         trackerReq.infoHash ←torrentFile.infoHash.
10:        trackerReq.isPseudoDownloader ← true.
11:
12:        SendTrackerAnnounceRequest(trackerReq)
13:        trackerRes ← ReceiveTrackerResponse.
14:        for each peer in trackerRes.peerList do
15:            connection ← ConnectAndSendHandshake(torrentFile.infoHash).
16:            ExchangePieces(connection)
17:        end for
18:        Wait(announceInterval)
19:
20:        if GetPassivelyEstablishedConnectionCountForTorrent(torrentFile.infoHash) = 0 then
21:            DisconnectActivelyEstablishedConnectionsForTorrent(torrentFile.infoHash)
22:            return
23:        end if
24:
25:    end while
26:
27: end function
```

Algorithm 3.3 Participate in Swarm as a True-downloader

```
1: function ParticipateInSwarmAsTrueDownloader(torrentFile)
2:
3:     InitializeTorrent(torrentFile)
4:     AddConnectionToTorrent( handShakeMessage.infoHash, connection)
5:     SendHandShake(connection, handShakeMessage.infoHash)
6:     ExchangePieces(connection)
7:
8:     while true do
9:         trackerReq.infoHash ←torrentFile.infoHash.
10:        trackerReq.isPseudoDownloader ← false.
11:
12:        SendTrackerAnnounceRequest(trackerReq)
13:        trackerRes ← ReceiveTrackerResponse.
14:        for each peer in trackerRes.peerList do
15:            connection ← ConnectAndSendHandshake(torrentFile.infoHash).
16:            ExchangePieces(connection)
17:        end for
18:        Wait(announceInterval)
19:    end while
20:
21: end function
```

3.3.4 Assessment of Benevolence and Safety of Pseudo-downloaders

This section describes how the proposed patch distribution system ensures the safety of pseudo-downloaders and their benevolence. In order to maintain the safety of the peer-assisted patch distribution system pseudo-downloaders should fulfil the following two requirements:

1. For their own protection pseudo-downloaders should not be vulnerable. Otherwise pseudo-downloaders will be attacked as they freely connect with other peers in the swarm.
2. For the protection of vulnerable true downloaders, pseudo-downloaders should not be attackers.

By selecting pseudo-downloaders from probable benevolent peer population, above two requirements are satisfied as follows. Probable benevolent peers are the peers who have not showed any interest about the particular patch until now, as described in Section 3.2.6. They are not interested about the patch probably because they are not vulnerable and not installed with the particular software. Thus, by selecting pseudo-downloaders from probable benevolent peer population we almost satisfy the requirement of pseudo-downloaders should not be vulnerable. Nevertheless, there is a possibility that there could be vulnerable peers in the probable benevolent peer population even though they have not shown any interest regarding the patch until now. Thus, there may be vulnerable peers in the selected pseudo-downloader population. Nevertheless, when a pseudo-downloader received the first connection from a true peer, the pseudo-downloader first requests the torrent file from the true peer prior to taking any other action. The torrent file contains the targeted software name and the version. Therefore, pseudo-downloader can safely determine whether it requires the particular patch by comparing details in torrent file and details about locally installed software. If pseudo-downloader is vulnerable, it will find out that patch is required for itself and instead of participating in the swarm as a pseudo-downloader, it will participate in the swarm as a true downloader. Thereafter, tracker will no longer use that peer as a pseudo-downloader, as the peer is already participating in the swarm as a true peer. To summarize, even though vulnerable peers are selected as pseudo-downloaders, they will be removed from the pseudo-downloader population as soon they started to participate in the swarm, causing minimal harm to pseudo-downloaders.

Mainly attackers that are considered in this work are the peers that are infected with the topological worm. In addition, infected peers are a subset of vulnerable peer population as only vulnerable peers can become infected. Hence, by selecting pseudo-downloaders from non-vulnerable population, the second requirement, which is pseudo-downloaders, should not be attackers, is satisfied as well. Still, there could be

vulnerable peers among pseudo-downloaders, and in turn there might be infected peers among pseudo-downloaders. In this work it is assumed that even if a peer is infected, the worm will not be able to influence the patch downloading process because they are independent to each other. Thus, when an infected peer is selected as a pseudo-downloader, the peer will detect that it needs the patch and will try to download the patch as a true downloader as soon as it receives a connection for the patch. Then tracker will remove that infected peer from pseudo-downloader population as it is already participating in the swarm as a true peer. However, if the connection initiator is a vulnerable true peer it will be exposed to the infected pseudo-downloader. Thus, the time window an infected peer is kept at pseudo-downloader population is minimal. Hence, the probability of an attack due to an infected pseudo-downloader is minimal. We validate this argument through simulations.

In another case someone might try to place a malicious peer intentionally into the pseudo-downloader population to find out patch interest of other peers. This type of malicious peer would not seek for the patch and therefore it will never participate as a true peer in the swarm. Thus, once such a peer is selected as a pseudo-downloader it will not be removed from the pseudo-downloader population as in the case of an infected peer with the topological worm. Nevertheless, we note that intentionally getting into the pseudo-downloader population is difficult because tracker selects pseudo-downloaders from a large population of peers as described in Section 3.3.1. However, if such a malicious peer is selected as a pseudo-downloader by chance, even such a pseudo-downloader cannot determine which peers share the patch actually. This is because, a pseudo-downloader receives a mix of true-seeders and pseudo-downloaders in the peer list from the tracker. As a result, a pseudo-downloader receives connections from true-peers and as well as pseudo-downloaders. Thus a pseudo-downloader cannot determine exactly which peers are true-downloaders.

3.3.5 Piece Caching Effect at Pseudo-Downloaders

As stated previously pseudo-downloaders do not act as relays but they participate in the swarm actively even though they are not interested in the swarm. Furthermore, Pseudo-downloaders keep the pieces they download until they participate in the swarm. This is called *piece caching* at pseudo-downloaders. Therefore, true peers join the swarm lately find all or most of the pieces of the file in the pseudo-downloaders.

Because of the piece caching at pseudo-downloaders, a true peer will not experience a performance bottleneck due to pseudo-downloaders because a pseudo-downloader behaves as another regular peer in the swarm while increasing the collaboration in the swarm.

3.4 Modification to the Tracker

The tracker of the proposed patch distribution system needs to carry out some additional tasks than a traditional a BitTorrent tracker. These additional tasks are:

1. Selecting pseudo-downloaders for a particular torrent.
2. Managing a pseudo-downloader pool for each torrent and dynamically adjust the size of each pseudo-downloader pool corresponding to the true peer count in each torrent.
3. Handling announce-requests from pseudo-downloaders and validating those announce requests are from genuine pseudo-downloaders.
4. Identifying situations where a selected pseudo-downloader starts to participate in the swarm as a true peer. In such situations that particular pseudo-downloader should be removed from the pseudo-downloader pool.

In order to carry out these tasks tracker of the proposed patch distribution system uses a different set of data structures and a different tracker announce request processing algorithm than a traditional tracker. How tracker manages its data structures and how it processes announce requests will be described in next two sections.

3.4.1 Tracker Data Structures

A traditional BitTorrent tracker only needs to keep peers associated with each torrent. However, the tracker of the proposed patch distribution system needs to manage a pool of pseudo-downloaders for each torrent. Moreover, to select pseudo-downloaders for a particular torrent, tracker should be aware of the target software of each torrent (i.e., each patch). In order to make pseudo-downloader selection efficient torrents need to be arranged in such a way that torrents that are not targeted for particular software can be straightforwardly discovered. Therefore, data structures of the tracker are arranged as shown in Figure 3.4.

Tracker keeps a map called "*Torrent Map*" to maintain associations between the info-hash and the Torrent Objects. The key of this map is the info-hash and value is the corresponding torrent object. Each Torrent object contains the target software details of the torrent, the map of true peers who participate in the swarm and the pseudo-downloader pool selected for the particular torrent. The *Torrent Map* is used to discover the corresponding torrent object when a peer makes an announce request as the peer sends the info-hash in the tracker request. In addition to the *Torrent Map*, tracker keeps another map called "*Software Map*" to discover pseudo-downloaders for each torrent. As there could be multiple patches and updates for particular software, key of the *Software Map* is the software name and value is a *Patch Map*. A *Patch Map* contains mappings of patch name to Torrent Objects.

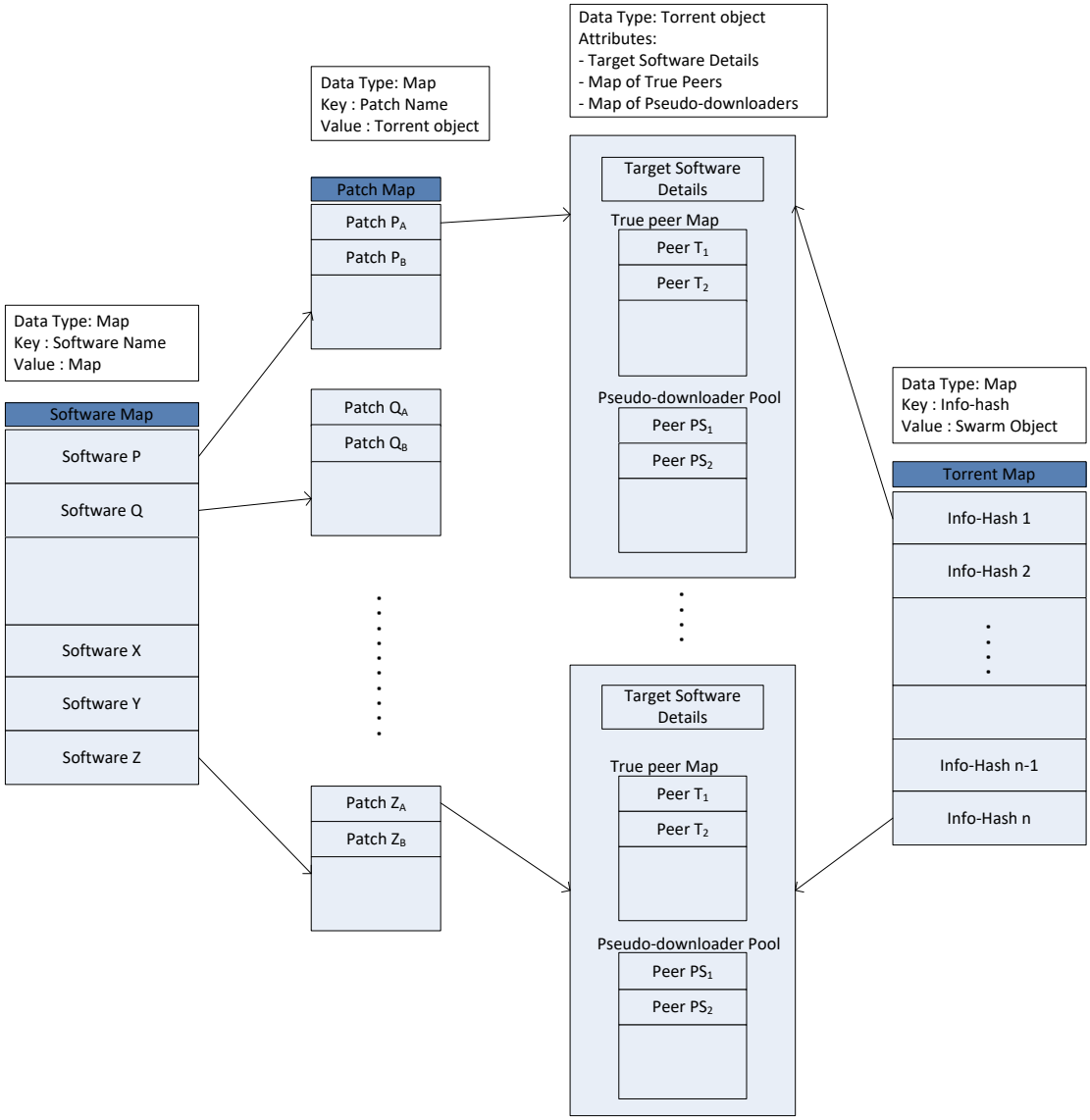

Figure 3.4: Tracker data strucutres.

Hence, when the tracker wants to discover pseudo-downloaders for a particular torrent *T*, tracker first derives the targeted software name *X* of the torrent. Targeted software name is an attribute of the Torrent Object. Then the tracker refers to the *Software Map* to find *Patch Map*s that are not related to the *X*. Those *Patch Map*s contains mappings to Torrent Objects that are not related to the *X*. Then the tracker selects pseudo-downloaders randomly from those Torrent Objects. However, before adding a selected pseudo-downloader P_S, to the pseudo-downloader pool *T*, the tracker first verifies that P_S is not participating in *T* as a true downloader. This is because peers who do not participate in *T* should be selected as pseudo-downloaders, for the protection of true peers and as well as protection of pseudo-downloaders, as specified in Section 3.3.1.

3.4.2 Tracker Announce Processing Algorithm

Both true-peers and pseudo-downloaders send announce requests to tracker for a particular torrent. However, tracker needs to differentiate between true-peers and pseudo-downloaders to process the announce request properly. Therefore, pseudo-downloaders set a flag called *isPseudoDownloader* in the tracker request to notify the tracker that a pseudo-downloader is making the announce request. *isPseudoDownloader* is a newly added field to BitTorrent tracker request in the proposed patch distribution system.

