Peer-to-Peer File Sharing with BitTorrent

Course Project Report

Ivan Ch., Ksenia K., Anton K., Ilya O.

April 28, 2025

Abstract

This report presents the design, implementation and evaluation of a simplified BitTorrent–style peer-to-peer (P2P) file-sharing system written in Go. The system implements chunked file transfer, rarest-first piece selection, and decentralised peer discovery through a lightweight Distributed Hash Table (DHT). We validate our implementation on a local four-node test-bed and demonstrate successful end-to-end transfers without a central tracker.

Contents

1 Introduction

BitTorrent popularised efficient P2P file distribution by splitting a file into many fixed-size *pieces* and exchanging them among peers in parallel. Modern content delivery still relies on the same principles (e.g. WebTorrent, IPFS). The aim of this project is to build a minimal yet functional BitTorrent-like client from scratch in Go, focusing on

- Decentralised peer discovery via a mini-DHT (no tracker);
- Chunk exchange over TCP with rarest-first scheduling;
- Structured JSON logging for observability.

We restrict ourselves to single-file torrents and ignore advanced incentives (tit-for-tat) to stay within course scope.

2 Background

2.1 BitTorrent Protocol

The official BitTorrent specification [1] defines five core messages (handshake, bitfield, request, piece, have) exchanged over TCP after an initial handshake that carries the *info-hash*. Peers maintain a *bitfield* indicating which pieces they already possess.
In our implementation, we also define such messages and handle them this way

```
func (peer *Peer) handle(message *protocol.Message) {
  if !peer.handshakeDone && message.ID != protocol.MsgHandshake {
     return
  }

  switch message.ID {
  case protocol.MsgHandshake:
     /* Compares expected hash and hash from handshake */

  case protocol.MsgBitfield:
     /* Receives the peer's available pieces bitfield */

  case protocol.MsgRequest:
     /* Peer requests a piece; server prepares and sends it */

  case protocol.MsgHave:
     /* Peer announces possession of a specific piece */

  case protocol.MsgPiece:
     /* Receives a piece of data and verifies its integrity*/
  }
}
```

No messages are accepted before the handshake is done

2.2 Kademlia DHT

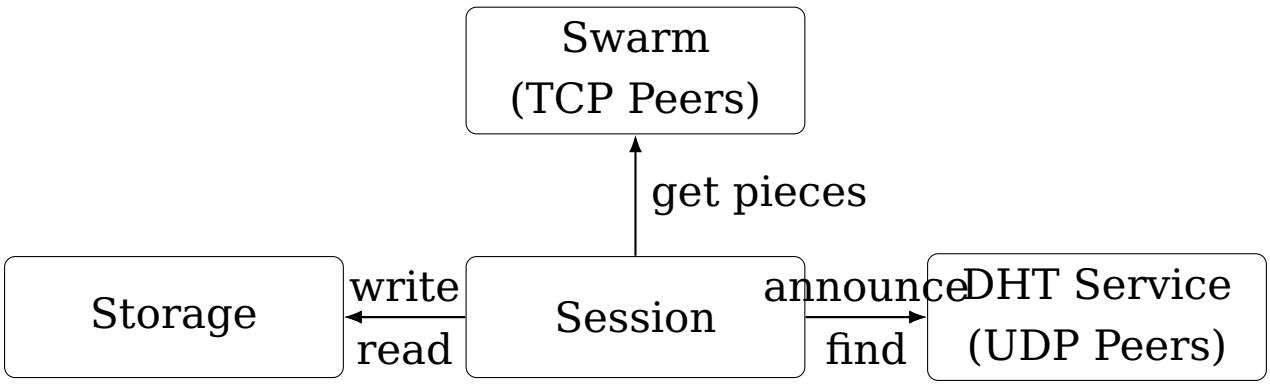
Kademlia [2] organises nodes in a 160-bit XOR metric space and allows logarithmic-time look-ups. Our DHT keeps the same bucket structure (k=8) but supports only three RPCs:

ping/pong, findPeers, and announce.
We handle RPC in such way:

```
 func (node *DHTNode) handle(msg Msg, adr *net.UDPAddr) {
  // Refresh routing table with sender's node-ID
  node.RoutingTable.Update(Peer{ID: id20, Addr: adr})

  switch msg.T {
  case "ping":
     // Receive ping request.
     // Collect 10 peers from routing table.
     // Send pong with collected peers.

  case "pong":
     // Receive pong response and update routing table with new peers

  case "announce":
     // Register a TCP address for a given infohash in the seeds list

  case "findPeers":
     // Lookup and send known seeders for a requested infohash
  }
}
```

