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.

Signature	

Summary

Distributed systems are challenging to design and build, Distributed hash tables can be used to implement many types of distributed system. Specifically DHTs are decentralised, self-organising, and able to adapt to changes in the set of individuals that make up the system. High performance network communication is important for implementing such a system if it is to be highly scalable, and Asynchronous I/O provide a highly scalable means of handling I/O operations.

The library produced provides a simple interface for building networks of large numbers of nodes and routing messages between them efficiently. The networks it builds are self organising and are resilient against failures of individual nodes in them. It uses consistent hashing for routing purposes and uses Asynchronous I/O for network communication allowing for high scalability.

This report outlines the inspiration for, and design and implementation of my high-performance DHT based library for building decentralised, self-organising, peer-to-peer systems. It explores similar existing solutions and design ideas used; details of the design and implementation of the library and example applications using the library.

Contents

1 Introduction

This project aims to create a system which enables the building of distributed computing applications, based on distributed hash table (DHT) lookup and routing. The system will allow the creation of programs which are run on multiple computers connected via a network (the internet) forming a self-organising decentralised peer-to-peer network, by implementing a consistent means of organising computing nodes and enabling communication between them without each node's applications needing direct 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 existing similar distributed systems, as well as asynchronous I/O implementations; 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

2.1 Distributed and DHT based systems

2.1.1 Distributed Hash Tables

Distributed Hash Tables (DHTs) are a method of distributing data or workloads evenly among a number of systems. As the name suggests they function like hash tables or hash maps, where data is mapped to *buckets* (separate computing nodes in a DHT) by hashing a *key* and using the hash to assign a bucket. In DHTs each node is given ownership of a section of the keyspace, with an *overlay network* connecting the nodes to each other to allow them to find the owner of a particular key. DHTs are generally self-organising and decentralised making them more robust than systems which rely on a central server, and are designed with a large number of nodes, in the thousands or even millions, in mind.

DHT specifically refers to a storage and retrieval system, though the concept can be adapted and used for much more general purposes such as dividing computational work between many systems.

2.1.2 The chord protocol

Chord [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

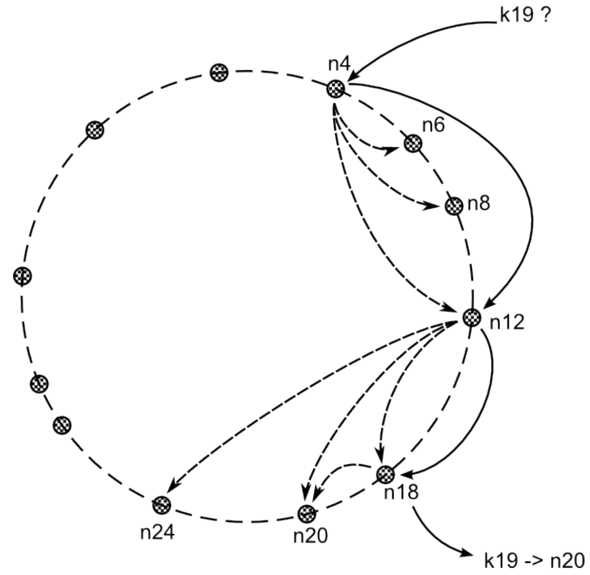


Figure 1: Finding the successor for key 19. Dashed arrows show which nodes a node has knowledge of (some have been omitted for clarity).

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 figure 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 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. Figure 2 shows the stabilisation process after a new node has joined the network.

There is an more extensive discussion of the chord protocol, along with my specific implementation in the design section.

2.1.3 Pastry

Pastry [4] is an implementation of a scalable, decentralised, self organising peer-to-peer routing system. Pastry assigns random 128 identifiers to nodes and routes messages by sending them to a node with a longer matching key prefix than the current node or to a node with a matching key prefix of the same length but with a closer value if none are known. To facilitate this a node's routing table contains several rows which have entries referring to nodes, where nodes in row n share an *n-digit* key prefix with the current node. Pastry nodes also has a *leaf set*, a list of the nodes in the network with the numerically closest identifiers both above and below that of the current node, and a *neighbourhood set* containing references to the nearest nodes according to a proximity metric (e.g. IP routing hops or geographical distance). Together the routing table, leaf set and neighbourhood set form the nodes state tables. Pastry favours routing through nodes close to the current one, and so routes chosen for messages are likely to be reasonably physically short.

