Chapter 1

Implementation

1.1 Implementation tools

Since I was building a distributed system, it was useful to find a way to simulate multiple replicas locally on one machine. I therefore used Docker [?] to run instances of an application in a *container*, with the host's X11 socket being shared with each *container*, so that I could interact with each instance's GUI when it was built. This allowed for some basic bug finding in the library and application and so helped me to write unit tests to cover those cases.

I also made use of the JetBrains PyCharm IDE [?] to easily manage large numbers of files and find and replace across them. Its feature-rich debugger was also very helpful in finding sources of errors in the code.

1.2 CRDT Ordered List Library

I began the project by creating classes for common things I would be representing in the project, making use of Python's Object Oriented features. As shown in figure 1.1, I decided to differentiate between operations that originated from this replica (*local*) and from another replica over some network (*remote*). Thus, performing a *local* operation executes both ATSOURCE and DOWNSTREAM from the updating functions, whilst performing a *remote* operation executes only DOWNSTREAM. Additionally, performing a local operation uses the *cursor*, state kept locally to each replica, which is an identifier of a vertex v_C. Performing `OpAddRightLocal(`a`)` will insert the character a immediately to the right of v_C. Performing a `OpDeleteLocal()` deletes v_C.

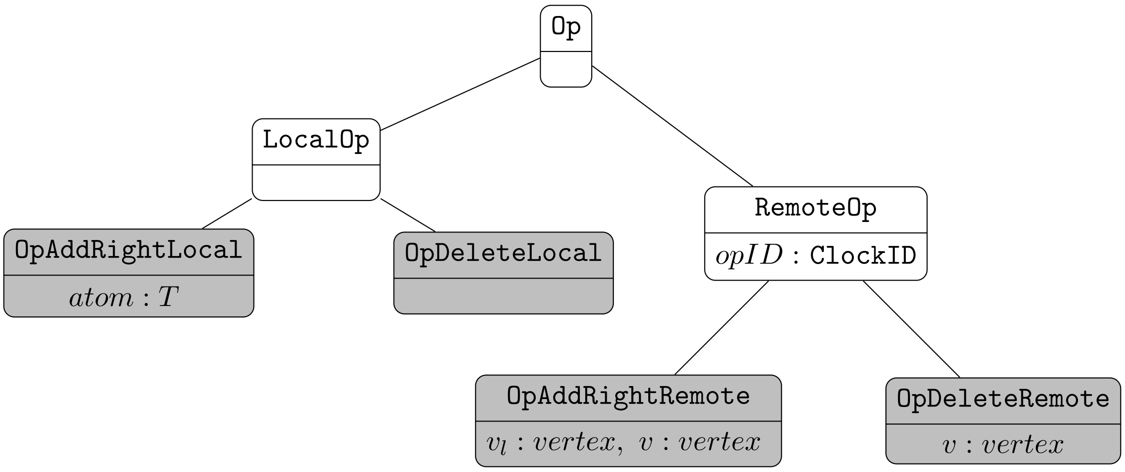

Figure 1.1: Class hierarchy for the supported CRDT operations. Shaded classes are concrete. Type T must be representable as a string.

I also created an `Identifier` abstract class to represent the identifiers in the vertices. Since RGA and LSEQ use different kinds of identifiers, they have different implementations of this (`ClockID` and `PathID` respectively).

I wanted to be able to perform operations on and get representations of the CRDT without knowing its underlying representation (the concept of encapsulation in OOP), so created another abstract class `BaseOrderedList` from which my implementations of RGA and LSEQ (`LLOrderedList` and `LSEQOrderedList` respectively) inherit.

Finally, the class `ListCRDT` was created to take a `BaseOrderedList`, be given `Op` objects to perform on it, and be queried for its representation. Performing a *local* operation returns the equivalent *remote* operation that should be performed at other replicas. For example, performing a `OpDeleteLocal()` with the cursor at v_C returns a `OpDeleteRemote(`v_C`)`, which will have the same effect on the representation of the list at other replicas.