-

3 Simplified System Design



3.1 Concurrency Model

The concurrency mechanisms of Go are extremely useful for the project. Each outbound or inbound TCP connection runs two goroutines: a reader and a writer, communicating with the rest of the program through a buffered channel of `protocol.Messages`:

```
func New(conn net.Conn, bf storage.Bitfield, id, desiredInfohash [20]byte) *Peer {
  peer := &Peer{ ... }
  go peer.writer()
  go peer.reader()
  return peer
}

// Writes messages into connection
func (peer *Peer) writer() {
  for msg := range peer.SendCh {
    msg.Encode(peer.Conn)
  }
}

// Reads messages from connection
func (peer *Peer) reader() {
  for {
    peer.handle(protocol.Decode(peer.Conn))
  }
}
```

The DHT runs its own UDP reader and a dispatcher goroutine.

```
// Creates and start a new DHT node listening on a specified address.
func New(listen string) (*DHTNode, error) {
  node := ...

  go node.udpLoop() // Socket loop
  go node.dispatchLoop() // message handler

  return node, nil
}
```

3.2 Data Structures

Session Owns every byte and handle that lives for the lifetime of *one* ‘.bit‘ file — metadata, piece cache, peer swarm and optional DHT helper.

```
type Session struct {
  // --- mutable ---
  Mu sync.Mutex // atomic piece writes
  Pieces [][]byte // RAM cache of pieces
  BF storage.Bitfield // pieces we own
  // --- constants ---
  InfoHash [20]byte
  Meta *metainfo.Meta
  // --- subsystems ---
  DHT *DHTService
  Swarm *Swarm
  cfg *Config // back-pointer to flags
```

```
}
```

Swarm Leecher-side brain that decides *what* piece to ask *which* peer (rarest-first) and announces completion.

```
type Swarm struct {
  Sess *Session // shared buffers
  Peers []*peer.Peer // active TCP conns
  // piece-picker state
  missing, availability []int
  ticker *time.Ticker
  destDir string
  keepSec int
}
```

Peer A goroutine pair (‘reader‘ / ‘writer‘) wrapping one TCP socket; pushes validated events upward to the swarm.

```
type Peer struct {
  Conn net.Conn
  Bitfield storage.Bitfield // peer’s inventory
  SendCh chan protocol.Message
  Meta *metainfo.Meta
  Pieces [][]byte // file pieces we have
  ID, RemoteID [20]byte
  OnHave func(int) // callback into Swarm
}
```

DHTNode A lightweight, JSON-encoded Kademlia actor that keeps a routing table, seeds map and two inboxes for UDP traffic.

```
type DHTNode struct {
  ID [20]byte
  Conn *net.UDPConn
  RoutingTable *dht.Table // 160 buckets
  Seeds map[string][]string // infoHash: []tcpAddr
  inbox, inboxPeer chan packet // demuxed receives
}
```

Msg The *only* packet that flies over UDP; optional fields let several RPCs share one wire format.

```
type Msg struct {
  T string ‘json:"t"‘ // "ping" | ...
  ID string ‘json:"id"‘ // hex(nodeID)
  Info string ‘json:"info,omitempty"‘ // hex(infoHash)
  Addr string ‘json:"addr,omitempty"‘
  TcpList []string ‘json:"tcp_list,omitempty"‘
  DHTPeers []MsgPeer‘json:"dht_peers,omitempty"‘
}
```

Meta A '.bit' manifest: file size, piece size and SHA-1 hash per piece (20 bytes each).

```
type Meta struct {
  FileName string // base name of payload
  FileLength int64 // total bytes in payload
  PieceSize int // bytes per piece (256 KiB by default)
  Hashes [][]byte // SHA-1 per piece
}
```

Bitfield A tiny compressed set that tells peers "I already own these pieces".

```
type Bitfield []byte // bf[i]==1 -> we have piece i
```

