“AL.I.CUZA” UNIVERSITY OF IAȘI, ROMANIA

DEPARTMENT OF COMPUTER SCIENCE

BATCHERLOR’S DEGREE

Torano - Torrenting Anonymously

candidate:

David-Andrei Viziteu

Session: June, 2022

Supervisor

Lect. Dr. Vârlan Cosmin

“AL.I.CUZA” UNIVERSITY OF IAȘI, ROMANIA

DEPARTMENT OF COMPUTER SCIENCE

Torano - Torrenting Anonymously

David-Andrei Viziteu

Session: June, 2022

Supervisor

Lect. Dr. Vârlan Cosmin

Avizat,

Îndrumător lucrare de licență,

Lect. Dr. Vârlan Cosmin.

Data: 22.06.2022 Semnătura: ....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Viziteu David-Andrei**, domiciliat în **România, jud. Iași, loc. Iași, str. Izvor 14, bl. 629, sc. C, ap. 26**, născut la data de 19.02.2000, identificat prin CNP **5000219226765**, absolvent al Universității „Alexandru Ioan Cuza" din Iași, Facultatea de Informatică, specializarea engleză, promoția 2022, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Torano - Torrenting Anonymously** elaborată sub îndrumarea domnului **Lect. Dr. Vârlan Cosmin**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: 22.06.2022 Semnătura: ....

Contents

Introduction **2**

Contributions . . . 3

Terminology and notations . . . 4

Notations . . . 4

Terminology . . . 4

Requirements . . . 5

Overview of the protocol . . . 5

Related work . . . 7

Thesis structure . . . 9

1 Objectives **11**

1.1 Resistant to brute force corruption attacks . . . 11

1.2 End to end encryption. Assuring anonymity . . . 11

1.3 Familiar user experience. Effortless adoption. Usability . . . 12

2 The Torano design **14**

2.1 Document distribution. The metainfo file. . . . 14

2.2 User experience . . . 15

2.3 Core entities . . . 15

2.3.1 The relay node . . . 15

2.3.2 The tracker . . . 16

2.3.3 The client . . . 16

2.4 Tracker interaction mechanism . . . 17

2.4.1 Announcing the tracker . . . 17

2.4.2 Scraping the tracker . . . 18

2.5 Onions and reply onions . . . 18

2.6 Peer to peer communication protocol . . . 19

2.7 Creating and publishing . . . 20

3 Threat model **21**

3.1 Passive attacker . . . 21

3.2 Active attacker . . . 23

4 The Torano design implementation **25**

4.1 Document distribution. The metainfo file. . . . 25

4.2 User experience . . . 26

4.3 Core entities . . . 27

4.3.1 The tracker . . . 27

4.3.2 The client . . . 27

4.4 Tracker interaction mechanism . . . 30

4.4.1 Announcing the tracker . . . 31

4.4.2 Scraping the tracker . . . 32

4.5 Onions and reply onions . . . 34

4.6 Peer to peer communication protocol . . . 35

4.7 Creating and publishing . . . 37

Conclusions **38**

Future work . . . 38

Client application improvements . . . 38

Other protocol designs . . . 39

Tracker application improvements . . . 39

Appendix 1 **41**

Appendix 2 **42**

Appendix 3 **43**

Appendix 4 **44**

Appendix 5 **45**

Bibliography **46**

Introduction

In the field of mathematics, the term ″anonymity″ refers to an element that exists within a well-defined set and has the property of not being able to be verified that it belongs to that set.

Excluding mathematics, the same word can as well refer to circumstances in which the decision of a person is obscured from the public eye. The act of voting is a perfect illustration of anonymity for this definition. A person is firstly identified and then allowed to vote. The linkability between that person and the voted candidate is the piece of information that must be destroyed. Others see anonymity as a tool for hiding their identity while exposing their thoughts to the world or to a specific group. As an instance, some victims of various trauma depend on such a services when seeking for assistance. Not only the abused are being rehabilitated, but their reputation remains unchanged. Others just like having this power to promote or sustain prohibited activities, which we are not going to exemplify.

So, depending on the context, it can be beneficial or detrimental. But can it be neutral? For example, rumor has it that a system similar to the first generation onion routing was developed by the American army to secure the communication between various teams or departments. Ignoring the truthfulness of this information, can this be a case where an anonymity system can be considered to be ″neutral″?

Online anonymity is a popular subject in university education, mainly due to all the details that gives such a system it's core ability, and due to the vast the potential of research (both practical and theoretical). Still, creators that have deployed incomplete but powerful anonymity systems have been constrained by an authority to stop the research [8]. Others, that have perfected and launched such a mechanism have been imprisoned.

Contributions

The thesis describes the Torano communication protocol for peer-to-peer file sharing, which enables users to distribute anonymously electronic files over the Internet in a decentralized manner. Moreover, it presents at a conceptual level how the Torano protocol should be implemented whilst providing examples on how the writers specifically applied these principles in a developed application.

In order to accomplish this, we had to study and combine procedures from the BitTorrent protocol for peer-to-peer file sharing protocol [3], Onion Routing [6] and Tor: The Second-Generation Onion Router [5] anonymity system. At the moment of writing, both the BitTorrent and The Second-Generation Onion Router are one of the most popular systems optimised for their purpose.

We modified the original BitTorrent protocol by adding additional steps when one interacts with both the tracker and the peers, to protect any communication partners from knowing each other's identity. Those are based on the anonymity design of the first generation Onion Routing: a message is encrypted multiple times - like an onion - and carried downstream on a path. Furthermore, one can build and distribute a reply onion containing layered encrypted data that enables others to contact the creator of this onion without knowing his identity. The technical details are briefly described in the Overview section below and more comprehensively in the Onions and reply onions section. Among the tracker and the peers, the Torano protocol added the presence of a new entity: the relay node. To send or receive files, users must use a Torano compliant client on their internet connected computer. Each client should act as a relay node to facilitate the protection of it's own anonymity. By designing the system like this, the protocol actively encourages those who mainly seek their own self-interest to participate in the network as relay nodes, avoiding to a certain extent the problem of the tragedy of the commons.

The current description of the Torano protocol should serve as a solid foundation for future improvements, study and development of comparable peer-to-peer anonymous file system sharing models.

Torano is a direct alternative to all the systems mentioned in the Related work chapter.

Terminology and notations

Notations

Msg	message: any sequence of bits
$Encrypt_X(Msg)$	encryption of Msg with the X key (any crypto system)
$Decrypt_X(Msg)$	decryption of Msg with the X key (any crypto system)
$SymEncr_X(Msg)$	encryption of Msg with X's public key
$SymDecr_X(Msg)$	decryption of Msg with X's private key
$AsymEncr_Z(Msg)$	encryption of Msg with the Z asymmetric key
$AsymDecr_Z(Msg)$	decryption of Msg with the Z asymmetric key

Throughout the paper we mention that a message (or structure) is encrypted with some public key - $SymEncr_X(Data)$ - and then it is sent to X to be decrypted. In reality, the data that Torano demands to be protected with a public-key cryptographic algorithm is too large to be processed in a reasonable timeframe. Because of this, in practice, assuming we have a considerable message (Msg) that must be sent securely to X, the following steps occur:

1. generate a asymmetric key Z
2. send to X $AsymEncr_Z(Msg)$ and $SymEncr_X(Z)$

Obviously, X will:

1. $SymDecr_X(Z)$ to obtain the asymmetric key Z
2. $AsymDecr_Z(Msg)$ to obtain the message

Terminology

This paper uses a number of terms to refer to the roles played by participants in, and objects of the Torano protocol, some of which have an identical meaning as the terms mentioned in the BitTorrent protocol specification [3].

*"The group of endpoints involved in the distribution of a particular file is called a **swarm**. The file is divided into several pieces."* [3]

The electronic resource that identifies a particular file (or set of files) a swarm can be interested of is called the **metainfo file**.

A particular file (or set of files) a swarm can be interested of can be called a **torrent**.

A swarm (and a torrent) is identified by an unique id, the **info hash**.

*"An endpoint that both downloads (because it does not have the complete file yet) and uploads pieces is called a **leecher** (note that this definition is counter intuitive because, in other contexts, a leecher normally means someone that takes but does not give)."* [3]

*"When an endpoint has the whole file (i.e., it has all the pieces of the file), it does not need to download any pieces any longer. Therefore, it only uploads pieces to other endpoints. Such an endpoint is called a **seeder**."* [3]

An endpoint that is either a seeder or a leecher can be called a **peer**.

An endpoint that receives and passes on information or messages in the network is called a relay node.

The path of ordered relay nodes from the network of any length greater than zero is called a **circuit**.

An **onion** is a structure of data prepared by a sender for a specific recipient, that is carried and processed through a circuit (the last node of that circuit is the recipient).

A **cell** or transit cell is a structure that encapsulates an onion and some external payload that travels directly from one endpoint to another.

A **reply onion** is an onion prepared by a sender in such a way to be received and decoded only by himself.

An endpoint that centralizes and can offer information about the network's participants is called a **tracker**.

The action of an endpoint that informs the tracker about it's presence and capabilities is called **announce**.

Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. [2]

Overview of the protocol

To send a message to a node (Bob), an user (Alice) can chose a path through the network that ends with the destinatary and build an onion that will traverse a path in which each routing node is aware of only it's successor and it's predecessor, but no other nodes in the circuit. The onion travels down the path where is unwrapped by a symmetric key at each node and is relayed to the next corresponding node. The destination node will do one last unwrapping (it assumes it's an onion to be relayed) and will read the message.

For the receiver to be able to respond, the sender (Alice) must create and include a reply onion in the message. This reply onion is being built in a similar way to the previously mentioned onion, with the following dissimilarities: the last node of it's circuit is the creator (Alice), it's message contains data generated by the sender (Alice) and at each layer of encryption it contains a different key (asymmetric or symmetric) that must be used by each relay node it reaches to encrypt the external payload (Bob's response). The response message will be encrypted the first time by Bob with the first key and sent to the next routing node in the response path alongside the return onion. If one of the relay nodes or the responder fails to do as mentioned, Alice will not be able to read the encrypted response message when the reply onion reaches back to her.

Furthermore, Bob is not aware of the person he is communicating with. If Alice were to anonymously provide multiple reply onions to Bob who distributes them to a pool of communication partners, she would not be able do discover the identity of any of the pool's message senders. One of the most important aspect is the fact that any pool communication partner cannot force an exchange of messages with Alice without a reply onion provided by her. Even if a pool participant would guess Alice's address and would try to reach to her in the same way she reached to Bob, Alice can chose to respond to the message (therefore confirming her identity and address) or can choose to ignore it (therefore protecting her identity).

The readers can anticipate that Alice is incapable to respond to a message from one of Bob's friends unless they also provide in the message a reply onion, whose circuit is entirely composed of nodes chosen in the network by it's creator. If the reply onion that Alice must use to respond has a circuit of length equal to one (the next and only node in the circuit is the person who reached to Alice via her return onion), Alice's identity would be instantly revealed. Alice must not trust any reply onion provided and will privately delegate to a relay node in the network the task of beginning that reply onion's circuit. To protect the content of the message, Alice encrypts the message herself with the first provided key and then forward the encrypted content to the delegated node.

This communication design assures Alice's anonymity and end-to-end encryption, leaving subsequently compromised nodes unable to decrypt the message or determine exactly it's source and destination.

To be able to distribute a file anonymously in this peer-to-peer file sharing system, Alice is ought to use a Torano compliant client that:

1. must announce the tracker it's presence as a relay node (and must relay onions)
2. must be provided by the user with a metainfo file corresponding to the data she wants to obtain from and distribute to the other leechers.
3. must scrape the tracker for other relay nodes such that the application will be able build secure circuits for the onions and for the reply onions.
4. must prepare and send the tracker at least one reply onion for a specific swarm (meaning that the person reachable with that reply onion is at least a leecher for the provided metainfo file)
5. must scrape the tracker for other peer's reply onions of the same swarm
6. must use the obtained reply onions to contact (with the previously highlighted consideration) and request the missing pieces of the wanted data

All these steps can be done using the anonymity-preserving communication mechanism mentioned above.

Related work

Anonymity systems derived from Chaum's Mix-Net design can be split into two categories: low-latency approaches that try to anonymize network traffic and high-latency designs that prioritizes the protection of one's anonymity at the expense of increased latency.

The currently described Torano protocol can be said to belong in the second category, as each participant relies on a static reply onion, a configurable number of announces as a seeder for a particular torent, thus influencing to a certain extent the duration it takes a leecher to become a seeder.

Nowadays, using a virtual private network service is the most well-known method of torrenting anonymously among undocumented users. However, relying on a centralized service for privacy might not be the wisest choice, because they are usually under a strict jurisdiction that imposes any routed traffic to be stored for a determined period of time.

Torrenting over TOR - the second generation onion routing [5] has also proven to be problematic [1]. On one hand, the TOR network is not able to properly handle the load. On the other hand, some BitTorrent trackers use the UDP/IP protocol for the announce procedures, a protocol that the TOR network does not and will not support. Even if you would configure a BitTorrent application to proxy it's traffic through the TOR network, many implementations will just communicate anonymously your IP address and port to other peers, leaking your identity. Last but not least, an attacker can observe the unencrypted traffic coming out of the exit node of the built tunnel.