Announce processing algorithm of the tracker is shown in Algorithm 3.4. When an announce request arrived, tracker first checks whether the sending peer is a true-peer or a pseudo-downloader. This is shown in line 3. If it is a pseudo-downloader, tracker then checks whether the particular peer is a valid pseudo-downloader. This is done by confirming that the particular peer is present in the selected pseudo-downloader pool as shown in line 4. Because of this validation, a peer cannot pretend to be a pseudo-downloader if it has not been selected as a pseudo-downloader by tracker. If it is a valid pseudo-downloader, tracker fills a mix of seeders and pseudo-downloaders in to the tracker response as shown in line 9 and 10. This mix is a parameter per torrent and it is specified as *pseudoDownloaderMixFraction*. For example, when *pseudoDownloaderMixFraction* = 0.2 and maximum number of peers that is allowed fill in to the peer list is 50, then tracker fills 10 pseudo-downloaders and 40 seeders in to the peer list. This calculation is shown in line 6 and 7. If there is not enough pseudo-downloaders or seeders available, tracker will fill peers accordingly, while keeping the pseudo-downloader fraction in the peer list at 0.2.

If the peer is not a valid pseudo-downloader, tracker sets a failure message in tracker response and does not fill any peer into the tracker response. This prevents leaking information of patch sharing peers to intruders who try to pretend as pseudo-downloaders.

If a true-peer is making the announce request, tracker first checks whether that it has already selected the particular peer as a pseudo-downloader as shown in line 16. The particular peer might already present in the pseudo-downloader pool if tracker selected the peer as a pseudo-downloader before it participates in the particular swarm as a true-peer. Even though tracker might select vulnerable peers as pseudo-downloaders, a vulnerable peer will never participate in the swarm as a pseudo-

downloader as described in Section 0. Thus, if the particular peer is present in the pseudo-downloader pool, tracker removes the peer from the pseudo-downloader pool as shown in line 17. After those validations tracker fill only pseudo-downloaders into the tracker response. Then at the end of the processing tracker sends the tracker response to the client.

Algorithm 3.4 Tracker Announce Processing

```
1:  function ProcessAnnounce(trackerRequest)
2:
3:      if trackerRequest.isPseudoDownloader = true then
4:          if IsPeerPresentInPseudoDownloaderPool(trackerRequest.peerID) = true then
5:
6:              pseudoDownloaderCount ← maxPeersInResponse * pseudoDownloaderMixFraction
7:              seederCount ← maxPeersInResponse − pseudoDownloaderCount
8:
9:              FillSeeders(trackerResponse, seederCount)
10:             FillPseudoDownloaders(trackerResponse, pseudoDownloaderCount)
11:         else
12:             trackerResponse.failureReason ← "Action un-allowed".
13:         end if
14:
15:     else                                          ▷ a true-peer making the announce request
16:         if IsPeerPresentInPseudoDownloaderPool(trackerRequest.peerID) = true then
17:             RemovePeerFromPseudoDownloaderPool(trackerRequest.peerID)
18:         end if
19:         if IsSeeder(trackerRequest) = false then
20:             FillPseudoDownloaders(trackerResponse, maxPeersInResponse)
21:         end if
22:     end if
23:
24:     SendReponseToClient(trackerResponse)
25:
26: end function
```

3.4.3 Pseudo-Downloader Pool Size

The number of pseudo-downloaders participate in the swarm is exactly the same as the size of the pseudo-downloader pool at the tracker. There should be sufficient number of pseudo-downloaders in the swarm to service all true-peers in the swarm, and if there are excessive number of pseudo-downloaders present in the swarm it will be a waste of resources of pseudo-downloaders. The number of pseudo-downloaders that should be placed in the swarm should be determined by the number of true-leechers present in the swarm. This is because if there is large number of true-leechers in the swarm, there should be significant number of pseudo-downloaders to service them. Therefore, tracker determines the size of the pseudo-downloader pool by considering the number of active leechers. Tracker is configured with optimum pseudo-downloaders to true-leechers ratio. This ratio is known as *pseudo-downloader*

pooling factor. Hence, tracker dynamically adjust the size of the pseudo-downloader pool to keep the pseudo-downloaders to true-leechers ratio constant by adding and removing pseudo-downloaders as required. We carry out simulations to find out best pseudo-downloader pooling factor.

3.4.4 Security Enforcements by Tracker

As described previously tracker enforces connections between peers by including different types of peers in the peer list when sending the tracker response to different types of peers, i.e., true leechers, true seeders and pseudo- downloaders. Furthermore, tracker selects pseudo-downloaders randomly. As a result of these enforcements following restrictions will be hold in the overlay:

1. Tracker selects pseudo-downloaders randomly. Furthermore, when a pseudo-downloader requests a peer list from the tracker, tracker first validates whether the particular pseudo-downloader is present in the selected pseudo-downloader pool. Thus no peer can elect itself as a pseudo-downloader deliberately and no peer can pretend to be a pseudo-downloader.
2. Tracker never includes true leechers, i.e., vulnerable peers or peer without the complete patch, in the peer list. Therefore, details of vulnerable peers are not published and no peer will be able to discover who are vulnerable peers or connect to a vulnerable peer.
3. When sending the peer list to true leechers, tracker only includes pseudo-downloaders in the peer list. Therefore, vulnerable peers only connect with randomly selected pseudo-downloaders. As these pseudo-downloaders are most probably benevolent, vulnerable peers are rarely exposed to attackers.
4. When sending the peer list to pseudo-downloaders, tracker sends a mix of pseudo-downloaders and true seeders. Therefore, a pseudo-downloader will connect with true seeders and other pseudo-downloaders as well. As a result, pseudo-downloaders cannot precisely determine the true seeders. Furthermore, as a pseudo-downloader receives connections from both vulnerable peers and other pseudo-downloaders, a pseudo-downloader cannot determine who are vulnerable peers precisely. Thus a pseudo-downloader cannot determine who are true peers in the swarm precisely. Therefore, patch interest of true peers is concealed from pseudo-downloaders as well.

5. When a seeder makes an announce request, tracker does not include any peer in the list. This is because, seeders have the complete patch and they do not need to initiate connections anymore. Therefore, a seeder will not be able to discover any other true-peer in the swarm.

As a result of above restrictions kept in the overlay, a true-peer will never be able to discover or will never get exposed to another true-peer participating in the overlay. Furthermore, even randomly selected pseudo-downloaders cannot determine who are true downloaders. Thus, the privacy of patch sharing is preserved, while preventing exposure of vulnerable peers to probable attackers.

3.5 Suggested Modifications to BitTorrent Protocol

In the proposed peer-assisted patch distribution system, three extensions are introduced to the BitTorrent protocol. Even though these extensions are mentioned in previous sections, they are described here as well, to be more detailed.

1. An additional field called *isPseudoDownloader* is added to tracker request message. This field helps tracker to identify whether a true peer or pseudo-downloader is making the announce request. When pseudo-downloaders send the tracker request, they set this field to true and when true-downloaders send tracker request they set this field to false.
2. An additional message exchange is introduced between peers, such that a peer is able to request and obtain the meta-info file or the torrent file from another peer. When a peer wants to obtain the torrent file for a particular info-hash it sends a Torrent-File-Request message to another peer who already have the torrent file. Torrent-File-Request message contains the info-hash of the required torrent file. When a peer receives a Torrent-File-Request message, it replies with the Torrent-File-Response message which contains the requested torrent file. Another alternative is to download the torrent file similar to any other files in the P2P overlay. However, this would introduce more overhead and latency as the typical file download process has to be followed.
3. The default behaviour of BitTorrent is to drop the connection when a connection for unknown info-hash is received. However, in the proposed patch

distribution system a peer starts to participate in the particular torrent as a pseudo-downloader if the info-hash is unknown.

3.6 Operational Constraints in Multi-Tracker Environment

As user base grows one tracker might not be able to handle the load. Furthermore, there should be multiple trackers in the proposed patch distribution system to prevent a single point of failure. However, there is a constraint in the proposed peer-assisted patch distribution system when multiple trackers are present in the network.

This constraint is that, one peer should communicate with only a single tracker at any given time instant for its every torrent it downloads. The reason for this is that a tracker might select a peer as a pseudo-downloader for a particular patch while the peer is already sharing the patch as a true-peer with another tracker. For instance if a peer registered with two trackers for two separate torrents in the same time, first tracker might select the peer as a pseudo-downloader for the second torrent, for which the peer is already registered with the second tracker as a true peer. Therefore, if a peer communicates with many trackers at the same time it might get exposed to attackers while sharing patches in the P2P network, because the peer might get selected as a pseudo-downloader while it is vulnerable. However, as every P2P patch download is launched through the P2P client application of the proposed solution, this constraint can be straightforwardly overcome by developing the P2P application in such a way that, at any given time frame it communicates with only a single tracker for all of its patch downloads. Exploration of a more comprehensive solution for this problem is left as future work.

3.7 Summary

This chapter described the design of the proposed peer-assisted patch distribution system. As the P2P file sharing protocol in the proposed solution, BitTorrent P2P protocol is used with minor modifications. A set of non-vulnerable peers which are called pseudo-downloaders, are used to preserve privacy of patch sharing peers. These pseudo-downloaders, act as intermediate hops between patch sharing peers. Pseudo-downloaders are selected from peers who do not share the particular patch in the P2P network and pseudo-downloader selection is done by tracker. To preserve the privacy

of patch sharing peers, tracker regulates connections between peers by responding with different types of peers in the peer list for different types of peers. For a true-leecher tracker only responds with pseudo-downloaders, for a pseudo-downloader tracker responds with mix of true-seeders and pseudo-downloaders and for a seeder tracker does not include any peer in the peer list. Tracker keeps a pool of pseudo-downloaders and size of the pseudo-downloader pool is a multiple of currently active true-leecher count. As tracker might select vulnerable peers as pseudo-downloaders, before participating in a swarm as a pseudo-downloader, a peer first checks whether the particular patch is applicable for it. If the patch is applicable to itself, instead of participating in the swarm as a pseudo-downloader, the peer participates in the swarm as the true-downloader. As soon as a selected pseudo-downloader started to participate in the swarm as a true-peer, tracker removes that peer from the pseudo-downloader pool. Therefore, the time window a vulnerable or an infected peer is kept in the pseudo-downloader pool is minimal. Next chapter presents and analyses the performance of the proposed peer-assisted patch distribution system.

Chapter 4

Performance Evaluation

To demonstrate the effectiveness of the proposed patch distribution system we performed an extensive set of simulation-based experiments. We simulated the proposed solution using OMNet++ [13] simulation framework. An existing BitTorrent module [14] for OMNet++, which was described in Section 2.6 is used to simulate a BitTorrent P2P network.

This chapter presents the evaluation methodology of the proposed patch distribution system and the results of the evaluation. The chapter is organised as follows. Section 4.1 describes the extensions made on the OMNet++ BitTorrent module. Section 4.2 describes about the experimental setup and the parameters that are used in simulations. Performance metrics are described in Section 4.3. Next, few subsections describe the experiments that are performed to validate the effectiveness of the proposed patch distribution system.

4.1 Extensions Made to OMNet++ BitTorrent Module

We extended the OMNET++ BitTorrent module to accommodate our solution. For this purpose several existing modules were extended. Names of these extended module and classes are appended with "SPD" (Secure Patch Dissemination) in order to distinguish them from original names.

A P2P node in our simulation model is represented by BTHostSPD module. The BTHostSPD module is extended from original BTHost module in OMNet++ BitTorrent simulation module. BTHostSPD module contains several sub-modules which are; Tracker-Client-SPD, Peer-Wire-SPD, Vulnerable-Software and Worm modules as shown in Figure 4.1. Tracker-Client-SPD module handles the communication with the tracker. Tracker-Client-SPD module is extended from original Tracker-Client module and this extension is made to accommodate the modification of the tracker request message. Peer-Wire-SPD module handles the peer–wire protocol which is the protocol between peers. Peer-Wire-SPD module is extended from the original Peer-Wire module. Peer-Wire-SPD module handles several functionalities; i) torrent file exchange between peers which happens when a pseudo-downloader do not have the torrent-file, ii) notify the patch download completion to the Vulnerable-Software module and iii) implement functionalities of the pseudo-downloader. Peer-Wire-SPD module communicates with the Tracker-Client-SPD

module to initiate the communication with the tracker and then to obtain the tracker response message which contains the peer list. Tracker-Client-SPD module and Peer-Wire-SPD module collectively represent the P2P application installed on a P2P node.

