Protecting BitTorrent: design and evaluation of effective countermeasures against DoS attacks

Marinho P. Barcellos
Federal University of Rio Grande do Sul & Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS 90619-900 – Brazil

Daniel Bauermann, Henrique Sant’anna
Matheus Lehmann, Rodrigo Mansilha
Universidade do Vale do Rio dos Sinos
São Leopoldo, RS 93022-000 – Brazil

Abstract

BitTorrent is a P2P file-sharing protocol that can be used to efficiently distribute files such as software updates and digital content to very large numbers of users. In a previous paper, we have shown that vulnerabilities can be exploited to launch Denial-of-Service attacks against BitTorrent swarms, which can substantially increase download times and network traffic. In this paper, we review the three most damaging attacks, and propose two algorithms as countermeasures to effectively tackle them. We implemented the attacks and countermeasures in a packet-level BitTorrent simulator. The results indicate that our proposed approach is effective when there is an ongoing attack while at the same time efficient when the countermeasure is active but there is no attack. To the best of our knowledge, this is the first proposal in the literature to make BitTorrent more robust against Denial-of-Service (DoS) attacks.

keywords: Event-based and Peer-to-Peer infrastructures, Internet-based systems and applications

1. Introduction

BitTorrent is a file sharing system based on the Peer-to-Peer (P2P) paradigm. It harnesses resources such as network bandwidth and storage of peers to efficiently distribute large volumes of data. The protocol has gained increased popularity, and now is virtually a “de facto” standard for content distribution. A new generation of content-distribution applications is being built on top of BitTorrent, to be employed, among others, by large media companies such as BBC and Twentieth-Century Fox and device makers such as D-Link.

In this context, security attacks to BitTorrent-based applications may cause substantial harm to their users (and to the media companies interested in disseminating the content they own). There have been studies in the literature about harmful peer behavior in BitTorrent networks. For instance, Locher et al [1] present BitThief, an agent which downloads but never contributes to the network, and Sirivianos et al [2] identify an attack in which a free-rider profits from having a wider view (in number of peers) of the swarm. The attack is called Large View Exploit. In [3] Piatek et al present BitTyrant, a modified version of Azureus that adheres to the protocol but uses different policies to improve download performance while reducing upload contributions. In [4] Shneidman et al present two strategies, namely multiple identities and garbage upload, that can be combined by a peer to perform free-riding. Finally, in [5] Liogkas et al present three techniques in which a peer does not respect the protocol and lies about piece availability or refuses to cooperate.

The common aspect among the previous papers is the focus on how a peer can maximize the benefit received from a BitTorrent swarm whilst minimizing the resources it contributes. In contrast, in our work a malicious peer is not interested in downloading; instead, its sole intention is to hinder one or more swarms employing the minimum amount of resources required. In a previous paper, we demonstrated that BitTorrent swarms were vulnerable to a set of DoS attacks [6]. In this paper, we introduce two countermeasure algorithms, which we coined as “PeerRotation” and “Anti-Corruption”, that effectively reduce the impact of the attacks previously identified without incurring overhead when there is no attack. As far as we know, there is no proposal of security countermeasures for BitTorrent in the literature. Thus, the contribution of our paper is to design and evaluate these two novel algorithms.

The proposed countermeasures were implemented and their efficacy was evaluated through simulation. Representative scenarios were prepared to compare situations with/without attack as well as with/without...
countermeasure. The results suggest that the proposed algorithms can be incorporated to the protocol, making BitTorrent user agent implementations and derived applications more secure.

The remainder of this paper is organized as follows. Section 2 reviews BitTorrent and briefly discusses the three attacks tackled. Sections 3 and 4 present the proposed algorithms, PeerRotation and Anti-Corruption, respectively, justifying design decisions and discussing how they operate. The algorithms were implemented using a detailed message-level BitTorrent simulator (called TorrentSim). This implementation enabled the evaluation of the proposed algorithms, as presented in Section 5. Section 6 ends with conclusions and perspectives of future work.

2. Background

BitTorrent is a reasonably well-known protocol, with several papers that describe how it works. However, there is no clear and unanimously accepted description of the protocol. We first present a set-oriented overview of it, which we will employ to describe the countermeasure algorithms. Then we describe the attacks performed by malicious peers, and finally, simple countermeasures available in some user agents.

2.1. Set-oriented View of BitTorrent

A tracker works as a bootstrap server for peers. Once an user agent loads a .torrent file with metadata, it can connect to the tracker. When it does so, it registers itself as part of the swarm, and periodically (say, every 30 minutes) reconnects to the tracker. The set \( S_t \) represents the tracker view of the swarm membership, and if a given peer fails to connect multiple periods in a row, it is removed from \( S_t \). Once connected to a peer, the tracker may supply a list of peers (called Peer List and denoted as \( L \)) randomly selecting peers from \( S_t \).

The “local peer” is denoted as \( p_i \), whereas \( p_{j}, p_{k}, \ldots \) represent “remote peers”. The list of “known peers” to \( p_i \) is called Peer Set and denoted as \( P_i \). When \( p_i \) receives \( L \) \((L', L'', L''', \ldots)\), it adds peers to \( P_i \): \( P_i \leftarrow P_i \cup L \). When \( p_i \) needs to open connections to new peers, it randomly selects entries from \( P_i \). If \( p_i \) attempts a connection with \( p_j \in P_i \), but \( p_j \) does not respond, then \( P_i \leftarrow P_i \setminus \{p_j\} \).

