Sussex University

Final year project

Scalable Distributed Computing using Hashing with Asynchronous I/O

Michael Sledge
CN:121677

Computer Science MComp
Department of Informatics

supervised by
George Parisis

2016

Statement of Originality

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Summary

Contents

1 Introduction

This project aims to create a system which enables the building of distributed computing applications. By implementing a consistent means of organising computing nodes and enabling communication between them without each node's applications needing knowledge of the other nodes in the system. The system will be scalable and allow a large number of nodes to communicate efficiently.

This report will cover professional and ethical considerations, related research and systems, more detailed technical requirements, details of the design and implementation of the system and evaluation of its functionality and performance,

1.1 Professional Considerations

The BCS Code of Conduct [1] sets out the following standards which may be relevant to this project: 2 c) "develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field." and 2 d) "ensure that you have the knowledge and understanding of Legislation* and that you comply with such Legislation, in carrying out your professional responsibilities."

From 'Professional Competence and Integrity' in Annex A Interpretation of the BCS Code of Conduct [2]: "You should seek out and observe good practice exemplified by rules, standards, conventions or protocols that are relevant in your area of specialism."

Existing libraries may be used in this project and their licenses must be followed, standards and conventions in languages used should be followed also.

2 Background Reading/Existing solutions

The chord protocol [3] uses consistent hashing to assign keys to nodes. It defines a set of functions which nodes use to build and maintain a chord network. Chord nodes in a network form a virtual ring in which each node is aware of its immediately adjacent nodes as well as several succeeding nodes. Nodes are mapped into the same keyspace used for identifiers for the application Each node knows how to find the node that an identifier maps to, illustrated in fig 1. If the id is between the current node and its successor then it maps to the successor, otherwise it asks the furthest node it knows about that is not past the id in the keyspace. The succeeding nodes that a node has knowledge of are stored in a 'finger' table (illustrated as dashed arrows in fig 1.) These are not necessary for functionality but they reduce the number of steps when mapping a key to a node.

The stabilisation that chord nodes run periodically is important to keep the network functional as the availability of nodes changes. Stabilisation is executed

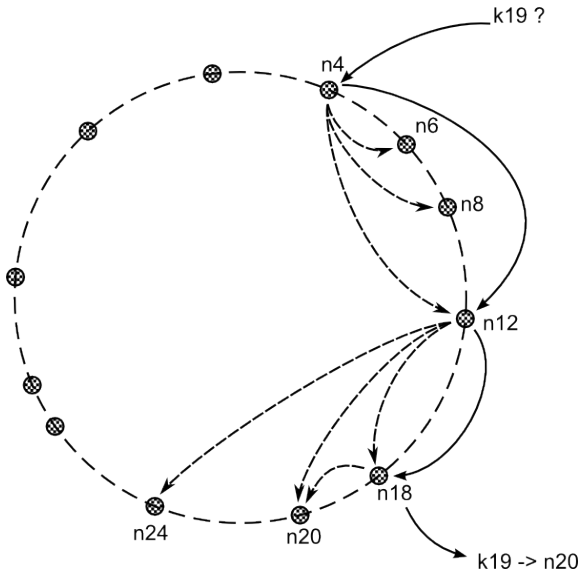

Figure 1: Finding the successor for key 19. Dashed arrows show which nodes a node has knowledge of (some have been omitted for clarity).

as follows: the node checks if there is a node between itself and its recorded successor, if there is it updates its recorded successor; then the node notifies its recorded successor of its existence so it can update its predecessor record if necessary. Nodes also periodically update their finger tables, and check whether their predecessor is still accessible.

Memcached [**memcached**] is a distributed in-memory caching system based on a distributed hash table intended for use in speeding up web applications by caching database entries, though it is possible to use it for other similar purposes. It is used by many large web companies.

Memcached uses spare memory on web servers to create a distributed in-memory cache which can be used to alleviate server load. This provides several benefits when compared to having each server managing it's own individual cache: the combined cache is much larger than individual caches; cached data is consistent any server requesting the a key in the cache will get the same value, whereas separate caches may not update data at the same time; improved scalability, as more servers are added to handle increasing load the size of the cache increases, making large scaling much easier than it would be with individual caches per server.

