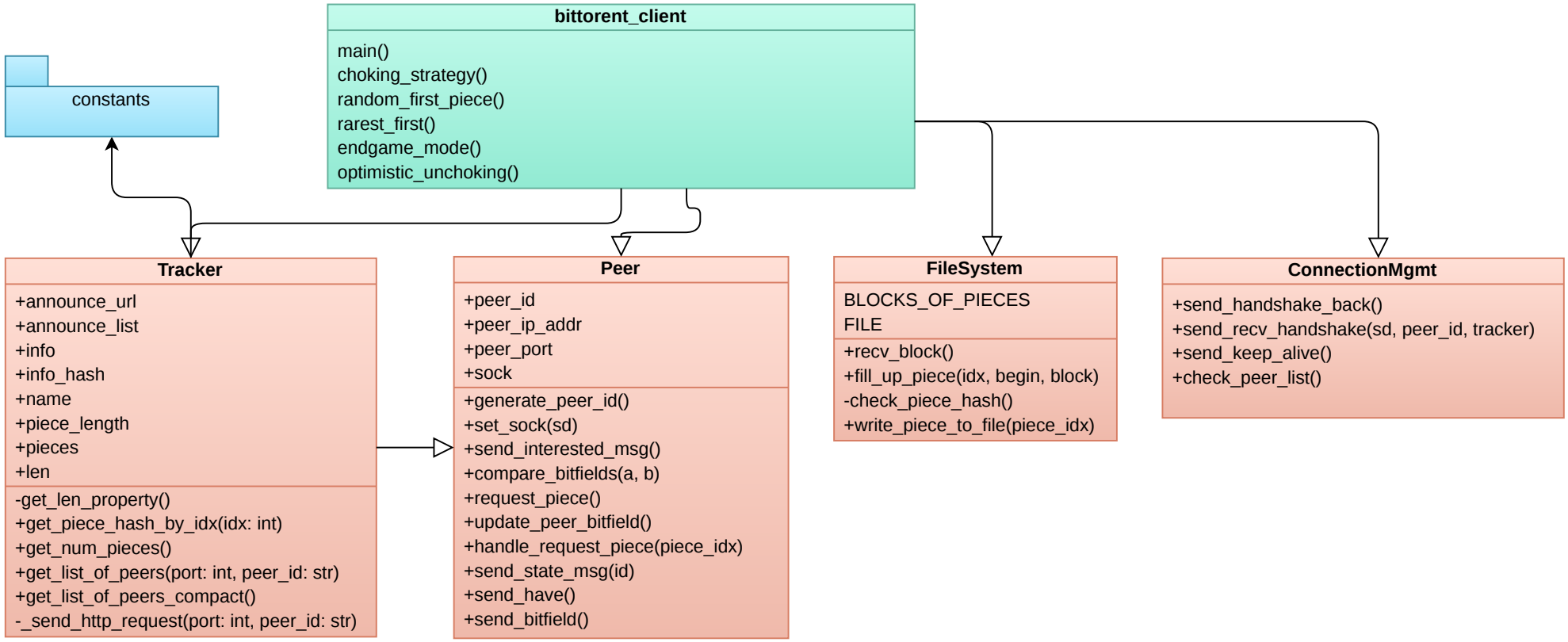

# Decomposition Document

*Note: The above UML diagram depicts the structure of the following functions.*

## Tracker

**Assigned To**: Saar Cohen
**Description:** The Tracker component is responsible for the communication with the tracker: from parsing a given torrent file to extract all necessary metadata, to sending a (http) request, and receiving a response from it.
**API:**