The list of peers currently connected to \( p_i \) is known as the Active Peer Set, and denoted as \( A_i \). (This set is of singular relevance in the context of this paper.) When a connection is opened between \( p_i \) and \( p_j \), \( A_i \leftarrow A_i \cup \{p_j\} \). When it is closed, \( A_i \leftarrow A_i \setminus \{p_j\} \). \( A_i \subseteq S_t \)

1. when necessary, an index is used to indicate a specific peer, such as \( p_i \).

is usually true, but not necessarily, since \( p_j \) may stop contacting the tracker but remain connected with \( p_i \).

The content is divided in “pieces” of constant size. Pieces are divided in “blocks” of 16KB. The number of blocks per piece is typically configured according to the content being shared, but constant for all pieces. Considering the relationship between pieces and blocks, we denote the \( x^{th} \) piece as \( b_x \), or \( b_x, e_x \) and the \( y^{th} \) block of \( b_x \) as \( b_{x,y} \).

The .torrent file contains information about the set of pieces that comprise the content, including the hash SHA1 of each piece \( b_x \). A peer \( p_i \) downloads a piece \( b_x \) and checks its integrity using the hash function and compares with the information obtained from the .torrent. We use \( h_x \) to abstractly denote a hash function that is true if \( b_x \) is correct, and false otherwise.

The set \( A_i \) represents the set of peers that are connected with \( p_i \); \( |A_i| \) is the number of peers in \( A_i \), \( A_{max} \) is the maximum size of \( A_i \), and thus \( |A_i| \leq A_{max} \). Peers enter \( A_i \) as the result from both outgoing and incoming connections; \( p_i \) tries to maintain at least \( A_{min} \) connections opened \((|A_i| \geq A_{min})\), attempting outgoing connections when \(|A_i| < A_{min} \) – as long as there are usable peers in \( P_i \). With \( A_{min} \) connected peers, \( p_i \) keeps \( A_{max} - A_{min} \) “slots” available in \( A_i \) for incoming connections from new peers. In some user agents, default values are \( A_{min} = 30 \) and \( A_{max} = 50 \); however the informal protocol specification does not define standard values and user agents may allow the user to configure them.

The presence of peer \( p_j \in A_i \) does not imply data exchange. Before \( p_i \) sends pieces to \( p_j \), \( p_j \) needs to be “interested” in \( p_i \) (possess pieces \( p_i \) does not), and \( p_j \) has to enable \( p_j \) to ask pieces \( (p_i \) unchokes \( p_j \)). In principle, \( p_i \) keeps up to 4 unchoked peers, and serves their requests as it can. According to the tit-for-tat policy, 3 peers are chosen according to reciprocity – the 3 peers that most contributed to \( p_i \) recently – plus one peer that is randomly chosen in \( A_i \) (this is called “optimistic unchoking”).

A peer that has all pieces is called “seeder”, and “leecher” otherwise. Peers are divided in two groups: honest and malicious. Leechers can be either, but we assume that they never change behavior. A seeder, too, could be honest or malicious; in the latter case, it would be distributing polluted content, which is a different kind of attack [7]. In the context of this paper, we assume that seeders are always honest. Thus, we abstractly define that the sets \( H \) and \( M \) represent the honest and malicious peers, respectively, and \( A_{H}^{S}, A_{M}^{S} \) their corresponding intersections with \( A_i \). When peers join or leave \( S_t \) they also join/leave either \( H \) or \( M \). Note that these sets are for representation only, and thus are not implemented by peers.

The elements (such as \( p_i \)) and the sets (such as \( P_i \) and \( A_i \)) allows us a less more succinct and less
ambiguous definition of the countermeasures and their algorithms.

2.2. Attacks

This paper proposes countermeasures against three attacks: Eclipse, Piece Lying and Piece Corruption. The first two attacks are only effective by employing a large number of peers; since BitTorrent does not require authentication, it allows a malicious user to fake dozens or hundreds of identities (called “sybils”) that map to a single machine or network [8]. Below, we review very briefly each attack.

Eclipse. It consists of a malicious user creating a large number of sybils \((p_m \in M)\) and having them connect to honest peers \((p_i \in H)\). If \(|M| \gg |H|\), then \(\frac{|M \cap H|}{|H|} \approx 1\). Since \(H\) is populated with entries selected from \(P\), plus incoming connections, the chance of establishing connections with malicious peers is much larger. If the attack is successful, then \(\frac{|M \cap H|}{|M|} \approx 1\); when \(H \subseteq M\), the honest peer is called “massive lying”.

Piece Lying. This attack aims at breaking the homogeneity of individual piece availability in the swarm [6]. With the help of sybils, a malicious user falsely announces the possession of a piece it does not actually have. This interferes in the priority given by honest peers in the choice of the next piece to be loaded. Consequently, the availability of some pieces tend to decrease and a piece might even disappear completely; if so, the swarm suffers a failure, which is partial if some peers managed to complete their downloads before they left. To be effective, this attack requires a large number of peers to lie about a few pieces; thus, one might regard it as an “advanced eclipse”. We will adopt this approach, and the attack will be called “massive lying”.