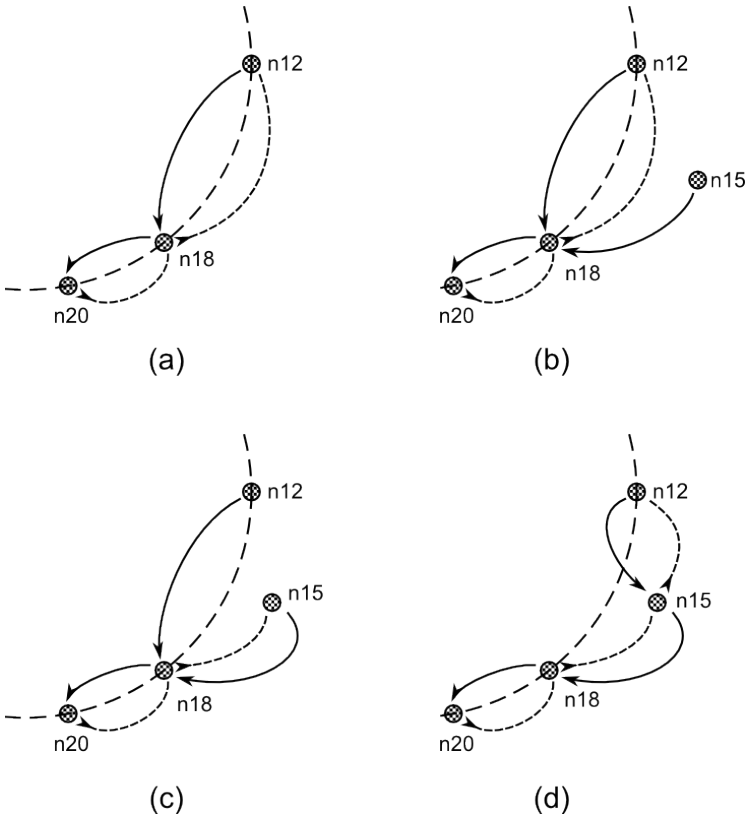

Figure 2: Node joining and stabilisation. (a) A chord network in a stable state. (b) new node n15 joins having found its successor (c) n15 runs its stabilisation and now n18 is aware of n15. (d) n12 runs its stabilisation and the network is back to as stable state

Mace [**mace**] is a toolkit for building distributed systems which "seeks to transform the way distributed systems are built by providing designers with a simple method for writing complex but correct and efficient implementations of distributed systems." [**mace˙quote**] It is designed to provide a common 'language' to design distributed systems without having to implement the entire system from scratch, it provides and uses libraries to build systems. The restrictions mace imposes in this way allow for better options for debugging a distributed system.

Mace consists of it's own domain-specific C++ language extension to allow users to implement node functionality, a set of libraries which implement the functionality necessary to create and maintain distributed systems as well as common features for such systems. This allows its users to concentrate on the system being distributed and spend less time on the implementation of the distribution itself.

Nodes in mace are made up of one or more layers and each layer is built on an event driven state machine. Like Chord, Mace creates a virtual overlay network in which each node keeps a record of adjacent nodes, periodically updating these records and notifying the application when changes occur.

Libevent [**libevent**] is a cross platform C library which provides a single API for implementing event driven network servers using one of a number of mechanisms systems provide for event notification, to allow for asynchronous I/O operations. Libevent checks for events and runs the callbacks specified for each event in the queue. An event can be a notification for a socket becoming read- or write-able, a new connection on a listen socket or a timer event among other things. Libevent also provides bufferevents to simplify the task of using read and write buffers with socket events. Libevent also includes a simple HTTP, DNS and RPC (remote procedure call) implementations.

Libevent provides a single API for event notification which can use a range of underlying mechanisms provided by different operating systems (currently /dev/poll, kqueue(2), event ports, POSIX select(2), Windows select(), poll(2), and epoll(4)).

Libevent is used in a range of widely used applications, systems and tools, including Memcached, the Chromium web browser, and the Transmission Bittorrent client, among many others.

Boost.Asio [**basio**] is a cross platform C++ library, similar to libevent, providing a consistent API for network and low level I/O programming with an asynchronous model. Its scope is slightly narrower than libevent's, so it doesn't include the simple protocol implementations.

3 Requirements Analysis

Build a high performance library for building DHT based systems with asynchronous I/O for scalability. Nodes should be independent and self organising, requiring no central authority, and be tolerant of joining, leaving and failing of nodes in the system.

3.1 Functional requirements

- Creating and joining a network
- Finding the node a key maps to from any node on the network. The algorithm finds the node a key maps to directly or finds the next hop and passes the job of finding it on to that node.
- Periodic stabilisation to account for change in availability of nodes. Periodic checking for failures of adjacent nodes and arrival of new nodes. New nodes must notify at least one node of their presence.
- Allow sending of messages or data to other nodes for the application. Once the node a key maps to has been found allow data to be sent to that node.
- Provide callbacks for the application when a message is received. Allow the application programmer to provide functions which are called when specific events occur.
- Provide callbacks for notifying the application of changes in the availability of nodes that may affect it, i.e. the next and previous nodes. Specifically when the set of keys a node is responsible for changes.