A design decision was made not to support nested CRDTs (each atom is itself a CRDT) for simplicity, as moving cursors around them quickly becomes non-trivial. So, as long as an atom can be represented as a string (which all python objects can through the method STR), it is valid, and the representation of that atom is STR(*atom*).

Here is example Python code for how `ListCRDT` performs a `OpAddRightLocal` operation. It maintains a `ClockID` object *clock*, which has a timestamp, the *rid*, and an INCREMENT function to increment the timestamp. The *clock* holds the timestamp and *rid* of the last locally inserted vertex, so is incremented before inserting a new one. This is the implementation of the NOW function as described in the previous chapter. *self.olist* is a reference to a `BaseOrderedList` object.

```python
def addRight(local_add_op):
  # the atom to insert
  atom = local_add_op.atom

  # generate fresh timestamp
  self.clock.increment()

  # olist insert returns the new vertex and its predecessor in the list
  left_vertex, vertex_added =
    self.olist.addRight(self.cursor, (atom, self.clock))

  # update the cursor so that a further insert will insert just after the new vertex
  self.cursor = vertex_added.identifier

  return OpAddRightRemote(left_vertex, vertex_added)
```

Figure 1.2: How `ListCRDT` performs a `OpAddRightLocal`

1.2.1 RGA

Since the data structure the CRDTs represent is an ordered list, it makes sense to hold the vertices as a linked list, as insertion and deletion on arbitrary positions in the list are independent of its size (unlike Python's builtin `list`, an indexed array structure). However, in a standard linked list, the time taken to find an element is linear in its size, so to improve this I store a hash table mapping a vertex's identifier to the node in the list, giving a constant time lookup. Hence, both the AddRight and Delete functions can be performed on the list in constant ($O(1)$) time in the average case (if the dictionary is sufficiently balanced).

1.2.2 LSEQ

For LSEQ, it is possible to know where to insert a vertex purely based on its position. Therefore, AddRight(v_l, v) can be simplified to Add(v) (this will come in useful when implementing *undo* in section 1.2.3). However, this means a data structure that can support this is needed. So, the requirements for such a data structure are:

- Easy to find next and previous elements
- Fast lookup of an element given its identifier
- Fast lookup of the element with the largest identifier smaller than a given identifier

The linked list approach used for RGA would satisfy the first two criteria, but would require a linear scan for the third. The tree-like nature of the positions in LSEQ suggests a tree-like data structure, and indeed I found the SortedList [?]. It offers roughly logarithmic time insertion and deletion (verified in the Evaluation chapter), and has a method BISECT_LEFT for finding the index of the smallest element greater than a given one (meaning one less than this index points to the largest element smaller than the given one).

While implementing LSEQ, I came across a further improvement to the scheme [?]. The motivation was that replicas would all be choosing different schemes, so it would be better if they could collude. The h-LSEQ scheme achieves this by having a pseudorandom sequence generated by a seed shared by all replicas. Thus, each replica uses LSEQ, but all replicas have the same strategies.

1.2.3 Undo/Redo

Having seen a paper implementing a global undo for a CRDT-based editor [?], I decided to also implement an undo function, but only locally. That is, a replica can undo and redo the operations it originated, but no others. Being able to undo all replicas' operations seemed messy and frustrating from a user's perspective.

After an operation is performed, it is stored in a list specific to its originating replica (see section 1.3 for details). As shown in figure 1.3, the operations originating at this replica can be undone by popping them from *opStore* (for this replica), performing their inverse and pushing that to a special stack of undone operations. To redo, pop from *undoStack*, perform the inverse operation, and push back to *opStore*. To avoid a branching history, when a new operation originating at this replica is performed (one that has not been undone or redone), *undoStack* is emptied.