Pastry nodes join the system by sending a join message to a nearby node, this message is routed to the node with closest numerical ID to the new node, all nodes that the join request is routed through

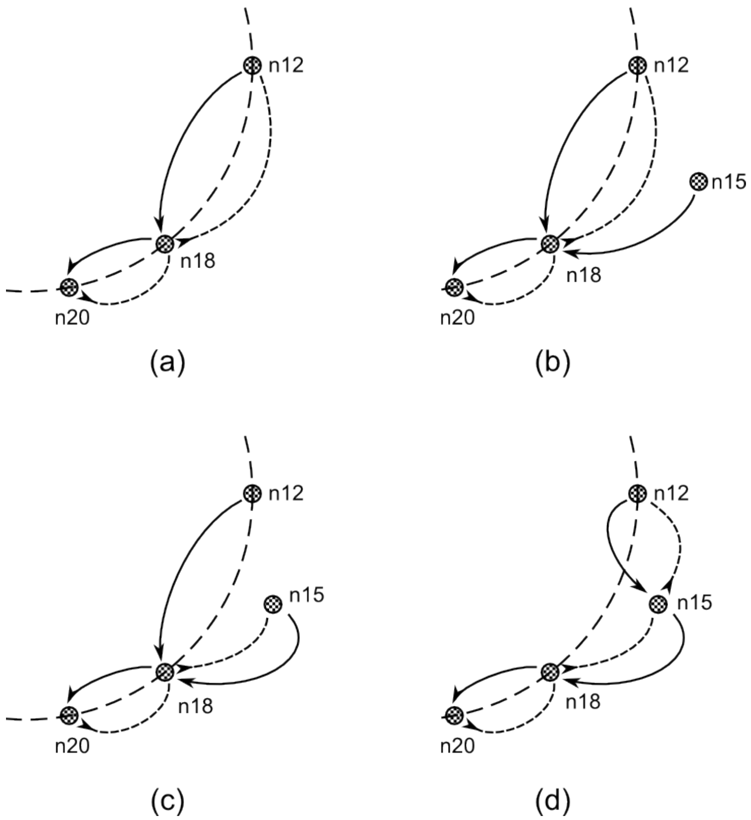


Figure 2: Node joining and stabilisation. (a) A chord network in a stable state. (b) new node n15 joins having found its successor (c) n15 runs its stabilisation and now n18 is aware of n15. (d) n12 runs its stabilisation and the network is back to as stable state
Based on Fig 7. from Chord[3]

send their state tables to the new node which uses this information to begin to populate its own state set and request more information from other nodes where necessary to complete it. Once this is done, the new node sends a copy of its state table to all nodes referenced in it which then update their state as necessary. The new node is now part of the network and messages can be routed to and through it.

When a node cannot be contacted by its immediate neighbours it is considered failed, to deal with such failures, nodes which had the failed node in their leaf sets get the leaf set of the node in the leaf set furthest from the current on the side of the failed node and uses the information to maintain its leaf set. If a node in a nodes routing table has failed, it will be detected when no reply is received from an attempt to route through it, when this happens the node can routes the message through another node instead, then it asks another node in the same row for an entry to replace the failed node.

2.1.4 Tapestry

Tapestry [5] is another DHT based scalable, self organising infrastructure that efficiently routes requests to content with emphasis on fault tolerance and robustness with regards to heavy load and node and network faults through redundancy. Similar to pastry, tapestry nodes have a *neighbour map* which takes the form of a table, storing lists of the closest network nodes with matching identifier suffixes of increasing length. Nodes also store a list of nodes which refer to them in their own neighbour maps. To provide better fault tolerance, where tapestry stores multiple copies of data nodes en route store the location of all copies, and allows applications which wish to retrieve such a copy to define a selection operator to choose which copy to retrieve.

When a node failure is detected, through TCP timeouts while routing or using heartbeat packets, the failed node is marked as invalid, and an alternative node is used to route messages. Marking a failed node as invalid instead of simply removing it from the routing table allows it to be re-enabled in the event that it was only temporarily unavailable. Redundancy is implemented by appending a sequence of 'salt' values to object IDs so that they can be mapped to several different nodes after hashing.

2.1.5 Memcached

Memcached[6] 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.

Memcached is made up of several components: client software, for interacting with the cache; a hashing algorithm, for mapping data to servers, as in most DHT based systems; The server software, which actually manages storing and retrieving data; and an LRU system for managing cached data and discarding old data when necessary. Memcached differs from DHTs in that its servers are not aware of each other, but the client software manages mapping keys to servers, this allows for high performance in data retrieval.

2.1.6 Mace

Mace[8] 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.

2.2 Asynchronous I/O

The most simple I/O calls are synchronous and therefore blocking, that is to say that when a function is called (e.g. read from a socket) it does not return until the operation has completed or encounters an error. This makes handling multiple operations concurrently without a separate thread for each which causes a large amount of overhead when scaling an application which uses such a system.

Asynchronous I/O allows an application to initiate I/O operations and continue with other tasks such as handling completed operations while waiting for them to complete. This system is much more scalable, allowing for instance a server to handle thousands of requests with as little overhead as possible. It can also be combined with threads to handle non-I/O based workloads.

Most operating systems provide an method for implementing Asynchronous I/O calls, although their functionality and differ somewhat, fortunately libraries which provide a consistent interface across multiple platforms exist to simplify their usage.

2.2.1 Libevent

Libevent[9] 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.
Using Libevent involves setting up an `event_base`, adding one or more events to it and running the event loop. Events can be setup to be triggered by a number of conditions: a file descriptor being ready to read from and/or write to; a timeout expiration; a signal occurring; or manual triggering.