- `_get_len_property` - a private function which calculates the length of the desired downloaded file according to the mode of the .torrent file (Single of Multiple File Mode)
  → Already implemented and tested against several .torrent files online.
- `get_piece_hash_by_idx` - calculates the hash of a piece by its index.
  → Already implemented and tested against several .torrent files online.
- `get_num_pieces` - returns the number of pieces in the file, which is determined by 'ceil (total length / piece size)'.
  → Already implemented and tested against the example .torrent files.
- `get_list_of_peers` - sends a HTTP request using `send_http_request`, and then waits for a response. It parses the response and extracts the peers list (creates Peers objects) which is returned from the function.
  → implemented and tested (not against UDP trackers).
- `get_list_of_peers_compact` - same as `get_list_of_peers`, but supports the compact mode (the client accepts a compact response).
  → Not yet implemented. Deadline: 12/02.
- `send_http_request` - send a http request (GET) to the address of the tracker (the announce url)
  → implemented and tested

## Peer

**Assigned To:** Sadia Nourin
**Description:** The peer component is responsible for the communication among different peers. For every peer it saves its state (id, ip, port, socket) and bits for choking/interested, so one peer can connect to it using TCP.
**API:**

- `send_state_msg(id)` - This function will be responsible for sending the "interested", "not interested", "choking", "unchoking" messages to peers. It will take in a message id for the message that it has to send (ranging from 0 to 3), corresponding to the message that the client will send.

    - Currently, we have some of this implemented via a `send_interested_msg()` function. However, after further deliberation between group members, we would rather have one function that sends these messages rather than divide it up into four functions and so will have to rework the current function.
- `generate_peer_id()`- Not a method of the class itself, but part of the Peer component. This function generates a peer id according to Azureus-style convention.
  → Already implemented by Saar Cohen during his Tracker implementation
- `set_sock(sd)`- Connects to a peer using a socket via TCP. It will take in a socket that has already been created.
- `compare_bitfields(a, b)` - This function will take in two bitfields, `a` and `b`, compare them, and return the next index that `b` has that a does not have. In this case, `a` is supposed to be the current client's bitfield while `b` is the peer's bitfield.
- `request_piece()` - This function will request a piece from a peer that the client itself does not have. It will use the request message format laid out by the BitTorrent Specification document: <len=0013><id=6><index><begin><length>
- `update_peer_bitfield()` - This function will update the bitfield for a peer. This will be triggered whenever we receive a `have` message from a peer that we are connected to.
- `handle_request_piece(piece_idx)` - This function will send a piece that the current client has to one of its peers. The piece that it will send is specified by the `piece_idx` parameter. This function will have to read the piece from the file, chunk up the piece into blocks, and then send those blocks.
- `send_have()` - Whenever the client writes a new piece to the file, the client will send a `have` message to all of the peers that it is connected to. It will use the request message format laid out by the BitTorrent Specification document: <len=0005><id=4><piece index>
- `send_bitfield()` - Whenever the client sends a handshake back to a peer that has initiated the handshake with it, the client will send a bitfield message, denoting the pieces that the client currently has, back to the peer. This is also in the BitTorrent Specification document and will be of the form: <len=0001+X><id=5><bitfield>

# File System

**Assigned To:** Jonathan Camberos
**Description:** The File System component is responsible for receiving blocks (pieces split up), checking a (fake) SHA-1 on complete pieces, and writing that piece to the specific part of the file if the hash matches the bitTorrent hash for that piece.
**API:**

- `blocks_of_pieces` - This global variable will hold the currently received blocks of a piece, for all pieces, in order to recv blocks from sender
- `file` - Global variable we will be writing valid pieces to
- `recv_block()` - This function will handle a receiving a block from a peer. The client will receive a 'piece' Bittorrent message. The 'piece' message follows the form:<len=0009+X><id=7><index><begin><block>

- `fill_up_piece(idx, begin, block)` - This function will handle storing a block corresponding to a piece in the global dictionary containing the current received blocks for a certain piece. (A piece can be sent via multiple blocks depending on how large).
- `check_piece_hash()` - This function will handle the case where all blocks corresponding to a certain piece have been received from a peer, in which case it will calculate the hash of the received piece and compare it to the hash of the torrent file. If the hashes match it will call “write_piece_to_file(piece_idx)
- `write_piece_to_file(piece_idx)` - This function will be where a valid piece has been received, in which case we write the piece to our file. Depending on the number of pieces, length of the piece, and current piece index, we will calculate the offset of where to write in the file