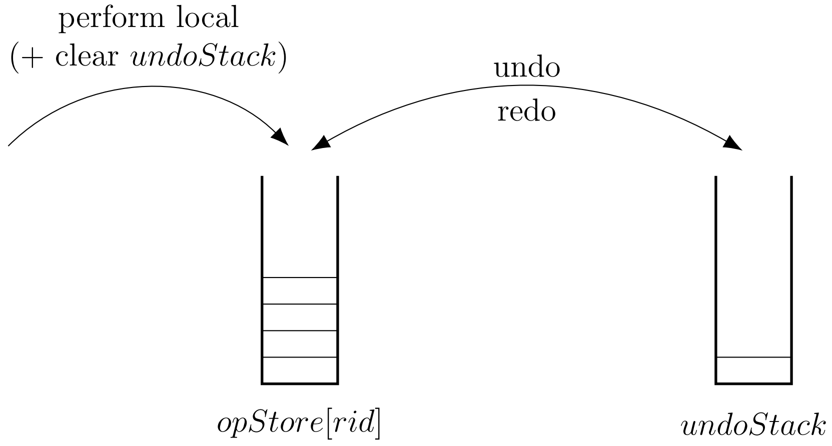

Figure 1.3: How operations are undone and redone at replica with id *rid*

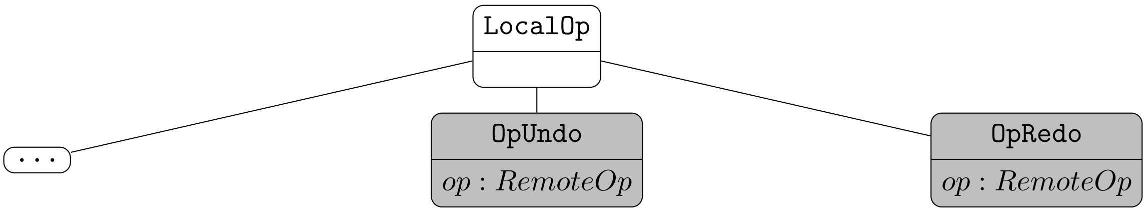

Figure 1.4: How OpUndo and OpRedo fit into the operation hierarchy

When the application initiates an undo (respectively redo), an `OpUndo` (`OpRedo`) operation is created. This is given the most recent operation from *opStack* of this replica (*undoStack*). The `ListCRDT` object has logic for taking an operation and performing the inverse of it on the `BaseOrderedList`. This UNDO function is the same for `OpUndo` and `OpRedo` as to redo an operation is to undo the effects of its undoing.

We see below that it is required for an `Add` to always insert a fresh vertex with the same atom, but a different identifier.

How UNDO inverts operations

```
function UNDO(OpUndo(Add($v$)))
    return Delete($v$)
```

```
function UNDO(OpUndo(Delete($v$)))
    let $(a, (p, rid, t)) \leftarrow v$
    let $t' \leftarrow$ NOW( )
    let $v' \leftarrow (a, (p, rid, t'))$
    return Add($v'$) !!!!!!!!!!!!!!!!!!!!!!!
```

With only the vertex to be deleted in stored in `OpDeleteRemote`, it is not possible to fully invert it to a `OpAddRightRemote` as the left vertex (v_l) is unknown. However, the LSEQ scheme doesn't necessarily require a left vertex, so we may leave out the left vertex property of the `OpAddRightRemote` and just consider a one argument constructor taking the vertex to be inserted. So, for brevity here I will refer to `OpAddRightRemote`(v) as `Add`(v) and `OpDeleteRemote` as `Delete`(v). This means undo isn't supported when using RGA.

