List of supported features:

1. All core features
2. Endgame mode
3. Direct (point to point) trackerless transfers
4. Seeding previously downloaded files
5. Resuming a partial download

Design and implementation choices that you made

1. **Asynchronous:** Since Bittorrent requires simultaneous communication with lots of peers and depends on independent cooperating operations (for example handling incoming blocks, requesting new blocks, finding which clients have blocks, fulfilling client requests, sending keepalives, querying the tracker are all discrete parts that can run continuously in their own loops) running simultaneously to accomplish a download, and because I didn't want to expose myself to the kinds of bugs that OS level threading can create I used python's asyncio and coroutines for my implementation.
2. **Independent discrete parts:** To improve code clarity and reduce the chance of introducing bugs, I tried separating my client into lots of independent modules which each had a specific task. There's a whole module dedicated just to serialization/deserialization, and a single gateway through which all packets are serialized and sent or deserialized and received. There's a PieceManager which is in charge of figuring out what requests to send next. There's a PieceIO class solely dedicated to storing pieces across files. Torrent files are accessed through their own class with a simple interface and the same is true of Swarms and SwarmPeers. Once a SwarmPeer connection is made, you can use it through a very simple and intuitive API with calls like choke(), unchoke(), take_interest(), remove_interest(), request_piece(), etc. In addition, all of this happens asynchronously so you can talk to other SwarmPeers while waiting for a response.

   One other separation of responsibility which made coding easier was separating receiving and sending. Calling methods on a SwarmPeer to request a block (for example), sends a request packet, but does not wait for the response. That is handled separately. All responses are vacuumed by one central loop, which dispatches async handlers for each type of packet. It's very easy to read because packets have already been deserialized into classes with nice names so it's essentially one long match statement. It also makes it easy to interact with peers because instead of having to have one method request a piece from a peer and then wait for a matching response and worry about what to do with responses to other requests which might come in first, SwarmPeer methods like request_piece are fire-and-forget; either the request succeeds in which case the

handle_incoming_messages loop will eventually pick up the block, or it fails in which case that block won't come in and it'll be re-requested at a later time.

3. **Reusability:** To ensure correctness, I tried to write general purpose code so I could focus on testing it thoroughly and then use it in lots of places instead of writing lots of different implementations for groups of similar operations. For example, all static length packet classes are created generically by a function that takes a static packet type and generates a class for it including serialization and deserialization methods. For another example, my client will load finished pieces from a partially downloaded file before downloading the remaining pieces. I could have written a separate function to load in and verify the already downloaded parts of a file, but I already had a PieceIO class which was thoroughly tested and could very reliably read and write blocks distributed across multiple torrented files. Instead of writing a separate method, my downloads start by asking the PieceManager for all the requests which would normally be sent to peers, and instead sending them to the PieceIO class to get the corresponding blocks on disk. These are then saved and checked just as if they were downloaded from the internet, again reusing code which I already knew worked.
4. **Resiliency / Eventual Correctness:** Bittorrent is a swarm, a distributed network of unreliable peers. Together they have the right answer, but any one of them might be dysfunctional, so I wanted to interact with peers primarily as a swarm, and no matter what one peer sent (or didn't send) back, I always wanted to be working toward the overall goal of downloading the whole file. To do this I took inspiration from Kubernetes which uses reconciliation loops to keep moving toward a desired state until eventually they get there, and not stop working because of one failure. My client's reconciliation loop asks the piece manager what blocks it needs to complete the next piece, and then asks the swarm to find some peer with that piece, and request it. Which blocks get requested is decided by the PieceManager, and it keeps all the requests for all the pieces in a circular queue until it receives a block for that request. Because all requests are kept in a rotating queue, requests that aren't answered will eventually be resent (giving it natural resiliency), but they won't be resent immediately because requested blocks are moved to the back of the queue to give peers time to send a response. This also means that I get endgame mode almost for free because as the number of blocks I need to request shrinks, the few remaining blocks will get requested repeatedly by the piece manager (because they keep cycling to the front of the queue) which means those requests get sent to lots of peers. The only missing piece was canceling fulfilled requests, which I implemented with a dict of outstanding requests.
5. **Efficiency:** I tried minimizing large copies of bytes back and forth by relying heavily on slicing (which performs a reference copy) and making wrapper classes like Bitfield and MutableBitfield which can interact with data in its wire format. I also chose not to write all pieces to one huge file which would be sliced up into smaller files when the download was done, because that could require 2x the space on disk of the actual files being downloaded (until it's finished slicing the

temporary file at which point it could be deleted.) Instead my PieceIO class reads and writes blocks directly to the file or files of which they are part. This is more space efficient and also makes seeding an arbitrary torrent for which you have the files very easy.

Problems that you encountered (and if/how you addressed them)

1. **Too much complex code:**
    a. Dividing work into discrete parts: Using a PieceManager, and PieceIO to store the pieces, a Swarm to make requests, a SwarmPeer class to manage individual connections, Torrent and TorrentFile classes to manage .torrent files, and creating and the files mentioned in the torrent and a download folder to store them.
2. **Unreliable code with lots of bugs:**
    a. Unit testing and property-based testing: Using hypothesis, a property based testing library for Python, I was able to very thoroughly test serialization/deserialization, block storage and retrieval, the generation of piece requests, and the verification of pieces, without spending time writing tons of unit tests. It found lots of edge cases, and made my core methods very reliable.
3. **Client not seeding/accepting peer connections**
    a. I identified this using wireshark and then was able to use wireshark and a debugger to see what packets I was failing to respond to correctly. For example, my unchoke method set a class variable am_choking to true instead of false which meant my client was never unchoking other peers. I was able to see this in wireshark and then track down the root cause with the debugger.

Known bugs in your implementation

None AFAIK. It could have nicer error handling.

Contributions made by each group member

I was the only member.