BitTorrent Multi-Peer Download Flow Example

Scenario: Downloading "BigMovie.mp4"

- **File Size**: 500MB
- **Piece Size**: 1MB each
- **Total Pieces**: 500 pieces (numbered 0-499)
- **Available Peers**: 3 peers (PeerA, PeerB, PeerC)
- **Worker Threads**: 3 workers

Phase 1: Initial Setup

User Command

```
./bittorrent download -o BigMovie.mp4 movie.torrent
```

1. `handle_download()` Function

What it does: Command handler that kicks off everything

```
TorrentInfo info = BitTorrentClient::parse_torrent("movie.torrent");
// Calls the multi-peer download function
BitTorrentClient::download_file_multi_peer(info, "BigMovie.mp4", 3);
```

Key Action: Parses torrent file and starts multi-peer download with 3 workers

---

Phase 2: Download Orchestration

2. `download_file_multi_peer()` Function

What it does: The main coordinator that sets up all components

Step 2a: Create Work Queue

```
WorkQueue work_queue;
for (int i = 0; i < 500; ++i) {  // 500 pieces
    work_queue.add_piece(i);
}
```

WorkQueue State: [0,1,2,3,4,...,499] - All piece numbers queued up

Step 2b: Create Progress Tracker

```
DownloadProgress progress(500, torrent_info);
```

DownloadProgress State:

- `completed_pieces` : [false,false,false,...] (500 false values)
- `piece_data` : 500 empty 1MB vectors pre-allocated
- `completed_count` : 0

Step 2c: Spawn Worker Threads

```
for (int i = 0; i < 3; ++i) {
    workers.emplace_back(download_worker, torrent_info, peers, work_queue, progress);
}
```

Result: 3 worker threads start running simultaneously

Phase 3: Parallel Download Execution

3. Worker Threads (`download_worker()`)

Worker 1 - Time 0:00

```
int piece_index;
work_queue.get_piece(piece_index);  // Gets piece 0
```

Action: Worker 1 grabs piece 0 from queue **WorkQueue State**: [1,2,3,4,...,499]

Worker 2 - Time 0:00

```
work_queue.get_piece(piece_index);  // Gets piece 1
```

Action: Worker 2 grabs piece 1 from queue **WorkQueue State**: [2,3,4,...,499]

Worker 3 - Time 0:00

```
work_queue.get_piece(piece_index);  // Gets piece 2
```

Action: Worker 3 grabs piece 2 from queue **WorkQueue State**: [3,4,5,...,499]

Phase 4: Individual Piece Downloads

4. `download_piece_from_peer()` Function

Worker 1 Downloads Piece 0

Step 4a: Connect to Peer

```
int sock = connect_to_peer(peerA);  // Connects to PeerA
```

Action: Establishes TCP connection to PeerA

Step 4b: Handshake

```
perform_handshake(sock, info_hash, peer_id);
```

Action: Exchanges BitTorrent protocol handshake

Step 4c: Send Interested

```
send_interested(sock);
```

Action: Tells peer "I want to download from you"

Step 4d: Wait for Unchoke

```
recv_message(sock, msg_id);  // Receives UNCHOKE message
```

Action: Peer says "OK, you can download"

Step 4e: Download 1MB in Chunks

```
for (offset = 0; offset < 1MB; offset += 16KB) {
    send_request(sock, 0, offset, 16KB);  // Request 16KB block
    payload = recv_message(sock, msg_id); // Receive PIECE message
    // Copy data: payload -> piece_data
    std::copy(payload.begin() + 8, payload.end(), piece_data.begin() + offset);
}
```

Action: Downloads piece 0 in 64 chunks of 16KB each

Step 4f: Hash Verification

```
SHA1(piece_data.data(), piece_data.size(), actual_hash);
if (memcmp(actual_hash, expected_hash, 20) == 0) {
    return true;  // Success!
}
```

Action: Verifies downloaded data matches expected hash

Phase 5: Progress Tracking

5. `DownloadProgress::mark_piece_complete()`

Worker 1 Completes Piece 0- Time 0:03

```
progress.mark_piece_complete(0, piece_data);
```

Inside mark_piece_complete():

```
std::lock_guard<std::mutex> lock(mtx);  // Lock for thread safety
completed_pieces[0] = true;             // Mark piece 0 as done
piece_data[0] = data;                   // Store the 1MB of data
completed_count++;                      // Now = 1
```

DownloadProgress State:

- `completed_pieces` : [true,false,false,...]
- `completed_count` : 1
- Console: "Downloaded piece 0 (1/500)"

Worker 1 Gets Next Piece- Time 0:03

```
work_queue.get_piece(piece_index);  // Gets piece 3
```

Action: Worker 1 immediately starts downloading piece 3 **WorkQueue State**: [4,5,6,...,499]

Phase 6: Parallel Progress

Time 0:10 - All Workers Active:

- **Worker 1**: Downloading piece 3 from PeerA
- **Worker 2**: Downloading piece 1 from PeerB
- **Worker 3**: Downloading piece 2 from PeerC

Time 0:15 - More Completions:

```
// Worker 2 finishes piece 1
progress.mark_piece_complete(1, piece_data);  // completed_count = 2
work_queue.get_piece(piece_index);            // Gets piece 4

// Worker 3 finishes piece 2
progress.mark_piece_complete(2, piece_data);  // completed_count = 3
work_queue.get_piece(piece_index);            // Gets piece 5
```

DownloadProgress State:

- `completed_pieces` : [true,true,true,false,false,...]
- `completed_count` : 3
- Console: "Downloaded piece 2 (3/500)"

Phase 7: Completion and Assembly

Time 5:30 - Download Complete:

```
// All 500 pieces downloaded
progress.is_download_complete();  // Returns true
work_queue.mark_finished();       // Tell workers to stop
```

6. `DownloadProgress::write_to_file()`

```
std::ofstream file("BigMovie.mp4", std::ios::binary);
for (int i = 0; i < 500; ++i) {
    // Write each 1MB piece in order: piece[0] + piece[1] + ... + piece[499]
    file.write(reinterpret_cast<const char*>(piece_data[i].data()), piece_data[i].size());
}
```

Action: Combines all 500 pieces into final BigMovie.mp4 file

Key Class Responsibilities Summary

WorkQueue Class:

- **Job**: Distributes work fairly among workers
- **Key Method**: `get_piece()` - "Give me the next piece to download"

DownloadProgress Class:

- **Job**: Safely tracks completion and stores data
- **Key Method**: `mark_piece_complete()` - "I finished this piece"

Worker Threads:

- **Job**: Actually download individual pieces
- **Key Loop**: Get piece → Download piece → Mark complete → Repeat

Main Thread:

- **Job**: Monitors progress and coordinates shutdown
- **Key Action**: Waits for `progress.is_download_complete()`

The Magic of Parallelism

Instead of downloading pieces sequentially (0→1→2→3...), all 3 workers download different pieces simultaneously:

- **Sequential**: 500 pieces × 3 seconds each = 25 minutes
- **Parallel (3 workers)**: 500 pieces ÷ 3 workers × 3 seconds = ~8.5 minutes

The **WorkQueue** prevents conflicts, **DownloadProgress** safely combines results, and **worker threads** maximize bandwidth usage.