Piece Corruption. A malicious peer can send one or more corrupted blocks to compromise the integrity of pieces. The piece with one or more incorrect blocks will not be accepted by the peer performing the download, and will have to be download again. It is not possible for a correct peer to find out which block (or blocks) is/are incorrect. Therefore, the entire piece is discarded and all its blocks are download again. A corruption attack by a malicious peer consists in sending corrupted blocks to all connected peers, attempting to compromise as many pieces as it can. To launch an attack of this nature, a \(p_m\) informs \(p_i\) \((p_m \in M, p_i \in H)\) that \(p_i\) is unchoked, then waits for a block request from \(p_i\), which is replied with random content instead. Immediately afterwards, \(p_m\) informs \(p_j\) that is being choked by \(p_m\) (or, else, \(p_j\) disconnects from \(p_i\), to reconnect later). If \(p_m\) choking or closes the connection, \(p_i\) will typically wait some time before requesting to other peers the remaining blocks of the desired piece. When the piece is completed, \(p_i\) finds out that it is corrupted and discards it. Then, the piece will be requested to a \(p_i \in A_i\).

To be effective, the attack launched by \(p_m\) to \(p_i\) must be repeated periodically. Otherwise, \(p_i\) will recover the piece (and others) from honest peers. The challenge for this attack is to determine the optimal periodicity, since it depends on many factors (including bandwidth, piece distribution, and so on). Lastly, if \(p_m\) wishes to force the transmission of a given \(b_x\) to \(p_i\), it may force \(p_i\) to chose this piece by announcing it as the single piece available to \(p_m\) (in its piece bitmap; to do so \(p_m\) might have to disconnect and reconnect to \(p_i\)).

2.3. Existing countermeasures

BitTorrent includes, from its inception [9], a mechanism to tackle inactive peers: it is called “anti-snubbing”. There is no consensus, however, on how it works. According to Cohen, anti-snubbing may increase the number of peers that are concurrently benefited by optimistic unchoking. In principle, the optimistic unchoke randomly selects a single peer to which it will concede upload. The idea behind is probing for new, better peers. In contrast, with anti-snubbing, \(p_i\) monitors the amount of data contributed by peers and if an unchoked peer \(p_j\) fails to provide a single complete block in 1 min, then \(p_j\) is choked and its slot is allocated to an additional optimistic unchoke. Hence, it is possible that in a given moment multiple optimistic unchokes are triggered because one or more peers did not contribute with any blocks.

With respect to corrupted pieces, popular BitTorrent user agents provide countermeasure mechanisms to punish peers that consistently send corrupted pieces. These mechanisms are often referred to as “IP filters”, since they filter requests from a set of peers that have been blacklisted. Such mechanisms may be simple, or even naive, in punishing only those peers that send all blocks of a corrupted piece. The more aggressive alternative is to ban all peers that have contributed to the corrupted piece. One of the drawbacks of this approach is that a honest peer \(p_j\) that contributes with larger amounts of data to \(p_i\) is more susceptible of punishment due to incorrect transmissions. Therefore, a more intelligent mechanism, as adopted in Azureus, would monitor how often a peer contributes to corrupted pieces, and then compare with the amount of correct pieces that this peer has sent. Peers that contribute with a considerable fraction of corrupted content may be then banned.

Another possibility is to allow the end user to manually control the filtering of peers. Therefore, any peer can mount and share a blacklist, from information recorded in a site that aggregates information about malicious peers. However, this gives rise to
other challenges, such as the dynamism of IPs in the Internet, scalability limitations, and the trustworthiness of information stored there.

3. PeerRotation Countermeasure

The massive lying attack both isolates honest peers and leads to an unbalance in piece availability. This section describes a countermeasure algorithm, coined PeerRotation, that mitigates the effect of such an attack. In general terms, the algorithm which runs in $p_i$ seeks to identify the peers that consistently remain “inactive”, that is, neither provide blocks to $p_i$ nor request from it. Such suspected peers are placed in a quarantine and temporarily disconnected by $p_i$. The newly available slots may be used to establish outgoing connections to other known peers as well as to accept incoming connection requests, depending on the current value of $|A|$. The algorithm in $p_i$ only interferes in situations in which it might gain some benefit to $p_i$; it may be worth rotating inactive peers only if all slots for outgoing connections are taken ($|A|$ ≥ $A_{\text{min}}$), there are some inactive peers, and $p_i$ knows “new” peers that could possibly occupy some of these peer slots. Further, no rotation would be useful if there are no new honest peers to be tried by $p_i$ (that is, $H \subseteq P$); however, $p_i$ cannot determine this condition for sure. Therefore, if there are multiple available slots in $A_i$, the motivation for $p_i$ disconnecting suspected peers is not so strong. Reflecting this, as a rule of thumb, the proposed strategy requires that $p_i$ be already connected to a substantial number of peers before evaluating potentially inactive peers. More precisely, if $|A| ≤ \frac{3}{4}A_{\text{min}}$ no rotations are allowed. Thus, we anticipate that in small swarms, without attack, the countermeasure will not introduce any overhead.

Peers become suspect to $p_i$ when they consistently refrain from doing any data transfer with $p_i$. A peer $p_i$ monitors the exchange of data with each $p_j \in A_i$ from the moment the connection is established. The amount of blocks exchanged with $p_j$ so far in the connection, denoted as $d_j$, is divided by the time elapsed for that connection, $t_j$, to produce the rate, $r_j = \frac{d_j}{t_j}$. These values do not persist among connections with the same peer. A peer is allowed some time before its rate is evaluated; denoted as $t_{\text{min}}$, this time is equal to all peers. $r_{\text{min}}$ denotes the minimum rate a peer $p_j$ must honor with $p_i$ to avoid being suspected by $p_i$.

