[S25] Distributed and Network Programming

TorrentInno

Project 16:A

Salavat Zaynulin, Evgenii Bortsov, George Selivanov,
Timofey Ivlev, and Bulat Gazizov

April 29, 2025

1 Introduction

Peer-to-peer (P2P) file sharing protocols allow users to distribute files directly between themselves without relying on a central server for the file data itself. Systems like **BitTorrent** are widely used for distributing large files efficiently by breaking them into smaller pieces that peers download from each other and upload to others simultaneously. A central component, the **tracker** helps peers discover each other for a specific file (identified by an **info-hash**).

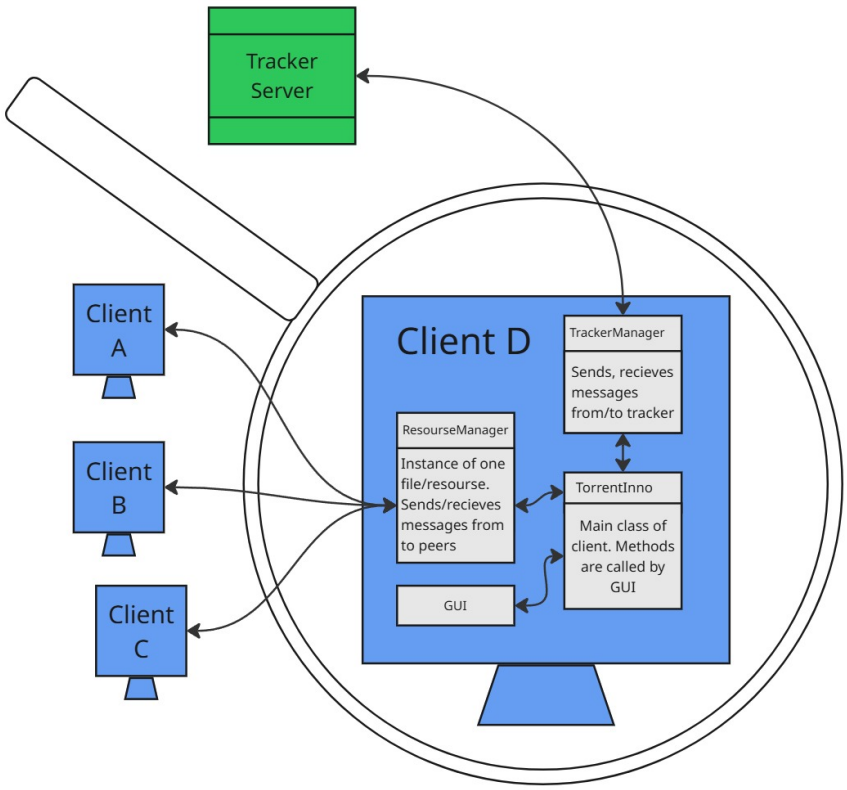
TorrentInno is a custom implementation inspired by these principles, providing a framework for P2P file exchange coordinated by a tracker. Such systems are relevant for decentralized file distribution, reducing server load, and potentially increasing download speeds through parallel sourcing.

2 Methods

2.1 Overall Architecture

The TorrentInno system consists of two main components:

1. **Tracker:** A server application (implemented in Go) responsible for registering peers interested in a specific resource (file) and providing lists of peers to each other.
2. **Client:** A client application (implemented in Python) that interacts with the tracker to find peers and then communicates directly with those peers to download or upload file pieces. It includes a core P2P engine and a GUI layer (see gui). Here is interaction schema:



2.2 Peer Discovery (Tracker Interaction)

As described in specifications README.md, the process begins with a peer announcing itself to the tracker:

1. **Compute Info-Hash:** The client calculates a unique identifier (`info-hash`) for the resource (e.g., the `.torrentinno` file). The details are mentioned to be in the `client.core.common.Resource` class (method `get_info_hash`).