1.3 Application

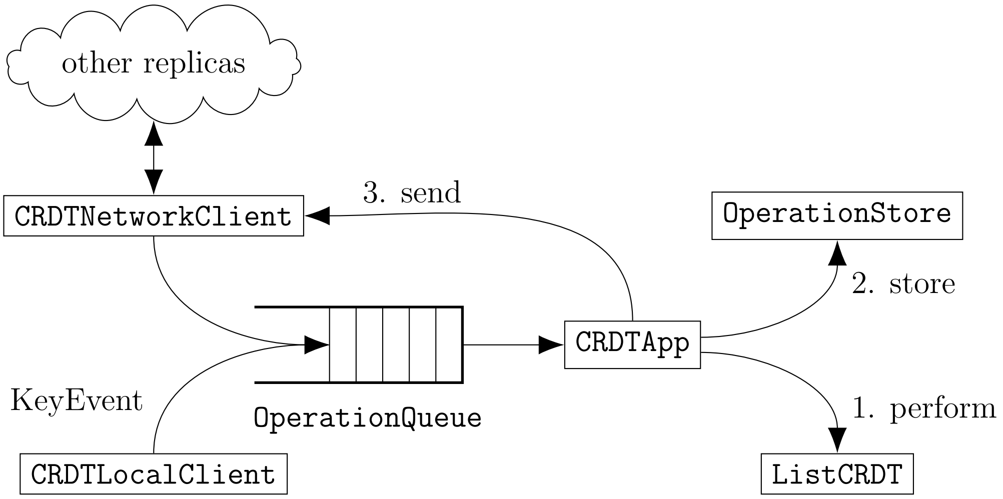

Figure 1.5: The layout of the application; arrows indicate the flow of operations. The numbers indicate the order in which `CRDTApp` does these actions

.

To build an application that uses the CRDT library required a few extra data structures to move around and store operations. The first problem to be dealt with is that operations will arrive asynchronously from both the network and any local GUI the application is connected to. It therefore makes sense to use the *producer-consumer* paradigm and have a queue of operations that the network and GUI feed into, and a separate thread pulls out of to perform them.

`OperationQueue` wraps around Python's `Queue` class, a thread-safe, double-ended queue. It adds a `Semaphore` (from Python's `threading` module) for condition synchronisation, such that the consuming thread can, on seeing an empty queue, block until something is available instead of busy waiting. When a producing thread adds to the queue, it increments the semaphore, and a consumer decrements it on receiving. If the semaphore is 0 a decrement will block until there is an increment. The application's implementation of the `OperationQueue` will be referred to just as *opQueue*.

I then created a class `OperationStore` which just has a `dict` of `list`s (Python builtins) and exposes operations on it.

Methods supported by `OperationStore`

```
function addOp(key, op)          ▷ stores op in the list indexed by key
function getOpsForKey(key)       ▷ returns the list indexed by key
```

Since it was the case that multiple threads may be accessing instances of this simultaneously, and Python has no *synchronised* primitive like Java, I needed some way of having thread-safe access to the structure. This was achieved with a *synchronised* decorator [**?**] to decorate the necessary functions such that only one thread can access the object they are tied to at a time.

Recall that operation-based CRDTs require that if A *happens-before* B, A should be delivered (and hence here performed) before B. To provide this, each operation sent (any `RemoteOps`) has an *opID* (instance of `ClockID`), a pair of a timestamp and *rid* showing the operation's source replica. Since a causal order on events implies the order of their timestamps, but not the other way round, from two *opID*s we can only infer one operation *happens-before* the other from their timestamps if their *rid*s are the same.

As will be seen in section 1.4, the way operations are sent guarantees operations with the same source replica will be ordered correctly. However, there needed to be a way of dealing with ordering dependent operations from different source replicas whose causal order cannot be inferred from their *opID*s.

As shown in figure 1.6, it may be that an operation arrives before a causal predecessor over a network. On seeing a timestamp more than one greater than the last we have seen from any replica, we don't know if we have missed one. For example, if there was also an operation 1:C, then on seeing 2:B, replica C wouldn't know if there was a 1:A coming, because it could have been 1:C that incremented B's clock. To resolve this ambiguity, I attempt to perform operations, and if they reference vertices that don't exist yet (ie have broken causal order), they get *held back* and are performed when that vertex does exist.