TheOnionBay TORrents [8] project resolves the problems mentioned in the previous paragraph: it's approach is to build a custom torrent client, tracker and a TOR-like network for routing messages, where each peer is being positioned behind a fixed tunnel composed of some of the network's relay nodes. Unlinke in Torano, it requires special deployments of relay nodes that have to be run by volunteers. For two peers to communicate, the tracker bridges the two exit nodes of their tunnels as illustrated in the Figure 1.1. The images have been obtained from the project's website and annotated. Unfortunately, the project has become inactive since 2018, due to several cease and desist notices coming from the FBI.

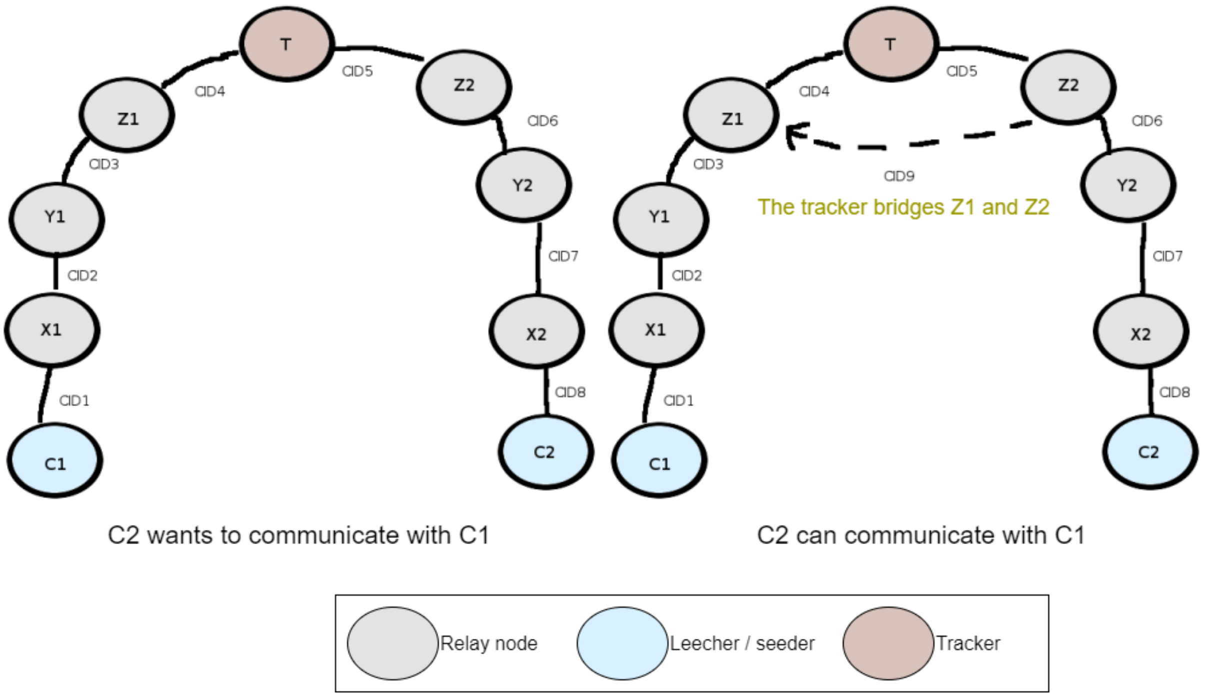

T
Z1
Z2
Y1
Y2
X1
X2
C1
C2
CID1
CID2
CID3
CID4
CID5
CID6
CID7
CID8
CID9
The tracker bridges Z1 and Z2
C2 wants to communicate with C1
C2 can communicate with C1
Relay node
Leecher / seeder
Tracker

Figure 1: TORrents peer to peer communication model

Anonymous Peer-to-peer File-Sharing (APFS) [7] is based on Onion Routing and hides the anonymity of a participant behind a proxi, requiring, Unlinke in Torano, special deployments of proxies nodes. The proxi is used to start onion circuits.

Return Address Spoofing makes use of the ability to alter the sender's address on UDP/IP packets. This technique is often forbidden by ISPs and directly associated with illegal activities.

Besides these projects, we were not able to find other similar BitTorrent inspired systems properly described that are optimized for peer-to-peer file sharing [4].

Thesis structure

It is composed of four chapters and five appendixes. It outlines a protocol for the internet community whilst providing concrete implementation examples, demonstrating that the fundamental idea behind the Torano anonymity system can be implemented in real-world applications.

The introduction presents our contributions, defines some terms and concepts used in this paper, established how certain key words should interpreted, an overview of the Torano protocol, other alternatives to the Torano protocol, and the structure of this thesis.

The first chapter highlights the objectives of the current version of the Torano network. It has three sections, each one presenting a unique goal of the protocol.

Chapter two proposes the Torano protocol specifications, according to the previously established goals. The majority of these are described at a conceptual level, while the vital ones are also more thoroughly exemplified. The first section presents how the files are being perceived and distributed by the software applications as well as the structure and format of the metainfo file that is used by the clients to form swarms. The second section suggests how the user experience should be. The third section specifies the core entities of the Torano protocol and their functionalities. The fourth section summarizes the content of the data that must travel between two endpoints of the network. The fifth section mentions the utilized peer to peer communication protocol. The sixth section describes how peers should exchange file data. The last section presents how can someone publish a new torrent to the network.

Chapter three demonstrates how the Torano network is resilient to different attacks coming from two attacker typologies, each described in it's corresponding section. It also defines the context in which an attack is considered to be successful.

The fourth chapter provides schematics and technical details of our application that respects the Torano protocol specification presented in this thesis. It has seven sections that have identical names to the sections of the second chapter. This correspondence associates each concept to a practical implementation.

The five appendixes contain the base algorithms that we implemented in our application. Chapter three clarifies with additional explanation the intent and applicability of these algorithms.

The conclusion summarises the protocol presented in this thesis and plans for future improvements of both the developed application and the protocol's specifications.

Chapter 1

Objectives

Torano's main objective is to exasperate attackers from correctly identifying the involvement of a particular leecher / seeder in a specific swarm. Besides this, we have also considered the following:

1.1 Resistant to brute force corruption attacks

Due to the nature of the onion routing architecture and of cryptography encryption mechanisms, a brute force software is able to decode merely one layer of an onion in years. So, the attackers will most certainly consider capturing traffic and corrupting legitimate nodes to seize their keys, as it is more time-efficient. [6]

Having established that, the Torano protocol provides a mechanism that offers the possibility for each network participant to refresh it's keys periodically. Any captured onions will be of no use to any attacker.

1.2 End to end encryption. Assuring anonymity

To increase the difficulty of tracking a packet across a circuit, there should not be two exact cells that will travel along a circuit. Every time Alice wants to send a message (either as a response or as an initiator of a conversation) she must first encrypt it with the public key of the receiver or with a key provided by the sender. Then, the resulted data will be encrypted or processed by each node in it's travelling path. This assures end-to-end encryption for a circuit, the anonymity of the initiator of a conversation, and the uniqueness of the contents of any two cells.

As mentioned in the introduction, when Alice uses a reply onion, she must assume that it can not be trusted. To respond to it's creator while protecting her identity, she should use a proxy node. This proxy node can be contacted with an onion whose message contains the reply onion and a specific flag (or message or instruction) with the meaning that the reader node should start it's circuit. This assures the anonymity for the responder in bidirectional conversation.

1.3 Familiar user experience. Effortless adoption. Usability

A protocol that is simple and easy to understand attracts users in need for it's use cases. Nonetheless, it will be easily implementable in real world applications by willing developers. Torano, being inspired by the BitTorrent protocol, facilitates active and potential customers of the BitTorrent protocol to join by delivering a nearly identical user experience. The difference is that a Torano client might have more options to configure than a usual BitTorrent client.

As the Torano protocol resides at the application level of the TPC/IP stack, a conforming client may be easily developed for all common operating systems. In the current implementation, Torano was built exclusively with JavaScript frameworks (ElectronJS, SolidJS, ExpressJS) thus making the applications ready for deployment on Windows, Linux and MacOS, with minor to no code modifications. A large user base will provide a more robust anonymity and reliability of the entire network, due to the fact that the protocol hides the peers among the participant users.

A Torano client application should be easy to deploy and use on a user's computer. The provided implementation was tested on a Windows machine, where the user only had to install the client and allow firewall access once if requested. Once the user loaded the software with a metainfo file, the client will begin downloading.

A Torano relay node should be easy to deploy and use on a user's computer. The provided implementation was tested on a Windows machine, where the user only had to install the the Torano client, allow firewall access once if requested, provide the application a ".torano" file and then delete the download.

Either of the previously mentioned programs should not be computationally expensive to run, allowing machines with reduced computational capabilities or bandwidth to be part of the network.

The tracker can be deployed on any hosting web service. In the current implementation, for it's base main purpose, it requires zero code configuration. The fact that it must manage the presence of the relay nodes and the swarms of the whole network implies the need of more processing power and more bandwidth than a standard Torano client.

Chapter 2

The Torano design

2.1 Document distribution. The metainfo file.

In a similar manner to the BitTorrent protocol, a file that is distributed in the Torano network is being perceived by the peers as an array of equally-sized ″pieces″ (except the last piece, that can have a variable length). The size of a piece is expressed in bits.

A metainfo file must contain at least identification and authenticity data of a file and it′s pieces (piece length, file size, the number of pieces). It must have the extension ″.torano″ and should also contain the tracker′s address.

A proprietary metainfo file structure or encoding for the Torano protocol is beyond the scope of this thesis, and it might not be needed, so we recommend using the BitTorrent specifications for the metainfo file.

A swarm′s unique identification code (the info hash) is the computed SHA1 hash of the contents of the ″.torano″ metainfo file.

Besides the ″announce″ field, a ″.torano″ file can contain other relevant fields. For example, if future implementation will commit to a different tracker entity design, obviously, the metainfo file should contain their location in a field that describes their use-case. Splitting the metainfo file into multiple files will inevitable abrupt the learning curve of the protocol, making the 1.3 goal harder to be achieved.

2.2 User experience

The user experience should be identical to that of the BitTorrent protocol - the consumer will mainly interact with ".torano" files and the installed Torano client. The source of the ".torano" files must be a trusted party (example: a web server).

2.3 Core entities

The core nodes of the network are of three types: relay nodes, clients (that also act as relay nodes) and trackers.

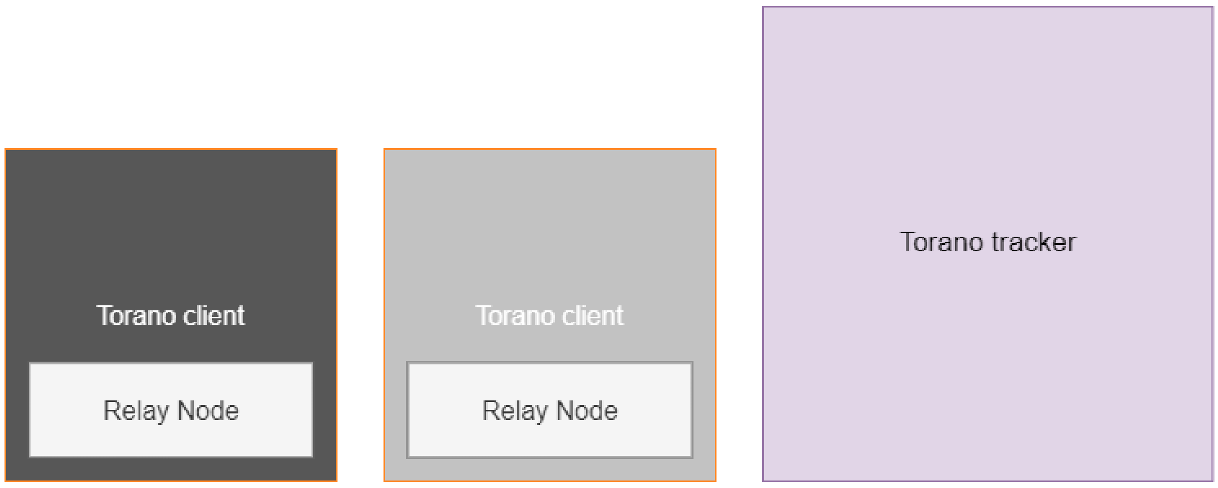


Figure 2.1: The Torano protocol core entities

2.3.1 The relay node

The relay node represents the base unit that this network has been developed on. It is exclusively capable of relaying onions. One mandatory task of a relay node is that it must strip out any extra data from an onion, to prevent an attacker from discovering a circuit between two endpoints.

The previous schema highlights that it is incorporated in each Torano client and that it can be deployed as a stand-alone participant. A Torano client that does not have any active torrents and that announced the tracker is just a relay node. Any volunteer can keep a Torano client running in the mentioned state just to act as a relay node, supporting the network.

2.3.2 The tracker

The tracker manages the relay nodes and the swarms. Swarms are identified using the SHA1 hash of the structure that describes the shared files (Section 2.4 of this chapter). The tracker must at least:

1. provide a constant IP address and port
2. provide a public key for the clients
3. provide a secure service for the clients to announce themselves as relay nodes
4. provide a secure service for the clients to announce themselves as leechers (for a swarm)
5. provide a secure service for the clients that dispenses a list of relay nodes
6. provide a secure service for the clients that dispenses a list of leechers (for a swarm)
7. must regenerate the provided public key (Item 2) as well as clear both leechers and relays lists every predetermined and fixed number of minutes.
8. provide a secure service for the clients that dispenses the timing details of the previous item (7)

A service is secured if the data exchanged between the tracker and a client can not be read by an observer. This result can be obtained if a client encrypts the message with the tracker′s public key and if the message also contains a key that must be used by the tracker to encrypt the response.