Message Length-prefixed frames travelling on every TCP link; helper constructors create typed variants such as `NewRequest` or `NewPiece`.

```
type Message struct {
  ID uint8 // Handshake | Bitfield | Request | ...
  Data []byte // payload depends on ID
}
```

Routing Table A 160-bucket Kademlia table. Peers are evicted by least-recently-seen (LRU) once a bucket exceeds $k = 8$.

```
type Table struct {
  mu sync.RWMutex
  self [20]byte
  bucket [160]bucket
}
```

3.3 Piece Selection

We implement *rarest-first*: every 2 s the leecher computes the availability of each still-missing piece across known peers and requests the least replicated one.

```
// Return rarest piece index by computing availability
func (sw *Swarm) choosePiece() int {
  // Reset availability
  for i := range sw.availability {
    sw.availability[i] = 0
  }

  // Compare bitfields and recompute avail
  for _, p := range sw.Peers {
    for i := range sw.availability {
      if p.Bitfield.Has(i) {
        sw.availability[i]++
      }
    }
  }

  // Find rarest piece
  best := -1
  for i, need := range sw.missing {
    if !need { // No need to ask available piece
      continue
    }
    if best == -1 || sw.availability[i] < sw.availability[best] {
      best = i
    }
  }
  return best
}
```

3.4 DHT Table

Motivation. Efficient peer discovery requires each node to remember only a *logarithmic* view of the network. Following Kademlia [2] we partition the 160-bit XOR metric space into 160 k-*buckets*; bucket b collects peers whose IDs share exactly b leading bits with our own `selfID`. With a constant bucket capacity k=8 this bounds memory to $k \times 160 = 1280$ entries while still yielding $O(\log N)$ routing guarantees.

Core data structure. The implementation (`internal/dht/table.go`) stores buckets as `[]Peer` slices guarded by an RWMutex:

Listing 1: Routing-table excerpt (simplified).

```go
const kSize = 8 // bucket capacity

type Peer struct {
   ID [20]byte
   Addr *net.UDPAddr
   Time time.Time // last seen
}

type bucket struct{ peers []Peer }

type Table struct {
   mu sync.RWMutex
   self [20]byte // this node's ID
   bucket [160]bucket // one per prefix length
}
```

Insertion / refresh (`Update`). When a UDP packet is received the sender is added via Table.Update. The algorithm (Listing 2) performs four steps under a write lock:

1. Ignore our own ID.
2. Compute bucket index $b = \text{prefixLen}(ID \oplus selfID)$.
3. Remove any previous occurrence of the peer (refresh).
4. Append the peer as *most-recent*; if the bucket now exceeds k entries the least-recent (LRU) element is evicted.

Listing 2: Peer refresh / LRU eviction

```go
func (t *Table) Update(p Peer) {
   if p.ID == t.self { return } // 0. never store self
   b := prefixLen(xor(p.ID, t.self)) // 1. bucket index
   bk := &t.bucket[b]

   // 2. refresh: drop previous instance
   for i, q := range bk.peers {
      if q.ID == p.ID {
         bk.peers = slices.Delete(bk.peers, i, i+1)
         break
      }
   }

   // 3. append as MRU
   p.Time = time.Now()
   bk.peers = append(bk.peers, p)

   // 4. evict LRU if over capacity
   if len(bk.peers) > kSize {
      bk.peers = bk.peers[1:]
   }
}
```

Look-ups (`Closest`). Queries such as findPeers need the n peers whose XOR distance to a target key is minimal. The routine first copies all bucket contents into a scratch slice, then sorts it by $dist(a, b) = \text{bigInt}(a \oplus b)$ and returns the first n entries:

Listing 3: Selecting the n closest peers.

```go
func (t *Table) Closest(target [20]byte, n int) []Peer {
   t.mu.RLock(); defer t.mu.RUnlock()

   var cand []Peer
   for _, b := range t.bucket {
      cand = append(cand, b.peers...)
   }
   sort.Slice(cand, func(i, j int) bool {
      return dist(cand[i].ID, target).Cmp(
           dist(cand[j].ID, target)) < 0
   })
   if len(cand) > n { cand = cand[:n] }
   return cand
}
```

The overall complexity is $O(k \cdot 160) = O(1)$ for `Update` and $O(k\,160 \log(k\,160)) \approx O(10^4)$ in the worst case for `Closest`, which is acceptable given the small constant factor.