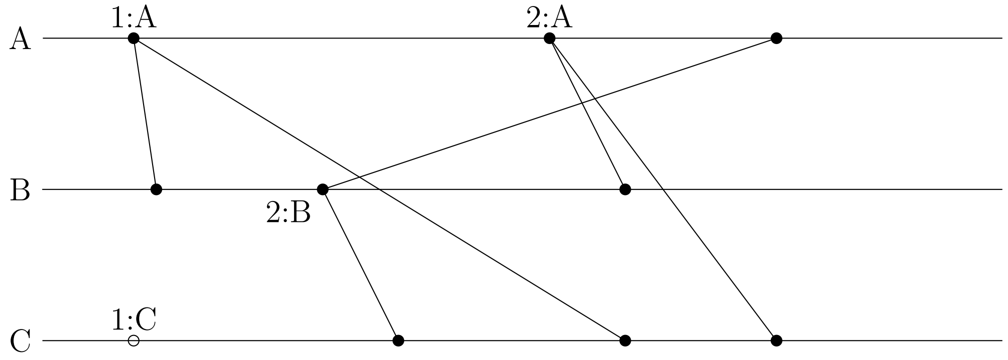

Figure 1.6: An example sending of operations with *opID*s 1:A, 2:A and 2:B

There is a implementation of `OperationStore` called *heldBackOps*, on which ADDOP(*vertexID*,*op*) is called for when the operation *op* fails which references a vertex with identifier *vertexID* (if `OpAddRightRemote` this is the identifier of the left vertex v_l and for a `OpDeleteRemote` is the identifier of the vertex to be deleted). Now, after performing an operation with identifier *opID*, we call *heldBackOps*.GETOPSFORKEY(*opI* and add any operations in the resulting list to the front of *opQueue* (called the *recovery* phase). This works because the *opID* of a `OpAddRightRemote` has the same timestamp and *rid* as the inserted vertex. So, on inserting the vertex, we free and perform any operations waiting on that vertex to exist.

Another structure needed is a way to keep track of which of whose operations a replica has performed, so that it may share operations with new replicas that come online to collaborate. Let this be known as *opStore*. This is an implementation of `OperationStore`, keeping lists of performed operations indexed in the `dict` by source replica. Operations in a list are kept in the order they were performed at this replica. So, each list will be ordered by the timestamps of the *opID*s as two operations with the same source replica cannot be concurrent.

As such, the `CRDTApp` has a thread which loops continuously over the following sequence:

- Pop from *opQueue*
- Perform the popped operation
- Store the operation that results from performing
- Send the operation to any other replicas connected
- Do *recovery* and check if any operations were waiting on this one to complete

Finally, in order to know who to send operations to (ie who you are currently collaborating with), there is a structure `ConnectedPeers` which wraps around Python's `dict` providing friendly methods for storing peer information and a thread-safe way to iterate over it. Applications each have an instance of this known as $connectedPeers$.

1.3.1 GUI

The first extension to the project I implemented was a simple graphical user interface (GUI) to allow quick identification of obvious errors in my code (helped even more by the use of Docker containers). This was achieved with the use of Tkinter [**?**], a Python interface to the Tk toolkit. Tk has a hierarchy of *widgets*, beginning with the *root*, which represent objects on the screen, where each child object is contained within its parent. Keyboard and mouse events are handled by a system of callbacks. A function may be bound to a specific event such that when that event occurs the function is called with the event information passed as a parameter.

With that in mind, I created a function to, given a keycode of a printable character, add a new `AddRightLocal` operation inserting that character to *opQueue*. If key was backspace, a `DeleteLocal` is created. This was bound to the *KeyDown* event so the relevant operations are performed when keys are pressed. If the Left (resp. Right) Arrow Key is pressed, code is executed which calls SHIFTCURSORLEFT(RIGHT) in ListCRDT. A reference to the latter function is passed to the object that contains the GUI code, `CRDTLocalClient`. These arrow keys could therefore easily be configured to traverse the document in some other way than just horizontally by one character. The keys to enact an undo or redo work by the same principle.

A Text widget from Tkinter is used to hold the CRDT's representation. This captures keyboard events and sends them to the aforementioned bound functions. It has a method for setting the position of its insertion cursor as well, which takes a line and column index. I decided to restrict the editor to only single-line text for ease of moving the cursor around (the only valid moves are to the next or previous vertices that aren't deleted) and so pressing *return* does not trigger an event as it is not a printable character.