3.2 Non-functional requirements

- Provide high performance network communication between nodes
- Be highly scalable both in terms of number of nodes and the load of individual nodes assuming the application is able to handle such loads.
- Ease of use.

4 Design and Building

4.1 Design Process

The design involves multiple modules or layers required considering both the internal functionality and the interface provided to users of the finished library.

4.2 Design Principals

Several design principals were employed to help reduce the chance of unhandled errors occuring and to simplify the task of building a complicated system.

No global state This makes it easier to create thread-safe code and allow multiple instances to be instantiated within a single process. It allows for a more functional approach wherein all state has to be passed to functions as a arguments, this will mean that they can be implemented so to, as far as possible, always behave consistently way when given the same input.

Layered structure The code will be organised into separate layers which utilise the layers below them but do not assume knowledge of layers above. this minimises interdependencies between modules. This allows easier management of the project and means each component can be understood on its own, thus simplifying the task of building them.

Defensive programming Errors in software are inevitable, however they can be reduced and managed with some thought. For example by checking all function input and checking function return codes to catch errors before they cause further problems, allowing

4.3 Design Details

Overlay network To facilitate the self-organisation of nodes and communication between them, a virtual 'overlay' network is created. It's design is based on the Chord protocol as referenced earlier. Each node on the network is assigned an identifier in the form of a hash of a user supplied parameter or 'name'. The overlay network itself takes the form af a virtual ring of nodes ordered by their identifiers, in which each node is aware the real location (i.e. IP address and port number) of and can communicate with one or more nodes with identifiers proceeding it's own. Figure 3 shows how nodes in a real network map into the virtual network.
To maintain the network, each node is aware of several other nodes, specifically their immediate successor and predecessor, as well as several nodes in the finger table for improved lookup performance.

Routing The main functionality of the network is mapping identifiers to nodes, because at minimum each node knows about its immediate successor, this is be achieved by checking if the ID is between that of the current node and that of its successor, if it is then the ID maps to the successor, otherwise the successor is asked to find the node which the ID maps to. This works correctly, however in the worst case it requires querying every node in the system in turn to find the mapping. An improvement to this system is for each node to keep a cache of several nodes succeeding it, this is called the *finger table* and consists of a list of nodes after the current at progressively larger intervals. Specifically

it contains m entries where m is the number of bits in identifiers. Entry k refers to the node that is the successor of $(n + 2^k) mod 2^m$ where n is the identifier of the current node. The *finger table* is used for mapping by asking the last node in the finger table with an ID lower than the one being mapped rather than just the successor. Figure 4 shows the improvement using a finger table can make for lookup.

Stabilisation There are several things which need to be done in order to maintain the network as nodes may join or leave at any time. The first stage is to check the availability of the nodes successor as this is the main point of contact with the network, for this reason each node keeps a list of several successor nodes for use in the event that the immediate successor becomes unavailable.
The stabilisation process consists of several stages, firstly a node checks that the successor nodes are available and removes any which aren't. Next the node verifies it is in the right place on the network by asking it's recorded successor for it's predecessor. This will either be itself, a new node that has joined the system or none in the case that it's original immediate successor has become unavailable. The node is now aware of its successor, either it hasn't changed or it has the been given a new one. The next step is to 'notify' the successor so that it is aware of its predecessor for use in future stabilisations. Nodes also periodically check the availability of their predecessors.
The last stage is updating the finger table, which is simply a matter of asking the successor to find the successor of several pre-determined identifiers as defined in the routing description.

The Network layer The network layer handles open connections as well as the listen socket. It mostly acts as a wrapper for libevent and provides a simplified interface for the layers above. The main data structure is `struct net_server` which is an instance of the net layer which needs to be passed to the majority of functions. It holds references to an event base (libevent's main data structure); open connections; and the listen socket. The layer contains functions for creating, activating and configuring connections.

Usage Initialise a new server instance by providing a listen port number and a callback function and argument for accepting incoming connections. Once any other setup is complete, the function `net_server_run` can be called which starts the event loop and does not return until the loop exits and as such users must start another thread if they wish for their application to do more than wait for incoming connections.
On receiving notification of an accepted connection (via callback), the application should set read, error/disconnect, and optionally write handlers for the newly accepted connection.
Users can create outgoing connections by providing a remote IP and port this sets up the connection internally but does not activate it, this allows the user to add data to the write buffer before connecting if desired.