2. **Announce:** The client sends an HTTP request to the tracker's `/peers` endpoint. The request body contains the peer's ID (`peerId`), the `infoHash`, and the peer's public IP and port (peer-announce.json). From torentinno.py:

   - Peer ID generation:

```python
def generate_random_bits(size) -> bytes:
    '''
    Generate random bits usign randint
    '''
    return bytes(random.randint(0, 255) for _ in range(size))

def generate_random_peer_id() -> str:
    '''
    function what generate peerid
    '''
    return generate_random_bits(32).hex()
```

3. **Receive Peer List:** The tracker responds with a list of other peers currently sharing the same `info-hash` (see tracker-response.json).

4. **Periodic Updates:** The client periodically re-announces itself (approx. every 30 seconds) to stay listed by the tracker. The tracker removes peers that haven't announced recently.

2.3 Peer-to-Peer Communication

Once a client has a list of peers, it attempts to establish direct connections (peer-message-exchange.md).

1. **Handshake:** The initiating peer (determined by comparing peer IDs; the one with the smaller ID initiates) sends a 75-byte handshake message:

   ```
   TorrentInno[peer-id (32 bytes)][info-hash (32 bytes)]
   ```

   The receiving peer verifies the `info-hash`. If it matches a resource it manages, it replies with its own handshake message containing its `peer-id`. A successful handshake establishes a persistent, bidirectional connection for that specific resource.

2. **Length-Prefixed Messages:** All subsequent communication uses length-prefixed messages:

   ```
   [body-length (4 bytes)][message-body]
   ```

3. **Message Body:** The `message-body` contains the message type and data:

   ```
   [message-type (1 byte)][message-data]
   ```

   Supported message types:

   - **Request (`0x01`):** Sent by a peer to request a block of data.
     - `[message-data]`: `[piece-index (4 bytes)][piece-inner-offset (4 bytes)][block-length (4 bytes)]`
   - **Piece (`0x02` - *assumed type based on common practice, not explicitly numbered in docs*):** Sent in response to a Request, containing the actual file data.
     - `[message-data]`: `[piece-index (4 bytes)][piece-inner-offset (4 bytes)][block-length (4 bytes)][data]`
   - **Bitfield (`0x03` - *assumed type*):** Sent by a peer to inform others which pieces it possesses.
     - `[message-data]`: `[bitfield]` (A byte array where each bit represents a piece, 0=missing, 1=present).

2.4 Client Core Logic (resource_manager.py)

The ResourceManager class manages the lifecycle of downloading or seeding a specific resource.

- **Initialization:** Takes the host peer ID, destination path, resource metadata, and whether the peer initially has the file. It sets up internal state, including piece status tracking. From resource_manager.py:

```
class ResourceManager:
    class PieceStatus(Enum):
        FREE = 1  # The piece is not in work
        IN_PROGRESS = 2  # Waiting for reply from some peer
        RECEIVED = 3  # The data has been fetched from network and now is saving on disk
        SAVED = 4  # Piece is successfully saved on disk

    def __init__(
            self,
            host_peer_id: str,
            destination: Path,
            resource: Resource,
            has_file
    ):
        self.host_peer_id = host_peer_id
        self.destination = destination
        self.resource = resource
        self.has_file = has_file
        # ...
        self.piece_status: list[ResourceManager.PieceStatus] = []
        if has_file:  # The caller claims to already have the file
            self.resource_file = ResourceFile(
                destination,
                resource,
                fresh_install=False,
                initial_state=ResourceFile.State.DOWNLOADED
            )
            self.piece_status = [ResourceManager.PieceStatus.SAVED] * len(self.resource.pieces)
        else:  # The caller does not the complete downloaded file
            self.resource_file = ResourceFile(
                destination,
                resource,
                fresh_install=False,
                initial_state=ResourceFile.State.DOWNLOADING
            )
            self.piece_status = [ResourceManager.PieceStatus.FREE] * len(self.resource.pieces)

        # Current peer id that handles the piece (empty string=no peer)
        self._peer_in_charge: list[str] = [''] * len(self.resource.pieces)
        # ...
```