Internally Libevent’s event loop checks for triggered events and marks them active, then runs the user set callbacks for all active events which have the highest priority and then moves to the next iteration.

Also provided are buffer events which use internal callbacks to read from the events file descriptor into a buffer and write from another buffer to the file descriptor when the file descriptor is ready and then call user set callbacks when the buffers are above or below user set watermarks. The Libevent book[10] provides a detailed overview and examples of the usage of much of libevent's functionality.

2.2.2 Boost.Asio

Boost.Asio[11] is a cross platform modern C++ library, similar to libevent, providing a consistent API for network and low level I/O programming with an asynchronous model, supporting IPv4 and IPv6, TCP and UDP, and SSL. It allows use of both synchronous and asynchronous strategies and provides interfaces compatible with std::iostream for ease of use. 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. (Desirable but not necessary)

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 occurring 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.

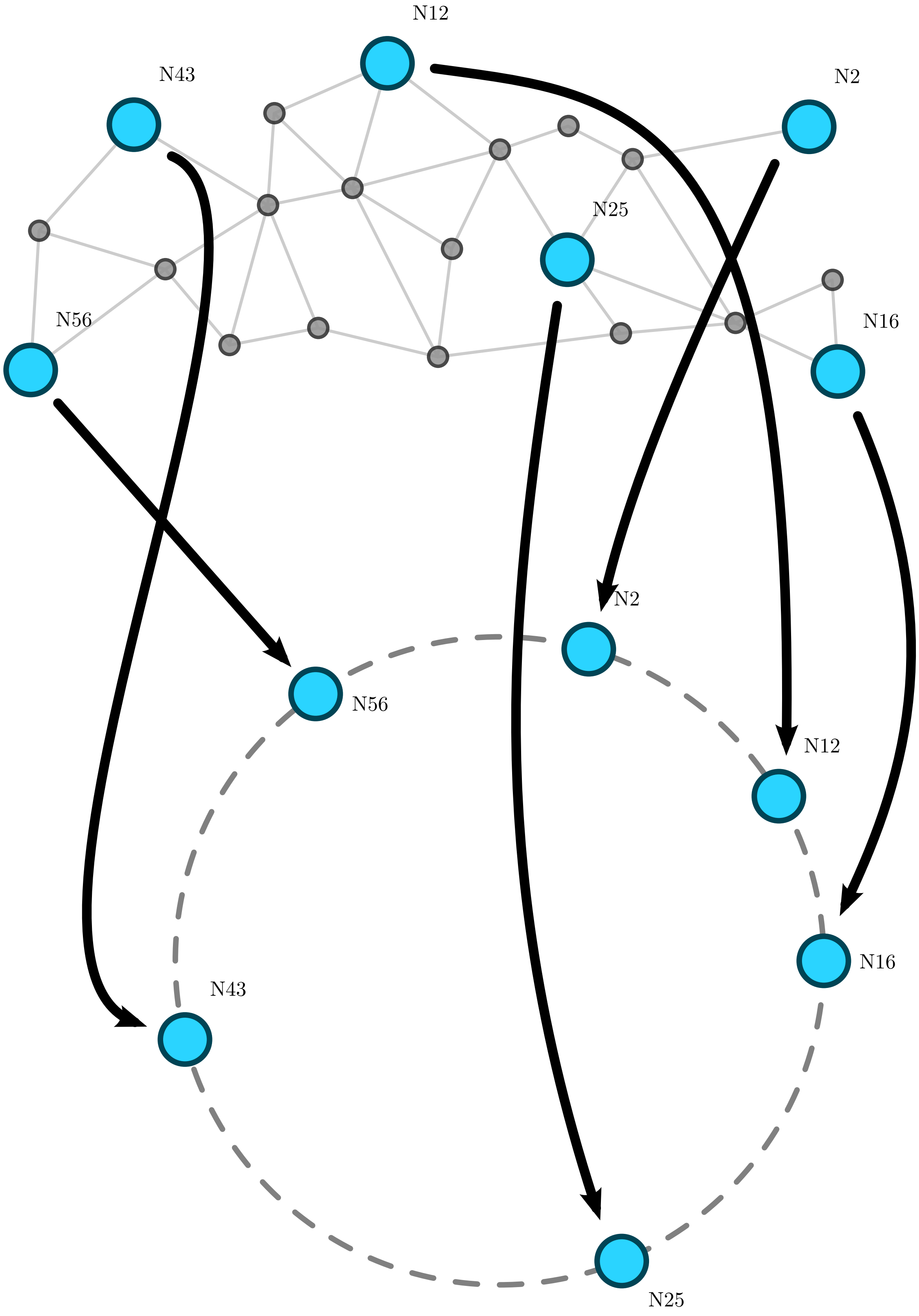
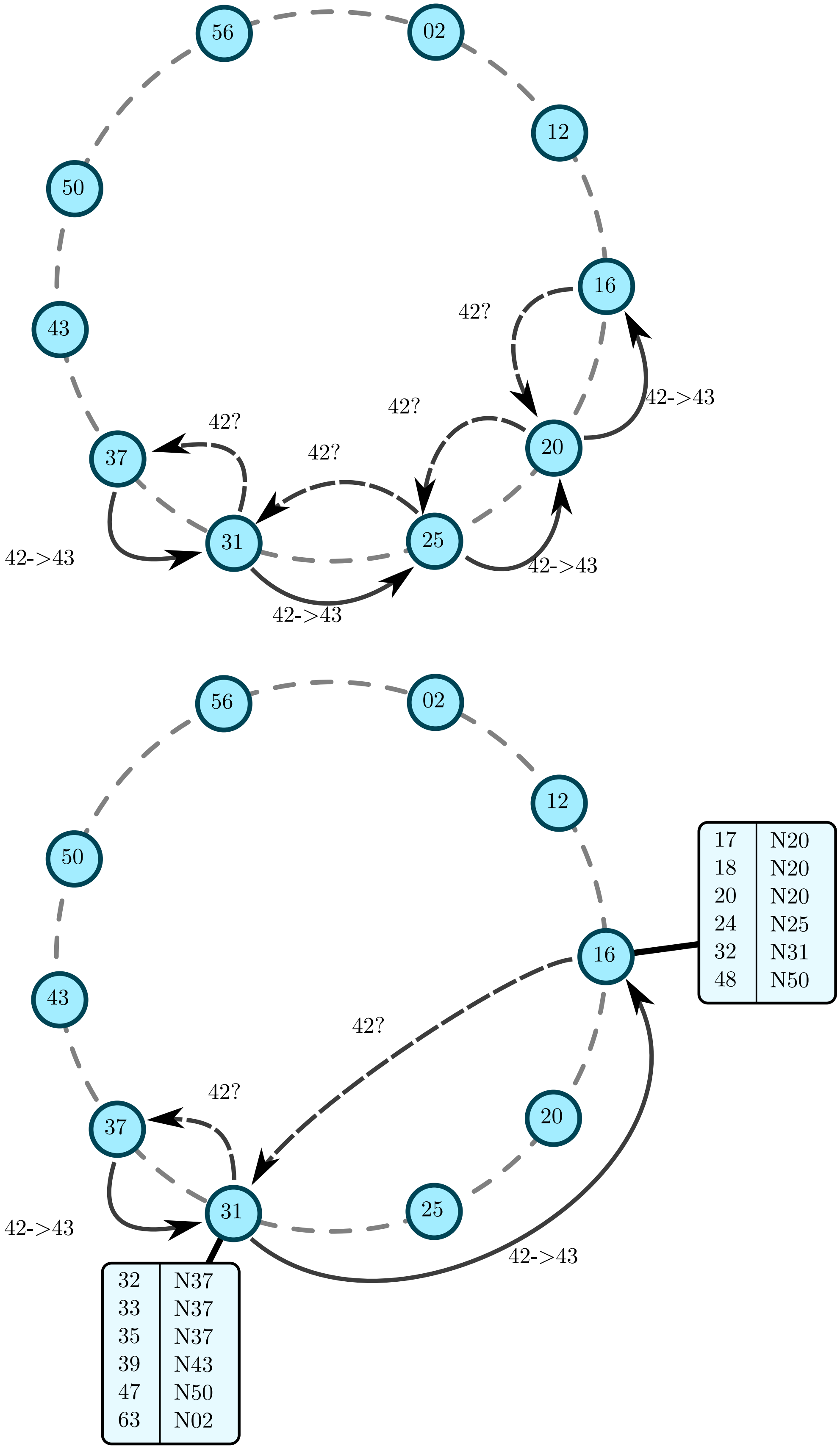