Concurrency considerations. Read-heavy operations (`Closest`) acquire only a shared RLock, allowing multiple look-ups to proceed in parallel, while insertions use Lock to guarantee LRU consistency.

Summary. This design keeps the well-known asymptotic properties of Kademlia while remaining implementation-friendly:

- **Memory**: ≤ 1280 peers (160 buckets × k=8).
- **Update**: constant-time LRU with mutex protection.
- **Query**: logarithmic network hops using PING/FINDPEERS.

It therefore provides a scalable yet lightweight substrate for the tracker-less BitTorrent network used in this project.

4 Protocol Details

4.1 TCP Messages

ID	Name	Payload
0	handshake	20-byte infoHash + 20-byte peerID
1	bitfield	N-byte (N= number of pieces)
2	request	4-byte piece-index + 4-byte offset (0)
3	piece	4-byte piece-index + 4-byte offset + N-bytes data
4	have	4-byte piece-index

Table 1: Implemented TCP message types.

4.2 UDP DHT Messages

ping, pong, announce, findPeers, peers;
All serialised as JSON for readability and convenience (see `internal/dht/msg.go`).

5 Implementation Highlights

- **Logger**: thread-safe structured JSON, allowing real-time inspection via `jq`.
- **Session**: owns per-torrent state (`Meta`, in-memory pieces, bitfield, DHT service, swarm).
- **Swarm**: orchestrates piece requests and maintains peer list.
- **Safety**: verified with Go race detector; no data races found.

6 Experiments & Results

6.1 Use Cases

Let's open four terminal and launch our torrent client there. Before diving into the experiment, let us explain the configuration flags

1. **-tcp-listen N** - specify address N for tcp listening. Seeders will listen for requests on this port
2. **-dht-listen N** - specify address N for udp listening. Every network node needs this.

3. **-bootstrap N,M,X** - specify udp addresses N,M,X (csv) for system bootstrapping. Without bootstrapping node will never reach the network, so one have to provide at least one node there
4. **-peer N,M,X** - specify tcp addresses N,M,X for udp listening. Not really needed if you specified some good UDP node, that will tell you about seeders. However, you might use it if you are not interested in UDP testing and you want to go straight to pieces exchange testing
5. **-seed path/to/file** - AS A SEEDER specify the filepath you want to seed. The file.bit (metadata) will be produced in the same directory as the file you seeded.
6. **-get path/to/file.bit** - AS A LEECHER specify filepath of .bit file (metadata).
7. **-dest download/file/to** - AS A LEECHER specify filepath where you desire to download seeded file.
8. **-keep N** - AS A LEECHER specify seconds amount you desire to seed after getting the file.

I also use **"| jq ."** so json logs are formatted pretty. Also I will refer to terminals 1-2-3-4 just by number.

Let the first terminal have the role of simple DHT-node, without seeding or leeching abilities.

```
go run ./cmd/bittorrent -dht-listen :10000 -tcp-listen :10001 | jq .
```

It will launch dht-node on localhost:10000 for udp, localhost:10001 for tcp. It is going to wait for nodes to connect.

Second terminal will have the role of seeder and will seed 524MB movie

```
go run ./cmd/bittorrent -seed ~/Documents/lenses_presentation.mov -bootstrap :10000
    -tcp-listen :20001 -dht-listen :20000 | jq .
```

I added flags -bootstrap in order to connect seeder and 1-node and -seed to seed the file. After running, one should expect to see the following two cases.