- **File Handling:** Uses the ResourceFile class to manage reading/writing pieces to a temporary file during download (`.torrentinno-filename`) and renaming it upon completion. From resource_file.py:

```
class ResourceFile:
    # ...
    class State(Enum):
        DOWNLOADING = 1
        DOWNLOADED = 2
    # ...
    async def save_block(self, piece_index: int, piece_inner_offset: int, data: bytes):
        if self.state == ResourceFile.State.DOWNLOADED:
            raise RuntimeError("Cannot perform write operation in DOWNLOADED state")
        # ...
        await self._ensure_downloading_destination()
        async with aiofiles.open(self.downloading_destination, mode='r+b') as f:
            await f.seek(offset)
            await f.write(data)
    # ...
    async def accept_download(self):
        await aiofiles.os.rename(self.downloading_destination, self.destination)
        self.state = ResourceFile.State.DOWNLOADED
```

- **Download Loop:** Runs as an `asyncio` task, continuously finding `FREE` pieces and assigning them to available peers (`_free_peers`) that have the piece (checked via `_bitfields`). From resource_manager.py:

```
async def _download_loop(self):
    # ...
    while True:
        # Find free pieces
        free_pieces: list[int] = []
        for i, status in enumerate(self.piece_status):
            if status == ResourceManager.PieceStatus.FREE:
                free_pieces.append(i)
        # Try to find piece and peer that has this piece
        found_work = False
        for piece_index in free_pieces:
            for peer_id in self._free_peers:
                if self._peer_has_piece(peer_id, piece_index): # Peer has this piece -> run the work
                    # Update the status and related peer
                    self.piece_status[piece_index] = ResourceManager.PieceStatus.IN_PROGRESS
                    self._peer_in_charge[piece_index] = peer_id
                    # ...
                    task = asyncio.create_task(self._download_work(peer_id, piece_index))
                    # ...
                    found_work = True
                    break
            if found_work:
                break
        await asyncio.sleep(0.2)
```

- **Connection Handling:** Listens for incoming connections and initiates outgoing connections uses `asyncio.StreamReader` and `asyncio.StreamWriter` to handle messages (see resource_manager.py).
- **Message Processing:**
  - `on_request`: Reads the requested block using ResourceFile.get_block and sends a `Piece` message back.
  - `on_piece`: Validates the received piece data against the expected hash from the resource metadata. If valid, saves it using ResourceFile.save_block, updates `piece_status` to `SAVED`, and broadcasts an updated `Bitfield` to all connected peers.
  - `on_bitfield`: Updates the internal record (`_bitfields`) of which pieces the sending peer possesses.
  - `on_close`: Cleans up connection state when a peer disconnects.

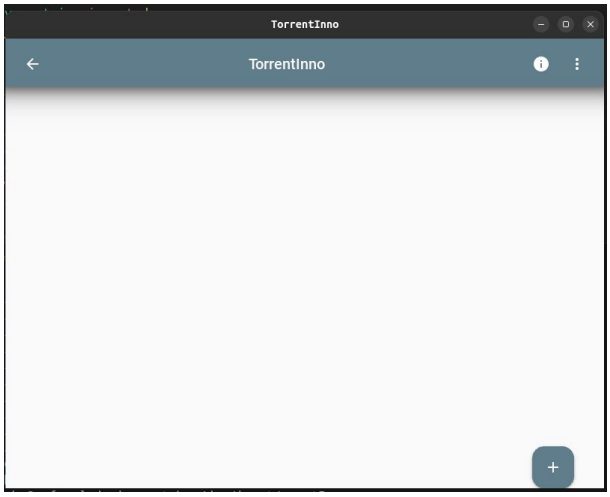
2.5 GUI Layer (see torrent_manager_docs.md)

A GUI torrent manager provides user interaction. Key functions include:

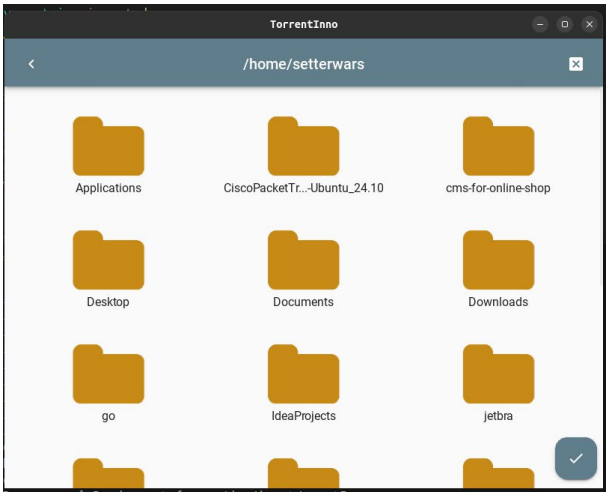
- `initialize()`: Loads saved state or uses test data.
- `shutdown()`: Shutdown work of the manager and saves the current state.
- `get_files()`: Returns a list of active torrents with details (name, size, speed, blocks).
- `update_file(file_name)`: Refreshes information (speed, blocks) for a specific torrent.
- `update_files()`: Refreshes information for all torrents.
- `get_file_info(url)`: Fetches metadata for a new torrent URL/magnet link.
- `add_torrent(file_info)`: Adds a new torrent to the manager.
- `remove_torrent(file_name)`: Removes a torrent.
- `get_mock_content(source)`: Provides mock file list data (likely for UI development/testing).

3 Results

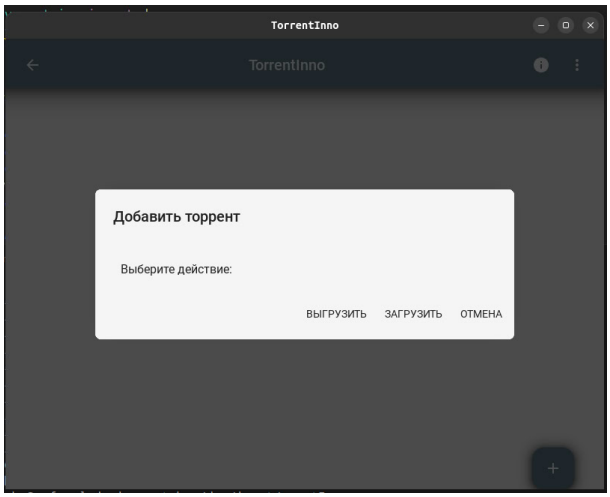
- In our work process we followed best practices. Before introducing new feature, we used thorough tests to ensure correct work of the application (see Test folder on TorrentInno repository).
- Tracker is made to be fault tolerant and reliable. We tried to develop tracker to be as separate, isolated component. This approach fasciate testability, and eases support and maintainability.