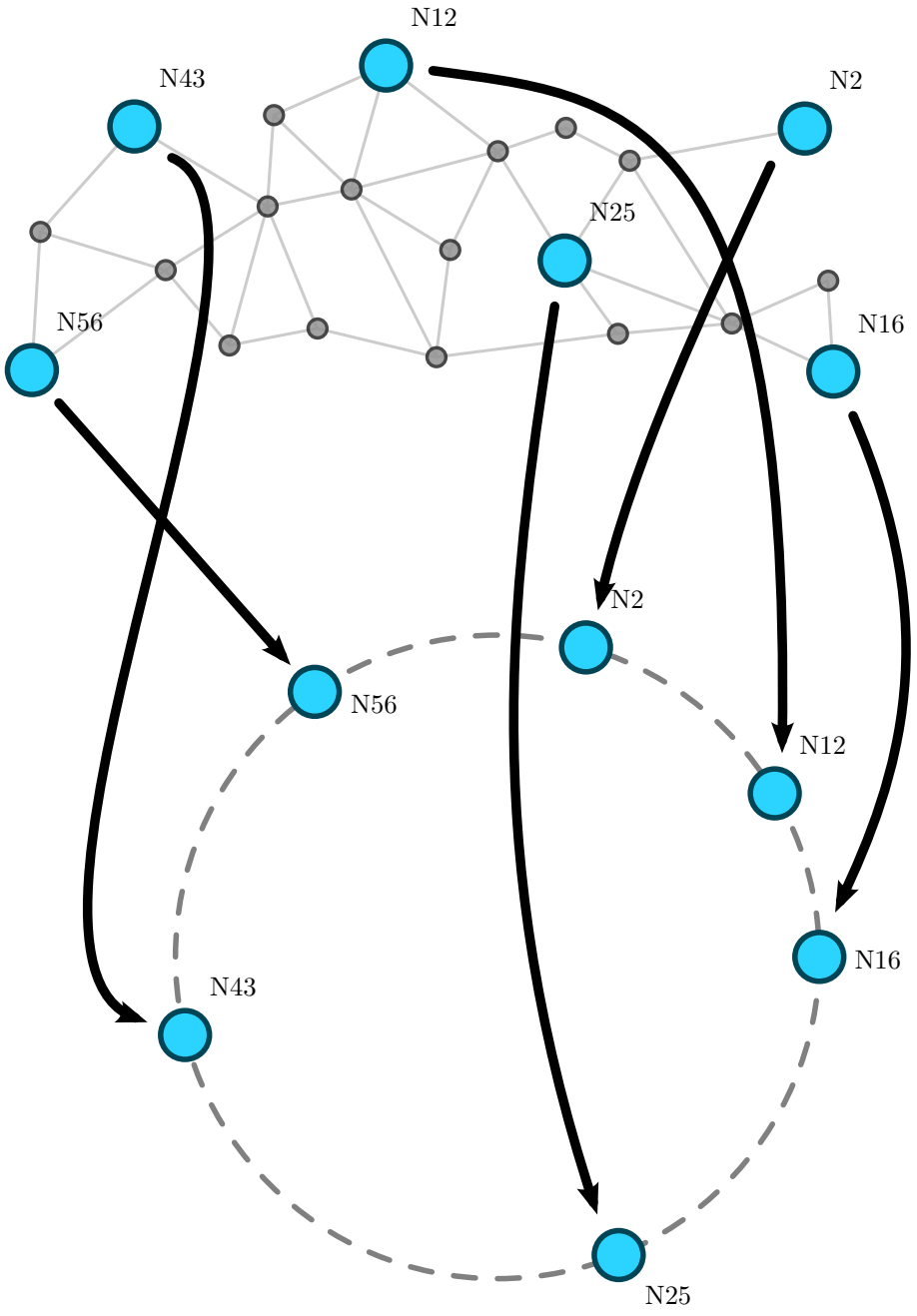

Figure 3: Mapping of nodes on a real network (e.g. the internet) to a virtual overlay network. Each node is aware of at least the next node in the ring