1. Everything worked fine (usually). DHT and Seeder connected. Following logs produced on seeder side (besides other ones)


```
{
  "event": "udp_send",
  "size": 130,
  "to": "127.0.0.1:10000",
  "ts": "2025-04-27T15:13:31.334306Z",
  "type": "announce"
}

{
  "event": "seeder_ready",
  "file": "/Users/ivanchabanov/Documents/lenses_presentation.mov",
  "infoHash": "904a09802b66917501f8cb5ce31c340fad128da0",
  "tcp": ":20001",
  "ts": "2025-04-27T15:13:31.33604Z"
}
```

2. You started seeder earlier and it sent ping not to DHT but networks darkness. This case you will never catch with DHT-node

```
{
  "event": "seeder did not find DHT yet... try again after 5 sec",
  "ts": "2025-04-27T15:35:37.793721Z"
}
```

Third terminal will be a leecher. Also keep it for 20 minutes to seed (-keep 1200).

```
go run ./cmd/bittorrent -get ~/Documents/lenses_presentation.mov.bit -bootstrap
    :10000 -dht-listen :30000 -tcp-listen :30001 -keep 1200 | jq .
```

1. After booting, leecher will ping-pong DHT-node (resulting in getting known both UDP address of 1 and 2-nodes)
2. Ask all the known DHT-nodes **FindPeers** (leecher will answer seeder, seeder will answer empty list since it does not know other seeders and dht-nodes do not answer themselves)
3. After that, leecher will connect to the seeder and get all the pieces from it.
4. Afterwards, leecher will become a seeder itself and broadcast **Announce** to all the known DHT-nodes (1 and 2-nodes).

Some of the Expected leecher-side logs:

```
// Leecher found other dht-nodes
{
  "event": "RT peers update",
  "new_peer": "127.0.0.1:20000",
  "new_peer_bucket": 1,
  "peers": [
    "127.0.0.1:10000",
    "127.0.0.1:20000"
  ],
  "ts": "2025-04-27T15:41:02.284966Z"
}
...
// Sending FindPeers to known nodes
{
  "event": "udp_send",
  "size": 115,
  "to": "127.0.0.1:10000",
  "ts": "2025-04-27T15:41:07.287684Z",
  "type": "findPeers"
}

{
  "event": "udp_send",
  "size": 115,
  "to": "127.0.0.1:20000",
  "ts": "2025-04-27T15:41:07.290157Z",
  "type": "findPeers"
```

```
}
...
// Receiving answers
{
  "event": "udp_recv",
  "from": "127.0.0.1:10000",
  "size": 133,
  "ts": "2025-04-27T15:41:07.288759Z",
  "type": "peers"
}

{
  "event": "udp_recv",
  "from": "127.0.0.1:20000",
  "size": 111,
  "ts": "2025-04-27T15:41:07.292984Z",
  "type": "peers"
}
...
// Gained TCP address of seeder from 127.0.0.1:10000
{
  "event": "leecher_bootstrap",
  "new_peers": [
    ":20001"
  ],
  "ts": "2025-04-27T15:41:07.294099Z"
}
```

After that leecher will connect to seeder. After handshake he will download the pieces

```
// Handshake validation
{
  "event": "joined_to_peer",
  "peer": ":20001",
  "ts": "2025-04-27T15:41:07.300349Z"
}

{
  "event": "send_handshake_dial",
  "infoHash": "904a09802b66917501f8cb5ce31c340fad128da0",
  "ts": "2025-04-27T15:41:07.30065Z"
}

{
  "event": "recv_handshake",
  "expected": "904a09802b66917501f8cb5ce31c340fad128da0",
  "infoHash": "904a09802b66917501f8cb5ce31c340fad128da0",
  "ts": "2025-04-27T15:41:07.304434Z"
}

{
  "event": "handshake_ok",
  "peer": "127.0.0.1:20001",
  "ts": "2025-04-27T15:41:07.304501Z"
}
...
```

This is how the download visualized

```
{
  "event": "request",
  "peer": "127.0.0.1:20001",
  "piece": 1,
  "ts": "2025-04-27T15:41:09.301866Z"
}

{
  "event": "have",
  "piece": 1,
  "totalPieces": 2,
  "ts": "2025-04-27T15:41:09.303978Z"
}

{
  "event": "request",
  "peer": "127.0.0.1:20001",
  "piece": 2,
  "ts": "2025-04-27T15:41:09.304019Z"
}

{
  "event": "have",
  "piece": 2,
  "totalPieces": 3,
  "ts": "2025-04-27T15:41:09.306124Z"
}
...
{
  "event": "request",
  "peer": "127.0.0.1:20001",
  "piece": 2001,
  "ts": "2025-04-27T15:41:11.072382Z"
}

{
  "event": "have",
  "piece": 2001,
  "totalPieces": 2002,
  "ts": "2025-04-27T15:41:11.07263Z"
}
// Leecher finished. Loaded the file.
{
  "event": "complete",
  "file": "lenses_presentation.mov",
  "ts": "2025-04-27T15:41:12.364185Z"
}
```