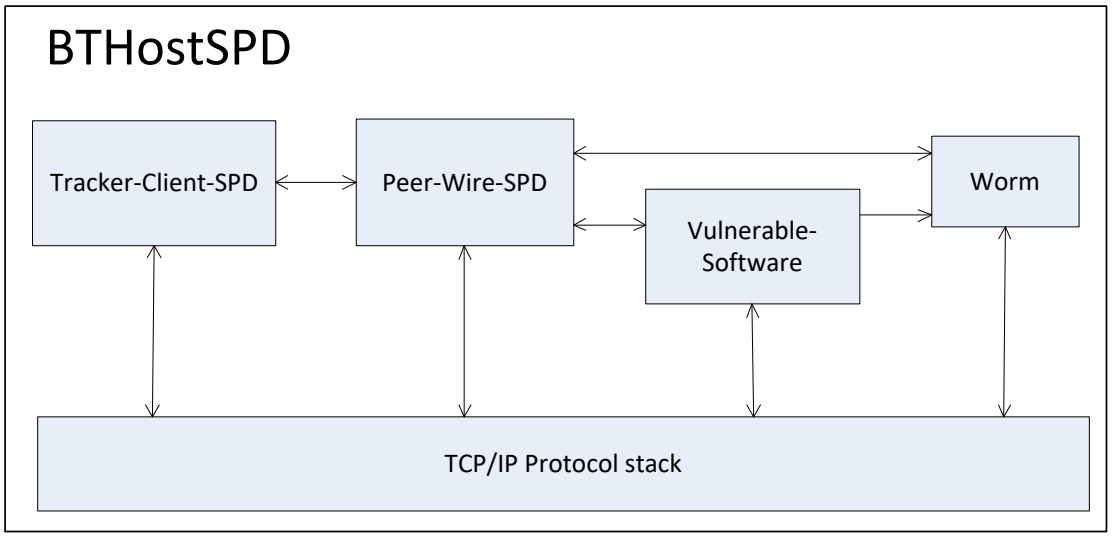

Figure 4.1: BTHostSPD OMNet++ simulation module.

Vulnerable software installed on a P2P node is represented by the Vulnerable-Software module. In order to represent the vulnerability, Vulnerable-Software module opens a TCP port to which attacker can connect and exploit the vulnerability by sending an attack message. Once the patch download is complete, Peer-Wire-SPD module notifies the completion to the Vulnerable-Software module. Subsequently, Vulnerable-Software module closes the TCP port to represent the fix of the vulnerability. Therefore, attackers cannot exploit the node once the patch download is complete. If some node is not vulnerable, Vulnerable-Software module is not activated and it does not open the TCP port, and as such, an attacker cannot exploit a non-vulnerable node.

If Vulnerable-Software module receives an attack message, it activates the Worm module. Worm module represents the topological-worm and a node is treated as infected if the Worm module of the node is active. When the Worm module is active, it obtains the address of every node connected to its occupying node from Peer-Wire-SPD module. Then Worm module connects to the TCP port of the Vulnerable-Software module of each of these remote nodes. If the remote node is vulnerable this connection will be successfully established and will be failed otherwise. If connection

is successful, Worm sends the attack message to the Vulnerable-Software module which simulates the exploitation of the vulnerability. Furthermore, Worm module is always notified when a new connection is established such that Worm module is always aware of the established connections of its parent node. This behaviour of the Worm module closely simulates the epidemic distribution of the topological worm.

4.2 Experimental Setup

In experiments, we assume all clients obtain the patch from the P2P network, except one node, which acts as the initial seeder. By assuming this fact we perform our experiments for the worst-case node exposure scenario, because every node has to use the P2P network in order to obtain the patch instead of using the central server. Therefore, there is no central server present in the simulation network. There is only single tracker present in the network.

There are two types of peers in the simulation network. First type is true peers, who are interested about the patch which we distribute though the P2P simulation network. We denote this patch as the primary patch as all measurements are taken with respect to this patch. Second type is the peers who are not interested in the primary patch but registered with the tracker for another patch file. These peers will be selected as pseudo-downloaders by the tracker. But these pseudo-downloaders might also become interested about the primary patch if they found out that they require the primary patch. In such a scenario their role will be switched to true-peers.

True-peers start to download the patch as soon as they have started. Therefore, these nodes should be created and started to reflect a real world node arrival process in a BitTorrent network. We use arrival process proposed in [29]. In [29] authors propose a peer arrival process based on a tracker trace analysis study. Authors propose peer arrival rate in BitTorrent follows an exponentially decreasing rule with time *t*,

$$\lambda(\mathrm{t}) = \lambda_0 \, e^{-t/\tau} \tag{4.1}$$

where λ_o is the initial peer arrival rate and τ is the file popularity. In [29] it is shown that $N = \lambda_o \tau$ where N is the total peer population size. Thus, the only configurable parameter of this arrival process is initial peer arrival rate. We use initial arrival rate of 0.001666 peers per second for both true-peers and pseudo-downloaders as in [14]. We

note that the value of this parameter is irrelevant to our final results as we compare our results with a typical BitTorrent download and where the parameter is same for both the cases. Furthermore, initially only the tracker and the initial seeder are present in the network and every other peer joins the network according to the above stated arrival process.

As node leaving pattern in a BitTorrent swarm is highly unpredictable [29], we use a uniformly random node leaving pattern in our simulations. We configured true-downloaders to leave the swarm after a random time which is no more than 100 seconds once their download is complete. Pseudo-downloaders leave the network after some random time which is no more than 2,000 seconds since they started to participate in the swarm. Initial seeder stays in the swarm until the end of the simulation.

Unless specified otherwise, we use 1,000 true-peers, one initial seeder and some number of pseudo-downloaders depending on the experiment. Even though we run most of experiments with 1,000 true downloaders, as we show in Section 4.7 performance of our solution does not degrade as we scale the number of true downloaders. Unless specifically stated otherwise, it is assumed that all pseudo-downloaders are not vulnerable and there are no infected peers in pseudo-downloader population. It is assumed that 5% of vulnerable nodes are initially infected. As number of initial infections is unpredictable we also run simulations for different initial infected percentages.

As the underlying network OverSim's InetUnderlayNetwork is used with 10 backbone routers and 100 access routers. These routers are connected to each other with 10 Gbps links. Each node is connected to one of these access routers with an access link. Based on [5] we set upload and download capacity of this access link as 5.56 Mbps and 13.69 Mbps, respectively. Again we note that network capacities of these links are irrelevant as typical BitTorrent download and our solution runs on the same network. Placement of nodes in the network is random. A true-downloader establishes connections with up to 30 peers. When participating in a swarm as a pseudo-downloader, a peer might not allocate its resources to the swarm as a true-peer because it plays the role of a helper. In our experiments, in order to take this fact in to account, a pseudo-downloader establishes connections with only up to 15 peers.

In practice, most patches are usually of few hundred kilobytes [4]. Therefore, other than in the experiment that runs simulations for various file sizes, we use file size as 1MB. Typically, we set pooling factor at tracker as five, i.e., tracker keeps the pseudo-downloader pool size as five times of currently active true-leecher count. In order to examine the effect of pooling factor, we run experiments for different pooling factors as well. We run every experiment three times and take the average of the results of three cases. When the experiment was executed with 500 pseudo-downloaders and 1,000 true-peers, for ten times average download time was 12.161 seconds. When it was executed for three times average download time was 12.533 seconds. As the average result of three runs is very close to the average result of ten runs, we selected to run three cases for each experiment. We conclude a simulation after all true-peers downloaded the patch file. Furthermore, the final infected node count indicated in results, only includes infections which occurred during sharing the patch and excludes the initial infection count unless specifically stated otherwise.

Table 4.1: Simulation parameters.

Parameter	**Value**
Initial seeder count	1
Number of trackers	1
True peer count	1,000
Pseudo-downloader count	Depends on the experiment (default value is 500).
λ_o (initial peer arrival rate)	0.001666
Maximum value for true-peer leave time	100 seconds, after download is complete
Maximum value for pseudo-downloader leave time	2,000 seconds, after starting to participate in the swarm
Initial seeder leave time	Initial seeder stays until the end of the simulation
Initial infected percentage	5% of total vulnerable population
Backbone router count	10
Access router count	100
Bandwidth of links between routers	10 Gbps
Peer upload capacity	5.56 Mbps
Peer download capacity	13.69 Mbps
Maximum connection count for a true-peer	30
Maximum connection count for a pseudo-downloader	15
File size	1MB
Pooling factor at tracker	5

4.3 Performance Metrics

The effectiveness of the proposed patch distribution can be quantified by two factors:

1. Final infection count – How many nodes additionally become infected at the end of a patch distribution session for a given number of initial infections.
2. Average download time – What is the growth of the average download time when the proposed patch distribution system is used with respect to the typical BitTorrent average download time.

Our evaluation had several goals. First, we wanted to investigate the effect on the download time and final infection count as the pseudo-downloader population size varies. Second, we wanted to find an optimal value for the pooling factor at the tracker. Third, we wanted to investigate the effect of vulnerable pseudo-downloaders. Finally, we wanted to investigate the effectiveness of the solution for different patch file sizes.

We compare the performance of our solution with the original BitTorrent protocol used for patch download (similar to downloading any other file) by measuring the total final infection count and the average download time. Therefore, throughout our experiments, we compare our results with a typical BitTorrent patch download where none of our solutions are used. We denote this typical BitTorrent patch download as the base-line scenario for our experiments. Only existing work similar to this research is [7]. However, we did not compare the results of the proposed solution with [7], as assumptions made in [7] are different from our assumptions. For example, in [7] authors considered that privacy of patch providing peers, i.e., seeders, is not important, whereas in our work we preserve privacy of seeders as well. Moreover, in [7] authors did not publish their solutions' impact on download time.

Patch download through a typical BitTorrent network without any of our methods in effect is considered as the base-line for our results. In a typical BitTorrent network with one initial seeder, if 1,000 downloaders downloaded the patch, mean download time results in 11.46 seconds and total of 923 peers (92.3%) additionally become infected at the end of the simulation when 5% of peers are infected at the beginning of the simulation. Therefore, the proposed solution would be effective, if it is capable of reducing the infection percentage significantly than 92.3% while achieving a download time comparable with 11.46 seconds.

4.4 Effect of Pseudo-Downloaders Population

Effectiveness of our methods depends on the number of available pseudo-downloaders while true peers download the file. As there is no correlation between the number of peers who are eligible to be pseudo-downloaders and the number of true downloaders, we run our simulations for different number of pseudo-downloaders to investigate the effect of available pseudo-downloader population size on download time. In this experiment we assume that there are no pseudo-downloaders available at the beginning and all pseudo-downloaders arrive during the simulation. Similarly in this experiment we assume that, none of the pseudo-downloaders are vulnerable or infected. Furthermore, we assume that 5% of true downloaders are initially infected. The experiment carried out with 1,000 true downloaders and one initial seeder.

Figure 4.2 shows the effect of pseudo-downloader population size on total infection count at the end of a patch distribution session. This infection count excludes the initially infected node count of 5% (i.e., 50). While a typical BitTorrent network results in 923 infected nodes, our solution produces zero infections at the end of the patch distribution in all cases provided that all pseudo-downloaders are not vulnerable or not infected. This is because, in our solution if pseudo-downloaders are not vulnerable or not infected, a vulnerable true peer never interacts with an infected peer.

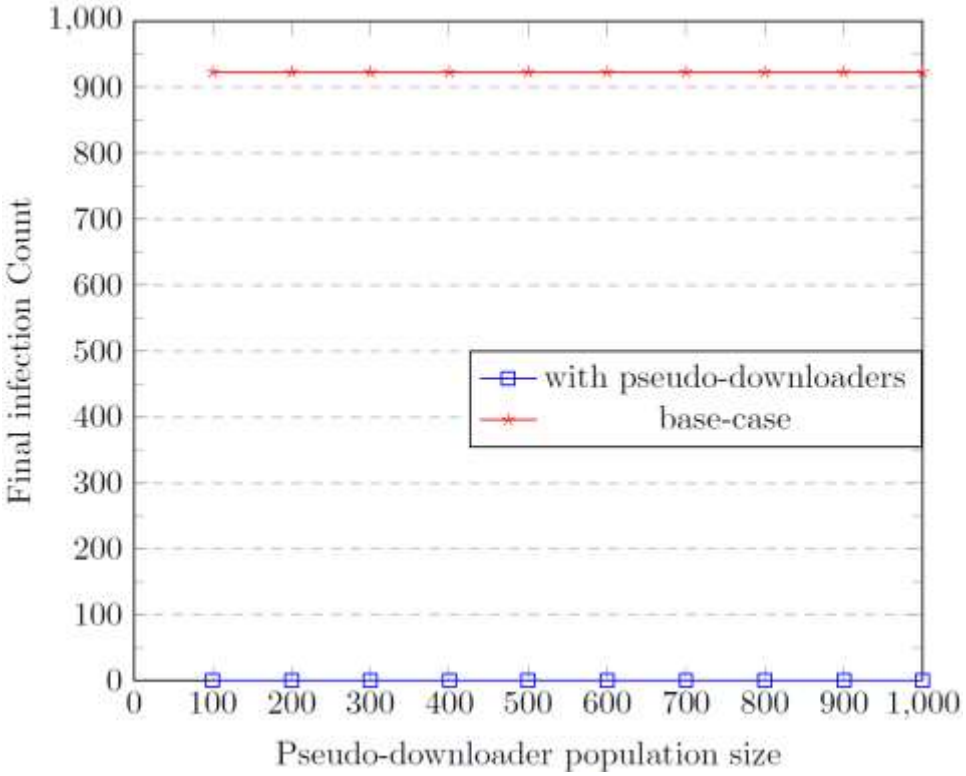

Figure 4.2: Effect of number of pseudo-downloaders on final infection count.