# Connection Management

**Assigned To:** Annie Zhou
**Description:** The communication management component is responsible for maintaining wire connections with peers and enforcing rules specified within the peer wire protocol .
**API:**

- `send_handshake_back()` - This function will handle a peer initiating a connection with the client. The client will receive and send a Bittorrent handshake message to complete the process of establishing a connection. The handshake message follows the form:<pstrlen><pstr><reserved><info_hash><peer_id>.
- `send_recv_handshake()` - This function will be called to initiate a connection to another peer from the client-side. As a result the client will send a Bittorrent handshake message. The handshake message follows the form: <pstrlen><pstr><reserved><info_hash><peer_id>.
  → Already implemented
- `send_keep_alive()` - This function will periodically send Bittorrent keep-alive messages to peers on the client’s peer list to avoid closing the connection. The keep-alive message follows the form: <len=0000>.
- `check_peer_list()` - This function will periodically check if the client should drop a connection with a peer as a result of lack/infrequent keep-alive messages sent to the client. In addition, the function will check if the current number of peers on the client’s peer list is below the threshold of peers. If the number of peers is below the threshold, the client will request a new peer list with `get_list_of_peers()` or `get_list_of_peers_compact()`.

# Strategies

## Piece Selection Strategy

**Assigned To:** Jonathan Camberos and Sadia Nourin

**Description:** Piece Selection is the component used to determine the order in which pieces should be requested in order to improve the performance of the torrenting process. For the base implementation of Bittorrent, the client will implement the Strict Priority Policy of the Piece Selection Algorithm where the client prioritizes downloading all blocks of the same piece before attempting to download another piece. This will be implemented within both the Peer and File System components.

**API:**

- `fill_up_piece()` - This function will enforce that once a single sub piece has been requested, downloading all the blocks of the same piece should be completed before attempting before downloading another piece.

### Top 4 Choking Strategy

**Assigned To:** Sadia Nourin

**Description:** A choking strategy needs to be implemented to ensure that our bittorrent client has the same, if not better, download speeds than reference bittorrent clients. Essentially, our client will only unchoke the top four peers that have the best download speed. This will be implemented within the Peer component.

**API:**

- `choking_strategy()` - This function will calculate the download speeds of all of the clients that it is connected to and unchoke the four that have the best download speed. This means that this function will call `send_state_msg(id)` from the Peer component above. The rest of the function will handle which uploaders to unchoke and which downloaders to choke as specified by the BitTorrent Specification here.

## Extra Credit Features:

*Note: We intend that the following API documentation would require minor alterations to our current design to add-in these features.*

### More Strategies for Piece Selection

**Assigned To:** Annie Zhou

**Description:** The following piece selection strategies will be implemented via boolean flags. If one of the flags is set, then we choose to use that specific strategy for piece selection. In all other cases, the strict priority policy will be used.

**API:**

- `random_first_piece()` - This function enforces the first piece that a client requests from the file-sharing process should be random.
- `rarest_first()` - This function enforces that the client should prioritize finding the rarest piece in the network.
- `endgame_mode()` - This function enforces that when finding the last pieces of a file, the client will request the piece from all peers.

### More Strategies for Choking

**Assigned To:** Sadia Nourin
**Description:** The optimistic unchoking strategy will be called when a boolean flag for it is set, similar to the piece selection strategies above. In all other cases, the top four choking strategy will be used,
**API:**

- `optimistic_unchoking()` - This function calls `send_state_msg(id)` and unchokes a random peer regardless of its upload rate every 30 seconds. We will give more weight to the new peers that have connected to us when it comes to this unchoking.