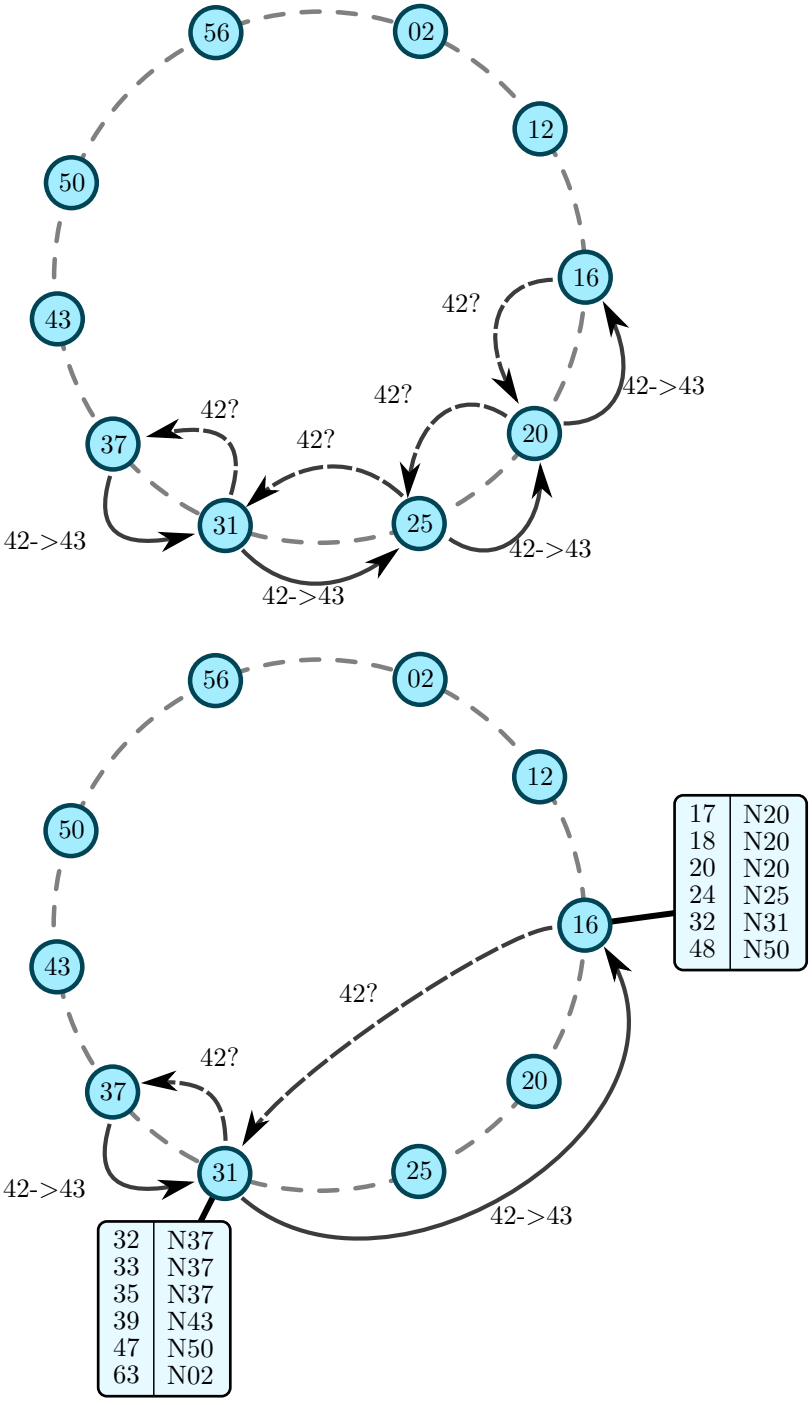

Figure 4: Use of a finger table improves lookup performance by reducing the number of hops to find the node an ID maps to.

Both accepted incoming and outgoing connections use the same interface, allowing setting of callback functions for read, write and other events; setting of the callbacks' custom argument; getting the connection's read and write buffers; setting the connection's timeouts; getting the remote address (useful for some types of messages on incoming connections); and getting the underlying buffer event for finer control

Internals The net server keeps a list of references to open connections and keeps track of the callbacks that have been set for each one. It also has a mutex lock for the connection list to prevent concurrent modification as it is likely that the event loop thread may accept a connection while another thread requests creation of an outgoing connection.
Connections use libevent's `buffer_event` which can be used to automatically read incoming data from a socket to a buffer and write data to a socket from a buffer as the socket becomes readable and writeable respectively, they can be set to invoke there callbacks when the buffers contain more or less than a given amount of data.
Most of the network layer's functionality is that for interacting with connections

The Node layer The node layer is the main body and contains code for creating, joining and maintaining the overlay network and routing, these are together due to the fact that joining and maintaining the network requires routing messages between nodes. Much like the net layer its main data type is a handle to a node instance, the interface similarly consists of a function to initialise a new node instance and functions to join an existing network or create a new one.

Usage Users of this library will setup a node by calling `node_create`, then setting up callbacks for incoming messages and join and leave notifications if necessary.
Then the user can call `node_network_join` with the address of an existing node to join an existing network or `node_network_create` to create a new network. Both functions will call a given callback when he network has been joined or created respectively.