Figure 4.3 shows the effect of pseudo-downloader population size on the download time. For each pseudo-downloader population size, download time of our solution is higher than the typical BitTorrent download time because there is an additional layer of pseudo-downloaders between seeders and leechers in the P2P overlay. When pseudo-downloader population size is small download time tend to be high. This is because when collectively considered there are not enough resources in pseudo-downloader population to provide service to all true downloaders. When available pseudo-downloader population exceeds 50% with respect to true peer population (i.e., 500 pseudo-downloaders) download time becomes stable. This is because after 500 pseudo-downloaders there is sufficient number of pseudo-downloaders to service true-peers and after that increasing the pseudo-downloader population does not make any notable difference. With 500 pseudo-downloaders final download time is 12.55 seconds, which is 9.51% increase from the base-line case of 11.46 seconds. Lowest download time, 12.1 seconds observed at 1,000 pseudo-downloaders which is 5.58% increase with respect to the base-line case.

Therefore, we note that our solution is quite effective when none of the pseudo-downloaders are vulnerable or infected because our solution produces zero additional infections and when available pseudo-downloader percentage is at least 50% of true

peer population size download time only increase by 9.5%. Furthermore, we note that download time can be further reduced even below the base-line case by increasing the pooling factor at the tracker as demonstrated in Section 4.5.

50% is the minimum pseudo-downloader percentage that produces a download time which is competitive with the base-case scenario. Therefore, in experiments where we experiment with only a single pseudo-downloader population size, we use available pseudo-downloader population size as 50% of true-downloaders (i.e., 500 pseudo-downloaders).

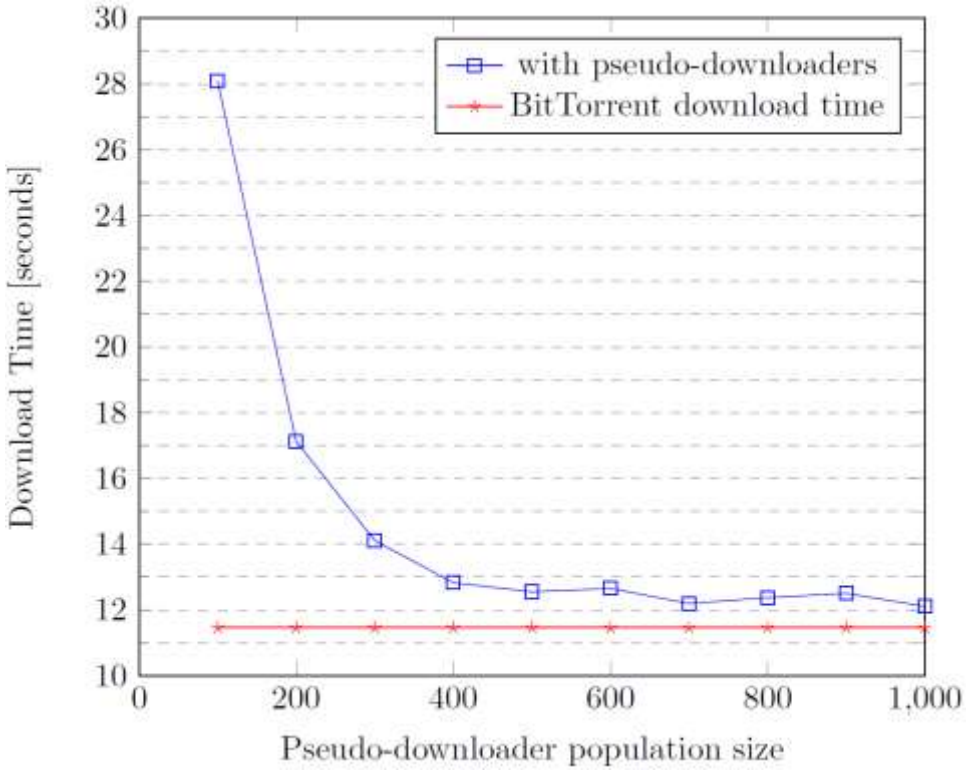

Figure 4.3: Effect of number of pseudo-downloaders on download time.

4.5 Effect of Pseudo-Downloader Pooling Factor

Irrespective of the available pseudo-downloader population size, pseudo-downloader pool size acts as the upper bound for the number of pseudo-downloaders that participate in the swarm at any given instant. In our solution tracker maintains the pseudo-downloader pool size as the multiplication of the pseudo-downloader pooling factor and the current true-leecher count. Therefore, selecting a proper pooling factor at the tracker has a crucial effect on the performance of our solution. In this experiment we investigate the effect of the pseudo-downloader pooling factor on the

download time. We use all the default parameters specified in Table 4.1, in this experiment as well.

Figure 4.4 shows the effect of pseudo-downloader pooling factor. Every curve for different pooling-factors follows the same shape. It is clear from Figure 4.4 that download time decreases with increasing pooling-factor. When pooling-factor is ten or above, download time is even better than the base-line case. However, we note that using higher pooling-factor means that higher number of pseudo-downloaders participates in the swarm and more resources of peers are allocated into a swarm for which they are not interested. Therefore, it is not recommended to increase the pooling-factor arbitrarily. As pooling factor of five is the minimum value which produces a download time close to the base case, we recommend using pooling factor as five. Furthermore, in next set of experiments we use pooling factor as five.

Infection count for each pooling factor was zero because pseudo-downloaders are not vulnerable or infected, and vulnerable peers only interact with the pseudo-downloaders.

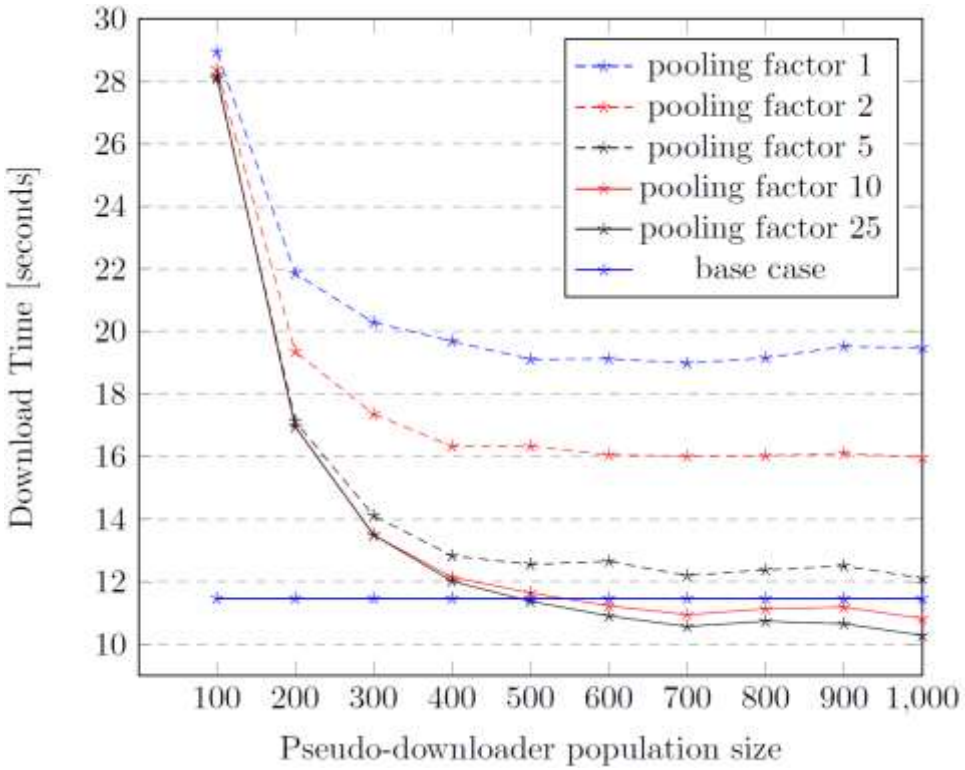

Figure 4.4: Effect of pseudo-downloader pooling factor on download time.

4.6 Swarm Dynamics

In this experiment we investigate the dynamics of the swarm as peers download the patch file. We are particularly interested in how peers arrive to the swarm, complete their download and leave the swarm over time. In addition to that, we investigate how average download time varies over time. The experiment is performed with 1,000 true peers and 500 pseudo-downloaders with none of them are vulnerable or infected while 5% of true-peers are initially infected. All other parameters remain same as specified in Section 4.2.

Figure 4.5 shows variation of different node counts over time. In Figure 4.5, cumulative download completed node count curve closely follow the cumulative started node count curve except at the beginning. This indicates that peers complete their download in a steady rate which is close to the peer arrival rate and this is a clear sign to indicate that swarm progress smoothly. Furthermore, after 100 seconds, active node count and seeder node count became stable and cumulative download completed node count curve started to follow the cumulative started node count curve. This is because, even though there is less number of seeders is initially present in the swarm, after 100 seconds, total seeder count is sufficient for peers to complete their download without any difficulty. Therefore, this point could be thought as the point where swarm enters its steady state and once swarm entered into the steady state, peers complete their download in a rate which is similar to the peer arrival rate.

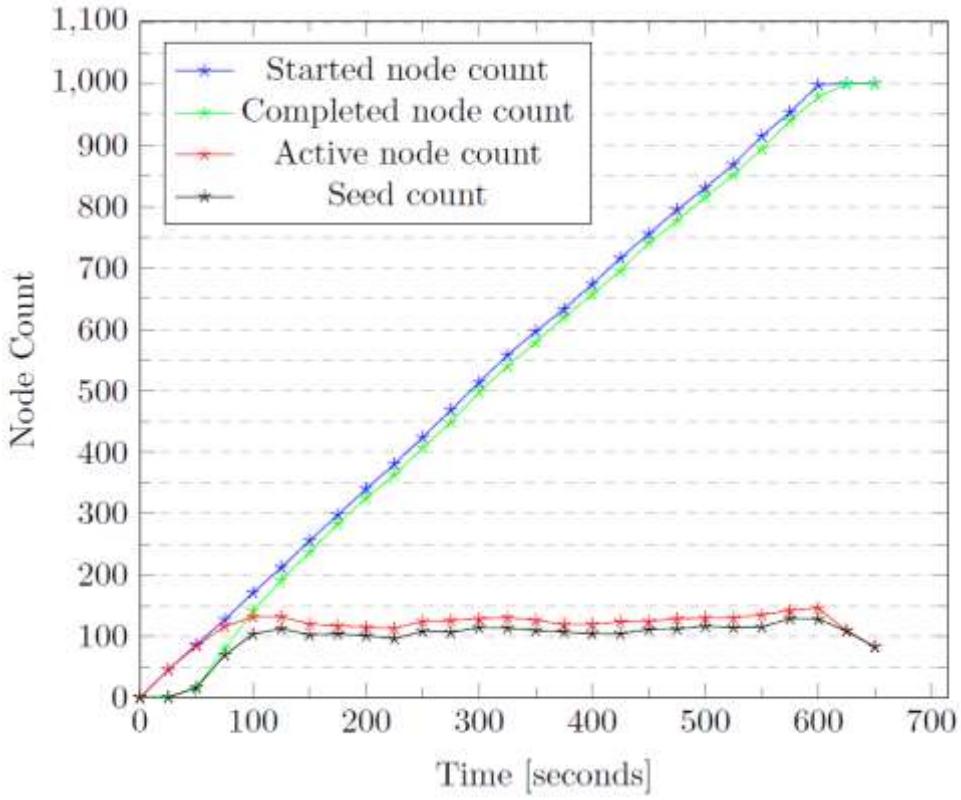

Figure 4.5: Variation of number of nodes over time.

Figure 4.6 shows the variation of average download time over time. Average download time closely follows the negative exponential function except when time is less than 100 seconds. As stated previously, 100 seconds is the point where the swarm enters in to the steady state. Therefore, it can be safely stated that average download time varies according to the negative exponential function once the swarm entered in to its steady state. When the swarm is started there is only one piece provider which is the initial seeder. Then as peers complete their download, number of piece providers gradually increases as shown by the completed node count curve in Figure 4.5. Therefore, peers who lately join the swarm enjoy more piece availability in the swarm and complete their download quickly. This is the reason for the gradual decrease of average download time in the swarm.

