Matthew Toseland

Efficient simulation of an anonymous peer-to-peer network

Computer Science Tripos – Part II

Wolfson College

May 10, 2016

Proforma

Name:	**Matthew Toseland**
College:	**Wolfson College**
Project Title:	**Efficient simulation of an anonymous peer-to-peer network**
Examination:	**Computer Science Tripos – Part II, June 2016**
Approximate Word Count:	**11733**
Project Originator:	Matthew Toseland
Supervisor:	Stephan Kollmann

Original Aims of the Project

To efficiently evaluate high-level changes to Freenet by simulating a network using the official Freenet source code for the higher layers while replacing the lower layers with more efficient bypass paths. To validate these simulations against models which use the official source code, and demonstrate which type of simulation is most appropriate for testing changes to routing and load management.

Work Completed

Simulations of sequential requests are working well: after improving the testing framework and fixing various bugs in Freenet, all three new simulation modes work correctly and the simplest bypass mode greatly improves performance when testing changes to routing. For parallel requests, I have constructed a new testing harness to evaluate changes to load management, quantitatively compared three different load management algorithms, confirmed some assumptions about how Freenet really works as a distributed system, and explored some unexpected behaviours related to high load levels.

Special Difficulties

None.

Declaration

I, Matthew Toseland of Wolfson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

This document is based on the LaTeX template provided by Martin Richards. Thanks are due to my supervisor, my director of studies, my parents and fellow Freenet developer Arne Babenhauserheide who all provided helpful feedback on draft versions.

Chapter 1

Introduction

Freenet is a peer-to-peer network which implements an anonymous, distributed datastore. It is designed to prevent and circumvent online censorship, by enabling anonymous, censorship-resistant communication and publishing of controversial opinions, leaks and other data within the distributed datastore ([2], [11]).

It is a complex distributed system where interactions between architecture, implementation bugs and end users may cause unexpected behaviours. End users expect anonymity and stability, and this makes it difficult to gather data from or experiment on the deployed system. My contribution is firstly to greatly improve the performance and reliability of the existing, low-level simulations, and secondly to create a new simulation framework for evaluating changes to load management. Both frameworks are based on the existing source code ([3]) rather than an abstract event-driven model, making them ideal for testing proposed changes before deployment.

1.1 Freenet

Freenet can function in "opennet" (traditional peer-to-peer) mode, where it connects automatically via a bootstrapping service and exchanges new connections continually, or in "darknet" (friend-to-friend) mode where a user's node only connects to the nodes operated by his/her friends. The latter provides much greater security by exploiting social trust, making it much more difficult for an attacker to monitor or control a large proportion of the network (a "Sybil attack"), as well as making Freenet harder to block. This is a feature which few other anonymity tools implement, in spite of some published work, e.g. PISCES [18]. In both cases the network has a small-world topology (see Kleinberg's recent book [13]) which enables greedy routing according to the node identifiers, as explained in [11] and [20]. Freenet supports publishing content including websites, larger files, a forums system, and a micro-blogging system, and has around 10,000 users in production use (Figure 1.1).

Content is divided into fixed-sized blocks, identified by a key. Keys are either Content Hash Keys (CHKs, [1]) which identify a 32 kilobyte block by the hash of its encrypted content, or Signed Subspace Keys (SSKs, [6]), which consist of a public key hash and a filename, identifying 1 KB of digitally signed, encrypted data. Only the owner of the

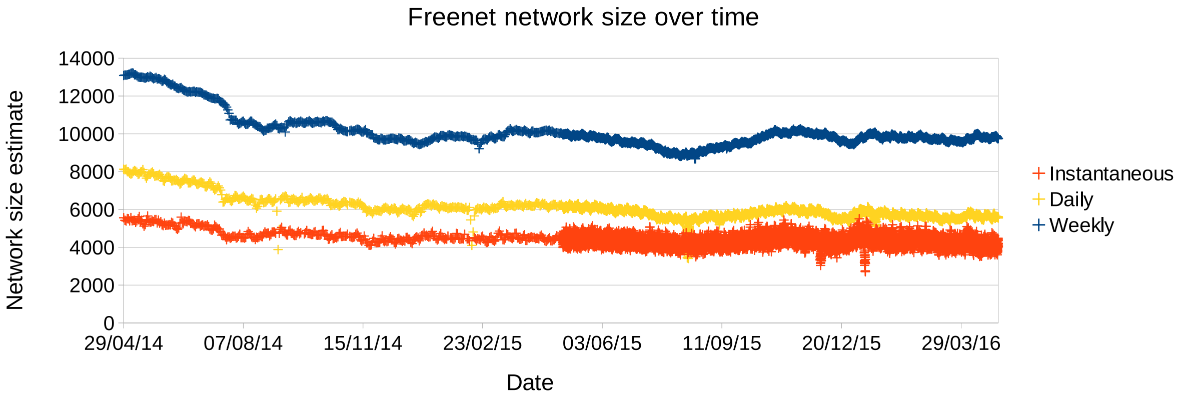

Figure 1.1: Estimated active Freenet nodes over the last 2 years. Data collected and computed by Steve Dougherty. Once an hour the network is scanned, giving the instantaneous estimate. The daily and weekly estimates are estimated based on the number of repeated versus unique node identifiers.

private key for an SSK may upload to that SSK. CHKs are used for bulk data storage. Files larger than 32 KB, and other structures such as compressed archives of web pages, are divided into many CHKs and stored in multi-layer structures similar to Merkle trees ([17]), with additional redundancy at each layer. SSKs are used when the key must be knowable in advance, in particular for the top block of an in-Freenet website or a post on an in-Freenet forum.

The decryption keys for a block are included in the Uniform Resource Identifier ([8]), but not in the key transmitted over the network, nor in the data stored on disk. URIs must be obtained “out of band”. They may be published on in-Freenet forums and web pages, or exchanged privately. Hence node operators have “plausible deniability” in that they will not have the URIs and decryption keys for most of the content cached on their nodes.