If you specified -keep flag, then leecher will become a seeder.

```
{
  "event": "seeder_ready",
  "file": "lenses_presentation.mov",
  "tcp": ":30001",
  "ts": "2025-04-27T15:41:12.364497Z"
```

```
}
{
  "event": "udp_send",
  "size": 130,
  "to": "127.0.0.1:10000",
  "ts": "2025-04-27T15:41:12.365163Z",
  "type": "announce"
}
{
  "event": "udp_send",
  "size": 130,
  "to": "127.0.0.1:20000",
  "ts": "2025-04-27T15:41:12.365185Z",
  "type": "announce"
}
```

After this, I will start a new 4-node leecher and bootstrap 1-node. Result: this leecher loads both from 2 and 3-nodes.

```
  go run ./cmd/bittorrent -get ~/Documents/lenses_presentation.mov.bit -bootstrap
      :10000 -dht-listen :40000 -tcp-listen :40001 -keep 1200 | jq .
```

Some Expected logs:

```
{
  "event": "request",
  "peer": "127.0.0.1:30001", // Asked 3-node
  "piece": 4,
  "ts": "2025-04-27T15:57:32.440545Z"
}
{
  "event": "have",
  "piece": 4,
  "totalPieces": 5,
  "ts": "2025-04-27T15:57:32.442222Z"
}
{
  "event": "request",
  "peer": "127.0.0.1:20001", // Asked 2-node
  "piece": 5,
  "ts": "2025-04-27T15:57:32.442265Z"
}
```

So, here it is! The network is living its live. At this point, 1-node will know seeders 2-3-4, 2-node will know seeders 3,4, third node will only know 4-seeder. Why? Because currently seeders state updates only after single-broadcasting. This might be enhanced in the future work, so every seeder rebroadcasts each 10 minutes.

```
// 1-node logs
{
  "event": "AVAILABLE_SEEDERS",
  "seeders": [
    "904a09802b66917501f8cb5ce31c340fad128da0: [:20001, :30001, :40001]"
  ],
  "ts": "2025-04-27T15:57:36.442336Z"
}
```

6.2 Validation Checklist

1. *Upload then download completely*: All the leechers received file `lenses_presentation.mov` (524 MiB) in about 4.1 s.
2. *Peer join/leave and chunk updates*: Observed via JSON logs (this is how leaving looks like)

   ```
   {
    "bye": "127.0.0.1:62059",
    "event": "bye_leecher",
    "ts": "2025-04-27T16:09:11.246471Z"
   }
   ```

3. *Visualised progress*: Also Visualized via logs.

7 Discussion

This bittorrent version is not absolute. Routing table is not recomputed after the node leaves the network, so some seeders might be unavailable. The version is only tested on

localhost and in nodes number under 10. Multiple files are available in network, however, one node can only hold one file at a time. So, in order to consume multiple files, you introduce multiple seeders.

8 Related Work

- **Official BitTorrent** client (C++) and `libtorrent-Rasterbar`.
- **go-torrent**, an MIT-licensed full client in Go; our codebase is an order of magnitude smaller and purposely single-file.

9 Conclusion & Future Work

We built a working BitTorrent-style client with autonomous peer discovery. Future enhancements include NAT traversal (uTP, hole-punching), multi-file torrents, and incentive mechanisms such as tit-for-tat.

References

[1] BEP 003: The BitTorrent Protocol Specification.

[2] Petar Maymounkov and David Mazieres. *Kademlia: A peer-to-peer information system based on the XOR metric*. IPTPS 2002.

A CLI Flags

```
--seed         path to payload to seed
--get          path to .bit file to download
--peer         comma-separated static peer list
--tcp-listen   TCP listen address (default :0)
--dest         download output directory
--dht-listen   UDP listen address for DHT ('' disables DHT)
--bootstrap    comma-separated UDP bootstrap nodes
--keep         seconds to keep seeding after download completes
```