2.3.3 The client

A client must at least:

1. generate a pair of public-private keys (should be constant at least in between tracker list refresh events)
2. open a port to provide the services specified in this list (the port number can vary but should be constant at least in between tracker list refresh events)
3. provide a secure service that handles the onion routing
4. provide a service that disposes the public key that should be regenerated simultaneously with the tracker list refresh event

The onion routing service must have the ability to:

1. receive transit cells from the other network participants
2. decrypt the onion of the received transit cell with the own private key (or discard it otherwise)
3. read the message presented at the current layer:
    - if the message contains another onion (or is flagged as not being a final message), it must be stripped of additional data and forwarded to the succeeding node
    - otherwise, act accordingly or disregard it

2.4 Tracker interaction mechanism

Firstly, a client must obtain the tracker's public key to be able to communicate with it. The key should be stored by the application to reduce the number of such interactions.

Secondly, a client must gather and store the current session details of the tracker (the time left until the next refresh and the refresh interval). Using these numbers, it must repeat at the correct timing the procedures outlined in the next sub-sections.

2.4.1 Announcing the tracker

Client announcing itself as relay node

The client will use the associated service to provide the tracker the opened port and the own public key. If the client's IP address appears in the list, the port and public key of the entry will be updated. Otherwise, the tracker will append the client's IP address, port and public key to the relay list. In both cases, the client's IP address is then sent a as a response / confirmation. The returned IP address should be stored by the client for future use.

Client announcing itself as a leecher

The client will use the associated service to provide the tracker the info hash of the swarm and (multiple) reply onions. The tracker will add them to the swarm's reply onion pool.

2.4.2 Scraping the tracker

The client will use the associated services to scrape the tracker for relay nodes or for leechers / seeders. For the latter, it must provide the swarm info hash and should filter the results of the own provided reply onions. We discuss more about filtering the created reply onions in the Conclusion.

2.5 Onions and reply onions

In all the following Algorithms, where we did not specifically mentioned the nature of a key (symmetric or asymmetric) or used the $Encr$ and $Decr$ notations, we did not intend to specify a cryptographic system, since any system that provides encryption and decryption of a message is supported.

To build an onion, Alice must have:

- a list of active volunteers (their addresses and public keys) that are willing to relay her onion
- a communication partner - Bob (B) - and his address and public key
- a message M to send

An $onions$'s structure must have following data:

- $nextNode$ - must contain relevant data for relaying the next onion to the next node in the circuit. Can be empty if the onion reached Alice's communication partner, Bob.
- $message$ - the message for Bob
- $encryptExternalPayload$ - a key to be used to encrypt the external payload (this field may be empty)
- $nextOnion$ - the next encrypted onion; the data that must be forwarded to the next node in the circuit. Can be empty if the onion reached Alice's communication partner, Bob.

A (transit) cell is composed of:

- $encryptedOnion$ - Example: $SymEncr_{Bob}(cell)$
- $externalPayload$ - may be empty or random

The steps of building a reply onion from Alice's perspective are presented in the Algorithm 1 from the Appendix section of this thesis.

Reply onion

To build a reply onion, Alice must have a list of active volunteers (their addresses and public keys) that are willing to relay an onion.

The pseudo-code of building a reply onion from Alice's perspective is presented in the Algorithm 2 from the Appendix section of this thesis.

The $currentOnion$ variable is ready to be used as a reply onion. The $next$ variable contains the first node of the reply circuit. The $encryptExternalPayload$ variable contains a key that the starter node of the reply circuit must use to encrypt the $externalPayload$ of the onion it is ought to relay. Alice must store the $arrayOfKeysId$ variable value and it's corresponding $arrayOfKeys$ to be able to decrypt a reply onion's external payload.

Reading messages and relying onions

To reply to Alice using the reply onion provided by her, Bob must use the Algorithm 4. Here, Bob's privacy is at risk, as Alice's reply onion might be not well-intended. To protect himself, he must designate a relay node of the network to start the reply circuit, as in Algorithm 5.

Lastly, the pseudo-code that a node (both Alice and Bob) must implement in order to relay onions and to be able to communicate is written in Algorithm 3 from the Appendix section of this thesis. In the "if" block at the line 11 it is shown how can Alice read the response Bob sent her.

2.6 Peer to peer communication protocol

The BitTorrent peer to peer communication method is using the TCP/IP protocol, and it firstly performs a handshake process between the peers and then the exchange of pieces begins. As our anonymous communication design is comparable to the UDP/IP protocol, a simplified communication method is needed. For a leecher to request pieces of a file from another peer, it must send to the network (directly or through a proxi) a reply onion obtained from the tracker with an external payload containing the list of needed pieces and a reply onion (used by the seeder to respond). The uploader will send the pieces it is willing to, in one or more messages.

Theoretically, a leecher can ask for the whole torrent files from a seeder, but it is impractical for him, as the download speed will decrease. The downloading client must individually optimize it's connections for the maximum download speed, and the uploading one must individually manage the maximum number of pieces offered in a response message, to prevent an observer from tracking any abnormal sized transit cell.

2.7 Creating and publishing

Publishing a new group of files only requires the creation of a ".torano" file and an announce operation for a given file stored locally. The Torano client should provide to the user the possibility of announcing the tracker and creating a ".tornao" file for a specific document.

Chapter 3

Threat model

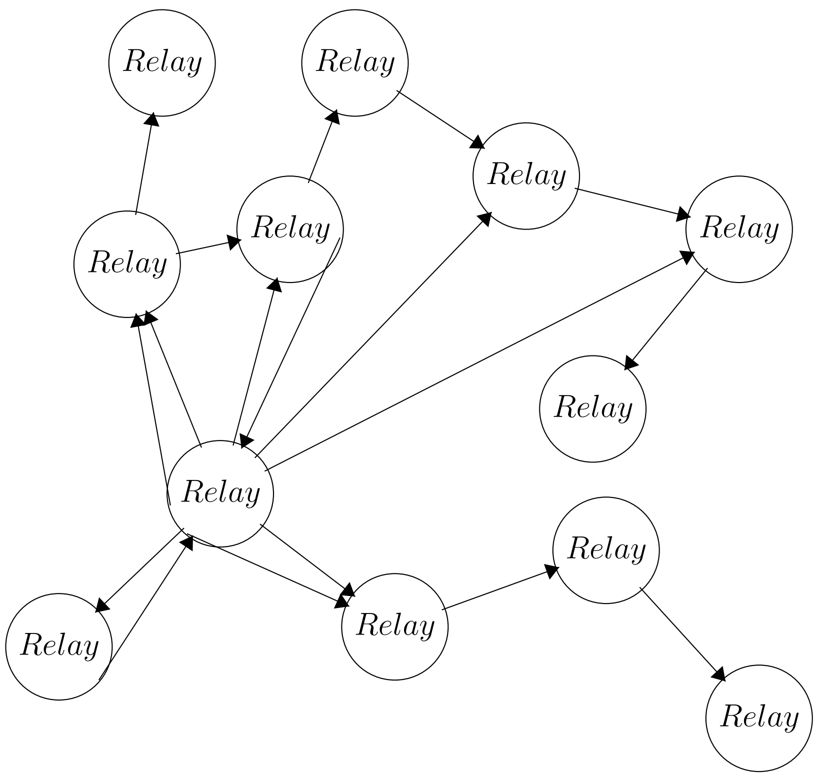
In this protocol, an attacker's purpose is usually to first select a specific endpoint and then to become able to confidently confirm it's association to a particular swarm.

To evaluate possible threats to the anonymity of a participant to the network, we have considered two types of attackers: a passive attacker that only observes the network traffic and an active attacker that in addition, has deployed corrupted nodes and corrupts legitimate nodes.

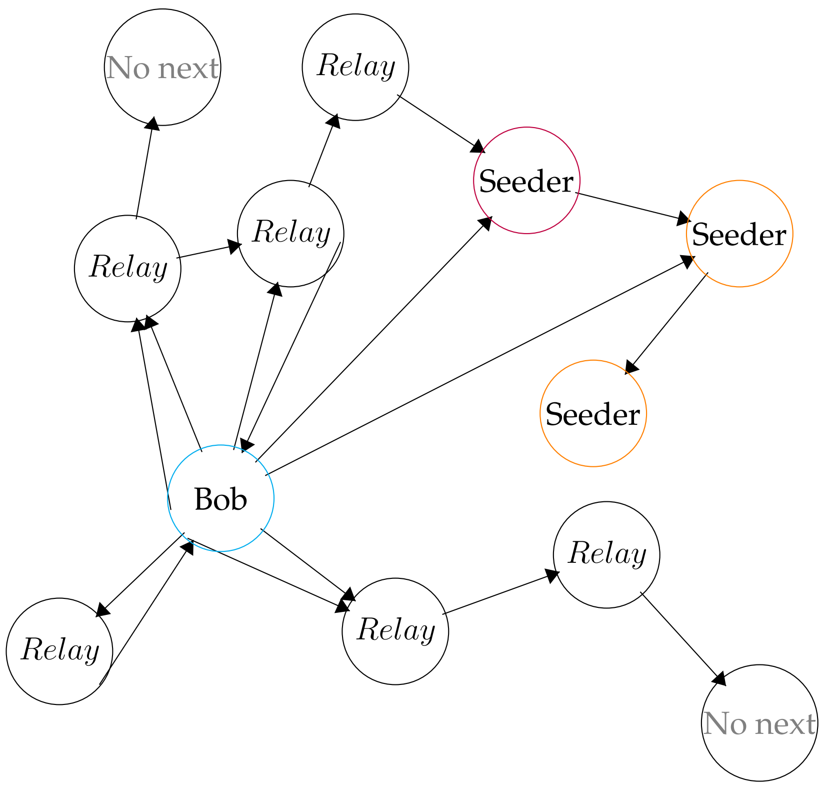
3.1 Passive attacker

The passive attacker is at most able to confirm that two endpoints exchanged messages, but as their content is encrypted, it is impossible for him to confidently accuse any of them for distributing files. The goal of the passive attacker is made more complicated as a Torano client is able to build disposable circuits for it's onions, meaning that a message can never be sent twice over the same path.

Figure 4.1 depicts the complexity of a network's state after all traffic unrelated to the potential suspect (Bob) was removed to his detriment. From the observer's perspective, we are unable to properly discover the leechers or seeders. Even so, it is impossible to group Bob and it's peers in swarms.



(a) Observer's perspective



(b) Reality - Bob performs in two swarms (The color of the seeder indicates the swarm)

Figure 3.1: A snapshot of the network having only Bob's traffic. Observer's perspective and the reality. A node labeled "No next" must be interpreted as "the next node in the onion's path is offline".

3.2 Active attacker

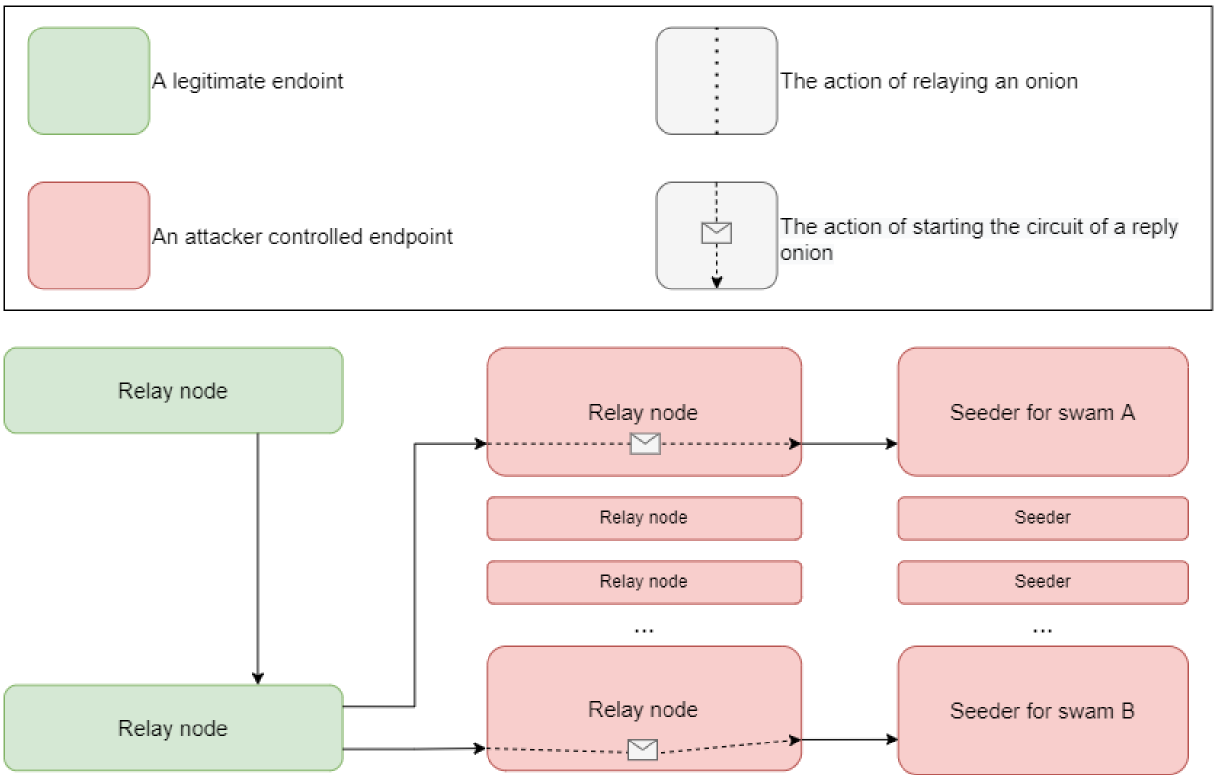
While an active attacker is a more challenging adversary, the Torano network still manages to make it's aim unattainable in real-world situations. As the tracker can only provide the reply onions for a particular swarm, an attacker can only partially discover their circuit in the short refresh period that the tracker fixed. In addition to that, a participant (Bob) can choose not to announce itself as a leecher (it only downloads) and because of that, there will be no reply onion that can be used to trace his address.