The fundamental operations on Freenet are inserting a data block into the distributed datastore, or requesting one from it (by its key). Almost all client functionality is built on these two operations. Every node has an identifier, which is a double precision float in the interval [0.0,1.0), and keys are hashed and mapped to this space. Both inserts and requests are routed greedily through a path converging on the nodes with the closest identifiers to the key. Hence Freenet is a distributed hash table, but relies on slightly different principles to the classic DHTs such as Kademlia ([16]), which the original Freenet paper ([10]) predates. Data is cached on (nearly) every node, and is stored more permanently on the closest nodes to the key. The local datastore is an on-disk hash table, so when the store is full, new data blocks replace pseudo-randomly chosen old data. Hence data persists in Freenet for as long as it remains popular, and disappears if it is not fetched for a long period.

When Freenet was first created in 1999, web sites were frequently served from single servers, and the “Slashdot effect” was the phenomenon of a link from a popular news site causing a site to become unavailable due to the server being overloaded. Similarly, a distributed denial of service attack against a controversial website often simply sends many requests to the same server [15]. Freenet is immune to the Slashdot effect: popular data is cached widely, so the more popular a file is, the faster it can be downloaded, and the less impact new downloads of that file have on the network. Today many sites on the open web are hosted on Content Distribution Networks which integrate layers of distributed caching, a similar although centrally controlled solution. This is one of many examples of parallel evolution in Freenet and the wider Internet.

Load management is similar to TCP. Each node estimates its own load and rejects requests if necessary ("hop-by-hop load limiting", similar to packet loss in an IP network), based on a worst-case estimate of how long it will take to transfer the data, and on fairness between peers. When a request is rejected, the previous node marks the rejecting node as "backed off", and will avoid sending requests to it for a period (binary exponential backoff). Also, a message is sent downstream to the request originator. As in TCP, the originator maintains a "request window" indicating how many requests it can have in flight simultaneously. This is decreased when feedback indicates that a request was rejected by some downstream node, and increased when a request completes without such feedback, using an Additive Increase Multiplicative Decrease formula (AIMD). On the originating node, this "client-side load limiting" determines when the request starters start requests from the client layer request queue.

Meanwhile the request itself continues, so it may be routed differently to how it would have been routed on a network with no load. If load is not managed correctly, this request re-routing may cause serious problems such as data not being found, being stored on the wrong nodes, or it may cause even more network load.

1.2 Simulating Freenet

In the past, the developers would try out changes to routing or load management on the main network. This was usually inconclusive and sometimes disastrous! In particular, on at least 2 occasions the network became temporarily unusable because of a change made by developers without adequate testing. One practical consequence was the implementation of a highly robust in-Freenet software updating system that does not rely on the ability to do requests at all! Another is that the developers are much more cautious about making changes to load management, but at the moment it is difficult to evaluate such changes in advance.

Furthermore, the current load management is very vulnerable to "cheating": a well-known patch [22] effectively turns off client-side load limiting, causing the node to send more requests than it should, giving better performance in the short term at the expense of other nodes. If enough nodes use the patch, the developers expect this will cause more rejections, backoff and incorrect routing. This is essentially a problem of "mechanism design" or microeconomics, that is, of aligning the individual end user's incentives better with the interests of the wider network. Note that this is also an unsolved problem for the wider Internet! However, alternative load management algorithms with better incentives behaviour have been proposed, and hopefully the simulation framework will allow evaluating them without breaking the production Freenet network.

Hence it would be very helpful to be able to simulate changes to load management before deploying them, not just for design exploration, but also for sanity-checking code before it breaks the network. Since Freenet has a lot of complexity including its own UDP-based reliable transport layer, congestion control and encryption, simulating many nodes using only the existing UDP code may be too slow to be useful. My project improves simulation performance substantially by only accurately simulating the layers needed for the specific task being modelled, as I explain in the next few chapters.

Chapter 2

Preparation

2.1 Starting point

Freenet is written largely in Java. The network protocols are still evolving, in spite of 17 years' work, hence the latest major version is called 0.7.5. I am familiar with the vast majority of the codebase as I was chief developer from 2002 to 2013. There are several existing simulations, as I describe below.

2.2 Related work

Many simple high-level event-driven simulators have been constructed for Freenet over the years for design space exploration. These were not based on the official source code, and so are not relevant to the question of whether a proposed change to load management will break the network due to bugs or interactions between features which were not simulated. None of them model load management on Freenet.

Freenet includes several simulations that create many nodes within a single Java Virtual Machine, using the official source code and connecting the nodes across localhost sockets. These tests are sometimes used for testing and debugging particular features. However, as they use the full transport layer, including encryption and reliable message delivery over UDP, they are too slow to simulate large networks, or even to form part of the automated test suite run during a full build. Most of the current simulations run one request at a time, which reduces the CPU cost and makes the results more predictable, but this is not acceptable for testing load management, where we need to see the effect of many simultaneous requests.

At the other end of the spectrum, there is NS-3 [5], an open source discrete event simulator written in C and designed for modelling networks, particularly IP and wireless. NS-3 also supports real-time scheduling, so we could use the official Freenet source code, and model the connections between simulated nodes with NS-3 models of the behaviour of a simulated IP network. This would be very useful for improving the performance of Freenet's transport layer, for example for performance on wireless networks where there may be high non-congestive packet loss and very variable latency. But that is not the problem I am addressing in this project.

I and other Freenet developers have believed for years that there are fundamental problems with higher-level load management and routing on the overlay network, and there appears to be a problem with “cheating” as explained in the previous chapter. These are high-level issues in Freenet itself which will occur even on idealised IP networks where there is no packet loss and latency is constant. For high-level simulations dealing with these problems, and especially for automated regression tests for changes to routing, request handling and load management, delivering messages directly as objects inside the same Java Virtual Machine should be much more efficient, allowing simulating larger networks and more requests as I will show later.

Hence the objective is a simulation based on the official source code, but which can avoid the transport layer when appropriate by replacing it with a simpler internal bypass path. This is similar to the approach taken for hardware by Electronic System Level Modelling (e.g. in SystemC), typically used to develop software and prototype high-level behaviour while the hardware itself is still under development ([7]). Here the goal is similar but different: to evaluate high-level changes efficiently without breaking a working distributed system. The other main outcome of the project is a test harness for evaluating changes to load management in simulation.