Internals A node instance contains a reference to its own net server instance. The node layer can be split into a few sub modules: Handling incoming messages from other nodes; network maintenance; message routing; initiating communications with other nodes
Routing: the most important functions are `node_find_successor` and `node_find_successor_remote` as they are used directly and indirectly by most other functions. These functions map IDs to nodes by checking if the current node or it's successor is the successor of the given ID and if not asking another node the same thing until the successor is found.
Stabilisation and network maintenance: There are three groups of functions

which are run periodically, they are: stabilise, which involves ordering nodes correctly in the overlay network; fix fingers, which ensures the node's finger table is up to date; and periodic checks of the availability of neighbouring nodes. The stabilise function updates the node's successor if necessary, by asking the node currently recorded as the successor for it's predecessor, and then notifies the updated successor that the current node is it's predecessor. The fix fingers function iterates through the positions in the finger table and finds the relevant node, again using `node_find_successor`, and updates the table. Periodically The availability of the node's predecessor is checked by sending it a message.
Incoming handling: When the network layer notifies the node layer that an incoming connection, the node layer sets up the callbacks for the connection. Once the header of the incoming message has been received, the type and content length are read and the relevant callbacks for each type of message are setup or called immediately if the content is already all in the connection's read buffer. There are functions for handling each type of incoming message.
Utility functions: There are several utility functions used by several other functions which simplify their own code. `node_send_message` takes a `struct node_msg` formats it and writes it to an open connection, the similarly named `node_connect_and_send_message` opens a new connection to the specified node and calls `node_send_message` with it. `node_id_compare` takes two identifier hashes and returns 1, 0, or -1 to indicate that the first is greater than, equal to, or less than the second respectively. `node_id_in_range` takes three ids identifier hashes and returns 'true' if the first falls between the second and third in the identifier space. The `get_id` function takes a string and maps it to an identifier hash

ID hashes The hashing function chosen for the identifier space is SHA-1. SHA-1 provides 160 bit hashes which is a relatively small space compared to other cryptographic hash functions and has a higher probability of collisions. This however, is not a cause for concern as collisions would only have an effect on the system if two nodes were to have the same ID, which due to the relatively low number of nodes is highly unlikely. The library uses the OpenSSL [**openssl**] SHA-1 implementation.

Communication Protocol Messages sent between nodes use a TLV (type-length-value) format, in other words they consist of a header, containing a single byte indicating the type of message and a 4 byte value indicating the length of the body, and the body itself which contains the actual information formatted according to the message type.

4.4 Using the Library

Most interaction with the library is with the node layer, though some interaction with the net layer is necessary. The first stage is initialising a node with a listen port and a unique name used for the nodes identifier by calling `node_create`.

This allocates all necessary data structures and sets up the network layer. Next calling `node_network_create` or `node_network_join` providing a callback for when the network has been successfully joined and, in the latter case, the address of an existing node in the network. This starts the event loop and as such does not return until the node is shut down, This is why the callback is necessary, the application should utilise it to start one or more threads for its functionality. When a node receives a message for the application, a users specified callback is called with the message, the application should use this to pass the relevant information to its processing thread(s) to prevent blocking of other network activity. Figure 5 shows an overview of the process of setting up and running a node, and figure 6 shows the main interactions that an application will have with the library as well as the major internal interactions

4.5 Example applications

1. textsend This is a simple command line application to demonstrate basic functionality. It will join an existing network if given the address of a node, or create a new one. Once in a network this application takes text input from `stdin` and sends it to the node it maps to. Upon receiving a message from a node it is printed to `stdout`

5 Testing

Much of the code in this project is difficult to test because of the fact that much of it depends on network communication and having the system running, for that reason most testing was done by setting up a small network of nodes running a simple application.

5.1 Simulated real world testing

6 Evaluation

7 Conclusion

have succeeded in building a library for implementing distributed systems, which provides an interface for sending messages between nodes and handles organising and maintaining the network using asynchronous I/O for network communication

7.1 Future work

There are several enhancements I would like to make beyond the current relatively simple functionality,