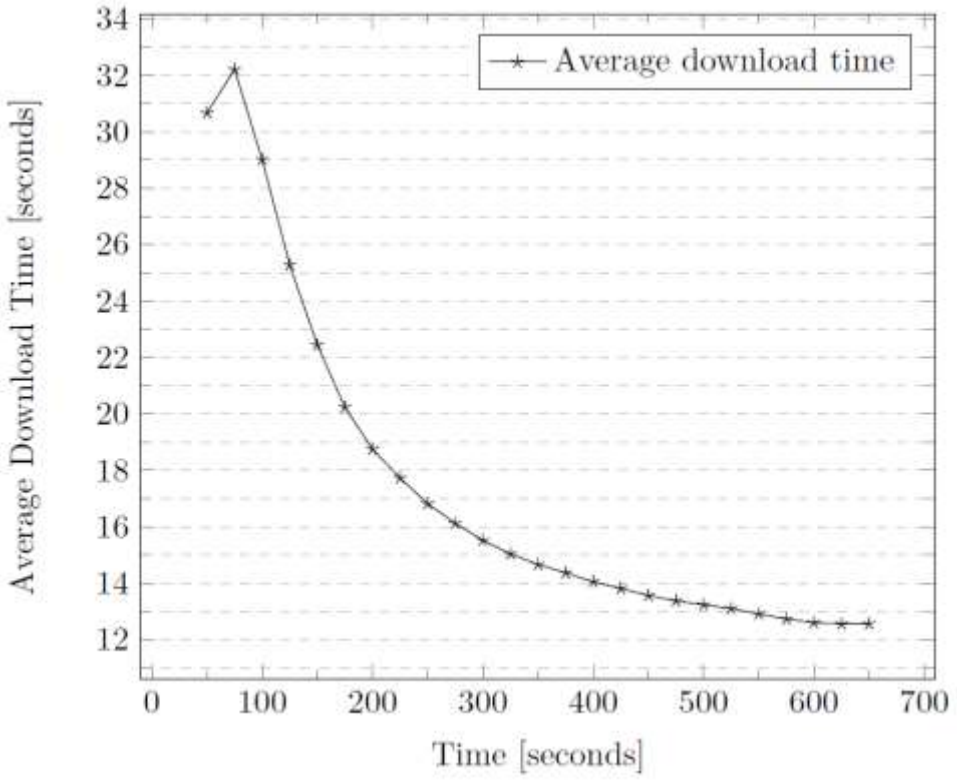

Figure 4.6: Variation of average download time over time.

4.7 Scalability of Proposed System

As we perform our experiments only for 1,000 peers in most cases, in this experiment, we investigate the performance of our system for a large number of peers. This experiment is performed with 5,000 true-downloaders while the size of pseudo-downloader population varies from 500 to 5,000 in 500 steps. We use default values for all other parameters as specified in Table 4.1.

As the base case for this experiment we simulated a BitTorrent network with 5,000 peers with none of our solutions applied. In a typical BitTorrent network with 5,000 peers, download time resulted in 21.24 seconds and total additional infection count was 4,604 nodes which is 92.08% of the total peer population. With 1,000 peers, in typical BitTorrent network, download time was 11.46 seconds and infection percentage was 92.3%. Therefore, even in the base case download time increased with the true-peer population size and the reason for this as follows. In both cases, where the peer population size is 1,000 and 5,000, we used the same initial peer arrival rate. However, in the peer arrival process we used, $N = \lambda_o \tau$ [29] (see Equation 4.1). Therefore, as number of peers N increases, file popularity increases and peer arrival

rate attenuates slowly over time. Consequently, in the case of 5,000 peers, peers arrive to the swarm more rapidly and there are more peers present in the network than the 1,000 peers case for any given time instant. However, only a single seeder is present initially in the network for both cases, and a seeder accepts only a limited number of concurrent connections. Therefore, in 5,000 case, many peers experience scarcity of some pieces due to the limited number of seeders in the network. Therefore, peers complete their download slowly and average download time increases.

Figure 4.7 shows the variation of download time with the pseudo-downloader population size for 5,000 true-peers. Download time curve has the same shape of the case where true-peer population size is 1,000 which we described in Section 4.4. However in this experiment, even with 1,000 pseudo-downloaders (i.e., 10% out of true-peers) our solution achieved a better download time than the base-case. Note that, as we described in Section 4.5, with 1,000 true-peers to beat the BitTorrent download time, we had to increase the pooling factor more than 10 and pseudo-downloader population size had to be more than 60% of true-peer population size. The reason for this as follows. In base-case with 5,000 peers, download time increased due to the limited number of seeders and the increase of hop count from most leechers to a seeder. But in our solution leechers only connect with pseudo-downloaders and pseudo-downloaders are connected with seeders. Therefore, every leecher has a seeder one hop away and leechers in our solution experience a less piece scarcity than the base-case with 5,000 peers. Therefore, in our solution, growth of average download time with the peer population size is less than the base case and scalability of our solution is better than a typical BitTorrent network.

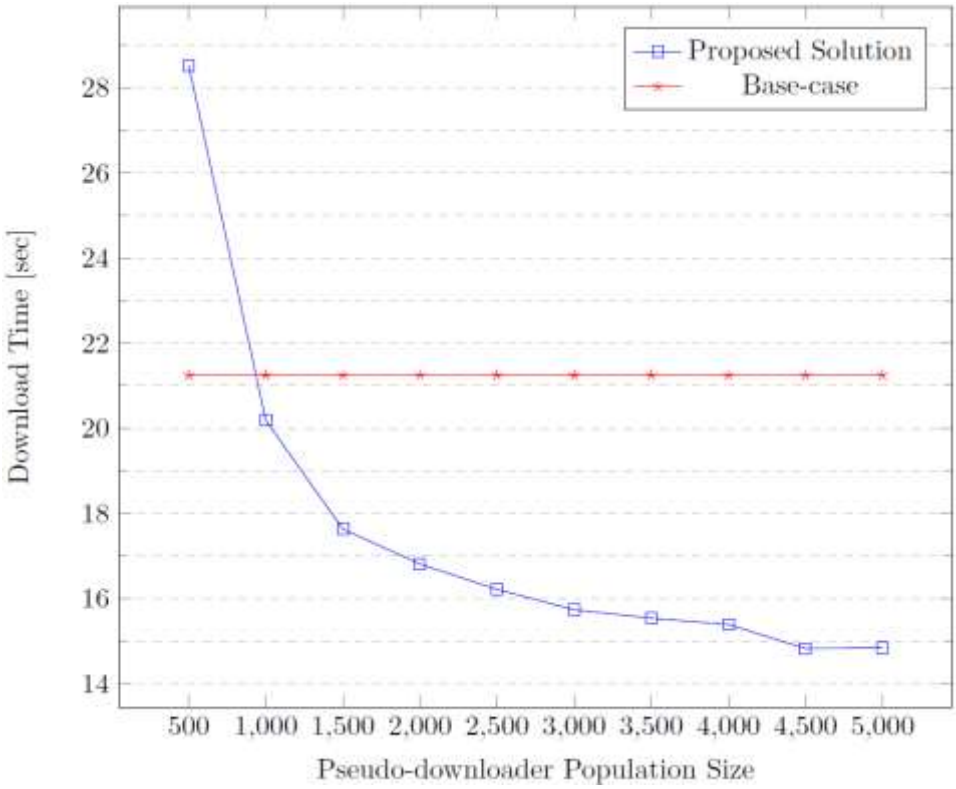

Figure 4.7: Effect of number of pseudo-downloaders on download time for 5,000 true-peers.

Figure 4.8 shows the variation of the infection count with the pseudo-downloader population size for 5,000 true-peers. Similar to the 1,000 true-peers case in Section 4.4, additional infection count is zero in our solution while base case produces 4,604 additional infections.

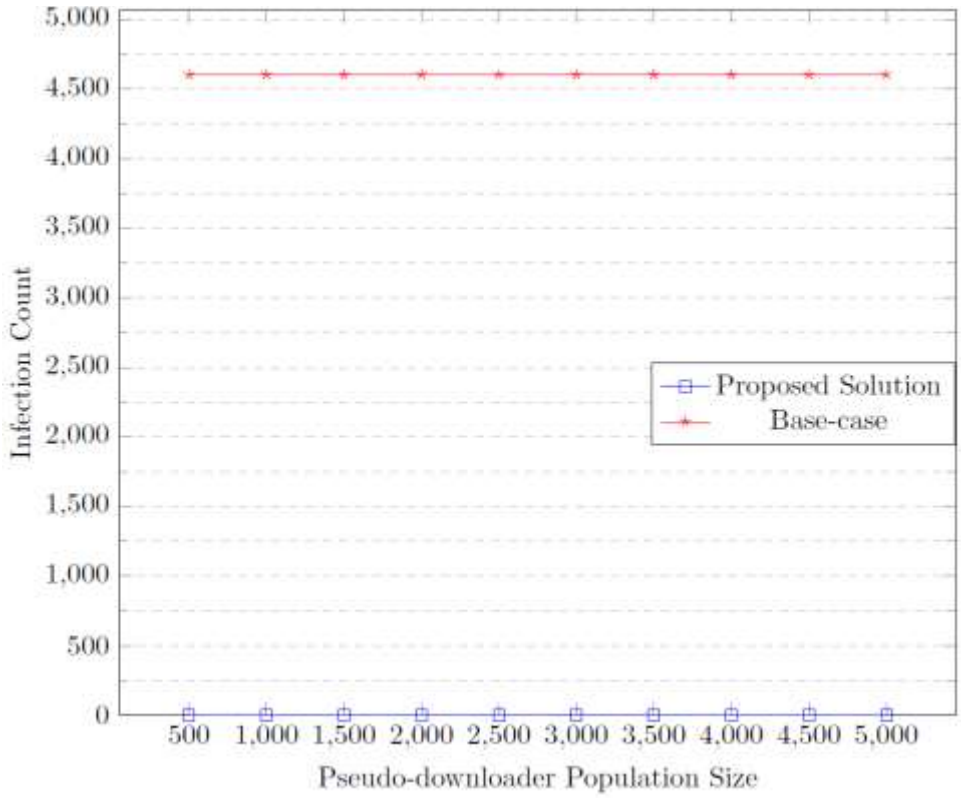

Figure 4.8: Effect of number of pseudo-downloaders on final infection count for 5,000 true-peers.

4.8 Effect of Vulnerable Pseudo-Downloaders

Previous experiments indicate that as long as there are no vulnerable or infected pseudo-downloaders, the proposed patch distribution system behaves quite well producing zero infections. In this experiment we explore the effect of the vulnerable and infected pseudo-downloaders. We use all the default parameters specified in Table 4.1. We vary the vulnerable pseudo-downloader percentage from 0% to 50% in steps of 10%. As we assume 5% of vulnerable peers are initially infected, the same initial infected percentage is assumed for the vulnerable pseudo-downloaders as well. For an instance, when 40% of pseudo-downloaders are vulnerable and total pseudo-downloader population size is 500, there would be 200 vulnerable pseudo-downloaders and out of that 10 pseudo-downloaders are initially infected.

In the proposed patch distribution system, pseudo-downloaders are removed from the pseudo-downloader pool as soon as they started to participate in the swarm to prevent the exposure of vulnerable or infected pseudo-downloaders to the rest of the swarm. However, there is a small time window between the time instants where a peer is

elected as a pseudo-downloader and the peer is removed from the pseudo-downloader pool once it started to participate in the swarm as a true-peer. In this particular time window, a vulnerable pseudo-downloader might get attacked by an infected peer or an infected pseudo-downloader might attack a vulnerable true-peer and this possibility increases as the vulnerable pseudo-downloader percentage increases. Therefore, infection count increases with the vulnerable pseudo-downloader percentage as shown in Figure 4.9 We note that, in Figure 4.9, the infection count is more of a linearly increasing curve than an exponentially increasing curve.

Furthermore, even when 50% of pseudo-downloaders are vulnerable, additional infected node count is 23.33 (fractional part is there because of taking the average) which is 1.86% of total vulnerable peer population. Total vulnerable peer population size in this case is 1,250 peers which includes 1,000 true peers and 250 pseudo-downloaders. Therefore, our solution is very effective even when 50% of pseudo-downloaders are vulnerable. In the base-case total additional infection count was 923, which is 97.15% of total vulnerable peer population. We did not indicate base-case value in the Figure 4.9 because, if base-case was included in the graph, it increases the range of y axis and reduce the focus on the important results.

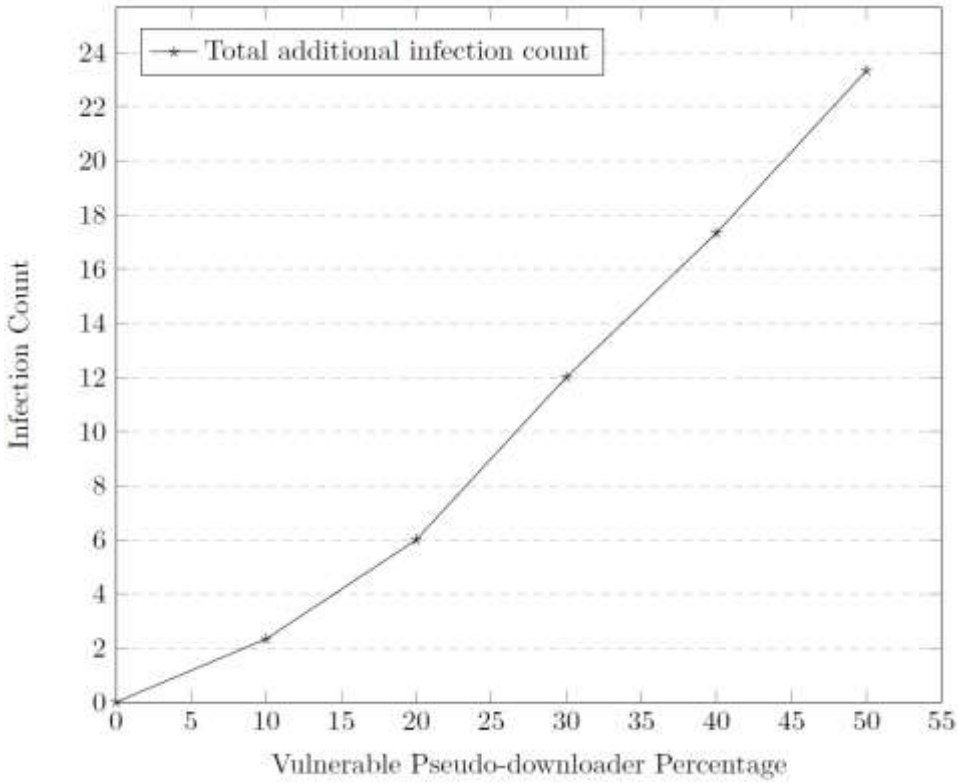

Figure 4.9: Variation of final infection count with increasing vulnerable pseudo-downloader percentage.