Figure 3: Mapping of nodes on a real network (e.g. the internet) to a virtual overlay network. Each node is aware of at least the next node in the ring



17	N20
18	N20
20	N20
24	N25
32	N31
48	N50

32	N37
33	N37
35	N37
39	N43
47	N50
63	N02

Figure 4: Use of a finger table improves lookup performance by reducing the number of hops to find the node an ID maps to. Note: the finger table also contains the addresses of nodes, though they are omitted here for clarity

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.

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 four 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; update successors which is similar to fix fingers but updates the successor list; and periodic checks of the availability of neighbouring nodes.

The stabilise function updates the node's primary 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.

The update successors function asks the node's primary successor for its successor and updates the second successor if necessary, then asks that node for its successor and so on until the successor list is fully populated.

Periodically The availability of the node's predecessor and successors are checked by sending them 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 [12] 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

A. 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`

There are a few other applications which could be implemented as follows

B. Publish subscribe system A publish-subscribe system allows data producers (publishers) send messages to receivers (subscribers) without either necessarily knowing about the existence of the other. Specifically publishers output data with a specific label or topic into some intermediary system which then forwards the data to subscribers to that topic. To implement such a system using the library, the application would map topics to IDs using the topic name and then the successor node of that ID would be responsible for that topic. Nodes could subscribe to a topic by sending a *subscribe* message to the node responsible for it, which would then add that node to its list of subscribers. Publishers would publish to topics in the same way, sending a *publish* message to the node responsible for the topic, which then sends the message on to all registered subscribers to the topic.