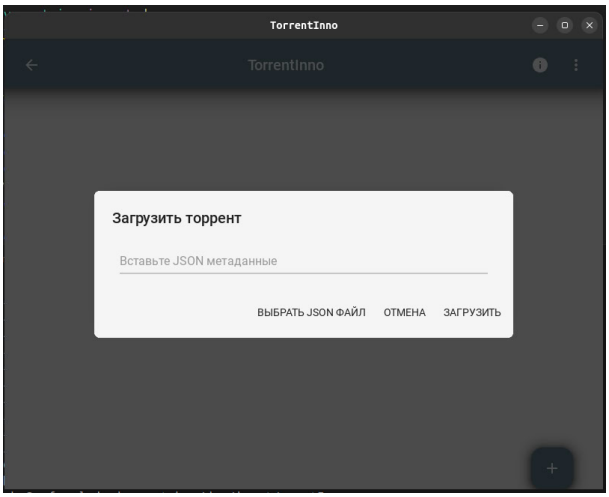
(a) Main screen



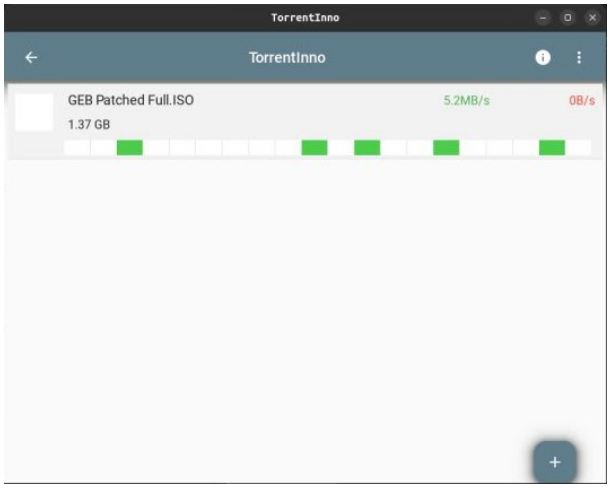
(b) Open file



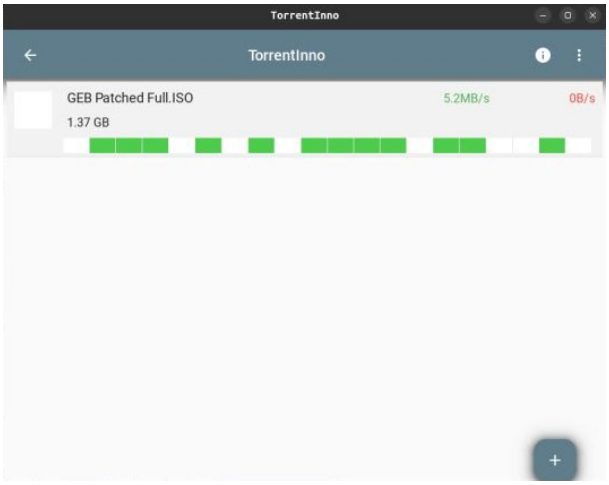
(c) Add file



(d) Download file



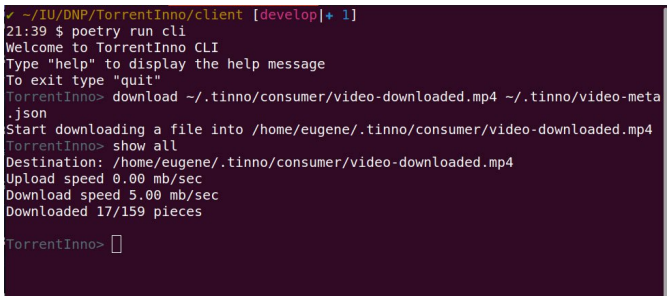
(e) Download in progress



(f) Download in progress 2

Figure 1: TorrentInno interface

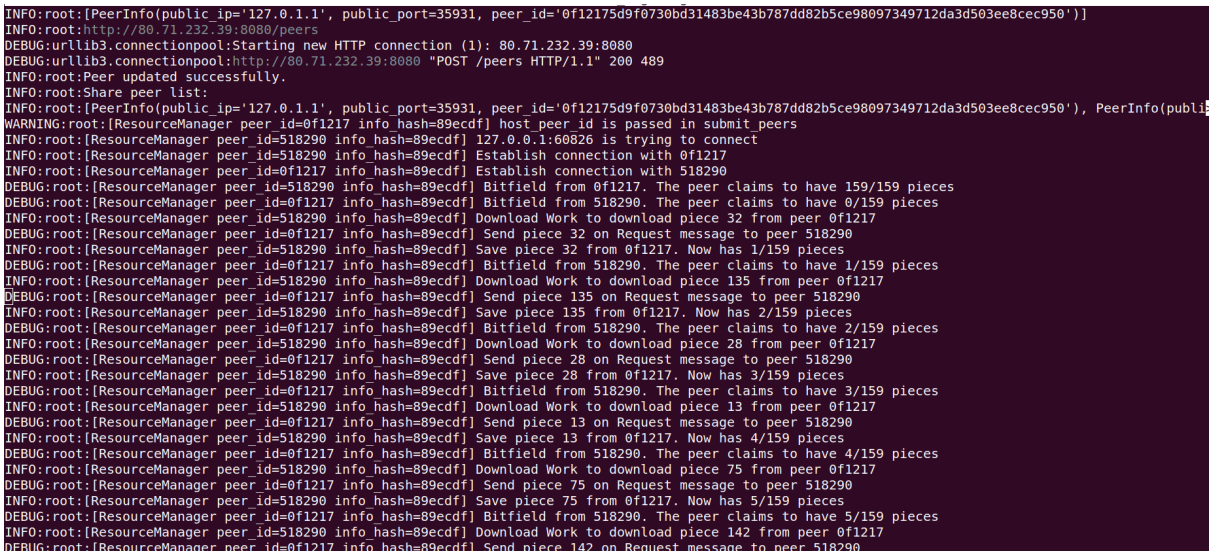
(a) Sharing peer



(b) Downloading peer

(c) Downloading peer: started

(d) Download peer: almost finished



(e) Logs

Figure 2: TorrentInno work from console

4 Discussion

The TorrentInno project successfully implements the fundamental components of a P2P file-sharing system, including a tracker for peer discovery and a client capable of exchanging file pieces according to a defined protocol. The use of `asyncio` in the Python client allows for efficient handling of concurrent network operations (multiple peer connections, downloads, uploads). The separation into core logic (core) and GUI (gui) follows Single Responsiblity Principle, enhancing maintainability and scalability.

Challenges:

- **Concurrency:** Managing the state of multiple pieces across numerous peer connections concurrently (assigning pieces, handling timeouts, validating data) is inherently complex. The current implementation uses `asyncio` tasks and status tracking (PieceStatus) to manage this.
- **Network Reliability:** P2P networks involve unreliable peers and network conditions. The system needs robust error handling for connection drops, timeouts (a simple 1-minute timeout is implemented in _download_work function), and potentially corrupt data (hash checking is implemented in on_piece function).
- **State Management:** Ensuring consistent state, especially when resuming downloads, can be challenging.

Potential Improvements/Optimizations:

- **State Restoration:** Implementing robust saving and loading of download progress to allow resuming interrupted downloads.
- **Piece Selection Strategy:** The current download loop shuffles free pieces randomly (resource_manager.py). We might need more advanced strategies (e.g., "rarest first") could improve download performance in swarms with uneven piece distribution.