Hence, when $t_j ≥ t_{\text{min}} ∧ r_j < r_{\text{min}}$, $p_j$ is considered suspect. As mentioned earlier, a suspected peer will not necessarily be disconnected. But when this does happen, peers disconnected by $p_i$ are placed in a quarantine, denoted by set $Q_i$, and initially empty. Connections to $p_i$ incoming from peers in $Q_i$ are rejected. The time a peer must spend in $Q_i$ given by $cq_i \in \mathbb{R}$, $cq_i ≥ 1$, is variable and grows geometrically according to a factor $f$, as $p_j$ becomes a suspect over and over. Initially, $cq_j$ is set according to a global quarantine time, but as the other parameters, is a choice of each individual peer. The remaining quarantine time of $p_j$ is given by $q_j \in \mathbb{N}$, $q_j ≥ 0$. When a peer enters $Q_i$, $q_j$ assumes the current value of $cq_j$. When $p_j$ leaves $Q_i$, $p_i$ allows connections between itself and $p_j$, but this does not necessarily occur. Algorithm 1 shows the pseudo-code of the countermeasure.

Algorithm 1 PeerRotation countermeasure

1: for all $p_i \in Q$ do
2: \hspace{1em} $q_i ← q_{i} - 1$
3: \hspace{1em} if $q_i = 0$ then
4: \hspace{2em} $Q ← Q \setminus \{p_i\}$
5: \hspace{1em} end if
6: end for
7: $a ← |P \setminus (A \cup Q)|$
8: $A' ← \{p_j, p_k, p_l \ldots \in A_i \mid \frac{d_j}{t_j} ≤ \frac{d_k}{t_k} ≤ \frac{d_l}{t_l} \ldots\}$
9: for all $p_j \in A'$ do
10: \hspace{1em} if $(t_j ≥ t_{\text{min}} ∧ r_j < r_{\text{min}} ∧ \{|A| > A_{\text{min}} ∨ \{|A| ≥ \frac{1}{2}A_{\text{min}} ∧ a > 0\})$ then
11: \hspace{2em} $A ← A \setminus \{p_j\}$
12: \hspace{2em} $Q ← Q \cup \{p_j\}$
13: \hspace{2em} $q_j ← cq_j$
14: \hspace{2em} $cq_j ← cq_j \times f$
15: \hspace{2em} if $|A| < A_{\text{min}}$ then
16: \hspace{3em} $a ← a - 1$
17: \hspace{2em} end if
18: \hspace{2em} end if
19: end for
20: while $|A| < A_{\text{min}} \land P \setminus (A \cup Q) \neq \emptyset$ do
21: \hspace{1em} $p_k ← \forall p_j \in P \setminus (A \cup Q)$
22: \hspace{1em} $A ← A \cup \{p_k\}$
23: \hspace{1em} $t_k ← 0$
24: \hspace{1em} $d_k ← 0$
25: end while

In lines 1-6 of Algorithm 1, the remaining quarantine time is updated for every peer $p_j \in Q$, removing each $p_j$ when its time counter reaches 0. In line 7, a variable $a$ is set to represent the number of peers that $p_i$ “knows” and that are potential candidates to take the place of peers that might be rotated. To be eligible, a peer $p_j$ may not be connected to $p_i$ nor be in quarantine. In line 8, the algorithm creates $A'$, which is $A$ sorted in increasing order of data exchange with each peer; that is, the first peers are more obvious candidates to be rotated than the last ones.

Lines 9-19 correspond to the process of evaluating peers in $A'$, potentially disconnecting and placing in quarantine some of them. In line 10, the algorithm first evaluates the exchange rate with $p_j$; if the connection has not elapsed enough time or the peer has sustained an adequate rate, then it must not be rotated. Otherwise, the situation is evaluated in regards to the number of connected peers, recalling that rotations are more useful when $|A|$ is large. So,
one of two conditions must be satisfied to rotate a peer: (i) while the number of connected peers exceeds \( A_{\text{min}} \), peers may be disconnected (thus decreasing \(|A|\)) to give an opportunity for incoming connections from other peers; (ii) while there exist in \( P_i \) a potential replacement, to which \( p_i \) could request a connection (\( a > 0 \)), and the number of connected peers is at least \( \frac{2}{3}A_{\text{min}} \). The choice of fraction is arbitrary, but is driven by the need to “renew” part of set \( A \) when there are a substantial number of candidate peers to replace the peers that have been observed as inactive. Lines 11 and 12 refer to the disconnection of \( p_j \) and its inclusion in the quarantine, respectively. Next, the quarantine time to be observed by \( p_j \) is set (line 13) and increased according to the factor \( f \) (line 14). In lines 15-17, \( a \) is decremented as long as we are assuming that \( p_i \) is going to connect to any peer \( p_k \in P_i \), to replace \( p_j \).

As a result of the disconnection of peers selected during analysis, resources in \( A \) are freed. In lines 20–25 of Algorithm 1, \( p_i \) makes outgoing connections to increase the number of peers in \( A \), until at most \( A_{\text{min}} \) peers. The slots left in \( A_i \) remain available to peers that request incoming connections with \( p_i \). Nonetheless, there is no guarantee that connections are in fact confirmed, for several reasons, including the fact that peers \( p_j \in P \) reject the connection (e.g. because \( |A_j| = A_{\text{max}} \) or because they may have left the swarm). In line 21, the algorithm chooses any peer that is neither connected nor in quarantine, attempting a connection with it (line 22) and resetting the corresponding data and time counters (lines 23–24).