2.3 Simplifying assumptions

Both the existing and new simulations are in “darknet” mode, i.e. fixed connections on an idealised Kleinberg network (see Kleinberg chapter 20 [13]). This models a real darknet, which we assume to be a small-world network of friend-to-friend connections. The real network is mostly opennet but the goal is to move towards more darknet, since that provides much better security and resistance to blocking.

It also assumes that all nodes are constantly online, which is clearly an unrealistic starting point in terms of long-term behaviour, but many important questions can be addressed with this model. Also, the developers strongly encourage users to run Freenet constantly, as this greatly improves performance, especially on darknet, and historically around one third of nodes have very high uptimes. Future work may include opennet simulations and realistic models of node uptimes; the probe data used to generate Figure 1.1 includes information on this.

2.4 Use cases

There are two main use cases relevant to this project, each requiring a different tradeoff between accuracy and performance:

2.4.1 Testing changes to routing and requests

This only requires the ability to simulate one request at a time. Hence the results are completely deterministic, and simulations can be very efficient because they can bypass the transport and message queueing layers.

2.4.2 Testing changes to load management

This requires simulating many simultaneous requests, which is potentially non-deterministic, and significantly slower, because of the need to model fairly sharing bandwidth both between different peers of the same node and between different requests on the same network connection.

2.5 Simulation modes

There are four simulation modes available in this project:

2.5.1 Message-level bypass

Individual messages are delivered immediately to their destination nodes, rather than being queued and sent in a packet. This is the fastest mode, but it does not simulate limited bandwidth at all, so it is not useful for load management. Because messages are delivered immediately, it can reveal race conditions which are very unlikely to happen in the real network (see section 3.6). However it improves performance dramatically for sequential tests as described in section 4.1.

2.5.2 Constant bit-rate message-level bypass

Messages are delayed to simulate fixed bandwidth and latency on each connection, unlike on the real network, where limited overall bandwidth is shared between peers of the same node. Also, there is no fair queueing between different requests, so it is possible for a message to be stuck behind other data transfers for so long that it causes a timeout. Hence this is only suitable for simulations which only do one request at a time.

2.5.3 Packet-level bypass

Simulates individual packets, keeping track of how many bytes are sent from each message and when they are acknowledged. Accurately shares bandwidth between different peers of a node and between different requests to the same peer, by using the existing packet scheduling and fair queueing code. Slower than the message-level bypasses, but can be used for parallel simulations within a limited load range, see section 4.4. Does not support simulating packet loss for reasons of efficiency.

2.5.4 UDP level (no bypass)

Simulation using the full transport and encryption layers, but with several nodes instantiated in the same Java Virtual Machine, communicating via UDP sockets on localhost. Used by existing tests prior to this project, the bypass modes are new. Packet loss can be simulated, so this is useful for debugging the transport layer, but there is no latency between nodes. More complex IP-level models can be implemented e.g. using NS-3.

Chapter 3

Implementation

3.1 Architecture

Figure 3.1 shows Freenet's layered architecture in some detail. For the message-level bypasses, the message queue itself is replaced, the peer management code remains, and everything below that is removed. The packet-level bypass retains the packet scheduler and message queue, but removes connection setup and everything below that level. Figures 3.2 and 3.3 show the layering with the message and packet bypasses respectively.

3.2 Scalability and resource usage of the simulator

The simulation is not event-driven: It runs in real time, and has parameters for bandwidth limits and the number of nodes simulated which must be adjusted according to available resources. Future work might include deriving an event-driven simulator from the official source code, using bytecode rewriting to change timeouts and clock measurements etc, which would be faster and reduce the need for overload detection, but the rest of this project was more urgent.

In general the tests can detect when the simulator host is overloaded by looking for timeouts. Packets should not be lost in a simulation, even in UDP mode, but if packets or asynchronous jobs handling them are severely delayed this causes timeouts. These are interpreted as protocol violations (because the preceding node should have detected the timeout first), and cause the node to disconnect temporarily to reset the peer's status. Similarly, if no messages are received on a connection for a period, the connection is lost. In normal operation, "keep alive" messages are sent periodically on idle connections to prevent this. In both cases, disconnections in a simulation mean there is severe CPU overload, so the test harness will fail with an error if nodes become disconnected.

These mechanisms could be turned off to allow more nodes to be simulated with constant full CPU usage, but this would cause worse problems: data packets would be delayed and then retransmitted, causing even more CPU usage, timeouts would occur at higher layers, and memory would be wasted waiting for packets that are no longer relevant. It is much better to allow disconnections to serve as a "canary in the coal mine".

When a node receives a request, it checks whether it has the resources to process the request before accepting it (this is called "hop-by-hop load limiting"). This is dominated in practice by a calculation of whether the node can process all of its current requests within a reasonable time, based on the available bandwidth, assuming the worst case where they all succeed and return data which must be relayed ("output bandwidth liability limiting"). A node may also reject a request if its average round-trip time to each of its peers increases above a configurable threshold, which on a deployed Freenet node is a reliable indication of either severe local network congestion or CPU overload, e.g. when the user has started a computer game.

Either way, when a request is rejected, the rejecting node is marked as "backed off" (meaning overloaded), so no more requests are routed to it for a period. For routing simulations, since only one request or insert runs at a time, any connection in backoff indicates CPU overload, so the simulation will terminate if backoff is detected. For load management simulations, backoff is allowed, but any serious overload will cause messages to be delayed long enough to cause timeouts and disconnections, which are detected and terminate the simulation.

3.3 Engineering approach and version control

In general the project was pre-planned, however there was significant iteration of the original detailed planning. The main reason for this is that the actual performance was very uncertain until large parts of the implementation had been completed. This project is essentially about optimising existing simulations and making new kinds of simulations possible on modest hardware, so it is very sensitive to what is actually achievable and what makes sense in light of testing and profiling, e.g. thread usage.