C. Distributed storage An implementation of a basic distributed storage system could be achieved by mapping files to nodes using hashes of their filenames and sending *store* messages containing the filename and content to the relevant node, and *retrieve* messages with the name of the file requested, the node receiving such messages would store or retrieve the correct file as necessary. This simple approach lacks redundancy, which could be added by adding a series of 'salt' values to filenames and hashing the results allowing mapping of a single file to multiple nodes like tapestry. It may also be necessary to occasionally 're-store' files periodically if the node set has changed.

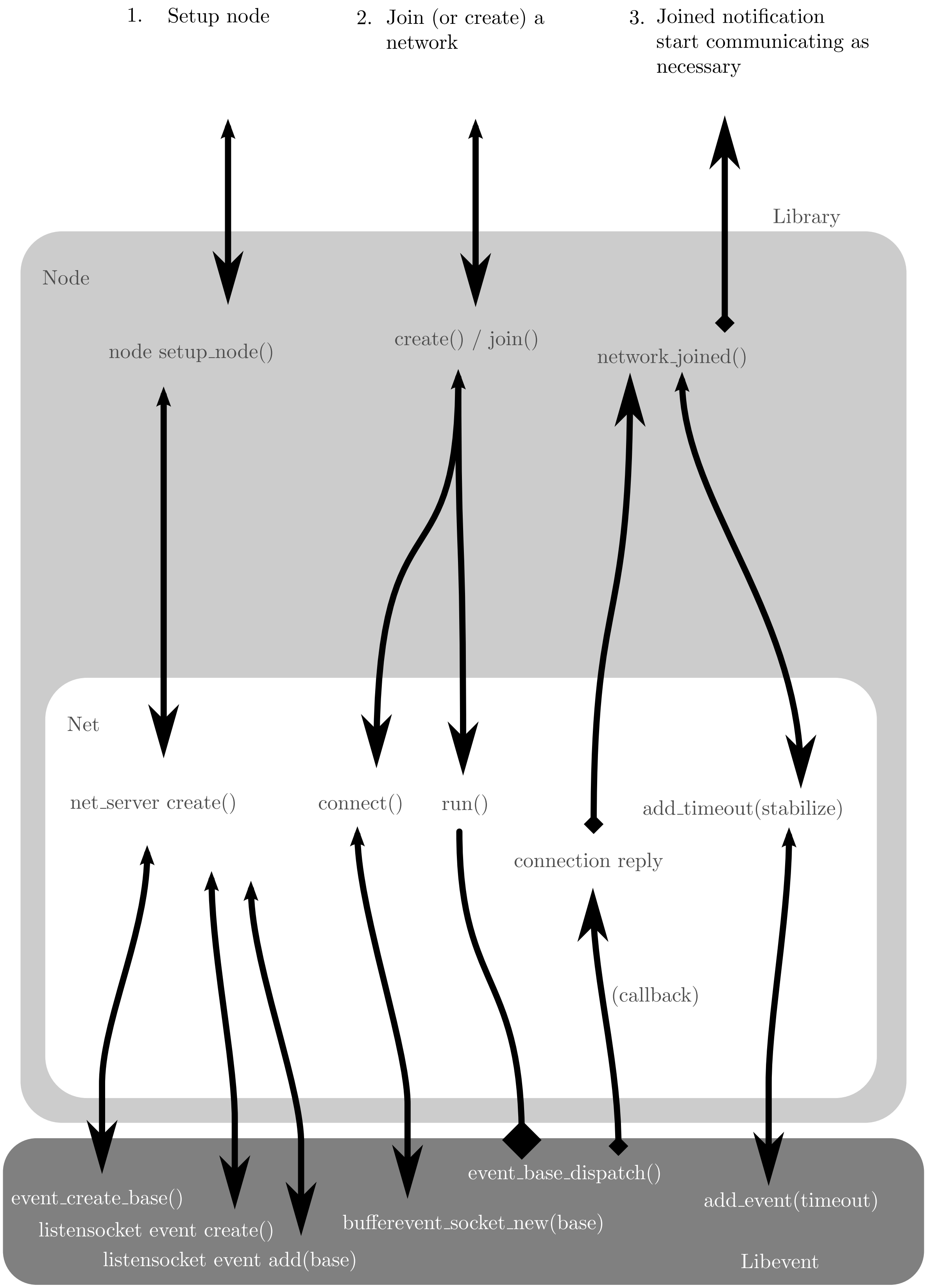


Figure 5: Process of setting up a node with the library, first call setup with a name and listen port number, then join or create a network, once this is done a given callback is called to allow the node to start its own functionality which should be run in separate threads.