In summary, the benefit of this countermeasure lies in replacing potentially malicious peers, which seem uninterested in exchanging data with \( p_i \), for other peers that might show themselves otherwise. The efficacy and efficiency of the proposed algorithm are evaluated in Section 5.

4. Anti-corruption Countermeasure

This section describes a countermeasure to address the corruption attack. The transmission of corrupted blocks harms the performance of the swarm as it forces the peers to waste their download/upload bandwidths to transfer repeated times the same piece or pieces. A countermeasure must identify which peer, or peers, among the set that have contributed with blocks to a given piece, were the ones that provided the corrupted blocks. The Anti-corruption countermeasure not only identifies malicious peers, but also attempts to reconstruct a piece without having necessarily to transfer all the piece blocks again.

To identify the corrupting peers, we propose an algorithm that is based on reputation. When peers are caught/suspected providing corrupted blocks of a piece, their reputation decreases. In contrast, when a peer helps by sending blocks of a correct piece, then its reputation increases. There are situations where nothing can be guaranteed and reputations are changed based on suspicion only. When the reputation of \( p_j \) for \( p_i \), reaches 0, \( p_j \) is disconnected from \( p_i \) and placed in a quarantine kept by \( p_i \), like in the previous case. In the context of this paper, we consider only the case in which the peer that goes to quarantine due to corruption never recovers.

Unlike the previous countermeasure, which was periodic, this one is event-driven, and activated when the transfer of a piece is completed. When a piece turns out to be corrupted, to prevent the peer from having to transfer all \( b_{x,s} \) blocks of this piece again, the algorithms tries to reconstruct it by loading only the corrupted blocks. In the best-case scenario, there is a single corrupted block in the piece. To identify which are the corrupted blocks, the algorithm relies on the following basic assumptions:

- an honest peer \( p_j \), while sending the blocks of a piece \( b_y \) to \( p_i \), tends to collaborate by providing more blocks of \( b_y - \) because \( p_j \) is not choking \( p_i \) and it has \( b_y \);
- a malicious peer, for it has limited resources such as bandwidth, will attempt to optimize the attack by sending a single corrupted block and then will choke the \( p_i \), or voluntarily disconnect.

Algorithm 2 presents the pseudo-code for this countermeasure. The reputation of a peer \( p_j \), from the point of view of \( p_i \), is denoted as \( o_{ij}^t \) (of “opinion”), with \( o \in [0; 1] \). In the algorithm, the first step is to verify the hash \( h_x \) of piece \( b_x \); if consistent (lines 2-4), the reputation of peers that contributed to the piece is increased according to a delta, defined as \( \delta_{\text{inc}} \), and respecting the maximum value for opinions. The case of a corrupted piece is discussed below.

In line 6, the original piece is saved for two reasons: its data blocks contribute to piece recovery as well as help identifying which specific blocks were corrupted (and ultimately which peer(s) provided them). In line 7, a set \( R \) is created with all peers that contributed to the piece; for such, we denote as \( r_{x,y} \) the peer that sent block \( b_{x,y} \) and \( U_x \) the set of peers that contributed to piece \( b_x \). There is one special peer whose importance to the countermeasure stands out among the peers of this set: the peer that sent the last block, the one that completed the piece. This peer will be denoted as \( p_c \). In line 8, a set is created with each of the blocks that have not been provided by \( p_c \). Each interaction of the loop (lines 9-12), \( p_i \) chooses a block in \( B \), reloads it from \( p_c \) (line 10) and then removes it from \( B \) (line 11), until one of the three conditions are verified: (i) the piece is recovered, as indicated by the hash function; (ii) all blocks originally in \( B \) were (re)loaded from \( p_c \), when \( B = \emptyset \); (iii) \( p_c \) choked \( p_i \) or disconnected, making it
Algorithm 2 Anti-corruption countermeasure

1: if $h_x$ then
2:     for all $p_i \in R$ do
3:         $a_i \leftarrow \min(a_i + d_{inc}, 1)$
4:     end for
5: else
6:     $b_x \leftarrow b_x$
7:     $R \leftarrow U_x$
8:     $B \leftarrow \{b_{x,y} | b_{x,y} \neq p_c\}$
9:     while $b'_{x,y} \land B \neq \emptyset \land U_nchoked(p_c)$ do
10:         $b'_{x,y} \leftarrow$ requests any $b'_{x,y} \in B$ from $p_c$
11:     end while
12: if $h_x$ then
13:     $a_c \leftarrow \min(a_c + d_{inc}, 1)$
14:     $R \leftarrow \{u_{x,y} | b_{x,y} \neq b_{x,y}\}$
15:     for all $p_i \in R$ do
16:         $a_i \leftarrow \max(a_i - d_{dec}, 0)$
17:     end for
18:     $b_x \leftarrow b_x$
19: else
20:     if $p_c \notin A \lor \neg U_nchoked(p_c)$ then
21:         $a_c \leftarrow \max(a_c - \frac{d_{dec}}{2}, 0)$
22:         $b_x \leftarrow b_x$
23:     else
24:         $a_c \leftarrow \max(a_c - 2 \times d_{dec}, 0)$
25:     end if
26: end if
27: end if
28: for all $p_i \in R$ do
29:     if $a_i = 0$ then
30:         $A \leftarrow A \setminus \{p_c\}$
31:         $Q \leftarrow Q \cup \{p_i\}$
32:     end if
33: end for
34: end if

impossible for $p_i$ to load new blocks from $p_c$.