The first core goal, implementing, validating and evaluating bypass paths for simulating sequential requests efficiently, took longer than expected because of debugging non-determinism issues. However this revealed opportunities for optimisations that allowed simulating a very large network on modest hardware, which seemed a useful extension (see the Evaluation section). Some of the detailed planning for the second core goal (load management simulations using concurrent requests) did not seem useful in light of the factors actually limiting the simulations' performance, as described later.

I intend to get this code merged into the official Freenet source. Freenet uses Git and JUnit, and all proposed changes are pushed as feature branches from a developer's own repository. The "pull request" is then discussed on Github [3] and once the release manager is happy with it, he merges it into the "next" branch. Releases are tagged and merged into the "master" branch. This is based on Driessen's branching model ([12]).

Hence I have merged changes from upstream periodically, and tried to break out features and bug-fixes which can be usefully separated into separate pull requests so that they can be discussed independently. This should reduce the size of the pull request for the simulator code itself, which is highly desirable since all code is reviewed by volunteers, and large patches can be hard to review and take a long time to be merged.

As usual in Freenet, which is heavily threaded, most of the debugging is via log files, but debuggers are occasionally useful. I have not tried to use a profiler on the simulation, but stack dumps have often been very informative for optimisation work. I intend to publish this document as documentation in addition to Javadocs in the source code. I have added JUnit tests using the new code for sequential requests.

3.4 Initial work during project selection phase (August 2015)

The message queue already had a well-defined interface, so I only needed to factor out a Java interface for the existing methods, and was able to replace the implementation reasonably easily. The NodeStarter run-time support class, used to start the Freenet node, already has methods for creating test nodes and for checking whether the node is a test node rather than a node exposed to the real network, hence allowing further configuration, optimisations and hooks that would be risky for a real node. I added a map keeping track of all the nodes in the JVM, so that the bypass code can find them, and implemented the first version of the fast message-level bypass (class BypassMessageQueue). I also fixed some of the bugs preventing the simulation from working, notably the keep-alive bug.

3.5 Test harness and infrastructure

3.5.1 Bypass implementation

The message-level bypasses use an alternative implementation of the MessageQueue class, which delivers messages immediately, or after a delay. The actual message delivery is scheduled on another thread using the Executor or Ticker infrastructure classes. Similarly the packet-level bypass replaces the PacketFormat class, which is ordinarily responsible for slicing up and packing messages into packets, encrypting them and passing them to the UDP layer for transport. For each virtual packet the bypass keeps track of how many bytes are sent from each message, which messages should be acknowledged and so on, but it delivers the messages directly, bypassing encryption, retransmission etc. There are numerous interactions with other classes, in particular the PeerNode class which keeps track of the state of each connection.

This could have been implemented by replacing bytecode in the class loader or similar mock-up solutions (e.g. Mockito [4]). However this would have made it harder to debug, made tests significantly more complex, and in particular such an approach would have increased maintenance costs: when the real message queue or packet format class is changed, the code no longer builds, and the need to change the bypass code is immediately obvious in any good Integrated Development Environment, without even running the tests.

The code changes in the existing core Freenet code are mostly in initialisation code, which is run on startup or once per connection, so there is negligible run-time overhead for a node which is not in a simulator. Similarly the data collection hooks described below have very low non-operational overhead.

The NodeStarter class keeps track of all the simulated nodes and their test parameters, as well as global test parameters such as the bypass mode. When a PeerNode is created, representing a connection to a peer (neighbour node), it checks the NodeStarter to see if a bypass mode is enabled, and if so it looks up the Node representing the destination, and if that is found it sets up the bypass (either a BypassMessageQueue or a BypassPacketFormat). Hence it is possible to have some peers of a node connected internally via a bypass path, while other nodes connect via UDP. This might allow for interesting extensions such as larger-scale simulations across a cluster, or distributing client layer load across multiple actual network-level nodes. In the past a "test network" has been deployed, with no expectation of anonymity, where volunteers would run nodes with extra logging and remote access for developers. One big problem was it was always very small. Making it easier for a user to set up multiple test nodes might enable a larger test network.

3.5.2 Sequential test harness

There is an existing test for sequential requests (RealNodeRequestInsertTest). This sets up a network, then inserts a block to one randomly chosen node and requests it from another, repeating until it reaches a specified number of successful requests. I have made many improvements to this test, including checking for overload, checking for unexpected requests, parsing command line options etc. I have also added hooks to determine the nodes visited by each request and insert, described in section 3.5.4. The data is stored on each node that the insert visits, so a request should succeed if and only if it reaches a node which was on the path of the insert, and the test now checks this. The test output includes the keys inserted and requested, the exact path of every request and insert, and the success or failure of each.

I have made the test as deterministic as possible: the same seeds result in the same generated network, and the same random blocks are inserted and retrieved. The driver script uses the same seeds across each of the four simulation modes, and compares the output, which should be identical (and in fact is identical, after fixing some bugs in Freenet). Similarly, the new JUnit tests simulate sequential requests on a 25-node network and verify that they work exactly as expected, computing a checksum of the test output, and comparing it to a known value. The build-time test suite uses the fast message-level bypass, while the extensive test suite uses all four simulation modes and a larger network. The test alternates between CHKs and SSKs, since each key type has different network behaviour, and should automatically detect any changes which break routing, requests or inserts.

3.5.3 Parallel test harness

For parallel requests (simulating load management), there are two tests based on a common base class. These tests insert data on one or more nodes on a simulated network, and fetch it on another, but allow many requests and inserts to be running simultaneously, according to a parameter specifying the target number of requests. Both requests and the inserts for the keys which will be requested later proceed simultaneously, partly to limit the storage requirements of the simulation, since requests will propagate data, causing it to be cached on the nodes on the request path.

This is a simplified model since only one node initiates requests, but it may be sufficient to evaluate changes to load management quickly, especially changes to client-side load limiting, while removing irrelevant variables. Any given block is inserted once and fetched once, which is unduly pessimistic as, on a deployed network, popular content will be fetched many times, and thus cached on more nodes and found more quickly. Thus the simulations model the worst case, for both data retention and load. But this is reasonable since the worst case behaviour determines how much data is available on Freenet.