Figure 4.10 shows download time linearly increases with vulnerable pseudo-downloader percentage. When vulnerable pseudo-downloader percentage increases, effectively available pseudo-downloader population size decreases. For an instance, if there are 500 peers initially in eligible pseudo-downloader population and 50% of them are vulnerable, only 250 peers will be used as pseudo-downloaders actually. As we noted in Section 4.4, when available number of pseudo-downloaders decreases average download time increases due to the lack of resources in pseudo-downloader population. Therefore, when vulnerable pseudo-downloader percentage increases, effectively available pseudo-downloader population size decreases and as a result average download time increases. Note that, base-case result indicated in the Figure 4.10 is only for comparison. In base-case experiment, there are no pseudo-downloaders or no vulnerable pseudo-downloaders present in the network.

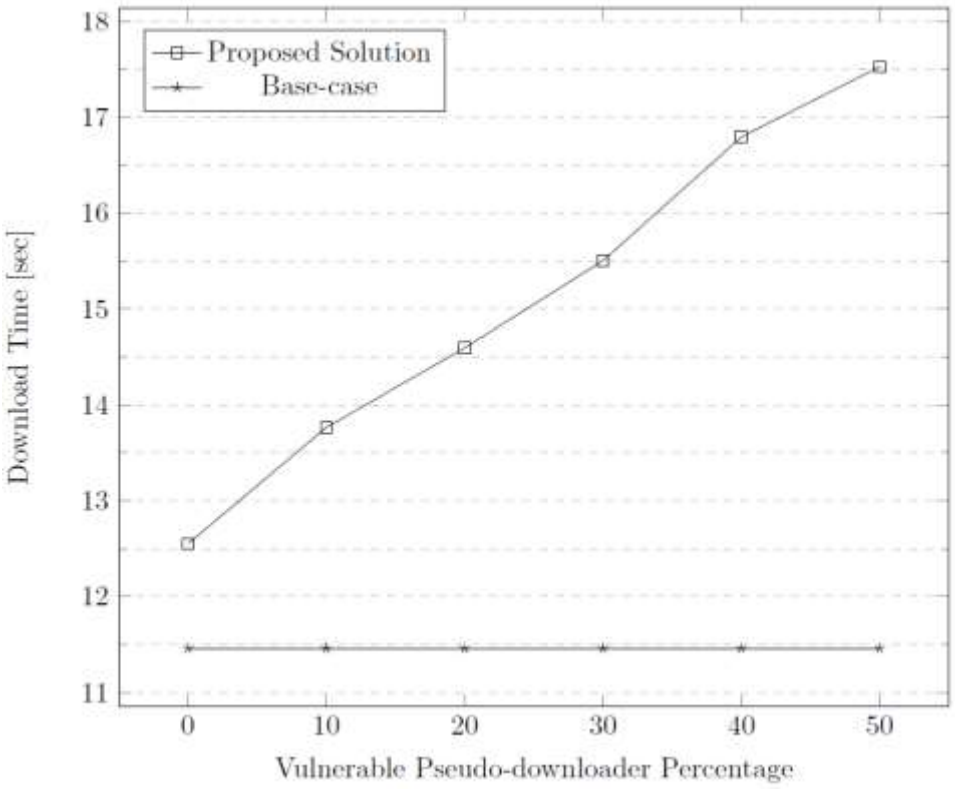

Figure 4.10: Variation of download time over increasing vulnerable pseudo-downloader percentage.

Figure 4.11 shows the vulnerability fixed pseudo-downloader percentage with increasing vulnerable pseudo-downloader percentage. Vulnerability fixed percentage is taken as the ratio of vulnerability fixed pseudo-downloader count to total vulnerable pseudo-downloader count. We note that almost 90% of pseudo-downloaders have fixed their vulnerability. The remaining pseudo-downloaders have not fixed their vulnerability because tracker has never selected those peers into the pseudo-downloader pool and they have never participated in the swarm as pseudo-downloaders. Therefore, we note that selecting a vulnerable peer as a pseudo-downloader also has a positive side, because that particular peer who was not aware about the patch until now becomes aware of the patch and get to download the patch early.

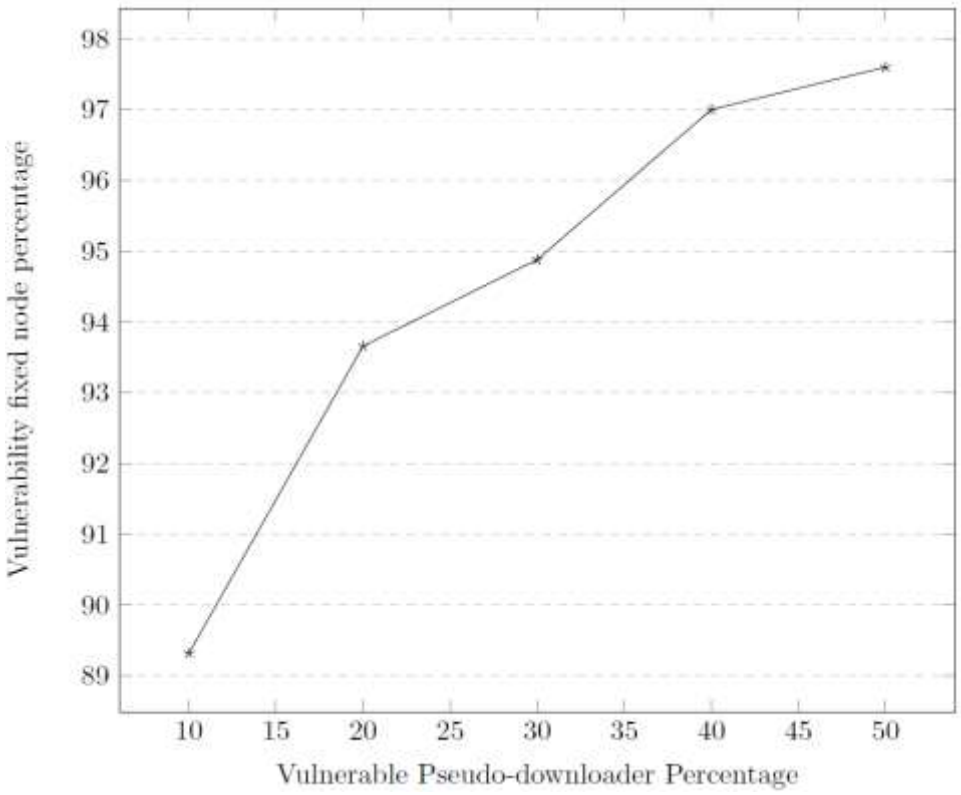

Figure 4.11: Variation of vulnerability fixed pseudo-downloader percentage with increasing vulnerable pseudo-downloader percentage.

4.9 Effect of Initial Infection Count

Next we investigate the effect of initially infected node count on the proposed patch distribution system. For this purpose, we vary the initially infected percentage from 10% - 50% in steps of 10% while keeping all other parameters in default as in Table 4.1. Initially infected percentage defines how many peers are infected out of the total vulnerable population. We record the final additional infection count and present the ratio of the final additional infection count to the total vulnerable population size. We do not take the initially infected node count into account in the presented results. In other words, we present how much percentage of vulnerable peers, those who were not infected initially, became infected at the end of a patch distribution session. We use this scheme to compare between different cases as with each case the total number of vulnerable peers changes. We run experiments for four cases, that are i) base case, ii) with 500 pseudo-downloaders while none of them are vulnerable, iii) with 500 pseudo-downloaders while 10% of them are vulnerable, and iv) with 500 pseudo-

downloaders while 25% of them are vulnerable. When there are vulnerable peers present in the pseudo-downloader population, there will be initially infected peers in the pseudo-downloader population as well. We use the same initially infected percentage for both true-peers and pseudo-downloaders. Therefore, as explained in Section 4.8, if initial infection percentage is 20% and 10% of pseudo-downloaders are vulnerable, in a population of 500 pseudo-downloaders there will be 10 (500 × 0.1 × 0.2) infected pseudo-downloaders initially.

Figure 4.12 shows the effect of initial infection percentage on final additional infection percentage. In a typical BitTorrent network, (i.e., base case) more than 99% of peers became infected at every initial infection percentage. When none of pseudo-downloaders are vulnerable our solution produces zero additional infections because as long as there are no vulnerable or infected pseudo-downloaders present, a vulnerable peer never interacts with an infected peer in our solution. When there are vulnerable pseudo-downloaders present in the network our solution produces some infections. This is because proposed solution removes a vulnerable pseudo-downloader from the pseudo-downloader pool after the peer started to participate in the swarm as a true peer. Therefore, there is a small time gap where a vulnerable pseudo-downloader or an infected pseudo-downloader is exposed to true-peers in the swarm. In this time frame, a vulnerable pseudo-downloader might get attacked or an infected pseudo-downloader might attack a vulnerable true peer. However, as this time gap is relatively small, final infection percentage is very small. For an instance, as shown in Figure 4.12 when 25% of pseudo-downloaders are vulnerable and 50% of vulnerable peers are initially infected our solution only produces only 11.56% of additional infections. Furthermore, when the initial infected percentage increases, there is more possibility for a vulnerable true-peer or vulnerable pseudo-downloader to connect with an already infected peer. Therefore, final infection percentage increases with the initial infected percentage. Similarly, when vulnerable percentage of pseudo-downloaders increases, there will be more vulnerable pseudo-downloaders and more infected pseudo-downloaders in the network. As a result there is more possibility for a vulnerable peer to connect with an infected peer. Therefore, final infection percentage also increases with the vulnerable pseudo-downloader percentage as shown in Figure 4.12.

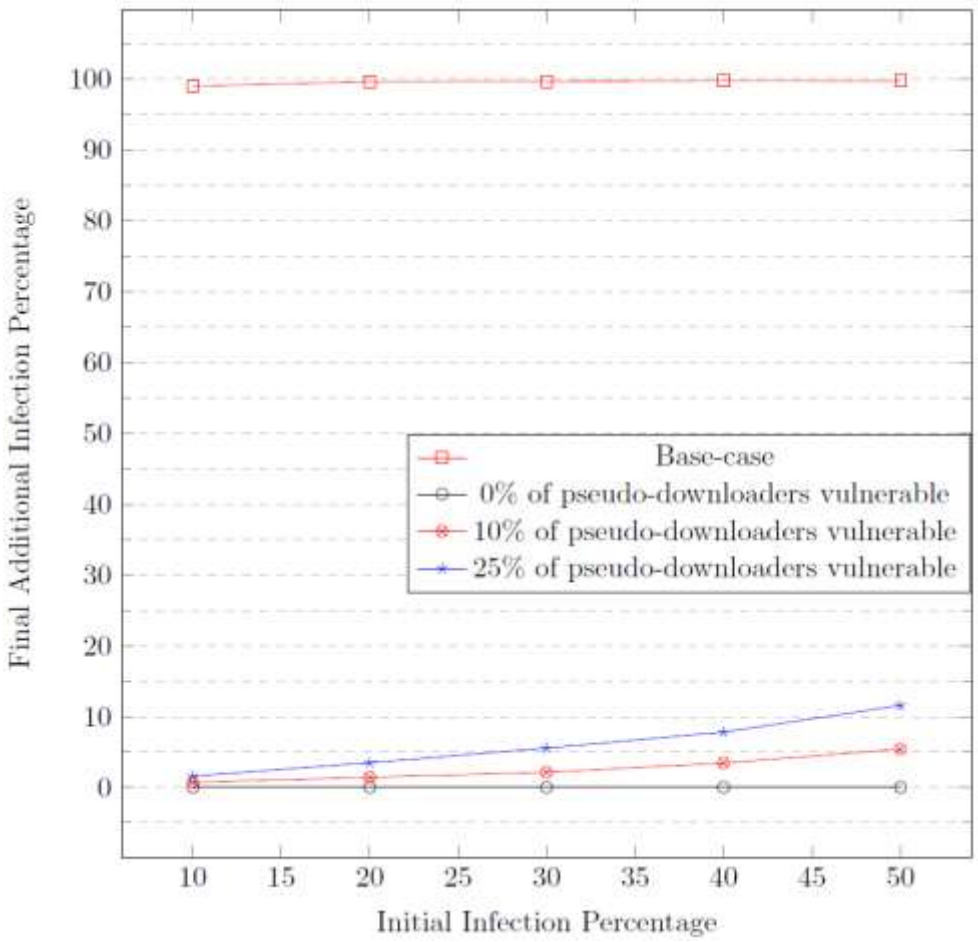

Figure 4.12: Effect of initial infection percentage on final additional infection percentage.