The GUI needs to be updated every time an operation is performed, as all operations have some effect on the representation of the CRDT. To do this, the list of vertices is iterated over and for each vertex (a, id), $\textsc{str}(a)$ is appended to the output (if the vertex isn't deleted). Also, when the identifier held in the *cursor* is encountered, the number of atoms output so far determines the column number that the cursor should appear in the final output. This is needed for specifying the column index when the insertion cursor in the Text widget is set.

1.4 Networking

To exchange operations with each other, replicas need to communicate over some network. I chose TCP/IP as the communication mechanism to allow potentially global collaboration. TCP allows for reliable, inorder delivery of data, which specifically means operations from the same replica will be delivered in the order they were sent, a required property for op-based CRDTs.

To implement such a mechanism, I used Python's builtin `socket` library to create sockets which can be read from and written to. One socket listens for connections (server) whilst the other actively connects to a listening socket (client) and hence a channel is created between them. However, TCP is a stream-oriented protocol, and I needed some way to delimit separate operations.

Firstly, operations are sent by sending the actual `Op` objects. To serialize such objects into a format that can be sent, the `pickle` library [?] is used. The pickled data is sent prefixed by a 4 byte representation of its length. In this way, a receiver knows to first read 4 bytes from the stream (= *length*), then read a further *length* bytes to get the actual data (a technique called framing). It then unpickles this to get the `Op` object. The RECV function on the socket object to read from it takes a maximum number of bytes you wish to receive, and returns the actual number read. As there is no guarantee that you will get the number of bytes you were expecting all at once, I call this method until *length* bytes of data are actually received.

In order to support pickling/unpickling, a Python object merely has to override the methods _GETSTATE_ and _SETSTATE_. In the former one must return a `dict` with values of the object's properties necessary to recover the object, and in the latter the properties are set with values from this `dict`.

As my project supports offline editing, when a replica comes online having performed operations offline, it is necessary to both share those operations with other replicas and receive any new operations from them. The operations needed by replica r_k is

$$\bigcup_{i \neq k} \{ops_{r_i}\} - ops_{r_k}$$

where ops_{r_k} are the operations r_k has either in *opStore*, *opQueue* or is currently performing. Any connections the replica makes are in different threads, so that requests for new operations to each replica are pipelined (all requests may be sent before even one response is received). This decision was made over waiting for each response to arrive as the waiting time is bounded only by the timeout of the local TCP implementation which was deemed to be too long.

To calculate this, each replica keeps a vector of `ClockID` objects (the vector clock) for each replica that it has seen an operation originating from. This is updated when *opQueue* receives a new operation. Each object's timestamp is the highest timestamp of all the operations it has seen originating from that object's replica. `OperationStore` has a method for taking a vector clock and outputting a list of operations stored that haven't been seen by that vector clock. That is, for each `ClockID` c in the vector, it finds, in the list of operations it has stored for the replica corresponding to that object, any operations o for which $\textsc{timestamp}(c) < \textsc{timestamp}(o)$, then combines those into one big list. Since *opStore*'s lists are kept sorted, one can use a variant of the binary search algorithm to find the smallest timestamp in the list greater than c's, then return that element and all subsequent ones in the list as the result. For example, given the vector clock 5:A | 4:B and the lists [1:A, 2:A, 4:A, 6:A, 7:A] and [3:B, 5:B], the procedure would return [6:A, 7:A, 5:B]. Importantly, this process preserves the ordering of the operations, so the operations from each replica are sent in increasing order of their timestamps.

- connect semaphore (canconsume)
- connected peers !!!!!!!!!!!!!!!!!!

1.4.1 Client-server Architecture

The first network architecture I implemented was one where there is a central server which all replicas connect to (it is not itself a replica). The server simply relays operations between replicas and stores them locally. Thus, any new replica only needs to send its vector clock to the server rather than all replicas separately.