One of the parallel tests starts requests directly, without client-side load limiting, ignoring feedback from the network. Apart from demonstrating how performance deteriorates with load (see the next chapter), with minor changes this might also be useful for showing how the network limits the impact of misbehaving nodes, hence the class name RealNodeSpammerContainmentTest.

The other test (class RealNodeParallelLimitedTest) starts requests in the usual way by adding them to the queue used by the request starter threads. The request starters limit the number of actual requests in flight according to the AIMD algorithm described earlier. To minimise CPU and memory usage, these are still low-level requests which avoid most of the client layer and in particular do not need to decrypt the downloaded data blocks. The code is based on a refactored version of an existing class used for propagating keys recently requested by other nodes (an important optimisation for chat applications).

The datastores for each simulated node are kept in memory, and are large enough that inserted data will not be dropped until after it has been requested. For longer lived tests a shared datastore might save memory, but it was not necessary for the testing implemented so far: for a sequential test, we can simulate a larger network than the official Freenet network, and the size is limited by CPU usage rather than by memory, while the parallel tests are limited by other factors, but are sufficient to compare load management algorithms in practice as shown later.

Requests are created in the simulator main loop and report in asynchronously, using inheritance and Java synchronisation. The simulator identifies requests by unique integers, which determine which requests are included in statistics (see below), and the blocks to be inserted and requested are generated from the request identifier. This should not be confused with the network-level request identifier used to prevent the same request or insert visiting the same node twice.

The parallel simulator can use either CHKs or SSKs, but the simulation runs evaluated here use CHKs. For high load it is possible for inserts to fail, so the test repeats each insert until it succeeds. When starting a new request, if the corresponding insert has not yet finished the simulator can "stall" waiting for the insert to finish, see section 4.4.1. The repository also includes shell scripts to run the simulations and parse their output into a more easily analysed Comma-Separated Values format.

3.5.4 Infrastructure and hooks

Messages received by a node are initially passed to the message matcher (MessageCore), which checks the message against those expected by currently running requests, block transfers etc, passing any unmatched messages to the dispatcher. The dispatcher handles new requests from other nodes, either rejecting them or accepting and starting a new request handler. The test replaces the standard dispatcher with a proxy dispatcher which additionally keeps track of which requests visited which nodes.

Similarly, each node already has a request tracker which keeps track of all requests running on that node, mainly for purposes of load limiting. When creating a simulated node, a subclass of the request tracker is used, which calls a callback when requests start and terminate on each node. This keeps track of the number of requests running across the network as well as locally at each node. Hence the sequential request test can wait until all requests have finished before starting the next one, ensuring that all resources have been freed, or fail with an error if there are unexpected requests (see below).

3.5.5 Data collected for the load limiting simulations

A single run of the load management simulator runs 1000 requests and gathers various statistics:

- The request success rate.
- The number of hops taken by a successful request.
- The number of requests actually running on the requesting node.
- The overall number of requests and inserts across the whole network.
- The number of times the simulator has to wait for an insert to finish before starting a new request (see section 4.4.1).
- The number of times an insert fails and has to be retried.
- The time taken by a request or insert.

Some of these quantities can be measured directly by the simulator when running requests, while others use the request or message hooks. For most of these quantities the simulator computes a running estimate of both the sample mean and the sample variance. The variance is computed using the naive textbook algorithm, using double precision floating point variables, which has numerical stability issues for large numbers of samples, but is acceptable here due to a relatively small sample size and fairly large actual variance (see Chan [9]). Some of the averages (the number of running requests in particular) are measured over time rather than over discrete events, using a sum weighted by the time the sample was valid for.

For n parallel requests, statistics are only recorded after $4n$ requests. The simulation records data for m total requests after that point, and then terminates. It continues to start new requests for all of that time. Hence the statistics should represent the steady state of the simulation, not including any artefacts from starting up and terminating (e.g. the fact that it takes time to reach the maximum number of simultaneous requests). The exact order in which requests complete will vary, and this partially explains why the error bars are still relatively large for larger values of n. I expect there could be artefacts if the ratio m/n is small. In the results in the Evaluation chapter, m is always 1000 and n is no more than 150. Both are set by command-line options.

3.6 Bugs in Freenet affecting the simulation

While testing the simulations I discovered several bugs in Freenet which caused non-deterministic behaviour in the routing simulations, and one important unrelated bug. I reasonably expect that these changes will be merged upstream eventually, so the simulations will correctly model the deployed code.

3.6.1 Bugs breaking UDP simulations

There was a bug causing keep-alive packets to not be sent, making simulations unreliable. I fixed this in August 2015, prior to the start of the project ([26]).

3.6.2 Bugs causing non-deterministic behaviour

Residual requests

In normal operation, every node starts a number of requests for maintenance purposes, e.g. to look for software updates. These requests are not helpful for simulations since they can take up significant resources and sometimes cause testing requests to be rejected, resulting in non-deterministic behaviour.

Before this project, the simulation framework already turned off the auto-updater when creating a node, but some of the auto-updater requests were running even when the auto-updater was turned off, which I fixed with this pull request [24]. The original simulation framework worked around the problem of residual requests by using an effectively infinite bandwidth limit, so no requests were rejected, but this was both inaccurate and inefficient. The simulator now checks for any residual requests using the request hooks described above, and normally terminates if any are found.

Random reinserts

1 in 200 successful requests triggers a reinsert, to redistribute semi-popular data which is found in a cache somewhere but may have dropped out of long-term storage, for example because the node it was stored on has gone offline. This would cause residual inserts and therefore non-deterministic behaviour. Random reinserts are now turned off by default for sequential simulations, since we only want to see the requests that the simulator has started explicitly. If they are enabled in the simulator settings, the checks for residual requests and inserts are disabled, and the sequential simulation may be less reliable.

Obsolete load limiting code

Some old hop-by-hop load limiting code, predating the current "output bandwidth liability limiting", was causing spurious rejections, especially soon after starting up, resulting in different simulation modes giving different results. On real nodes the new code is overwhelmingly dominant, and it is not a reliable fallback on new nodes since it is based on measurements of the number of bytes sent for a request. So it should be removed from Freenet ([34]).