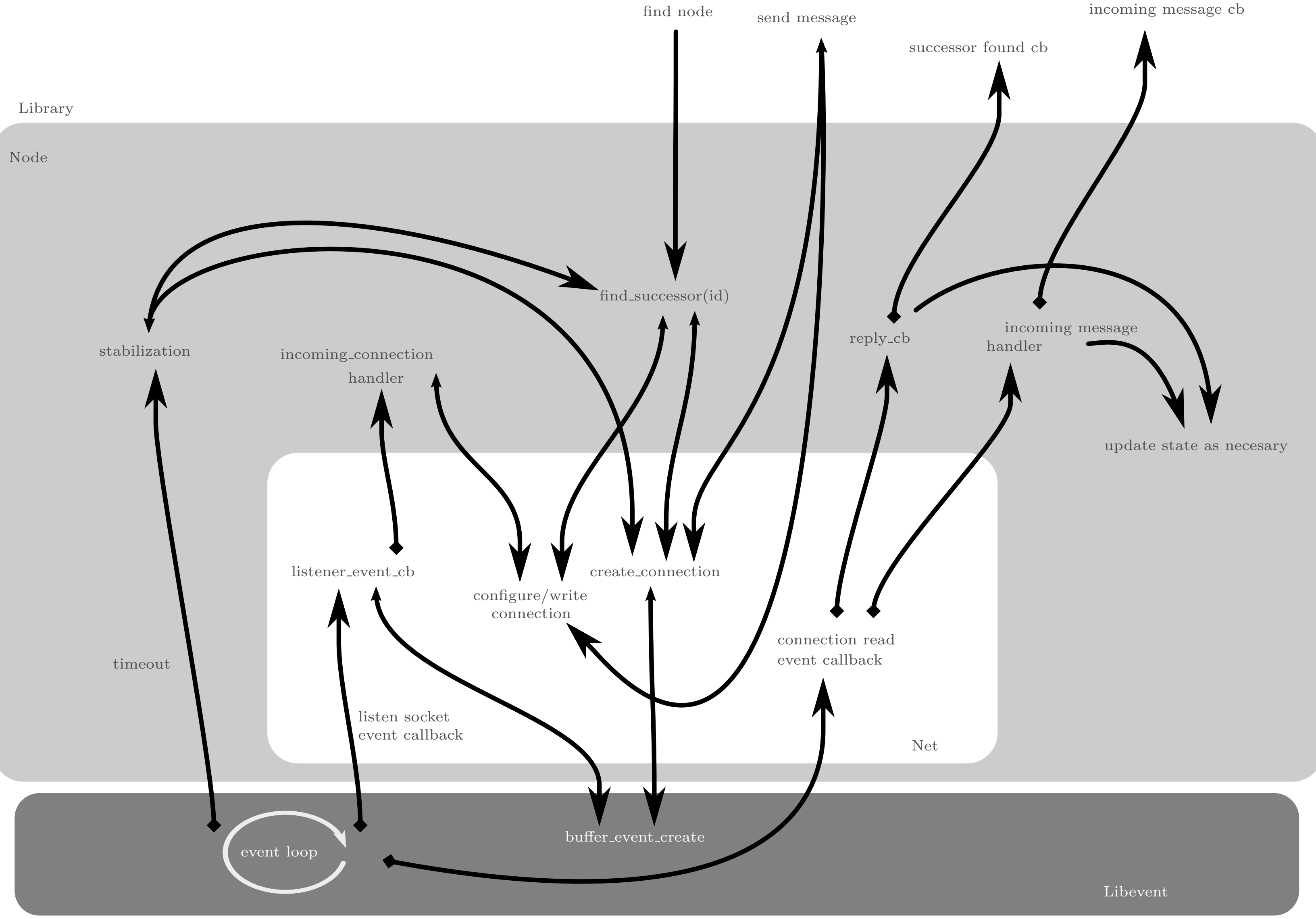


Figure 6: Main interactions with and within the library once running

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

Once the minimal required functionality had been implemented, I was able to test the functionality of the library using a simple application with minimal functionality, specifically I used the textsend application to test that nodes were able to join and that the network stabilized itself and routed messages correctly. Starting with a single instance routing messages to itself and gradually adding nodes Using a combination of debug logging and GDB, I was able to diagnose and fix major errors and bugs in the system.

Once basic functionality was working, I was able to begin testing the system with larger numbers of nodes, running over a hundred nodes on a relatively low end machine. To test that the network correctly stabilised and routed messages, I used instances of textsend to send the route same message from all nodes and check that they all arrived at the same node. After adding the ability to count the number of hops required to find the correct node to route a message to this same method was used to test routing performance, results shown in fig. 7.

Another aspect of the system tested in this way was failure tolerance. By setting up a network then manually terminating several random instances and retesting the routing of messages, I was able to verify that the network can manage failures and repair its routing infrastructure.

6 Evaluation

The library I have produced for this project succeeds in meeting the functional requirements set out, I have been able to use it to build an application to demonstrate its functionality. Testing has shown that nodes are able to create and join networks and self-organise to allow for message routing. The system is able to accommodate additional nodes joining a network. To demonstrate this, I added an additional 32 nodes to an existing network simultaneously. After a small amount of time had passed, I tested message routing and all nodes routed messages correctly.