Figure 4.13 shows the effect of initial infection percentage on average download time. In each curve of Figure 4.13, download time decreases with the initial infection percentage. This is because, in our simulations, initially infected true-peers stays in the swarm until the end of the simulation while other true peers leave the swarm after some random time which no more than 100 seconds once their download is complete. This has been done to keep the attackers in the swarm until simulation finishes, because if initially infected nodes (i.e., attackers) also left the swarm early, there would be no more attackers in the network after some time and that would be unrealistic. However, it is also noted that by enforcing attackers to stay in the network, we demonstrate a worst-case behaviour compared to a real-world P2P system with random churn. Therefore, as the initial infected percentage increases, number of seeders who stays in the swarm until simulation finishes will increase and as a result average download time decreases with the initial infection percentage. We note that

infected pseudo-downloaders do not stay in the swarm until the end of the simulation as infected-true peers. Infected pseudo-downloaders leave the swarm after some random time which no more than 100 seconds once their download is complete. In Figure 4.13 download time increase with the vulnerable pseudo-downloader percentage. This is because, as we explained in Section 4.8, as vulnerable pseudo-downloader count increases, effectively available pseudo-downloader population decreases and small effective pseudo-downloader population size produces a higher average download time.

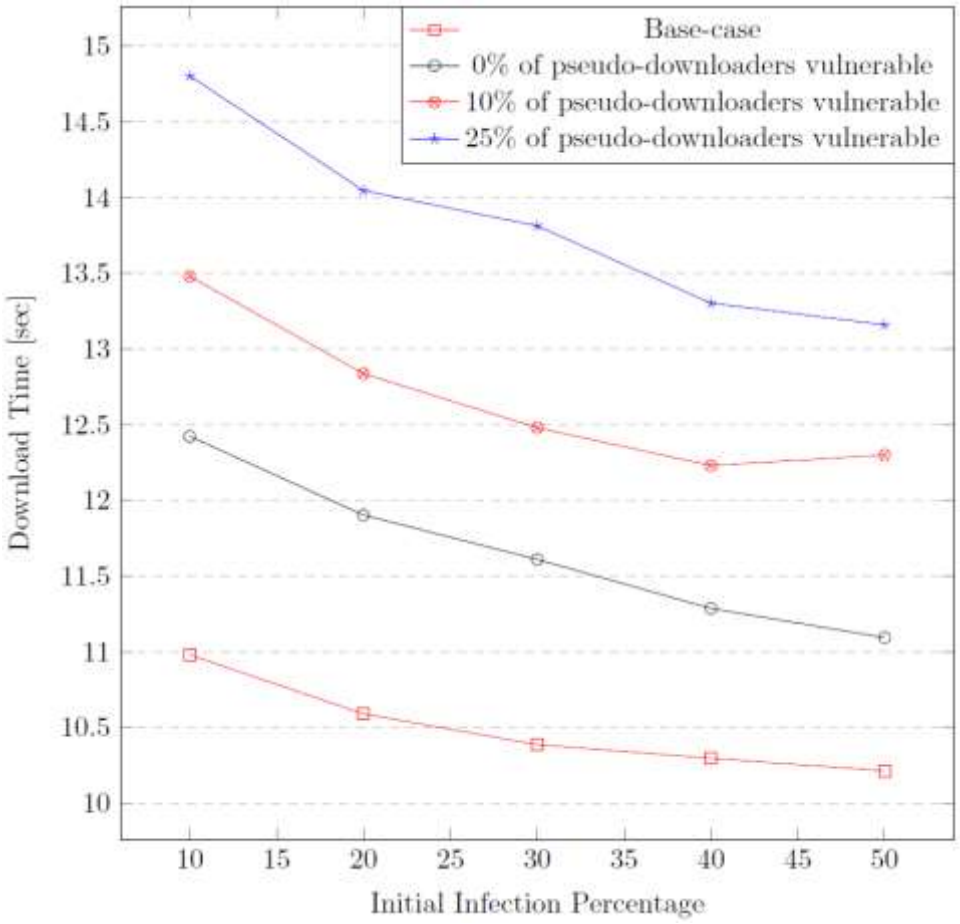

Figure 4.13: Effect of initial infection percentage on average download time.

4.10 Effect of File Size

In order to investigate the effect of file size on our solution we performed several experiments for files sizes, 100KB, 1MB, 5MB, and 25 MB. We considered three scenarios for these file sizes that are base-case, when none of pseudo-downloaders are

vulnerable and when 25% of pseudo-downloaders are vulnerable. All other parameters are set to default values as specified in Section 4.2.

Figure 4.14 shows the effect of file size on download time. Download time penalty of the proposed solution increases with the file size due to the presence of pseudo-downloaders. For example, when none of pseudo-downloaders are vulnerable, for 1MB file download time increases by 9.5% compared to base case, whereas for 5MB file download time increases by 17.39% and for a 25MB file it is 32.72%. However, for a 100KB file download time increases only by 2.1%. Therefore, we note that, download time increase with the files size would not be a major drawback as most patch files are small [4]. When 25% of pseudo-downloaders are vulnerable, download time is higher than when none of the pseudo-downloaders are vulnerable due to the decrease of the effective pseudo-downloader population size. Furthermore, when 25% of pseudo-downloaders are vulnerable, download time growth is similar to the case of none of the pseudo-downloaders are vulnerable.

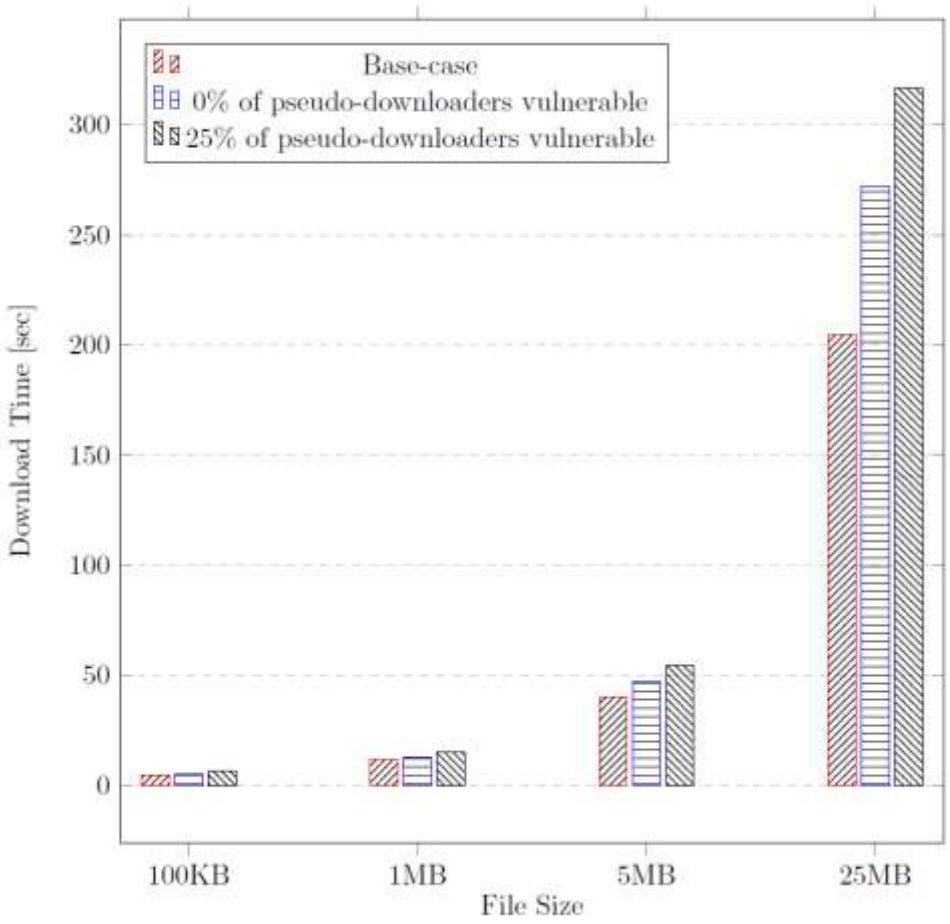

Figure 4.14: Effect of file size on download time.

Figure 4.15 shows the effect of file size on the final additional infection count. Infection count increased with the file size in base case and when 25% of pseudo-downloaders are vulnerable. This is because when file is large, peers had to stay longer in the swarm, which increased their infection probability. However, when none of the pseudo-downloaders are vulnerable additional infection count was zero in all four scenarios. Furthermore, when 25% of pseudo-downloaders are vulnerable and file size is 25MB, additional infection count is 24.33 (count is a fraction number because of taking the average), which is 2.27% of entire vulnerable peer population. Therefore, we state that file size has very little impact on the final infection count in our solution.

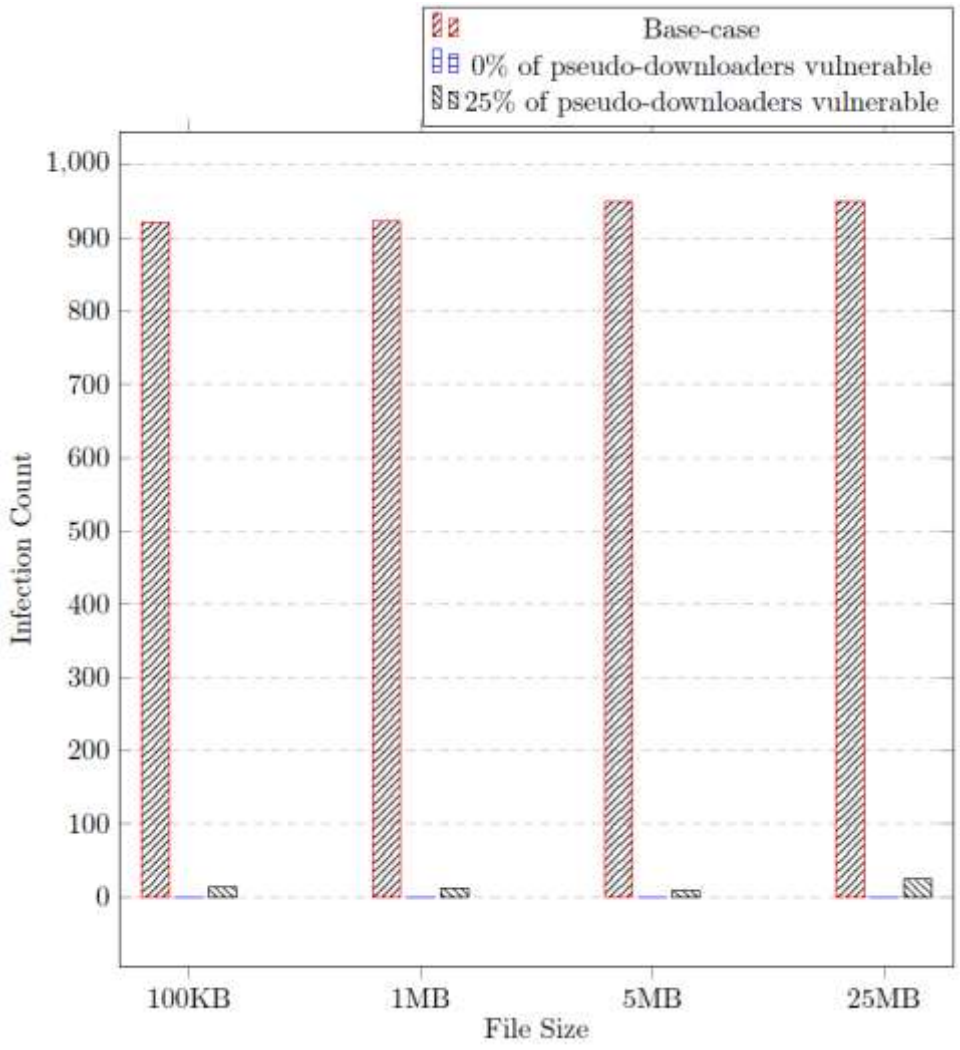

Figure 4.15: Effect of file size on final infection count.

4.11 Summary

This chapter investigated the performance of the proposed peer-assisted patch distribution system. Experiments validated that when none of the pseudo-downloaders are vulnerable, the proposed solution produces zero infections, where as a typical BitTorrent network end up with more than more than 90% of peers are infected, if 5% of peers are infected at the beginning of a patch distribution session. With 1,000 true-peers, proposed solution is capable of achieving a download time which is similar to the typical BitTorrent download time when the pooling factor is five and available pseudo-downloader population size is more than 50% of true-peer population size. Moreover, proposed solution is capable of achieving a better download time than the typical BitTorrent download time, if the pooling factor is increased. Furthermore, when true-peer population size is increased to 5,000, download time of the proposed solution begin to beat the typical BitTorrent download time when pseudo-downloader population size is more than 10% of true-population size for a pooling factor of 5. Therefore, performance of the proposed solution increases with the true-peer population size. Proposed solution performs very well even when there are vulnerable peers present in the pseudo-downloader population. When 50% of pseudo-downloaders are vulnerable, additional infection percentage is only 1.86%. Furthermore, when vulnerable pseudo-downloaders are present in the network, almost 90% of those peers fixed their vulnerability. Although the files size has very little impact on final infection count, average download time of our solution increases with the file size. However, as most patch files are small, this performance loss is not a major drawback.