Simulator configuration issues

For security reasons, a data block will not be cached in the local datastore for the first few nodes on the insert's path across the network. Hence once an insert is far enough away from the originator to be cached, its network-level identifier (not its simulator request ID) must be changed, so that it can be routed back to the previous nodes if they are closer to the target location. This complicates the simulator traces, but can be used for a large simulated network. For a small simulated network it is generally better to just cache everything. This setting is per-insert, and was not being propagated properly due to another bug which I have fixed.

Concurrency issues

For inserts of SSKs, inconsistent ordering caused a race condition where the data might not have been committed before the insert finished on the request originator. This caused problems in sequential simulations with the fastest bypass mode, and I have posted a fix [35]. Debugging the fast message-level bypass revealed race conditions in various layers of Freenet, which I have fixed.

3.6.3 Other Freenet bugs discovered while debugging the project

Opennet bugs

After a CHK request completes successfully, there is an opportunity for some of the nodes involved in the request to set up a new connection by exchanging "node references", which are 3 KB files including the keys and IP addresses of the nodes. This is the basis of the opennet "path folding" mechanism described by Sandberg [21] and more practically in Freenet's 2007 paper [11].

The simulations run in darknet mode, but this mechanism still operates, as path folding connection requests can be relayed through darknet nodes if there are opennet nodes on the same path. This is one reason why the sequential simulator needs to wait until all requests have finished before starting the next one, as described in section 3.5.2: even after the requests have returned data, they may not yet have completed on the network, and may cause later requests to be rejected.

An interesting aspect of this is that if the node is the request originator, it will always reply immediately to a path folding request, either with its own node reference or a refusal message. This is exploitable, and I have filed a pull request [32] which introduces a random delay before replying. In general an attacker directly connected to every Freenet node is a Sybil attack and outside of Freenet's threat model, and can do a number of other devastating attacks, and this is one of the main reasons for darknet. However it is still worth fixing this.

An alternative solution is for path folding to start a few hops away from the request originator. This was originally proposed years ago, but not deployed because it is regarded as risky for performance (my recent implementation: [31]). It could be tested by simulating opennet, which has not yet been implemented.

3.7 Optimisations

3.7.1 Fake connection setup

The simulator uses a "fake connect" method, which avoids the need for expensive cryptographic handshaking and sending UDP packets, and prevents some nasty race conditions when the bypass is delivering messages while the two sides of the connection disagree over whether the connection is up. It improves start-up performance significantly for bypass simulations, since otherwise the simulation must wait for all the connections to complete, and for the network to settle: the CPU usage peak causes high ping times and therefore backed off nodes, and the simulator must wait until the average ping times fall and the backoffs expire.

When a bypass is enabled, ordinary handshaking is disabled, and the fake connections are set up during initialisation, passing in dummy objects for the encryption and packet tracking data structures. For example, the method for completing a connection requires a block cipher, but it is not used by the bypass code. I pass a dummy block cipher which not only does not encrypt anything, it will throw an exception if it is ever used. This avoids the risk associated with adding a "dummy encryption" mode, while allowing reuse of the existing code called when a handshake is completed. The details vary depending on the bypass mode.

3.7.2 Thread usage and plugins

For the sequential simulation, the number of threads is a significant limiting factor for performance. Developers can increase the operating system thread limits, but it would be preferable to avoid the need to change kernel settings to run a simulation, especially if it is part of the automated test suite! Also, each thread has a significant memory overhead. Each Freenet node has a number of threads that always run in normal operation, even if it is idle. The packet scheduler, UDP listener, IP address detection and DNS lookup code all have one thread each, but are disabled when the message-level bypass is in use. Others include:

1. 8 request starter threads, one for each possible request or insert type.

2. The datastore checker thread. Requests check the local on-disk cache on this thread, to avoid having large numbers of threads blocked for disk input.

3. The plugin garbage collection thread. This is part of keeping track of optional add-ons such as the plugin used to configure home routers.

I added configuration options to start these threads lazily ([27], [28] and [30]) on simulated nodes. These options might also be useful for some memory-limited production nodes. The lazy request starters patch also revealed some client layer bugs. My fixes ([25]) will need to be tested more thoroughly before the code can be merged.

The job scheduling classes Executor and Ticker are shared between simulated nodes: one of each is created for each processor, to reduce the number of threads, CPU usage and lock contention. Note that Freenet does not use the standard Java Timer class, partly because of the need for reliable per-thread priorities on Linux.