- **Error Handling:** Enhance error handling needed for network issues and peer misbehavior.
- **Throttling/Rate Limiting:** Implement upload/download rate limiting.
- **Security:** Adding encryption to peer communication.
- **Tracker Reliability :** One simple decision is to make tracker copies (mirrors). Other options include compact peer lists or UDP tracker protocol support to decrease load (for analogy with DNS servers).
- **GUI Enhancements:** Adding more detailed statistics and configuration options

5 References

- BitTorrent Protocol Specification (BEP_0003)
  https://www.bittorrent.org/beps/bep_0003.html

- Useful articles on Russian
  - About NAT hole punching
    https://habr.com/ru/companies/ruvds/articles/761188/
  - BitTorrent from zero using Go
    https://habr.com/ru/companies/skillfactory/articles/714044/
- Links to relevant academic papers
  - P. Sharma, A. Bhakuni and R. Kaushal, "Performance analysis of BitTorrent protocol," 2013 National Conference on Communications (NCC), New Delhi, India, 2013, pp. 1-5, doi: 10.1109/NCC.2013.6488040. keywords: {Protocols;Peer-to-peer computing;Internet;Servers;Bandwidth;Libraries;Thin film transistors;BitTorrent protocol;Peer to Peer (P2P);Internet;Networks}
    https://ieeexplore.ieee.org/abstract/document/6488040
  - E. Costa-Montenegro, J.C. Burguillo-Rial, F. Gil-Castiñeira, F.J. González-Castaño, Implementation and analysis of the BitTorrent protocol with a multi-agent model,Journal of Network and Computer Applications, Volume 34, Issue 1, 2011, Pages 368-383, ISSN 1084-8045, https://doi.org/10.1016/j.jnca.2010.06.010. (https://www.sciencedirect.com/science/article/pii/S1084804510001086) Keywords: P2P; BitTorrent; Multi-agent; Model; JADE
  - Arnaud Legout, Guillaume Urvoy-Keller, Pietro Michiardi. Understanding BitTorrent: An Experimental Perspective. [Technical Report] 2005, pp.16. ⟨inria-00000156v3⟩
    https://inria.hal.science/inria-00000156/
- Links to documentation for key libraries:
  - Python
    - `asyncio` https://docs.python.org/3/library/asyncio.html
    - `aiofiles` https://pypi.org/project/aiofiles/
  - Go
    - `Gin` https://github.com/gin-gonic/gin