To more comprehensively prove this, let us demonstrate that Bob's identity can still be at most questionable in an absurd case, where the attacker controls an unrealistic proportion of the relay nodes in the network: an attacker overflows the tracker's memory with malicious relay nodes and malicious seeders (reply onions) for a certain torrent. We helped our attacker, by offering an information that Bob might be interested in torrent A's files. The scenario of Bob joining a swarm / network composed exclusively of corrupted endpoints is excluded, and we assume that the tracker is incorruptible.

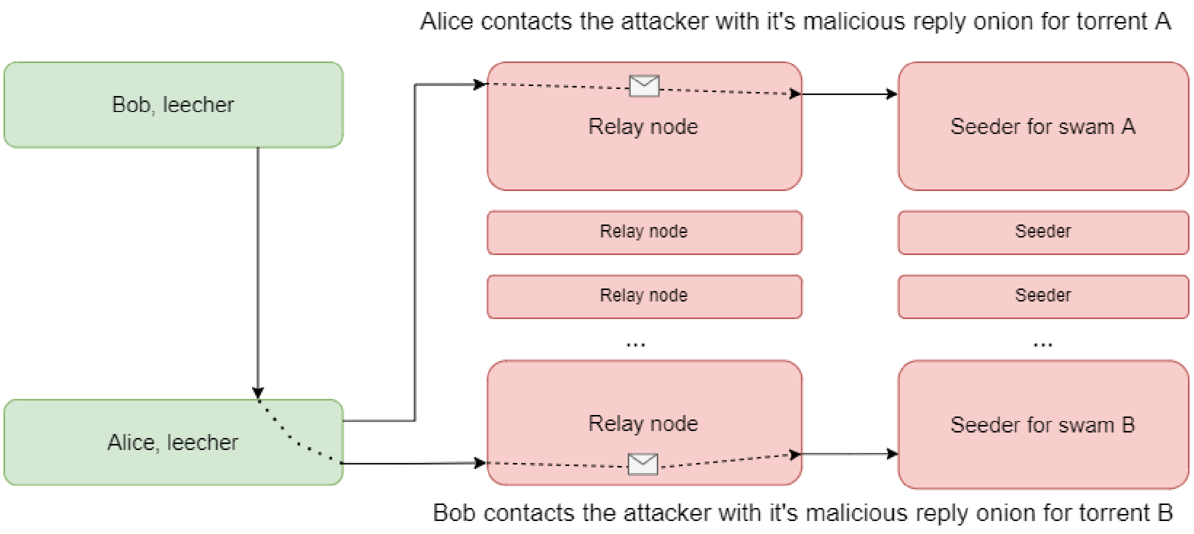
From the attacker's perspective we are unable to confidently identify each relay node's swarm. Even by delaying and correlating the timing of each packet, this ambiguity can be hardly reduced.

Altering any cell's contents or will determine the relay nodes to reject that specific cell. In this regard, an attacker can perform denial of service attacks on the network. However, the task of selecting an endpoint to determine it's swarm is difficult, as can be seen in the Observer's perspective in Figure 3.1.

The tracker refresh time also discourages the attacker, as any progress to building a map of the network will be lost once all the traffic will be carried on different routes. Even so, by capturing an onion and corrupting all the nodes in it's path, the public keys used to encrypt the data might have already been changed.



(a) Attacker's perspective


Alice contacts the attacker with it's malicious reply onion for torrent A
Bob, leecher
Relay node
Seeder for swam A
Relay node
Seeder
Relay node
Seeder
...
...
Alice, leecher
Relay node
Seeder for swam B
Bob contacts the attacker with it's malicious reply onion for torrent B

(b) Reality - Alice is associated to swarm A, Bob is associated to swarm B

Figure 3.2: A snapshot of the network having only Alice's and Bob's traffic. Active attacker's perspective and the reality.

Chapter 4

The Torano design implementation

We tested 7 Torano clients paired with a Torano tracker on a computer using the Windows 11 operating system. The network behaved as expected. We have also tried running the application on Windows Subsystem for Linux, the core functionalities are working, but the graphics used in buttons appeared corrupted. Our aim was to prove the functionality of the protocol. The readers might find numerous possible optimizations to the source code.

Any variable mentioned in this thesis or in any figure that can be referred to as a ″structure″ is implemented as dictionary (or JSON) with each field named as presented.

For our encryption algorithms, we have used RSA with a key size of 2048 bits and AES, with a key size of 256 bits.

4.1 Document distribution. The metainfo file.

As recommended, existing ″.torrent″ creation and parsing tools were used. We changed the default tracker′s addresses to our proprietary tracker′s address.

After parsing a metainfo file, the client application will have the following information:

- the tracker′s IP address and port
- the file name and size (length)
- the number of pieces the file has be split into
- the piece size and the last piece size
- the SHA1 hash of each piece

- the info hash of the swarm

4.2 User experience

As shown in the C4 diagram, we believe we successfully managed to replicate the exact user experience of the BitTorrent protocol, as a user will interact with the following:

- ".torano" files
- his Torano client

To start downloading

1. Install the Torano client, or have actually done that before
2. Find a ".torano" file and copy it locally
3. Open it with the Torano client
4. Select where to save the files locally
5. Wait for the client to finish downloading

To start relaying

1. Install the Torano client, or have actually done that befoe
2. Find a ".torano" file and copy it locally
3. Open it with the Torano client
4. Select where to save the files locally
5. Delete the download

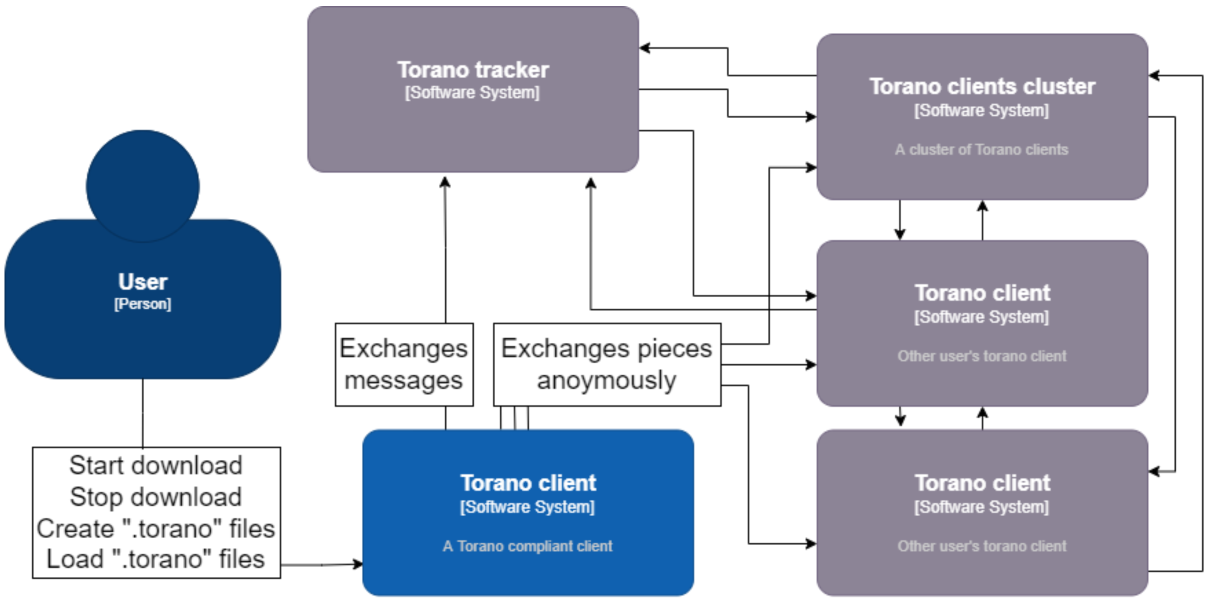


Figure 4.1: The C4 (level 1) diagram of the Torano protocol

4.3 Core entities

4.3.1 The tracker

In the current implementation, it is a HTTP server running on port 6969 (as the BitTorrent trackers do) that exposes the following routes:

- POST `/announce/relay` - announce as relay node service
- POST `/announce` - announce as leecher service
- POST `/scrape/relay` - distribution of relay nodes service
- POST `/scrape` - distribution of leechers service
- POST `/session` - session timing details service
- GET `/public-key` - distribution of the public key

4.3.2 The client

In the current implementation, it is a HTTP server running on an available port (as the BitTorrent clients do) that exposes the following routes:

- POST `/relay` - onion routing handler service
- GET `/publicKey` - public key distribution service

In order to integrate the user interface with the core functionalities, in our application we added more routes. As these are not part of the protocol specification, they will not be explicitly mentioned or described.

The following diagrams are illustrating the Torano client's structure and the most important components of the application.