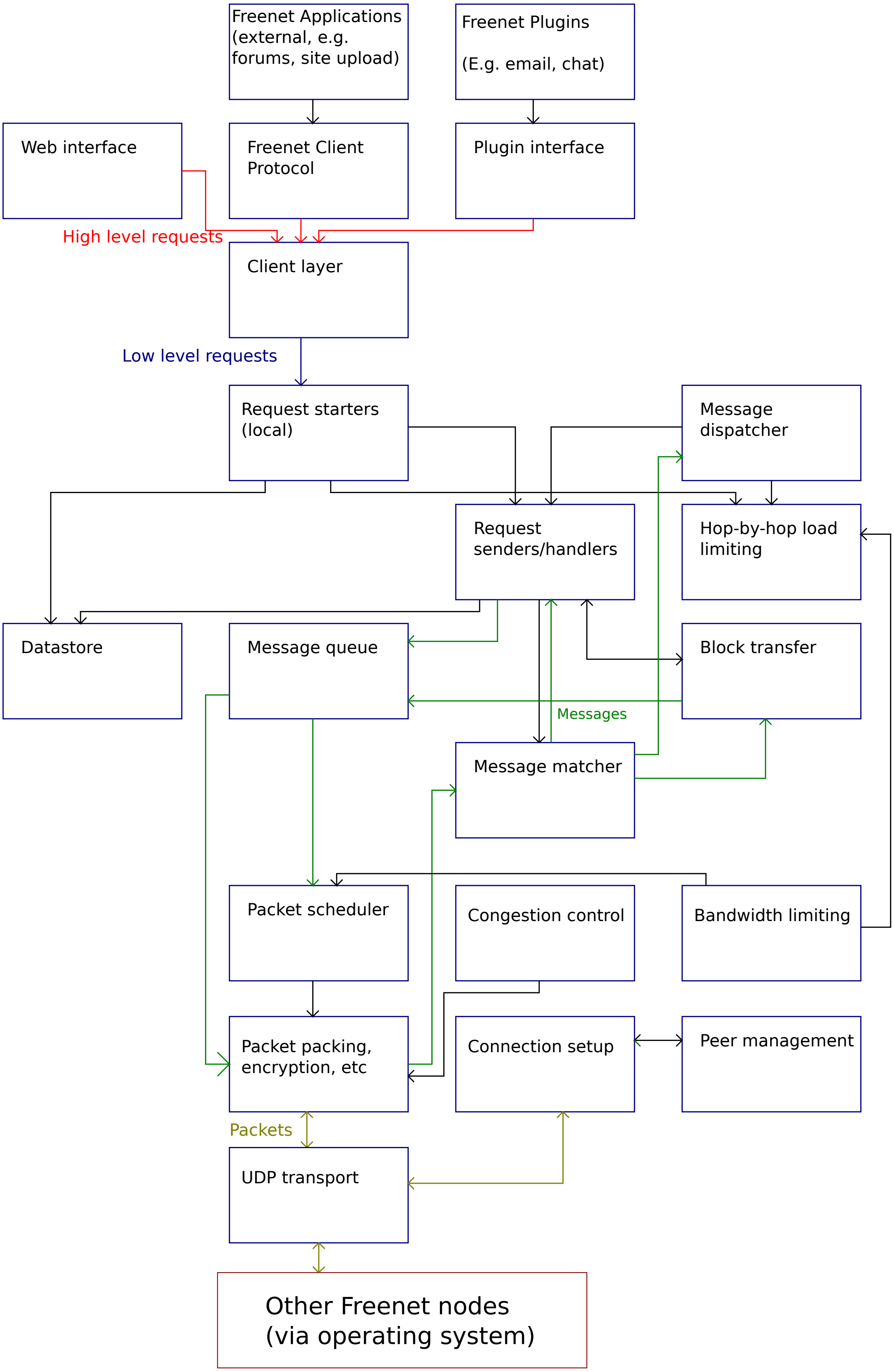

Figure 3.1: Diagram of Freenet layers relevant to this project. Blue shows the flow of high-level requests, red shows low-level requests, green is for messages and brown for packets. All other interactions between blocks are black.

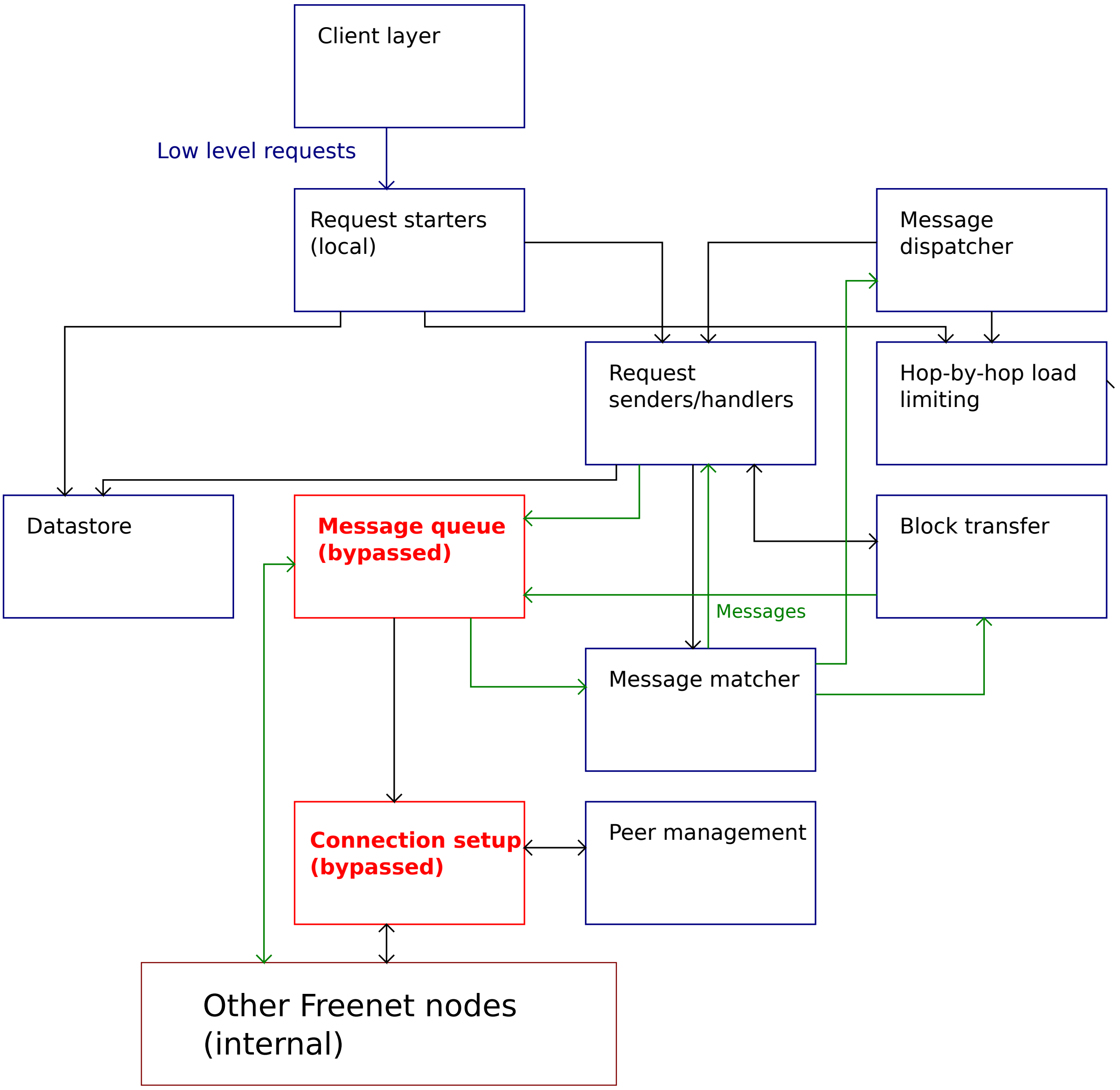

Figure 3.2: Freenet layers below the client layer, with the message-level bypass.