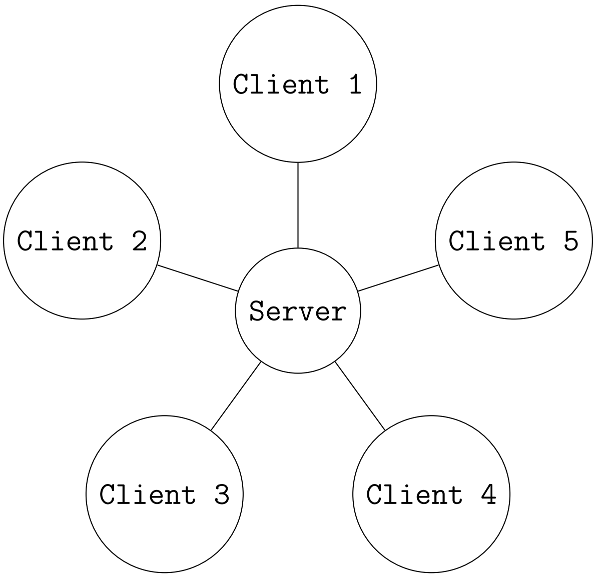

Figure 1.7: The standard client-server architecture with 5 clients

- server operation pseudocode onclientconnect and onrecvoperation

1.4.2 P2P Architecture

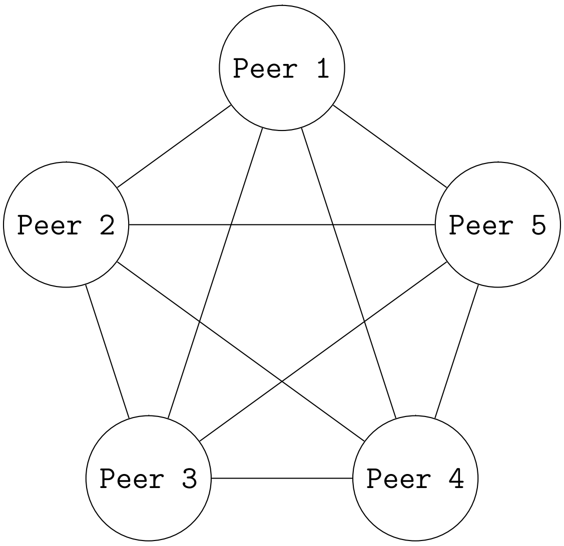

Figure 1.8: The standard P2P architecture with 5 clients

The P2P architecture (as shown in figure 1.8) consists of links from every node to every other node, making a complete graph. Therefore, it is necessary:

1. For each node to know how to reach every other node
2. For each node to listen for connections from as well as initiate connections to all other nodes

To deal with 1, a design decision was made to assume nodes who wish to collaborate know through some other channel the addresses of their collaborators. Another option considered was to have a centralized directory service which stores mappings of documents to node addresses. However, this mirrors the shortcomings of the client-server approach; namely that the directory is a single point of failure, and if compromised could reveal addresses people might want to keep private. Besides, to implement such a directory would require a new client to provide some identifier (such as a key) to the directory so they may be linked to a document. The identifier would have to be shared through some other channel anyway. Moreover, the Tor Hidden Service protocol deals with the issue of collaborators wishing their location to be hidden from the others.

For 2, it was necessary for each peer application to have a server socket accepting incoming connections and to try and open a socket to every other peer. However, this would result in two connections for each link shown in the diagram, which is undesirable. Therefore, it is necessary to check in *connectedPeers*, and not complete a connection to an already connected peer. Unfortunately, this is still not enough, as due to the socket threads running simultaneously, both connections could make progress simultaneously. So, asymmetry must be introduced so that both ends don't act in the same way. Hence, as shown in figure 1.10, peers send their name, and on detecting multiple connections to the same peer, an incoming connection is allowed only if the other peer has a lower identifier than yours (a total ordering).

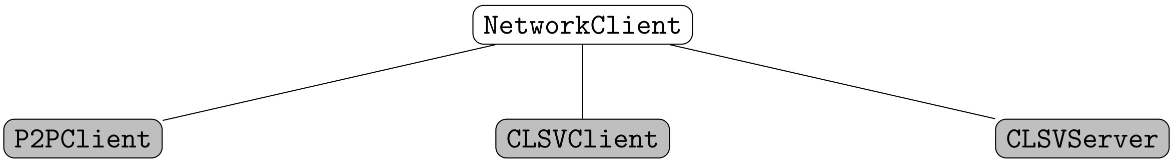

Figure 1.9: The networking classes used in the application. `NetworkClient` contains common code for sending and receiving data as described above.