Chapter 5

Summary

5.1 Conclusions

In peer-assisted patch distribution, patch sharing peers are exposed to each other even though the peer-assisted patch distribution provides a scalable and cost effective alternative to centralized patch distribution. Such peer exposure is a threat to the patch sharing peers' privacy and could be destructive in the face of attackers similar to topological worms. This thesis has reviewed the possibility of eliminating the peer exposure problem in peer-assisted patch distribution by utilizing set of non-vulnerable peers that are called pseudo-downloaders.

Chapter 3 presented the proposed solution to eliminate the peer exposure in peer-assisted patch distribution. A key feature of the proposed patch distribution system is that it is capable of distributing patches from multiple software vendors by utilizing the same infrastructure. The proposed patch distribution system is built upon BitTorrent. BitTorrent protocol was adopted by introducing minor modifications, such that it would not expose patch sharing peers to each other and it would prevent interaction between vulnerable peers and attackers. The proposed patch distribution system consists of a tracker, central patch distribution servers and a P2P client application which is intended to install on client machines. P2P client application provides a service to other software installed in the client machine to download patches from the P2P network. The main attacker which we considered in this work is a topological worm which exploits the vulnerability fixed by the patch that is being distributed. In addition to that, proposed patch distribution system provides the protection against attackers those who try to harvest addresses of patch sharing peers. In the proposed patch distribution system, a set of peers that are called pseudo-downloaders are utilized to avoid direct contact between patch sharing peers. Pseudo-downloaders act as intermediate hops between patch sharing peers while facilitating file transfer between them and as such, peers who actually share the patch are not exposed to each other. The selection criterion for a pseudo-downloader is that they should not be vulnerable. In order to fulfil this fact, tracker selects pseudo-downloaders randomly from peers those who do not share the particular patch in the P2P network. For example, for a Windows patch, peers who share a Linux patch or update are selected as pseudo-downloaders.

In the proposed solution, tracker regulates the connections between peers to avoid the peer exposure. A true-peer which does not have the complete patch is only allowed

connect with pseudo-downloaders. Pseudo-downloaders are only allowed initiate connections to other pseudo-downloaders and seeders. Seeders are not allowed to initiate connections to any other peer. As a result of these enforcements, a true-peer will never exposed to another true-peer and even randomly selected pseudo-downloaders are unable to determine which peers share the patch actually. Tracker enforces these restrictions in the P2P network by including different kind of peers in the peer list when sending the tracker response to a peer. Tracker has to keep special data structures to facilitate this process and these data structures have been discussed in Section 3.4.

In the proposed solution, tracker might select a vulnerable peer as a pseudo-downloader for some patch, if the particular peer is not aware of the release of the particular patch yet. In order to provide the protection against such situations, when a peer receives a request to participate in a swarm as a pseudo-downloader, the peer only participate in the swarm as a pseudo-downloader only if the particular patch is not applicable to itself. If the patch is applicable to itself, instead of participating as a pseudo-downloader, the peer starts to download the patch as a true peer. Complete behaviour of a pseudo-downloader was detailed in Section 3.3.3. If a peer who is selected as a pseudo-downloader is started to participate in the swarm as a true-peer, tracker removes that particular peer from the pseudo-downloader pool. Therefore, the time window, where a vulnerable peer kept in the pseudo-downloader pool is negligible and risk of vulnerable pseudo-downloader become infected is minimum.

Chapter 4 presented the evaluation of the proposed patch distribution system. Evaluation is performed by running simulations on OMNet++ simulation frame work. Experiments validated that when none of the pseudo-downloaders are vulnerable, the proposed solution produces zero infections, where as a typical BitTorrent network end up with more than more than 90% of peers are infected, if 5% of peers are infected at the beginning of a patch distribution session. Proposed solution is capable of achieving a download time which is similar to the typical BitTorrent download time when the pooling factor is five and available pseudo-downloader population size is more than 50% of true-peer population size. Moreover, proposed solution is capable of achieving a better download time than the typical BitTorrent download time, if the pooling factor is increased. Furthermore, performance of the proposed solution increases with the true-peer population size. Experiments that are performed in

Section 4.7 revealed that when true-peer population size is increased to 5,000, download time of the proposed solution begin to beat the typical BitTorrent download time when pseudo-downloader population size is more than 10% of true-population size for a pooling factor of five.

Experiments performed in Section 4.8 revealed that proposed solution performs very well even when there are vulnerable peers present in the pseudo-downloader population. For example, if half of peers of pseudo-downloader population is vulnerable, additional infection count resulted in 23.33 when pseudo-downloader population size is 500 and true peer population size is 1,000. That is, when 50% of pseudo-downloaders are vulnerable, additional infection percentage is only 1.86%. Furthermore, when vulnerable pseudo-downloaders are present in the network, almost 90% of those peers fixed their vulnerability. Performance of our solution degrades with the file size. However, as most patch files are small, this performance loss is not a major drawback.

Thus, the proposed patch distribution is capable of completely preventing infections due to peer exposure in ideal conditions without sacrificing the download time. Even when the conditions are not ideal, i.e., when there are vulnerable peers in pseudo-downloader population, proposed solution produces a negligible infection count. Furthermore, proposed patch distribution system scales better in terms of peer population size than an ordinary BitTorrent network.

One limitation of the proposed approach is that when there are multiple trackers present in the patch distribution network, at any given time instant, a peer should register with only a single tracker for all of its patch downloads. However, as every P2P patch download is launched through the P2P client application of the proposed solution, this constraint can be straightforwardly overcome by developing the P2P application in such a way that, at any given time frame it only registers with a single tracker.

5.2 Future Work

We plan to extend our study on following directions.

5.2.1 Allow Peers to Locally Decide How Much Protection They Desire

In the current implementation tracker attempts to completely protect vulnerable peers by avoiding their interaction with other true-peers in the swarm by sending only pseudo-downloaders in the peer list. If there are not sufficient pseudo-downloaders available in the network download time penalty will be high. In such situations some peers might want to keep their download time not impacted by exposing themselves to other peers to some extent. This can be achieved by introducing additional field into the tracker request which communicates the required pseudo-downloader percentage in the peer list. Then a peer start with 100% required pseudo-downloader percentage and if the download rate is not sufficient it will gradually reduce the required pseudo-downloader percentage. Thus, a peer will get an opportunity to connect with seeders in the swarm if there are not sufficient pseudo-downloaders present in the P2P network. However to protect the privacy of seeders, tracker should not respond to requests which specifies required pseudo-downloaders percentage as zero. Therefore, there will be a minimum limit for the required pseudo-downloaders percentage which is configured at the tracker.

5.2.2 Blacklisting Free Riders

In the current implementation, no incentive is provided for a peer when it participate in a swarm as a pseudo-downloaders. Therefore, when a peer receives a request to participate as a pseudo-downloader in a swarm it might neglect the request while it enjoys the service of other pseudo-downloaders. As such, we propose to blacklist free riders at the tracker. Tracker can always detect peers who reject to participate as pseudo-downloaders because after selecting a particular peer as a pseudo-downloader, if it does not send announce requests as a pseudo-downloader that implies that the peer rejects to participate as a pseudo-downloader. In addition to that individual peers can report about peers who reject to participate as pseudo-downloaders. By collecting

those data, tracker will form a black list of peers, and will reject to service those peers in future.

5.2.3 Modelling Proposed Peer-Assisted Patch Dissemination

In this work we evaluated the proposed peer-assisted patch distribution system only through experiments. If a mathematical model can be developed to capture the dynamics of the proposed peer-assisted patch distribution system, it would give better insight on system dynamics and impact of various system parameters. The mathematical model should also be able to capture, how download time varies over time, how number of download completed peers progress over time and how number of infected peers progress over time when vulnerable pseudo-downloaders are present.

REFERENCES

[1] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A Taxonomy of Computer Worms," in Proc. *2003 ACM workshop on Rapid malcode*. Washington, DC, USA, 2003, pp. 11-18.

[2] S. Shakkottai and R. Srikant, "Peer to Peer Networks for Defense Against Internet Worms," *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 9, pp. 1745-1752, December 2007.

[3] L. Xie and S. Zhu, "A Feasibility Study on Defending Against Ultra-Fast Topological Worms," in Proc. 7th *IEEE International Conference on Peer-to-Peer Computing*, Galway, Sept. 2007, pp. 61-70.

[4] D. Serenyi and B. Witten, "RapidUpdate: peer-assisted distribution of security content," in Proc. *7th International Conference on Peer-to-peer Systems*, 2008.

[5] M. S. Rahman, G. Yan, H. V. Madhyastha, M. Faloutsos, S. Eidenbenz, and M. Fisk, "iDispatcher: A unified platform for secure planet-scale information dissemination," *Peer-to-Peer Networking and Applications*, vol. 6, no. 1, pp. 46-60, 2013.

[6] C. Gkantsidis, T. Karagiannis, and M. VojnoviC, "Planet scale software updates," in *Proc. 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pisa, Italy, August 2006, pp. 423-434.

[7] D. Wu, C. Tang, P. Dhungel, N. Saxena, and K.W.Ross, "On the Privacy of Peer-Assisted Distribution of Security Patches," in Proc. *2010 IEEE 10th International Conference on Peer-to-Peer Computing*, Delft, Aug. 2010, pp. 1-10.

[8] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in Proc. *13th USENIX Security Symposium*, San Diego, August 2004, pp. 303-320.

[9] K. Bauer, D. McCoy, D. Grunwald, and D. Sicker, "BitBlender: Light-weight anonymity for BitTorrent," in Proc. *workshop on Applications of private and*

anonymous communications, Istanbul, Turkey, September 2008, pp. 1-8.

[10] R. Petrocco, M. Capotă, J. Pouwelse, and D. H. J. Epema, "Hiding user content interest while preserving P2P performance," in Proc. *29th Annual ACM Symposium on Applied Computing*, New York, USA, March 2014, pp. 501-508.

[11] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson, "Privacy-preserving P2P data sharing with OneSwarm," in Proc. *ACM SIGCOMM 2010 conference*, New York, USA, October 2010, pp. 111-122.

[12] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[13] A. Varga, "The OMNeT++ discrete event simulation system," in Proc. *European simulation multiconference*, Prague, June 2001, pp. 65-72.

[14] K. Katsaros, V. P. Kemerlis, C. Stais, and G. Xylomenos, "A BitTorrent module for the OMNeT++ simulator," in Proc. *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, London, Sept. 2009, pp. 1-10.

[15] Ported BitTorrent Module for OMNet++ version 4. [Online]. Available: https://github.com/manojds/BitTorrent_Omnet4

[16] C. Jia, X. Liu, Z. Hu, G. Liu, and Z. Wang, *Advances in Electric and Electronics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. Defending P2P Networks against Malicious Worms Based on Benign Worms, pp. 653–660..

[17] M. VojnoviĆ and A. Ganesh, "On the Effectiveness of Automatic Patching," in Proc. *2005 ACM Workshop on Rapid Malcode*. Fairfax, VA, USA, November 2005, pp. 41-50.

[18] G. Chen and R. S. Gray, "Simulating Non-Scanning Worms on Peer-to-Peer Networks," in Proc. *1st International Conference on Scalable Information Systems*. New York, NY, USA, May 2006, pp. 29-41.

[19] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys & Tutorials*, vol. 7, no. 2, pp. 72-93, 2005.

[20] B. Yang and H. Garcia-Molina, "Designing a Super-Peer Network," in Proc. *19th International Conference on Data Engineering*, Bangalore, India, March 2003, pp. 49-60.

[21] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17-32, Feb 2003.

[22] D. Doval and D. O'Mahony, "Overlay networks: A scalable alternative for P2P," *IEEE Internet Computing*, vol. 7, no. 4, pp. 79-82, July 2003.

[23] BitTorrentSpecification. [Online]. Available: https://wiki.theory.org/BitTorrentSpecification

[24] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," *in* Proc. *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 46–66.

[25] PlanetLab. An open platform for developing, and accessing planetary-scale services. [Online]. Available: https://www.planet-lab.org/

[26] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Revised Papers from 1st International Workshop on Peer-to-Peer Systems*, ser. IPTPS'01. London, UK, UK: Springer-Verlag, 2002, pp. 53–65.

[27] T. Chen, X. Zhang, H. Li, X. Li, and Y. Wu, "Fast quarantining of proactive worms in unstructured P2P networks," *Journal of Network and Computer Applications*, vol. 34, no. 5, pp. 1648-1659, September 2011.

[28] INET Framework. [Online]. Available: https://inet.omnetpp.org/

[29] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang, "A Performance Study of BitTorrent-like Peer-to-Peer Systems," *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 1, pp. 155-169, Jan. 2007.

[30] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in Proc. *2007 IEEE Global Internet Symposium*, Anchorage, Alaska, May 2007, pp. 79 - 84.