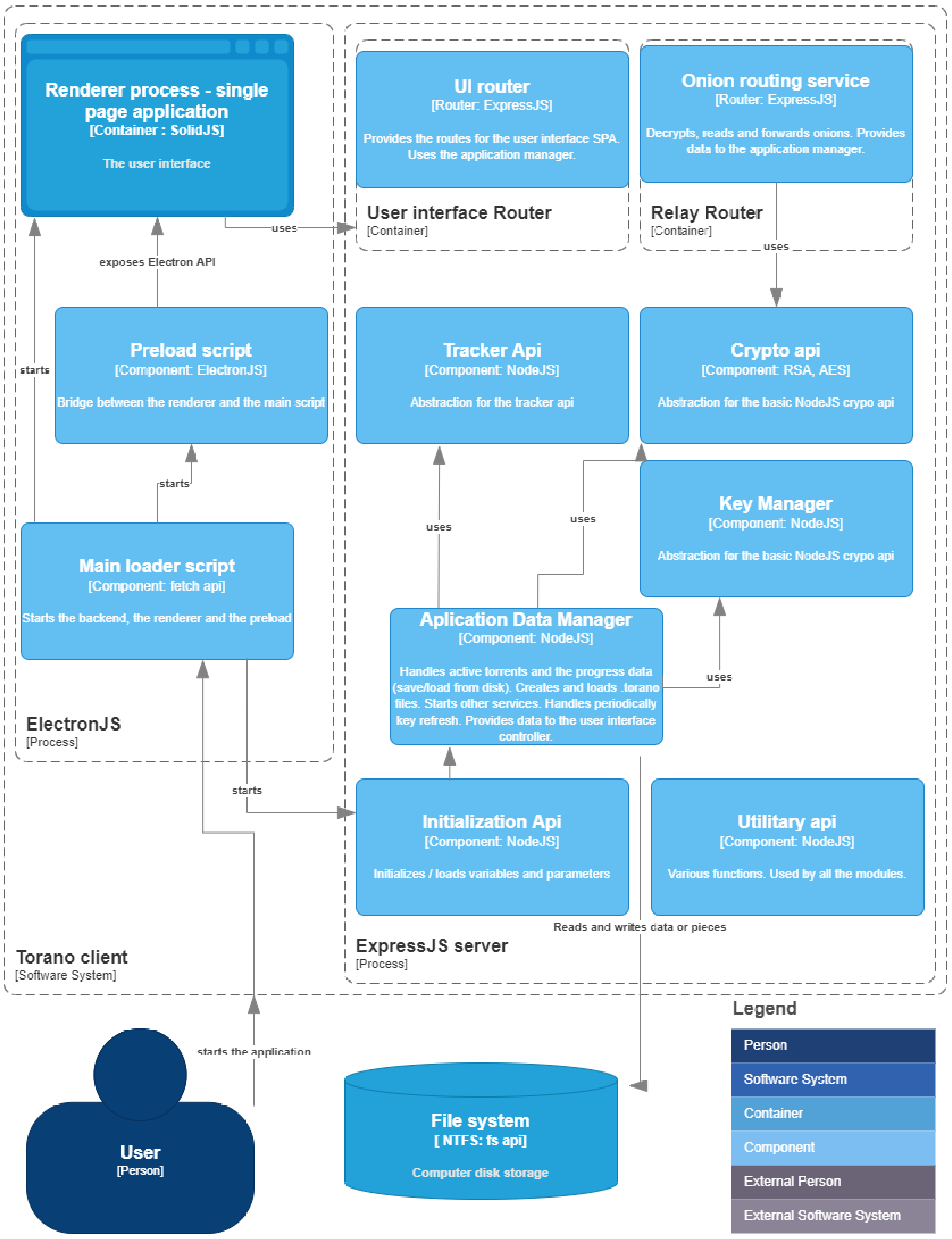


Figure 4.2: The C4 (level 2) diagram of the Torano client

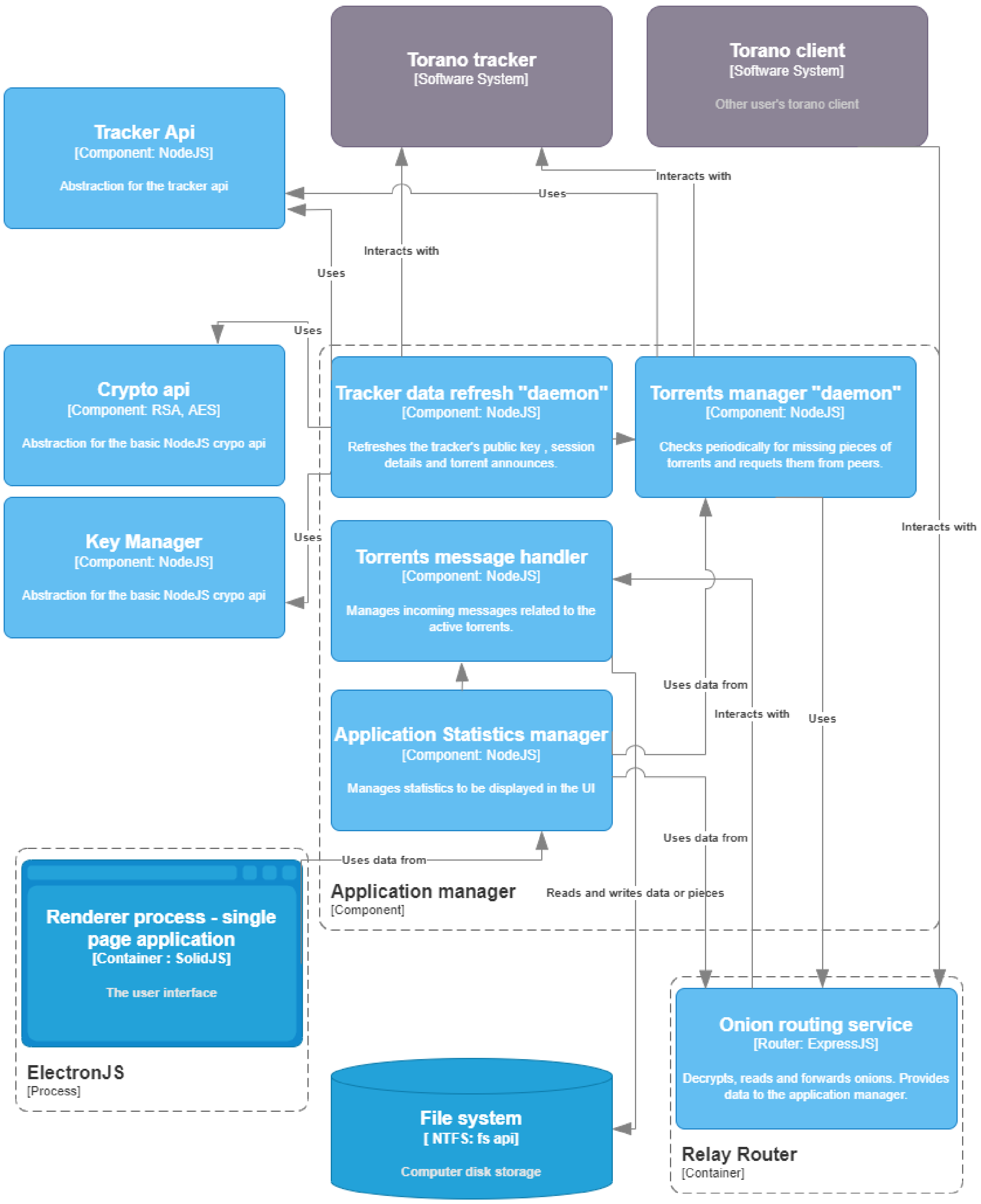

Torano tracker
[Software System]
Torano client
[Software System]
Other user's torano client
Tracker Api
[Component: NodeJS]
Abstraction for the tracker api
Interacts with
Uses
Interacts with
Uses
Uses
Crypto api
[Component: RSA, AES]
Abstraction for the basic NodeJS crypo api
Tracker data refresh "daemon"
[Component: NodeJS]
Refreshes the tracker's public key , session details and torrent announces.
Torrents manager "daemon"
[Component: NodeJS]
Checks periodically for missing pieces of torrents and requets them from peers.
Key Manager
[Component: NodeJS]
Abstraction for the basic NodeJS crypo api
Uses
Interacts with
Torrents message handler
[Component: NodeJS]
Manages incoming messages related to the active torrents.
Uses data from
Interacts with
Uses
Application Statistics manager
[Component: NodeJS]
Manages statistics to be displayed in the UI
Uses data from
Uses data from
Application manager
[Component]
Reads and writes data or pieces
Renderer process - single page application
[Container : SolidJS]
The user interface
ElectronJS
[Process]
File system
[NTFS: fs api]
Computer disk storage
Onion routing service
[Router: ExpressJS]
Decrypts, reads and forwards onions. Provides data to the application manager.
Relay Router
[Container]

Figure 4.3: The C4 (level 3) diagram of the most important components

4.4 Tracker interaction mechanism

The following c4 (level 4) diagrams present the procedure and data structures used when a client interacts with the tracker.

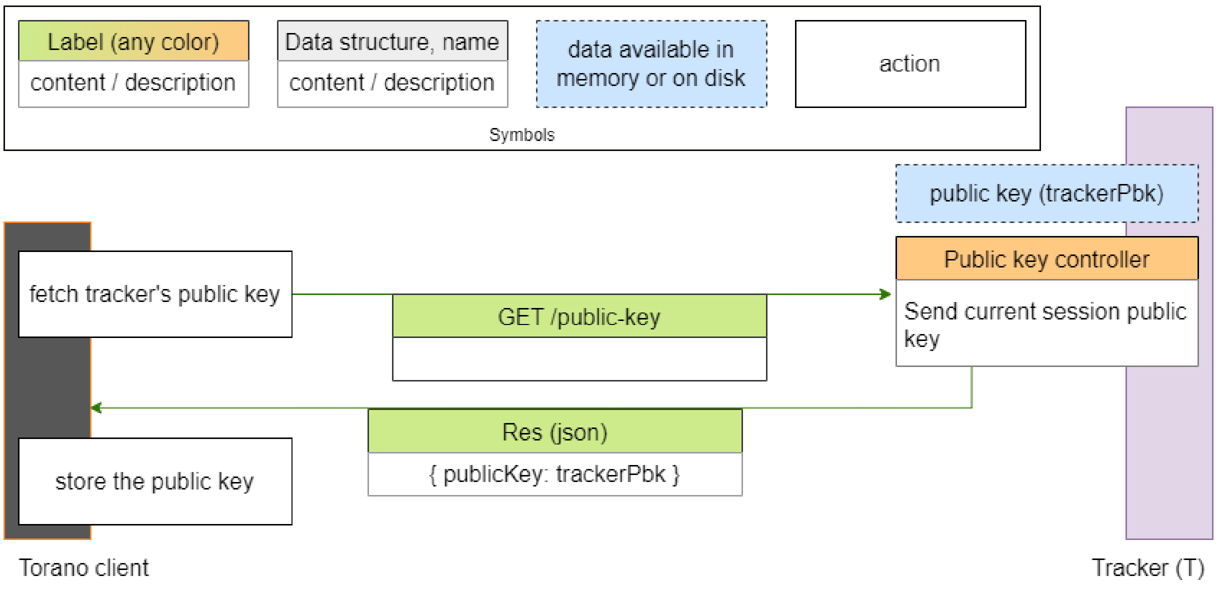


Figure 4.4: The C4 (level 4) diagram of a client fetching tracker's public key

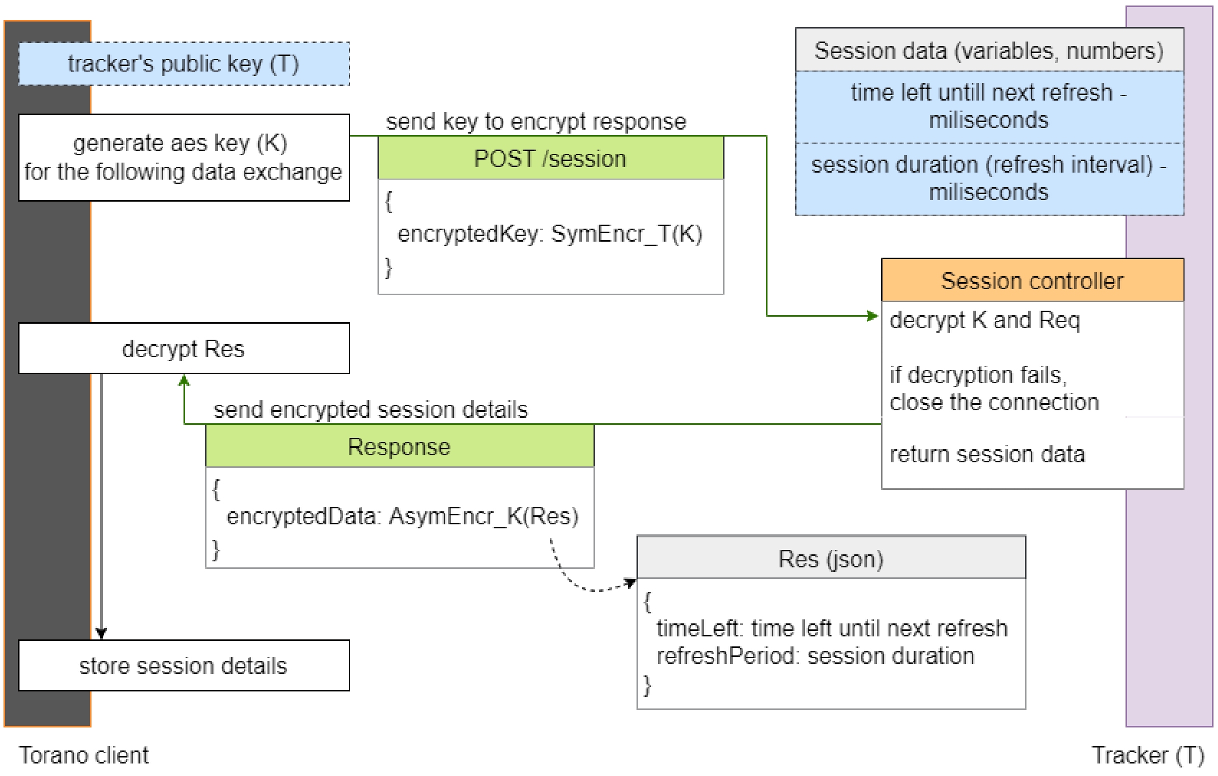
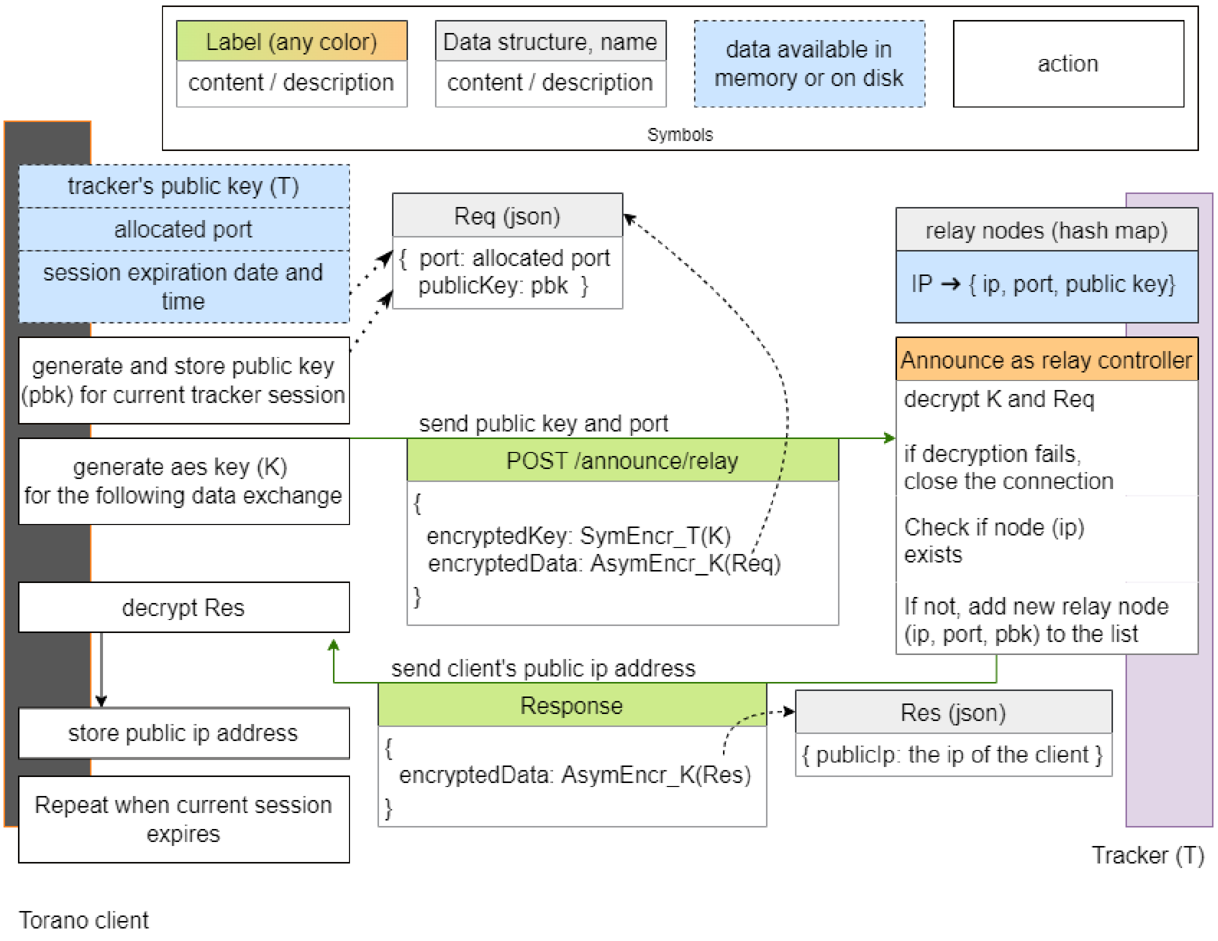