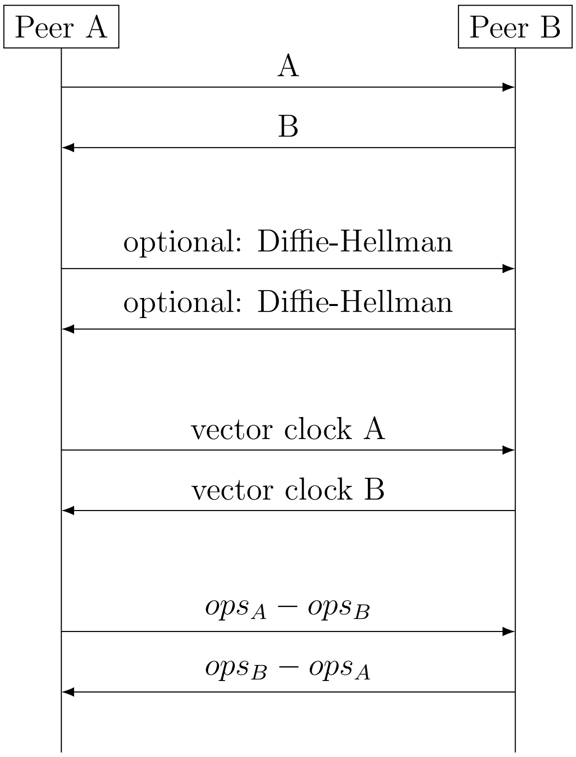

Figure 1.10: The protocol for peers connecting in the P2P architecture. After exchanging peer identifiers, they synchronize their operations.

- disconnecting

1.5 Encryption

As the project has a theme of privacy, encrypting the channels between nodes seemed sensible. This was achieved by adding an optional Diffie-Hellman (DH) key exchange [?] to the protocol, hashing the computed shared secret to generate an AES key, then encrypting all further communications over that channel using the key and AES CCM mode [?]. This is an authenticated encryption scheme that, provided it is used correctly, allows for confidentiality and integrity of all messages sent on the channel. Negotiating session keys for each channel means one can get *perfect forward secrecy*, meaning compromise of nodes doesn't reveal all past keys and thereby breaking confidentiality of all past messages. This use of asymmetric encryption (in DH) to bootstrap a symmetric encryption scheme is commonly used as symmetric encryption generally is faster.

1.6 Tor

The final extension added to the project was the use of the Tor network for communication in the P2P architecture. The idea is that each peer advertises a Hidden Service, and other collaborators know its *onion address* (again through some other channel), so connect to it through Tor's HS protocol.

Stem, the Python library used for interacting with the Tor process, exposes a method createEphemeralHiddenService, which can be called to setup a HS on a machine. It can be given a private key from which the public key (and subsequently the *onion address*) for the service can be derived. 'Ephemeral' here refers to how the HS created is kept entirely in memory and does not touch the filesystem at all.

Once the Tor process is running, to connect to a hidden service, one simply routes traffic through the local SOCKS5 proxy server the process creates. To do this, the `PySocks` library contains a socket object with the same interface as the Python's own `socket`, except with a setting to send traffic through a proxy server.

1.6.1 Basic Authentication

Using Stem and the createEphemeralHiddenService call, basic authentication can be implemented by simply passing an extra parameter. Namely, a `dict` with names as keys. The value for each of these can be `None`, in which case when the call returns and the HS is created a `dict` is returned mapping those names to randomly generated cookies. Alternatively, already generated cookies can be specified instead of passing `None`, in which case those are returned. The cookies are somehow distributed to other peers, then they use Stem to set their Tor process's `HidServAuth` parameter to, for each other peer c, a pair of c's onion address and the cookie for that peer.