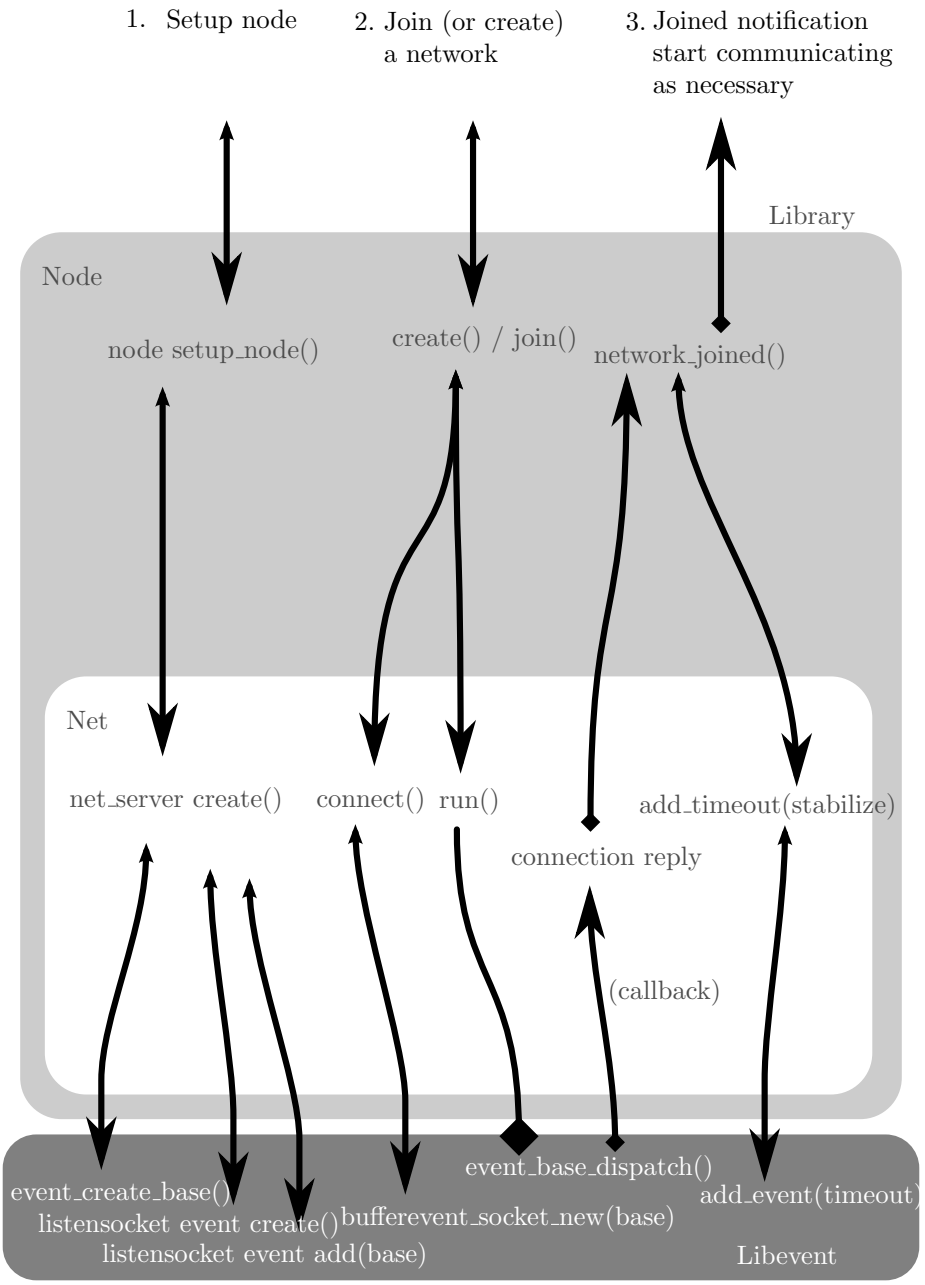

Figure 5: Process of setting up a node with the library, first call setup with a name and listen port number, then join or create a network, once this is done a given callback is called to allow the node to start its own functionality which should be run in separate threads.