tracker's public key (T)
generate aes key (K) for the following data exchange
send key to encrypt response
POST /session
{ encryptedKey: SymEncr_T(K) }
Session data (variables, numbers)
time left untill next refresh - miliseconds
session duration (refresh interval) - miliseconds
Session controller
decrypt K and Req
if decryption fails, close the connection
return session data
decrypt Res
send encrypted session details
Response
{ encryptedData: AsymEncr_K(Res) }
Res (json)
{ timeLeft: time left until next refresh refreshPeriod: session duration }
store session details
Torano client
Tracker (T)

Figure 4.5: The C4 (level 4) diagram of a client fetching tracker's session details

4.4.1 Announcing the tracker

Client announcing itself as relay node



Label (any color)
content / description
Data structure, name
content / description
data available in memory or on disk
action
Symbols
tracker's public key (T)
allocated port
session expiration date and time
generate and store public key (pbk) for current tracker session
generate aes key (K) for the following data exchange
decrypt Res
store public ip address
Repeat when current session expires
Torano client
Req (json)
{ port: allocated port publicKey: pbk }
send public key and port
POST /announce/relay
{ encryptedKey: SymEncr_T(K) encryptedData: AsymEncr_K(Req) }
send client's public ip address
Response
{ encryptedData: AsymEncr_K(Res) }
Res (json)
{ publicIp: the ip of the client }
relay nodes (hash map)
IP ➔ { ip, port, public key}
Announce as relay controller
decrypt K and Req
if decryption fails, close the connection
Check if node (ip) exists
If not, add new relay node (ip, port, pbk) to the list
Tracker (T)

Figure 4.6: The C4 (level 4) diagram of a client announcing itself as relay node

Client announcing itself as a leecher

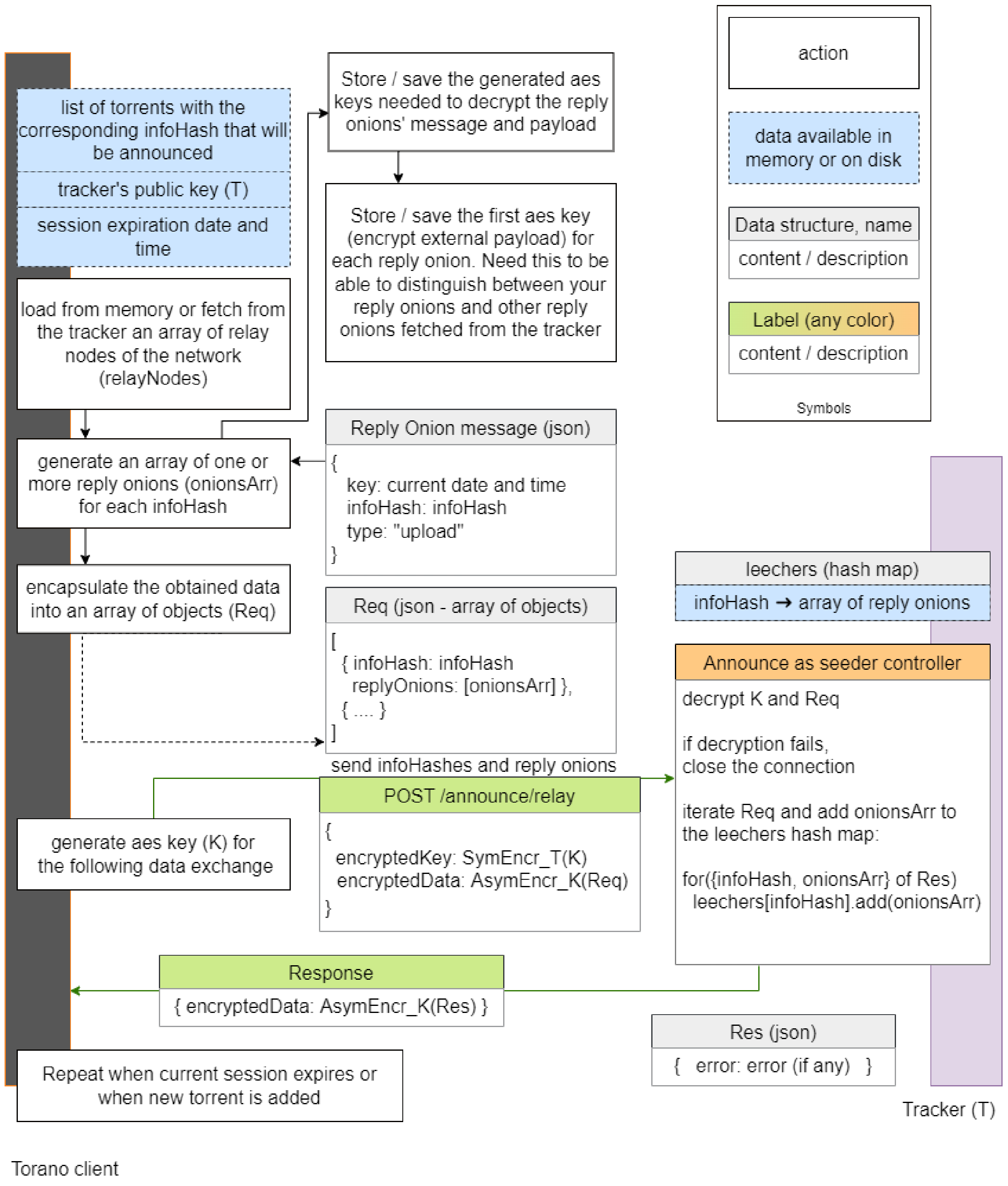

action
data available in memory or on disk
Data structure, name
content / description
Label (any color)
content / description
Symbols
list of torrents with the corresponding infoHash that will be announced
tracker's public key (T)
session expiration date and time
load from memory or fetch from the tracker an array of relay nodes of the network (relayNodes)
generate an array of one or more reply onions (onionsArr) for each infoHash
encapsulate the obtained data into an array of objects (Req)
generate aes key (K) for the following data exchange
Repeat when current session expires or when new torrent is added
Torano client
Store / save the generated aes keys needed to decrypt the reply onions' message and payload
Store / save the first aes key (encrypt external payload) for each reply onion. Need this to be able to distinguish between your reply onions and other reply onions fetched from the tracker
Reply Onion message (json)
{ key: current date and time infoHash: infoHash type: "upload" }
Req (json - array of objects)
[{ infoHash: infoHash replyOnions: [onionsArr] }, { .... }]
send infoHashes and reply onions
POST /announce/relay
{ encryptedKey: SymEncr_T(K) encryptedData: AsymEncr_K(Req) }
Response
{ encryptedData: AsymEncr_K(Res) }
leechers (hash map)
infoHash ➜ array of reply onions
Announce as seeder controller
decrypt K and Req
if decryption fails, close the connection
iterate Req and add onionsArr to the leechers hash map:
for({infoHash, onionsArr} of Res)
leechers[infoHash].add(onionsArr)
Res (json)
{ error: error (if any) }
Tracker (T)

Figure 4.7: The C4 (level 4) diagram of a client announcing itself as a leecher

4.4.2 Scraping the tracker

Client scraping leechers

In our implementation, we opted to filter the creator's reply onions by the ncyptExternalPayload field (see section 4.5, Onions and reply onions).

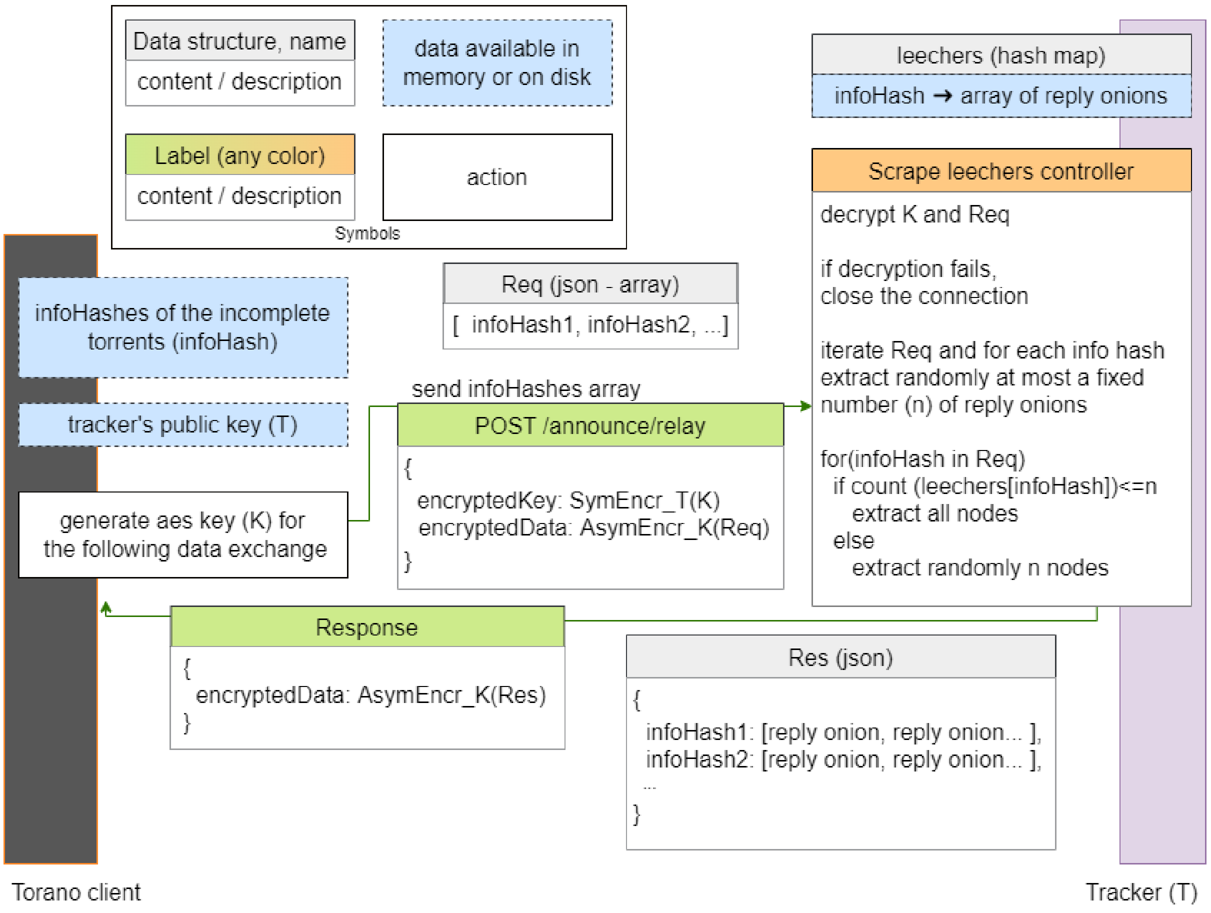

Data structure, name
content / description
data available in memory or on disk
Label (any color)
content / description
action
Symbols
leechers (hash map)
infoHash ➜ array of reply onions
Scrape leechers controller
decrypt K and Req
if decryption fails, close the connection
iterate Req and for each info hash extract randomly at most a fixed number (n) of reply onions
for(infoHash in Req)
if count (leechers[infoHash])<=n
extract all nodes
else
extract randomly n nodes
infoHashes of the incomplete torrents (infoHash)
Req (json - array)
[infoHash1, infoHash2, ...]
send infoHashes array
tracker's public key (T)
POST /announce/relay
{ encryptedKey: SymEncr_T(K) encryptedData: AsymEncr_K(Req) }
generate aes key (K) for the following data exchange
Response
{ encryptedData: AsymEncr_K(Res) }
Res (json)
{ infoHash1: [reply onion, reply onion...], infoHash2: [reply onion, reply onion...], ... }
Torano client
Tracker (T)