# Deadlines

As noted above, most of the functions for the tracker, a few functions for the peer, and one function for the connection management has been implemented. The remaining of our implementation will follow the schedule below, up until `Dec 5, 2022`:

- `Dec 1, 2022`: Jonathan will have implemented all of the functions for the File System
- `Dec 2, 2022`: Sadia will have implemented all of the functions for the Peer. This will be done independently from Jonathan, so all of the necessary functions for the uploading and downloading should be completely, but not integrated.
- `Dec 3, 2022`: Jonathan and Sadia will combine their File System and Peer code together. This is will be done via a pair programming session
- `Dec 4, 2022`: Saar will finish the compact version of the Tracker and implement functions that allow the client to call the Tracker when its peer list dips below the threshold. Annie will have implemented the functions for the Connection Management. We will also start working on the progress document that is due on `Dec 5, 2022`.
- `Dec 5, 2022`: We will finish the progress document and submit it. Afterwards, we hope to combine all of the components together in a group programming session, start planning out the extra credit functionality that we would like to implement by `Dec 13, 2022`, and plan out when to start working on the final report.

# Testing

- **Tracker:** Testing for the tracker is quite easy as the tracker can send an HTTP request to a real tracker and receive an HTTP response back from it. This is how we have been testing the tracker up until now and how we will continue to test the tracker as we implement the compact version, implement repeated calls to the tracker, and as we conduct the integration of all of the components. The non compact version of the tracker has already been implemented with the peer and tested.
- **Peer:** Testing for the peer components that do not need to rely on the file system components is trivial, such as when sending choking/interested messages, receiving the bitfield and choking/interested messages from other peers, etc. However, when it comes

to functions that will call file system components (such as `handle_request_piece` that needs to read from a file), Sadia will test using "dummy" bits that she comes up with and check via Wireshark and `tcpdump` whether those same bits are being correctly transmitted on the network.

- **File System:** Similar to the Peer Testing, the File System testing relies on function calls from the Peer. Therefore, instead of writing to the file the exact bits that the client receives from its peers, Jonathan will have to come up with his own "dummy" bits, buffer these within the global dictionary, and write them to the file once the entire buffer storing a piece has been filled. Unfortunately, one issue that Jonathan will face is that the SHA-1 hash of the piece will not be correct unless he uses the real bits for testing. This is unavoidable and so, the best he can do is write the code as correctly as possible, merge with Sadia's code once she is done, and conduct integration tests afterwards.
- **Connection Management:** Connection Management is also not heavily reliant on any of the other components, much like the Tracker. Annie can send and receive handshakes, send periodic updates, and check whether peers have dropped the connection all independently by checking whether the correct response is received from the peer that she has sent the message to. She can do this by analyzing incoming packets on Wireshark.
- **Strategy:** The piece selection algorithm and the choking algorithm will essentially be implemented via the Peer and File System components and thus, the testing of these components explained above will automatically test these strategies.
- **Integration Testing:**
    - Peer and File System: As mentioned previously, the Peer and File System components are the most interdependent. When Sadia and Jonathan have finished their portions, they will test the workflow together, ensuring that the "dummy" bits that Sadia was reading from the file are the actual bits that are on the file and the "dummy" bits that Jonathan was writing to the file are the actual bits that Sadia was receiving from the peers. This will also be the point when Jonathan can check whether the SHA-1 hash code works correctly, as the hashes should now be correct when the entire piece comes in from a peer.
    - Connection Management: This component will be integrated after the Peer and File System are completely integrated. The integration testing for this should also be trivial, as it's simply a matter of triggering the correct Peer calls whenever we receive or send a connection management message. For example, if another client has initiated a handshake with us, we must send a handshake back and immediately call the `send_bitfield()` function to send the client our current bitfield. During the integration process, we will check whether this workflow is accurately executed by our client by sniffing packets on Wireshark.
    - Compact Tracker: Since the regular tracker is already implemented within the Peer, the Compact Tracker's integration testing simply consists of swapping out the current call to the tracker with the compact call and checking to see whether it can return and parse out the peer list.