In line 13 the algorithm verifies if the piece was recovered. If so, in line 14, the reputation of $p_c$ is increased: $p_i$ can be sure that $p_c$ provided only correct blocks. In line 15, a set is created containing all peers that have provided blocks $b_x$ that did not match with the corresponding block in the correct piece, $b^c_x$. In lines 16-18, the reputation of these peers is decreased. In line 19 the contents of the recovered piece $b_c^c$ are copied to the original one. Lines 21-26 represent the situation in which it was not possible to recover the piece, as discussed next.

Line 21 verifies if the loop was terminated because $p_c$ disconnected or choked $p_i$; if so, $p_c$ reputation is decremented, although less strongly since it means only a suspicion (line 22). The fact that a peer $p_i$ is entitled to choke $p_c$, such “punishment” is applied because a non-cooperative strategy could be employed by a malicious peer. Even if the piece was not recovered, its new version is restored on top of the original (line 23), hoping that the former will have a larger number of correct blocks. The countermeasure seeks in this case to have less providing peers, but each with more blocks to a given piece. In line 25, the reputation of $p_c$ is strongly decremented, since the evidence against it is stronger: it was the single peer to contribute to the piece and even so it was corrupted.

Finally, in lines 28-33, the algorithm verifies the reputation of all peers that took part in the corrupted piece. Any peer whose reputation reaches 0 is disconnected and placed in a quarantine (as mentioned earlier, in the present context, it means banning the peer).

In summary, the countermeasure seeks to identify peers that are responsible for corrupting pieces with incorrect blocks, by associating a reputation index to each peer. In principle, a single event – such as sending blocks to a piece that turns out to be corrupted – should not make a peer enter the quarantine. A peer that contributes to correct pieces will have its reputation increased. In contrast, when a peer consistently behaves (apparently) in a malicious manner, that is, its participation is only or mostly associated to corrupted pieces, its reputation will quickly decrease and the peer will be disconnected.

5. Evaluation of Countermeasure Algorithms

The algorithms presented in Sections 3 and 4 were implemented in a BitTorrent simulator. Called TorrentSim [10], it mimics in detail the message-level communication of BitTorrent and was experimentally validated against actual swarms. In this section, we report on results obtained through a simulation campaign with the algorithms. We evaluated the efficacy of the proposed countermeasures in reducing the impact of attacks and their efficiency in avoiding overheads when there is no attack; we also studied how well the malicious detection scheme works.

We organized the performance study attempting to answer four fundamental questions: Q1. What is the impact of the massive lying and corruption attacks (contrasting the scenarios with vs. without attack)? Q2. What is the efficacy of the proposed countermeasures when facing the attack mentioned in Q1? In scenarios with attack, how well the countermeasure mitigates its negative impact? Besides, how this scenario with attack/countermeasure compares with the optimal one, without attack (and no countermeasure)? Q3. Regarding efficiency, what is the “cost” introduced by the countermeasures when there is no attack? Q4. Do PeerRotation and Anti-corruption algorithms manage to identify malicious peers promptly and correctly?

Next, we present the scenarios evaluated, the metrics collected, the assumptions made and the choice of parameters for the countermeasures.
5.1. Simulation model

The range of scenarios evaluated and reported in this paper refer to the initial phase of a swarm, when the peer arrival rate is initially high but decreases exponentially. The experiments consider one initial seeder and the arrival of 250 leechers exponentially distributed in 60 min, with a concentration around the first 10 min. This initial seeder is permanent, but has no special resources or capacity.

The evaluation focused on the following metrics: “time spent by each peer to download the content”, and “time required by all honest peers to obtain (correct) pieces”. The first one refers to successful download completion. Thus, a corresponding \( f(x) \) is a monotonic non-decreasing function that shows how long it takes for \( x \) peers to complete their download, with \( x \in [1; 250] \). It is possible, under attack, that one or more peers fail to complete their download. We consider a (partial) swarm failure if one or more peers fail to complete their download within “reasonable time”.

The second metric evaluates the evolution of a swarm in terms of correct pieces that have been successfully downloaded. From the point of view of a single peer, a failure with 1% of pieces may be very different from one with 99% of pieces. If partial content is deemed useful (e.g. when comprised of hundreds of high resolution photos), it is better having 99% than 1%. On the other hand, there are cases when the full content is required (e.g., an operating system patch to be applied to millions of computers) and having 1% is better than 99% because in the latter case much more resources are wasted. For the sake of evaluation, we adopt the first case, but we do not claim that it is more common than the second. So, the time taken to download (\( y \)) the global amount of correct, downloaded pieces (\( x \)) reflects the evolution of a swarm, and would provide an \( f(x) \) that is monotonic and non-decreasing. Results will be presented for both metrics above.

The parameters of the simulation model can be organized in four categories: (i) environment, (ii) BitTorrent protocol, (iii) attack, and (iv) countermeasure. The two first categories are common to all experiments, since they do depend neither on attacks nor countermeasures. The environment is defined as follows: the number of leechers, 250; initial seeders, 1; arrival of leechers, exponential from time 0; bandwidth of initial seeder and leechers, 1024 Kbps/256 Kbps (bandwidths are represented in pairs, download/upload); and tracker bandwidths, 4096 Kbps/4096 Kbps.