Figure 4.8: The C4 (level 4) diagram of a client scraping leechers

Client scraping relay nodes

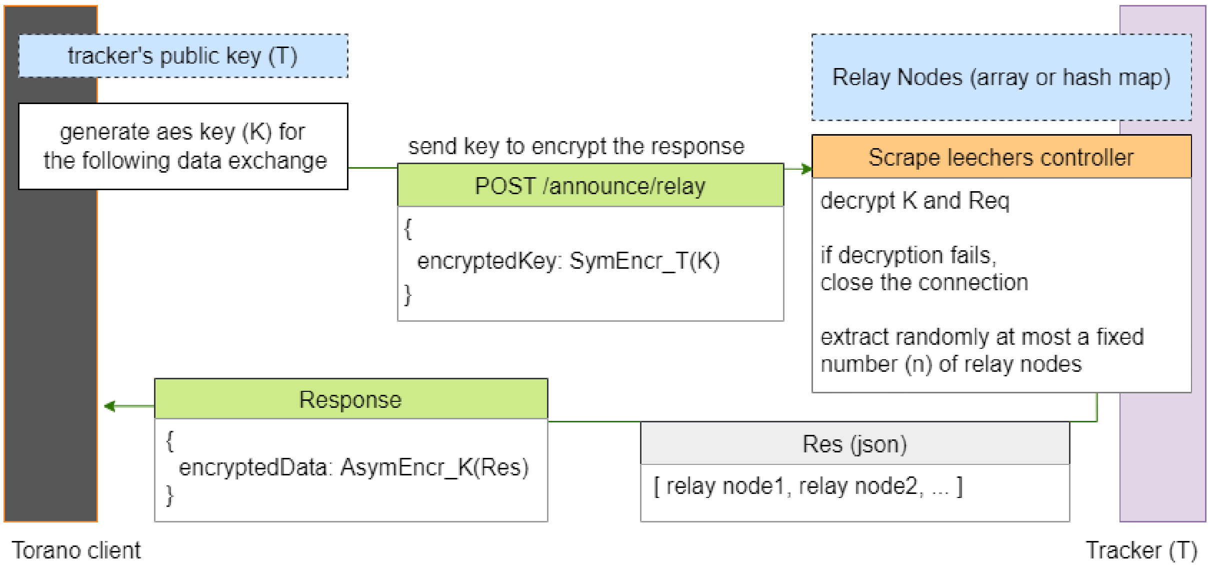

tracker's public key (T)
Relay Nodes (array or hash map)
generate aes key (K) for the following data exchange
send key to encrypt the response
Scrape leechers controller
POST /announce/relay
{ encryptedKey: SymEncr_T(K) }
decrypt K and Req
if decryption fails, close the connection
extract randomly at most a fixed number (n) of relay nodes
Response
{ encryptedData: AsymEncr_K(Res) }
Res (json)
[relay node1, relay node2, ...]
Torano client
Tracker (T)

Figure 4.9: The C4 (level 4) diagram of a client scraping relay nodes

4.5 Onions and reply onions

A cell or transit cell is a JSON containing:

- onion (aes encrypted onion for the next node)
- encryptedAesKey (the aes key to decrypt the onion. this key is encrypted with the public key of the next node)
- (optional) externalPayload (the external payload of the onion)

An onion is dictionary having the following attributes:

- message (if the message field is missing, the onion must be relayed)
- next:
  - ip (of the next node of the path)
  - port (of the next node of the path)
  - encryptedAesKey (aes key encrypted with the public key of the next node of the path)
- onionLayer (aes key encrypted onion layer for the next node of the path)
- (optional) encryptExternalPayload (aes key used to encrypt the cell's external-Payload data)

Reply onions

In our current implementation, the reply onion is a multiply encrypted onion, having this custom message:

- key (a value that corresponds to the array of encryptExternalPayload keys)
- infohash (a value used to identify the torrent this reply onion is addressed for)
- type ("upload" or "pieces" - the first meaning the reply onion's external payload contains an array of indexes of pieces that should be uploaded to a leecher, and the latter meaning that the reply onion's external payload contains pieces of a torrent requested by the reader)

From the application's perspective, a reply onion is a JSON:

- encryptedAesKey (aes key encrypted with the public key of the next node of the path)

- onion (aes encrypted onion for the next node of the path)
- port (port of the next node)
- ip (ip of the next node)
- encyptExternalPayload (aes key that must be used to encrypt a payload before relaying it to the next cell)

4.6 Peer to peer communication protocol

To request pieces of a file, having the info hash and a list of reply onions, a client must prepare a specific message (see the ″Message to request piece″ data structure in the figure below). It must be a dictionary with two attributes: requestPieces, containing the array of pieces indexes that the node requested and replyOnion, a dictionary as illustrated in the last paragraph of the Section 4.5. This message will be placed (after the encryption with encyptExternalPayload key of the reply onion) in the transit cell's externalPayload field.

When a seeder receives such a reply onion with the message type ″upload″ and a external payload as described in the previous paragraph, it can respond with the available pieces, in an array of dictionaries (see the ″Message to upload pieces″ data structure in the figure below), that will be encypted and attached as the ″externalPayload″ of the reply onion.

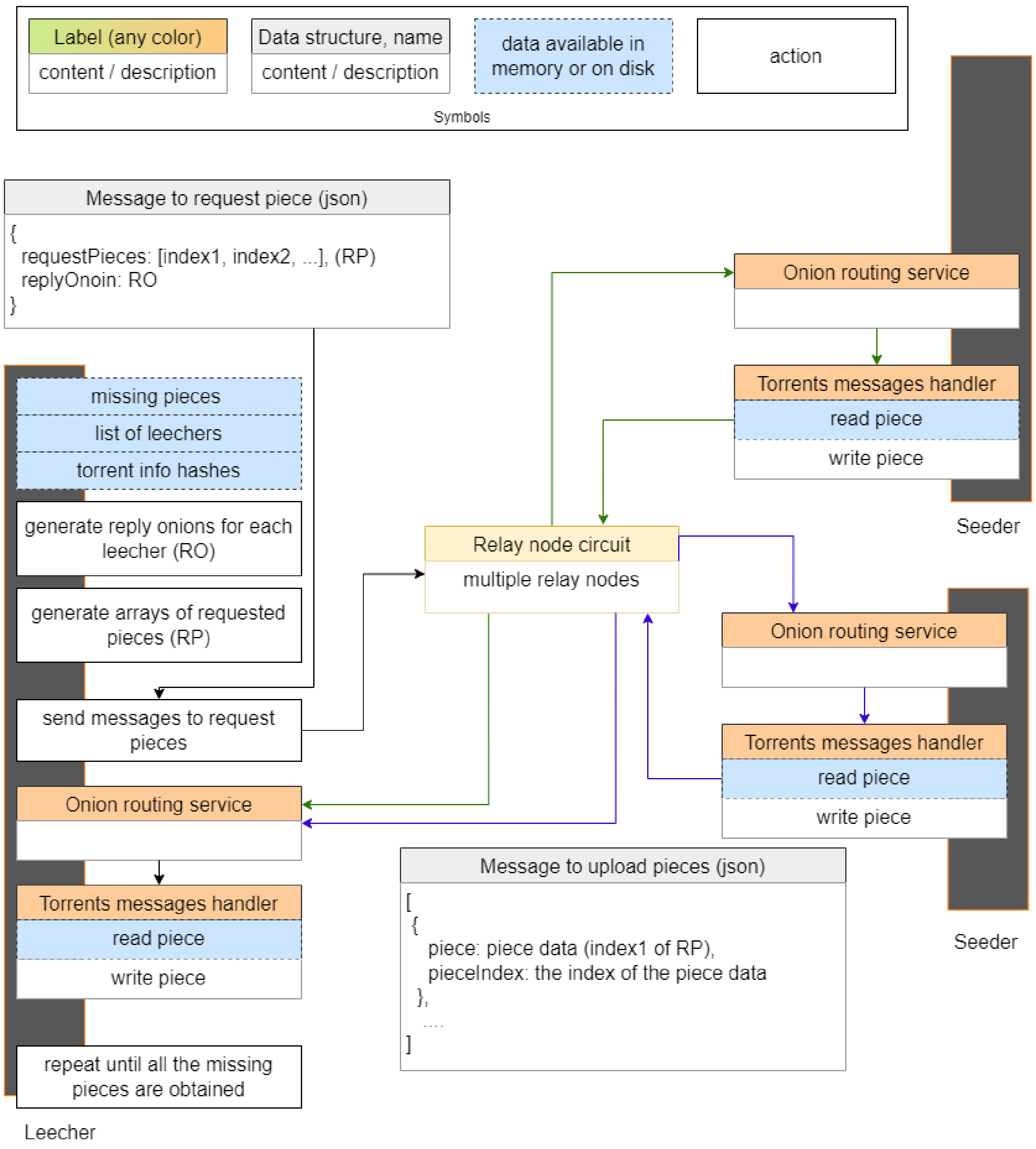
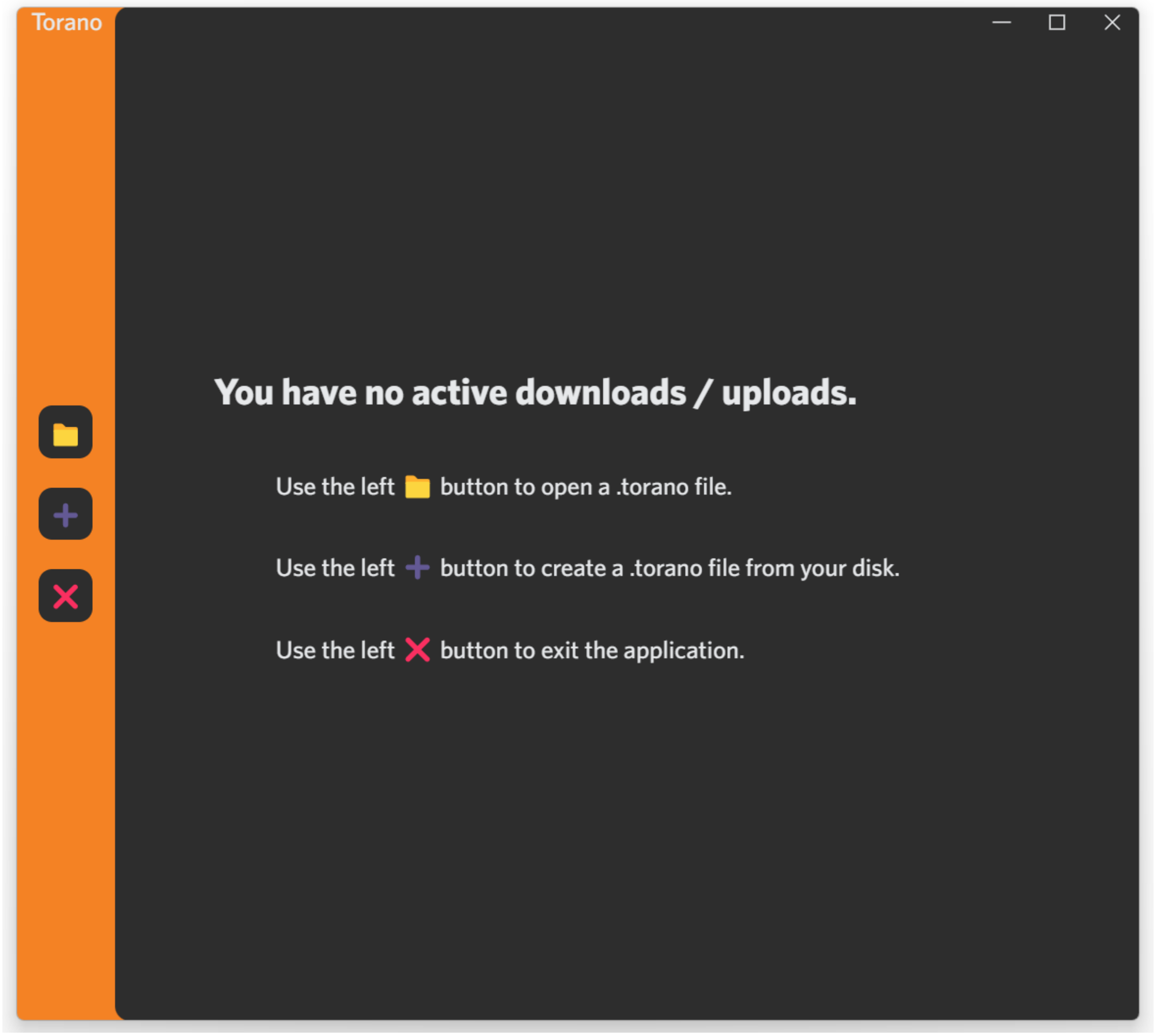

Label (any color)
content / description
Data structure, name
content / description
data available in memory or on disk
action
Symbols
Message to request piece (json)
{
requestPieces: [index1, index2, ...], (RP)
replyOnoin: RO
}
missing pieces
list of leechers
torrent info hashes
generate reply onions for each leecher (RO)
generate arrays of requested pieces (RP)
send messages to request pieces
Onion routing service
Torrents messages handler
read piece
write piece
repeat until all the missing pieces are obtained
Leecher
Relay node circuit
multiple relay nodes
Onion routing service
Torrents messages handler
read piece
write piece
Seeder
Onion routing service
Torrents messages handler
read piece
write piece
Seeder
Message to upload pieces (json)
[
{
piece: piece data (index1 of RP),
pieceIndex: the index of the piece data
},
....
]

Figure 4.10: The C4 (level 4) diagram of multiple clients exchanging pieces

4.7 Creating and publishing

To publish some locally stored files, the user must press the middle button from the left bar. The application will ask the user to select the file that should be published in the network, will create a ".torano" file, will begin seeding the file and will prompt the user to provide a location where the created metainfo file will be saved.