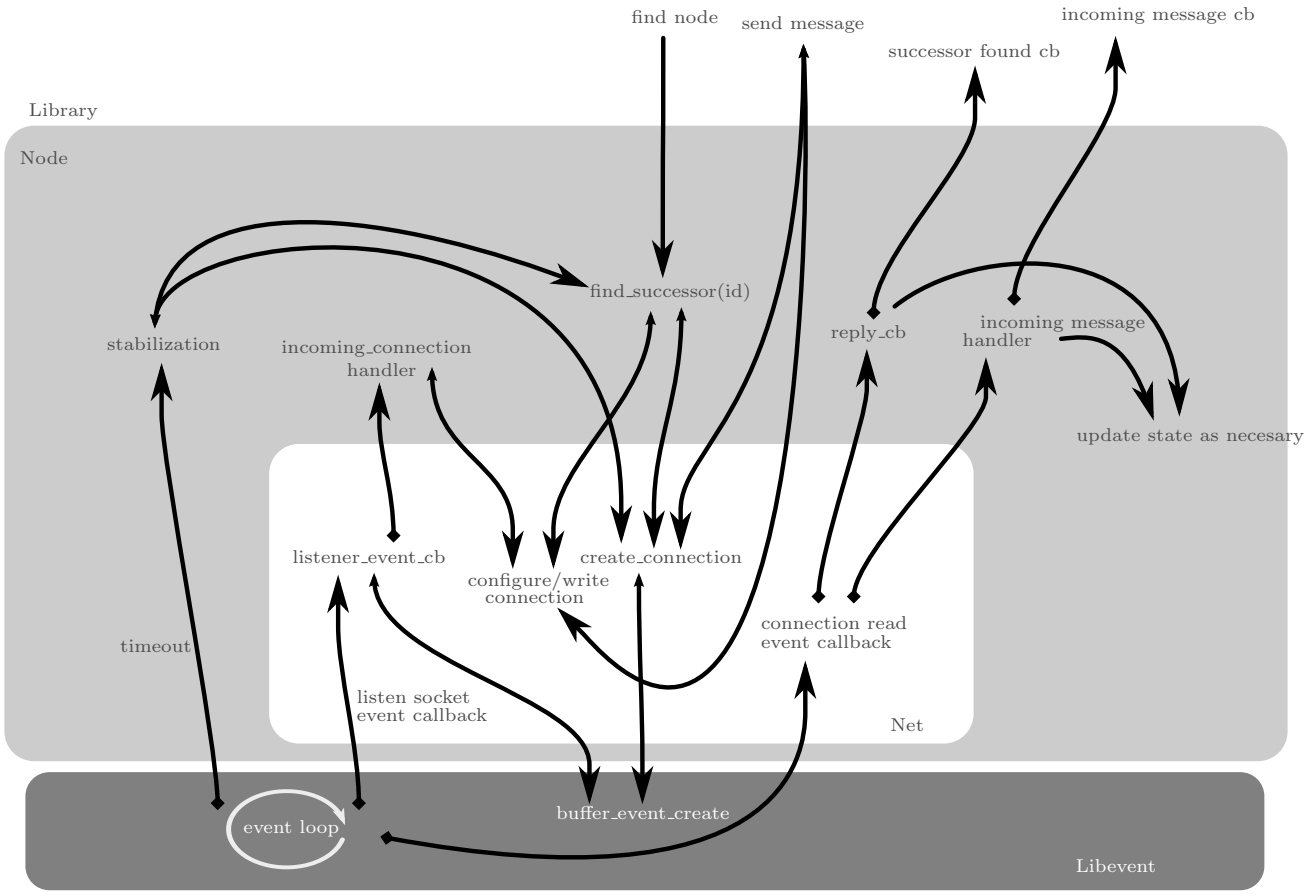

Figure 6: Main interactions with and within the library once running

Security I would like to implement SSL based connections between nodes to allow for secure communication and to allow for verification of message origin

8 References

References

[1] *BCS Code of Conduct*. URL: `http://www.bcs.org/category/6030`.

[2] *BCS Code of Conduct, Annex A, Interpretation of the BCS Code of Conduct*. URL: `http://www.bcs.org/content/ConWebDoc/39988`.

[3] Ion Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications". In: *IEEE/ACM TRANSACTIONS ON NETWORKING* 11.1 (Feb. 2003), pp. 17–32.

9 Appendices

A Message protocol

Messages are made up of a single bit defining the type of the message; a 4 byte value for the length of the message body; and the optional, variable length message body.

A.1 Type bit values

`'S'`	Ask node for successor of id
`'s'`	Return successor of id
`'P'`	Ask a node for its predecessor
`'p'`	Returning a node's predecessor
`'N'`	Notify another node that the sender believes it is that node's predecessor
`'A'`	Used to check the availability of a node
`'a'`	Reply to confirm that a node is operating
`'M'`	The contents of the message is for the application using the library
`'O'`	Signifies Unknown message type

A.2 Body Layouts

'S'	ID	The ID to find the successor of
's'	'Y' ID IP port	the char 'Y' to indicate success followed by the i.d. i.p. and port of the node found
'P'	N/A	no body necessary
'p'	'Y' ID IP port	the char 'Y' to indicate success followed by the i.d. i.p. and port of the node's prede
'N'	ID port	the i.d. and listen port of the node dong the notifying (i.p. address obtained from co
'A'	N/A	no body necessary
'a'	Y	simple confirmation message
'M'	N/A	Application dependant