The BitTorrent protocol configuration, on its turn, is defined as follows: number of pieces, 64; piece size, 1 MB (64 blocks); target ratio used by leechers when becoming seeders, 1.0; target ratio of initial seeder, \( \infty \); maximum seeding time for any peer, \( \infty \); minimum number of connections desired (\( A_{min} \)), 30; maximum number of connections (\( A_{max} \)), 50; maximum number of concurrent uploads, 4; optimistic unchoke execution interval, 30s; number of entries dedicated to optimistic unchoking (besides anti-snubbing), 1; tracker connection interval, 10 min; number of peers provided by tracker in \( L \), 50.

Regarding massive lying attacks, the sybils are hosted by a single malicious peer, which shares the information of its \( P \) among its sybils. Each new sybil that is activated opens a connection with the tracker, registers itself (joining the swarm), and receives as reply a new \( L \) (randomly generated based on the current \( S_t \)). The malicious peer then updates its list of known peers, removing its own sybils from the list: \( P \leftarrow P \cup (L \setminus M) \). The parameters that describe an attack are defined as follows. With respect to the mass lying attack: number of malicious sybils, 500; arrival time of first malicious peer, 0.1s (right after initial seeder); sybils “spawning” rate, 1 sybil/3s; bandwidths of the malicious peer (shared by sybils), 8 Mbps/8 Mbps; number of lied pieces, 16 (out of 64). Regarding the corruption attack, the malicious peers arrive every 3s from time 0; interval between the transmission of unchoke messages by each attacking peer, 8s. The choice of parameters was oriented towards increasing the impact of attacks, following an experimental study we conducted with swarms populated by Azureus agents.

Finally, the parameters assumed chosen for countermeasures are listed below. The PeerRotation countermeasure was evaluated with the following choices: evaluation cycle (\( T \)), 1 min; initial quarantine time, 4 evaluation cycles (4 min); elapsed connection time before a peer is evaluated (\( \ell_{min} \)), 5 min; geometric increasing factor for quarantine time (\( f \)), 2.0; minimum required exchange rate (\( r_{min} \)), 0.2 Kbps. The parameters for the Anti-corruption countermeasure are: initial peer reputation (\( \sigma_{i} \)), 0.5; basic decrementation delta upon negative evidence (\( \delta_{dec} \)), 0.2; basic incrementation delta upon positive evidence (\( \delta_{inc} \)), 0.1.

5.2. Evaluation of PeerRotation

This subsection presents an evaluation of the efficacy and efficiency of the PeerRotation countermeasure. In Figure 1(a), the \( x \) and \( y \) axis show respectively the number of completed downloads by honest peers and download duration in minutes. The swarm is entirely successful when all 250 peers manage to complete their downloads, that is, the curve reaches \( x = 250 \). Otherwise, there is a swarm failure at some point (e.g., at time 80.8 min, when only 30 downloads have been completed). Figure 1(b) shows in the \( x \) axis the global number of correct pieces already downloaded,
while the y axis shows the time required (to download such pieces). In both cases, there are four curves, reflecting the cases of interest: “No Countermeasure – No Attack” (NCNA), “No Countermeasure – With Attack” (NCWA), “With Countermeasure – With Attack” (WCWA), and “With Countermeasure – No Attack” (WCNA).

The comparison of the curves with no countermeasure, with vs. without the massive lying attack, helps answering Q1. As we can observe in Figure 1(a), the curve NCNA indicates that the fastest peers complete their downloads at around 60 min, while the slowest ones at around 80 min – when the last download is finished. In contrast, the curve NCWA, depicting the effects of an attack without countermeasure, shows that around 30 peers manage to complete their downloads between 68 and 82 min; the remaining 220 peers fail because of the attack. In terms of completed pieces, Figure 1(b) shows, through NCWA, the impact of an attack. Peers manage to download around 3000 pieces in 70 min, and thereafter very few pieces can be downloaded. Around 138 min, there is a failure in the swarm. Thus, answering Q1, there is a huge impact when a massive lying attack with 500 sybils is launched against a swarm with 250 honest peers without the protection of a countermeasure.

Question Q2, about efficacy of the countermeasure, is answered by comparing in Figure 1(a) the curves with attack, NCWA and WCWA. WCWA allows all 250 peers to complete their downloads; the 250th peer finishes at 93 min, which is only 13 min later than the “ideal” case of NCNA. Examining the same pair of curves, NCWA and WCWA, in Figure 1(b), it is clear that the countermeasure (WCWA) mitigates most of the negative impact of the attack. All downloads are completed, and do so within under 93 min. In comparison with the ideal case (NCNA), the proposed algorithm (WCWA) performs well: 80 min, 16% slower when the number of sybils is twice the number of honest peers.

To answer Q3, about overheads that might be introduced by the PeerRotation countermeasure, we compare in Figure 1(a) the curves without attack (NCNA and WCNA). We anticipated that the PeerRotation algorithm would introduce some delays, because of the (improper) rotation of honest peers, which just happened to be inactive. However, surprisingly, the results obtained with PeerRotation (WCNA) were slightly (around 3-5 min), but consistently, superior to BitTorrent without PeerRotation (NCNA). The reason for this advantage in downloads can be confirmed in Figure 1(b): WCNA is lower (faster) than NCNA by a margin of up to 20 min. We explored other scenarios and found similar results. Our explanation to the improvement lies in the fact that the algorithm may rotate out up to the $A_{max} = \frac{3}{4}A_{min}$ worst peers (with lowest exchange rate). The slots made available in A are randomly assigned to new peers and/or left to new incoming connections, allowing new, better peers to be found.