The network created by the library is generally robust and able to handle individual node failures without detriment except in a few specific (and unlikely) situations i.e. several nodes adjacent in the keyspace failing simultaneously, such that all nodes in a nodes successor list become unavailable between stabilisations which is unlikely due to the nature of identifier hashes meaning that adjacent nodes are higly likely to be divers in terms of geography etc.; or the successor of a new node failing as it joins in which case the network as a whole would remain functional but the new node would fail to join.

I was not able to fully implement the ability to notify the application of changes to the keyset it is responsible for, however this is not core functionality and not necessary for a working system.

Ease of use can be difficult to evaluate, however there are only a small number of functions needed to be used to build an application using the library, combined with the fact that the `textsend` application is so small gives a good indication that the library provides an easy to use interface.

The use of asynchronous I/O allows applications using the library to effectively utilise the resources they have available, the

As for performance, there are several aspects to consider, firstly, network communication, the performance of lookup and routing can

I was able to run over one hundred instances of textsend on a single relatively low-end machine with ease Routing performance Figure 7 shows the mean number of hops to find the successor of a given identifier

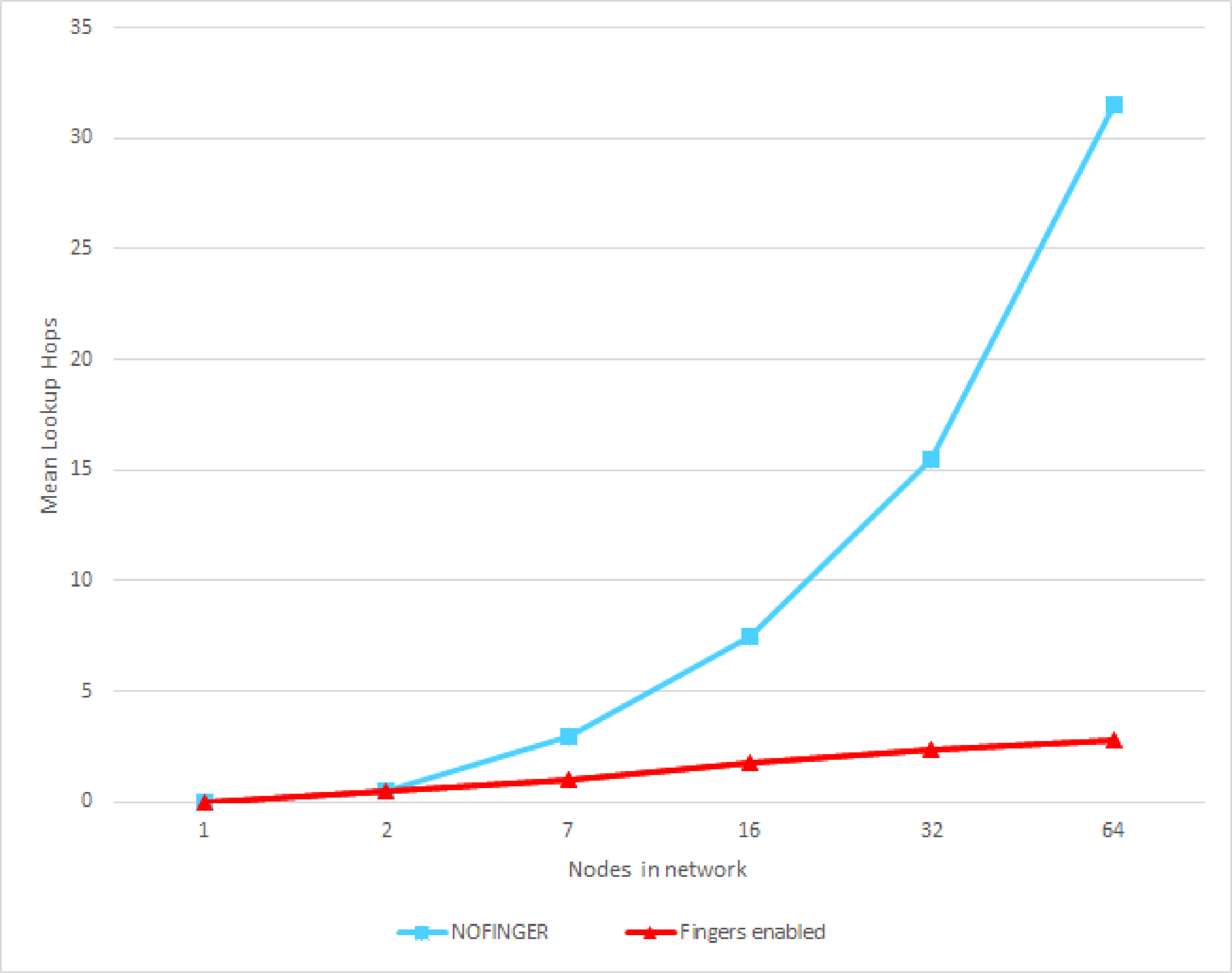


Figure 7: Graph showing average hops (nodes which need to be asked) to find the successor of an I.D. with and without the finger table

7 Conclusion

In this report I have provided an overview of the problem being tackled, details of the design and implementation and evaluation of my solution.