You have no active downloads / uploads.

Use the left button to open a .torano file.

Use the left button to create a .torano file from your disk.

Use the left button to exit the application.

Conclusions

This thesis presented the Torano peer-to-peer file sharing protocol and justifies how is anonymity of the peers maintained.

We have successfully designed and implemented an application for the internet society and for future search, that utilizes a familiar user experience with the already very popular BitTorrent protocol. We have established in the Related Work section that people are actively looking for such protocols and / or applications and that there are not many usable alternatives. As there is potential for a substantial mass adoption, we have described our security measures and illustrated our future users the attacker's perspective to earn their trust.

Moreover, we have stated some guidelines with the "should" keyword, to stimulate further research activities and curiosity of the readers for possible optimizations.

Can this be a "neutral" or "good" anonymity protocol? We believe that the majority of our target audience is mostly harmless, already distributing inoffensive and even educative media content, books, courses etc. This protocol can improve the reach of such large files in restricted areas of the world, to the benefit of the viewers' knowledge.

Future work

For the possible future research work, we have considered at least the following:

Client application improvements

Our client application only implements the relaying, onion and reply onion creation and handling, the core logic of the protocol. We will carefully study if we can implement the feature of starting of a reply circuit of an onion in future versions of the application. The latency and security of the whole network in our application can be improved by removing the HTTP layer entirely and by encoding an onion's data more efficiently. Because the HTTP headers might not be encrypted, an observer is able to see the route of the tracker accessed by a node.

Moreover, in order to better accomplish the adoption goal, future versions of the application will allow the following settings to be configurable by an user:

1. the maximum bandwidth allowed
2. the anonymity level:
   (a) the number of announce messages sent to the tracker per torrent
   (b) the number of seeders to contact per torrent
   (c) the number of leechers allowed per torrent
   (d) the circuit length
3. the number of torrents allowed to be active (seeding and / or leeching) at the same time
4. the refresh period of the announce messages (this must be shorter or equal to the one required by the tracker)
5. the maximum number of pieces sent per message

We could also have mentioned in the list ″the maximum size of a cell to be processed″ but this configurable parameter would directly affect the users who chose to build longer circuits. Hence it can also negatively impact the stability and latency of the entire network by overloading it with prone to be rejected cells, we consider that the benefits of this option currently do not outbalance it's disadvantages. Until it is proven otherwise, we remain reticent to recommending such a filter for a Torano cell.

Regardless of the fact that the source code already supports some of the mentioned improvements, it's current focus is to prove the main principle of the system's anonymity, and to provide concrete examples of the critical abstract notions of the protocol hereby stated.

Other protocol designs

Trackerless design

Some BitTorrent improvements suggests that a ″trackerless″ metainfo file can be used: besides the torrent name, pieces and pieces hashes, the metainfo file can contain a list of peers that act as a tracker for that torrent, further decentralizing the protocol. We are looking forward to implementing this ”trackerless” design both for our ”.torano” files and for the relay nodes, to remove the current tracker implementation as a stand alone entity.

Tracker as the main relay node

Some anonymity service was presented in various papers as The Anonymizer. This project has no official specification and the website (http://anonymizer.com) is unavailable.

As opposed to a ”trackerless design”, the main idea of this project was to route all the possible traffic through one node (hence the name), offering the users anonymity among untraceable high traffic.

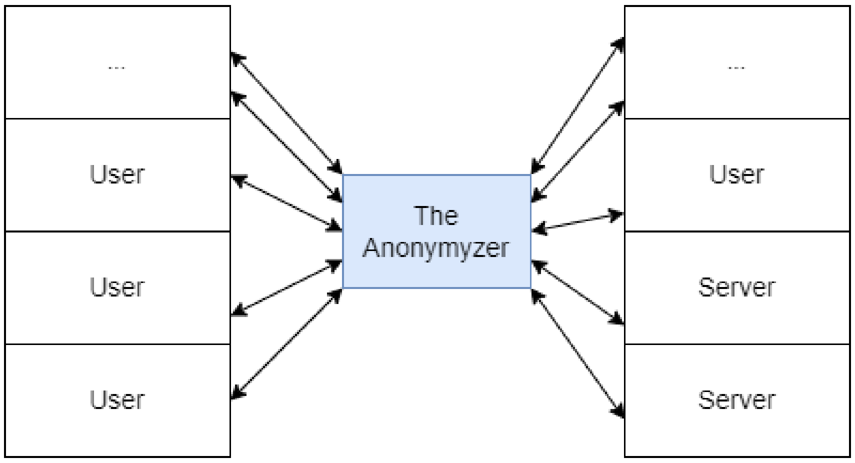


Figure 4.11: The Anonymyzer illustration

Assuming that the necessary improved hardware capabilities are not an issue, this approach would further centralize the protocol and would make the tracker a perfect target for the attackers to corrupt.

Dummy traffic with own reply onions

When building request piece messages, one can also pack them to his own reply onions 2.4.2 to further obfuscate traffic. This approach would only be viable if the nodes would send dummy traffic periodically between them. Otherwise, an observer can notice the temporary increase of network traffic that passes a node, traffic caused by the sending own reply onions to the network.

Appendix 1

```
Data: a list of relay nodes; the recipient - Bob; a message M
$lastOnion \leftarrow empty\ Onion$;
$lastOnion.message \leftarrow M$;
$previousOnion \leftarrow lastOnion$;
append $Bob$ at the beginning of the $relayNodesList$ ;
$relayNodesListLength \leftarrow$ length($relayNodesList$) ;
for $i \leftarrow 1$ to $relayNodesListLength - 1$ // skip last node
do
    $relayNode \leftarrow relayNodesList[i]$ ;
    $currentOnion \leftarrow empty\ Onion$ ;
    $currentOnion.message \leftarrow$ 'forward to next' ;
    $currentOnion.nextOnion \leftarrow SymEncr_{relayNode}(previousOnion)$ ;
    $currentOnion.next \leftarrow relayNode$ ;
    $currentOnion.encryptExternalPayload \leftarrow$ new encryption key
        // optional
    $previousOnion \leftarrow currentOnion$ ;
$cell \leftarrow$ empty Cell ;
$relayNode \leftarrow relayNodesList[relayNodesListLength]$ // the last node
$cell.encryptedOnion \leftarrow SymEncr_{relayNode}(previousOnion)$ ;
$cell.externalPayload \leftarrow data$ // optional
// the $cell$ variable is now ready to be securely sent to
    the node from the $previousOnion.next$ variable
```

Algorithm 1: How to create an onion

Appendix 2

```
Data: a list of relay nodes
append $Alice$ at the end of the $relayNodesList$ ;
$relayNodesListLength \leftarrow$ length($relayNodesList$) ;
generate an array of keys ($arrayOfKeys$) for encrypting the $externalPayload$.
  Must be of equal length to the list of relay nodes ;
generate a unique id for the $arrayOfKeys$ variable ($arrayOfKeysId$) ;
store the tuple ($arrayOfKeysId$, $arrayOfKeys$) $lastOnion \leftarrow empty\ onion$;
$lastOnion.message \leftarrow arrayOfKeysId$;
$previousOnion \leftarrow lastOnion$;
for $i \leftarrow relayNodesListLength$ to $2$ // skip the first node
 do
    $relayNode \leftarrow relayNodesList[i]$ ;
    $currentOnion \leftarrow empty\ Onion$ ;
    $currentOnion.message \leftarrow$ 'forward to next' ;
    $currentOnion.nextOnion \leftarrow SymEncr_{relayNode}(previousOnion)$ ;
    $currentOnion.next \leftarrow relayNode$ ;
    $currentOnion.encryptExternalPayload \leftarrow$
      $arrayOfKeys[relayNodesListLength - i + 1]$ ;
    $previousOnion \leftarrow currentOnion$ ;
$relayNode \leftarrow relayNodesList[1]$ // the first node
$currentOnion \leftarrow SymEncr_{relayNode}(previousOnion)$ ;
$encryptExternalPayload \leftarrow arrayOfKeys[relayNodesListLength]$ ;
$next \leftarrow previousOnion$ ;
```

Algorithm 2: How to create a reply onion

Appendix 3

```
Data: $incomingTransitOnion$ - an onion received from a relay node
1  $decryptedOnion \leftarrow SymDecr_{Alice}(incomingTransitOnion.cell)$ ;
2  // if the $SymDecr$ operation fails, the onion is ignored
3  $externalPayload \leftarrow incomingTransitOnion.externalPayload$ ;
4  if $decryptedOnion.message$ = 'forward to next' then
5      if $\exists$ $decryptedOnion.encryptExternalPayload$ then
6          $key \leftarrow decryptedOnion.encryptExternalPayload$ ;
7          $externalPayload \leftarrow Encr_{key}(externalPayload)$
8      $cell \leftarrow$ empty Cell ;
9      $cell.encryptedOnion \leftarrow decryptedOnion.nextOnion$ ;
10     $cell.externalPayload \leftarrow externalPayload$ ;
11     send $cell$ to $decryptedOnion.next$ ;
12 else if $decryptedOnion.message$ is flagged as a reply onion message then
13     // retrieve the tuple ($decryptedOnion.message$, $arrayOfKeys$)
           that has been stored.  See Algorithm 2 at line 5
14     if $\exists$ ($decryptedOnion.message$, $arrayOfKeys$) then
15         for $key$ in $arrayOfKeys$ do
16             $externalPayload \leftarrow Decrypt_{key}(externalPayload)$;
17         delete ($decryptedOnion.message$, $arrayOfKeys$) from storage
18     else
19         ignore this onion
20     // $externalPayload$ now contains Bob's message for Alice
21 else
22     read the onion's message and act accordingly
```

Algorithm 3: How to relay an onion

Appendix 4

```
Data: $replyOnion$ - the $currentOnion$ of the previous algorithm,
      $encryptExternalPayload$ - an the first encryption key provided in the
      reply onion, $next$ - the first node in the reply circuit, $Message$ - a message
      to Alice
1 $cell \leftarrow$ empty Cell ;
2 $cell.encryptedOnion \leftarrow replyOnion$ ;
3 $cell.externalPayload \leftarrow Encr_{encryptExternalPayload}(Message)$ ;
4 send the obtained $cell$ to $next$
```

Algorithm 4: How can Bob reply to Alice

Appendix 5

```
Data: $replyOnion$ - the $currentOnion$ of the previous algorithm,
      $encryptExternalPayload$ - an the first encryption key provided in the
      reply onion, $next$ - the first node in the reply circuit, $Message$ - a message
      to Alice
1 $cell \leftarrow$ empty Cell ;
2 $cell.encryptedOnion \leftarrow replyOnion$ ;
3 $cell.externalPayload \leftarrow Encr_{encryptExternalPayload}(Message)$ ;
4 create message containing the obtained $cell$ and the next node information
    from the variable $next$
5 flag the message as being a reply onion that must be sent as is to $next$
6 gather a list of active relay nodes of the network
7 build an onion using the Algorithm 1 with the list created at line 6, the
    responder being the last node of the list and the message being the one
    created at line 4
8 send the created onion to the first node of it' path ($previousOnion.next$)
```

Algorithm 5: How Bob can message a relay node to start the reply circuit of an onion

Bibliography

[1] arama. Bittorrent over Tor isn't a good idea. 2010. https://blog.torproject.org/bittorrent-over-tor-isnt-good-idea/.

[2] Scott O. Bradner. Key words for use in rfcs to indicate requirement levels. *RFC*, 2119:1–3, 1997.

[3] Gonzalo Camarillo and Internet Architecture Board. Peer-to-peer (P2P) architecture: Definition, taxonomies, examples, and applicability. *RFC*, 5694:1–26, 2009.

[4] Tom Chothia and Konstantinos Chatzikokolakis. A survey of anonymous peer-to-peer file-sharing. In Tomoya Enokido, Lu Yan, Bin Xiao, Daeyoung Kim, Yuan-Shun Dai, and Laurence Tianruo Yang, editors, *Embedded and Ubiquitous Computing - EUC 2005 Workshops, EUC 2005 Workshops: UISW, NCUS, SecUbiq, USN, and TAUES, Nagasaki, Japan, December 6-9, 2005, Proceedings*, volume 3823 of *Lecture Notes in Computer Science*, pages 744–755. Springer, 2005.

[5] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX, 2004.

[6] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE J. Sel. Areas Commun.*, 16(4):482–494, 1998.

[7] V. Scarlata, Brian Neil Levine, and Clay Shields. Responder anonymity and anonymous peer-to-peer file sharing. In *9th International Conference on Network Protocols (ICNP 2001), 11-14 November 2001, Riverside, CA, USA*, pages 272–280. IEEE Computer Society, 2001.

[8] TheOnionBay. TORrents - "Torrents" over TOR. 2017. https://theonionbay.github.io/TORrents/.