5.3. Anti-corruption Evaluation

This subsection provides results regarding the Anti-corruption countermeasure. Like in the previous subsection, we aim to answer questions Q1, Q2, and Q3. The meaning of the curves as well as the pair of axis remain the same. Figure 2 shows a pair of plots in which a corruption attack is launched by 15 corruptors, each sending one corrupted block every 4s.

The first point that stands out from Figure 2(a) is the curve NCWA: unlike NCNA, it grows very rapidly to over 100 min already within the few first downloads (around 3). In contrast, with NCNA all peers complete their downloads (and in up to 80 min). Answering Q1, the attack is devastating: all other 247 peers (insist for 600 min) but fail to download the content. With respect to pieces, the contrast between NCNA and NCWA in Figure 2(b) is not as striking. That is, despite the attack, honest peers manage to download nearly all 16000 pieces, albeit at a lower rate (recall that peers discard corrupt pieces and download them again). Peers manage reasonably well to obtain the first 12000 pieces; however, when there are fewer pieces to be downloaded, it becomes easier for the malicious peers to “target” those specific pieces. The end result is that
peers promptly and correctly? To answer this with respect to each countermeasure, we recorded at every minute, for each honest peer $p_i \in H$, the number of malicious peers as well as of honest peers in $A_i$, and obtained averages by dividing them by the number of peers.

We present curves for the average number of honest and malicious peers connected to honest ones. Figure 3 illustrates the peer population of $A$ through time, averaged among all honest peers (in case of corruption attack, seeders are ignored).

Finally, regarding Q3, note in Figure 2(a) that the curves NCNA and WCNA are superimposed. In other words, despite some negligible fluctuation with respect to the download of pieces shown in Figure 2(b), the success rate and time taken by downloads is identical. In summary, the Anti-corruption is efficient as it does not introduce any overhead when there is no attack. This was the expected result, since the Anti-corruption algorithm does not really quick in unless corrupted pieces are received.

5.4. Accuracy of countermeasures

This subsection attempts to answer the fourth and last question, Q4: do PeerRotation and Anti-corruption algorithms manage to identify malicious peers promptly and correctly? To answer this with respect to each countermeasure, we recorded at every minute, for each honest peer $p_i \in H$, the number of malicious peers as well as of honest peers in $A_i$, and obtained averages by dividing them by the number of peers.

We present curves for the average number of honest and malicious peers connected to honest ones. Figure 3 illustrates the peer population of $A$ through time, averaged among all honest peers (in case of corruption attack, seeders are ignored).

With respect to the massive lying attack, results in Figure 3(a) indicate that in the early moments of the swarm the average numbers of malicious peers connected rises rapidly up to 42 peers. In contrast, the number of honest peers grows to 25 peers in the first minute or so, but then drops to 13 peers in the subsequent minutes. Around 7 min, the proportion of malicious peers to honest ones reaches its highest. Thereafter, from 7 min, an up to 30 min, PeerRotation properly identifies and disconnects from the malicious peers. The PeerRotation is effective as it keeps in 5 peers or less the average number of malicious peers that remain connected to honest peers. Around time 85 min, some honest peers start leaving en masse the swarm, leading to a sharp drop in the number of honest ones. However, the number of malicious peers connected also decreases, for peers that had
connections with malicious peers also leave the swarm.

Figure 3(b), on its turn, shows curves for the average number of malicious peers that are connected to each honest peer through time. Since the corruption attack is performed with a smaller number of receivers, unlike the previous case, initially the number of honest peers exceeds by far the number of malicious ones (there are in total 15 malicious peers). From the third minute onwards, the Anti-corruption countermeasure detects and isolates malicious peers. The curve for malicious peers decreases smoothly up 7 peers, at around 82 min. Conversely, as the slots in \( A \) are freed by the countermeasure algorithm, new connections are established (incoming and/or outgoing) with honest peers, while malicious peers remain in the quarantine. Note that not a single honest peer was placed in the quarantine, so the proposed countermeasure is quite effective in detecting and isolating corrupting peers.

6. Conclusions and Future Work

BitTorrent, a key Internet application, is a popular file sharing protocol that allows the efficient dissemination of files to large amount of users. In a previous paper, we identify DoS attacks to BitTorrent and evaluated their impact [6], showing that even attacks with limited resources can be substantially damaging to the swarm. In this paper, we propose two countermeasure algorithms that mitigate the impact of those attacks. To accomplish this, we first revisited BitTorrent by providing a set-oriented view of the protocol (suitable for the specification of the countermeasures), and then described and discussed the algorithms proposed as countermeasures.

The principle behind the countermeasures is the same: allow the local peer to detect the malicious peers (among the connected ones) and disconnect from them, giving the opportunity to other peers. We proposed as countermeasures the algorithms PeerRotation and Anti-Corruption. We evaluated both countermeasures under a range of scenarios. Results have shown that the proposed algorithms are both effective and efficient against massive lying and corruption attacks.

As future work, we are pursuing two lines of investigation. First, we are studying the combination of countermeasures, evaluating their efficacy and efficiency in scenarios without attack, with one of the attacks, and with both attacks. Second, although TorrentSim – used in our evaluation – has been validated and provides results that are close to real life, it is imperative that our findings are confirmed through comparison with experiments in real, live networks. We have implemented the attacks in user agents and are currently implementing the countermeasures.

References