Overall I would say that I 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.

In hindsight, being able to have set up additional, possibly automated testing such as unit tests may have improved my ability to find and diagnose some errors in the library.

7.1 Future work

There are several enhancements I would like to make beyond the current functionality, which could make real world usage more viable.

Configurability and tuning Allowing configuration and tuning of parameters such as the stabilisation interval could help improve performance and stability of the network, depending on the use case, for instance one system may expect the availability of nodes to remain fairly constant so heigh frequency stabilisation may be unnecessary, while another may expect a high rate of *churn* (nodes joining and leaving frequently) and so would need much more frequent stabilisation.

Security I would like to implement SSL based connections between nodes to allow for secure communication and verification of message origin. Additionally, allowing use of domain names for addressing nodes in addition to SSL would allow for implementation of a system which only allows specific nodes to join the system.

Network monitoring Currently, there is no means of monitoring the state of the network as a whole, other than using the debugging info from each individual node. It should be relatively simple to add more types of message for gathering information about the network, for instance the number of nodes could be determined by a message sent around the entire ring from each node to its successor until it arrived at the node which originated the message.

Other It should also be possible to implement the ability to automatically find and join a sufficiently large network using an algorithm such as 'expanding ring'.

8 References

References

[1] *BCS Code of Conduct.* URL: `http://www.bcs.org/category/6030` (visited on 03/17/2016).

[2] *BCS Code of Conduct, Annex A, Interpretation of the BCS Code of Conduct.* URL: `http://www.bcs.org/content/ConWebDoc/39988` (visited on 03/17/2016).

[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.

[4] Antony Rowstron and Peter Druschel. *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.* Appears in Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Plat- forms (Middleware 2001). Heidelberg, Germany, November 2001.

[5] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-tolerantWide-area Location and Routing.* Apr. 2001.

[6] *Memcached, a free and open source, high-performance, distributed memory object caching system.* URL: `http://memcached.org/` (visited on 03/17/2016).

[7] *Overview · memcached/memcached Wiki.* URL: `https://github.com/memcached/memcached/wiki/Overview` (visited on 04/15/2016).

[8] Charles Killian et al. *Mace: Language Support for Building Distributed Systems.* 2007.

[9] *libevent, an event notification library.* URL: `http://libevent.org/` (visited on 03/17/2016).

[10] Nick Mathewson. *Fast portable non-blocking network programming with Libevent.* URL: `http://www.wangafu.net/~nickm/libevent-book/` (visited on 03/17/2016).

[11] Christopher Kohlhoff. *Boost.Asio.* URL: `http://www.boost.org/doc/libs/1_60_0/doc/html/boost_asio.html` (visited on 03/17/2016).

[12] *OpenSSL, Cryptography and SSL/TLS Toolkit.* URL: `https://www.openssl.org/` (visited on 03/17/2016).

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 predecessor
`'N'`	ID port	the i.d. and listen port of the node dong the notifying (i.p. address obtained from connection)
`'A'`	N/A	no body necessary
`'a'`	Y	simple confirmation message
`'M'`	N/A	Application dependant

B quantifiable testing results

B.1 Lookup Hops

nodes	NOFINGER	fingers enabled
1	0	0
2	0	0
2	1	1
7	0	1
7	1	1
7	2	1
7	3	1
7	4	0
7	5	1
7	6	2
16	0	1
16	1	3
16	2	1
16	3	2
16	4	4
16	5	2
16	6	2
16	7	2
16	8	1
16	9	1
16	10	1
16	11	2
16	12	2
16	13	1
16	14	0
16	15	2
32	0	3
32	1	3
32	2	2
32	3	4
32	4	2
32	5	2
32	6	2
32	7	2
32	8	3
32	9	1
32	10	2
32	11	3
32	12	3
32	13	2
32	14	2
32	15	3
32	16	3
32	17	3
32	18	1
32	19	1
32	20	3
32	21	2
32	22	3
32	23	2
32	24	3
32	25	2
32	26	3
32	27	3
32	28	3
32	29	3
32	30	0
32	31	2
64	0	3
64	1	3
64	2	1
64	3	3
64	4	5
64	5	2
64	6	1
64	7	2
64	8	3
64	9	3
64	10	2
64	11	3
64	12	3
64	13	2
64	14	3
64	15	3
64	16	4
64	17	5
64	18	2
64	19	2
64	20	2
64	21	4
64	22	4
64	23	1
64	24	3
64	25	3
64	26	3
64	27	2
64	28	4
64	29	3
64	30	5
64	31	4
64	32	2
64	33	2
64	34	3

64	35	2
64	36	5
64	37	2
64	38	3
64	39	2
64	40	2
64	41	3
64	42	2
64	43	5
64	44	2
64	45	3
64	46	5
64	47	2
64	48	4
64	49	4
64	50	1
64	51	2
64	52	3
64	53	1
64	54	2
64	55	2
64	56	3
64	57	2
64	58	4
64	59	3
64	60	0
64	61	3
64	62	3
64	63	2