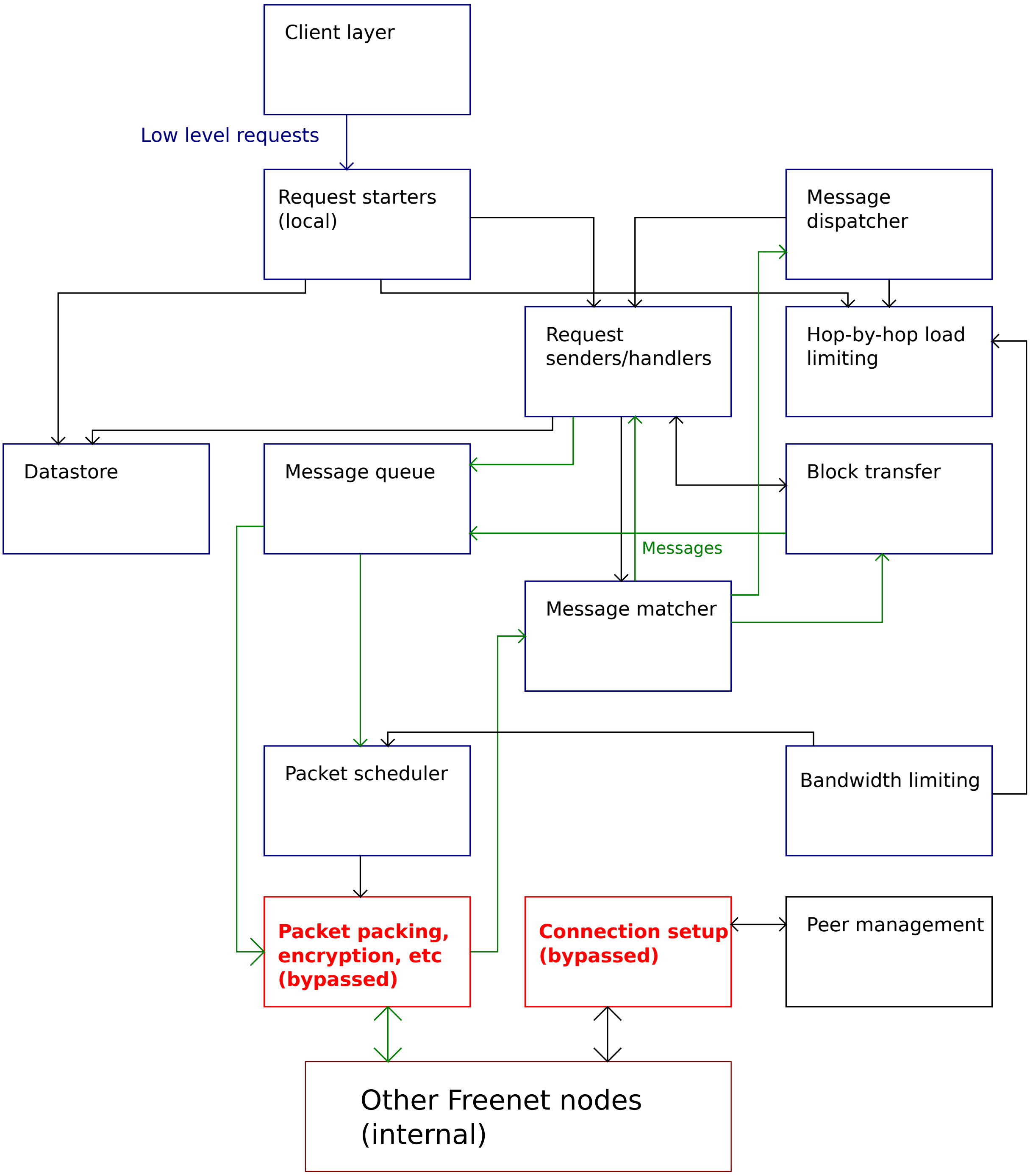

Figure 3.3: Freenet layers below the client layer, with the packet-level bypass

Chapter 4

Evaluation

4.1 Routing simulations using sequential requests

4.1.1 Validation of the sequential simulations

As described in section 3.5.2, routing changes can be tested deterministically by sending one request at a time across the simulated network. We can therefore compare behaviour exactly between different simulation bypass modes. The baseline is the relatively costly option of running full nodes and connecting them via UDP.

The current test code generates a 100-node network and repeatedly inserts random data through a random node and then fetches it through a different random node, until it has 20 successful requests. The test script runs 10 such tests with different random seeds and records the traces of all requests, and compares the log traces for each simulation mode. The log traces are identical, after fixing various bugs in Freenet (section 3.6).

4.1.2 Performance evaluation of the sequential simulations

Figures 4.1 and 4.2 show the effect of the new simulation modes on performance for sequential (routing) simulations. The simplest message bypass can be used here, and it reduces both the total CPU time and run time by over a factor of 10 relative to the baseline UDP simulation mode, while increasing the number of nodes that can be simulated from 1,400 to 11,000, which is larger than the deployed network. This makes the test fast enough to be used in regular build-time JUnit tests. For the message-level bypass and baseline UDP simulations, scalability is limited by CPU usage, as described in section 3.2.

The packet-level bypass is slower because it includes the packet scheduler, but it still reduces the CPU time by over a factor of 2. The packet scheduler and fair queueing have significant overhead as currently implemented. Another developer has proposed a patch recently [14] to improve this but were not evaluated here. However the surprisingly poor maximum network size of 1,800 nodes appears to be due to memory usage rather than CPU. The packet-level bypass is not as heavily optimised as the message-level code, and uses more threads, which may partly explain this, as each thread has a large stack.

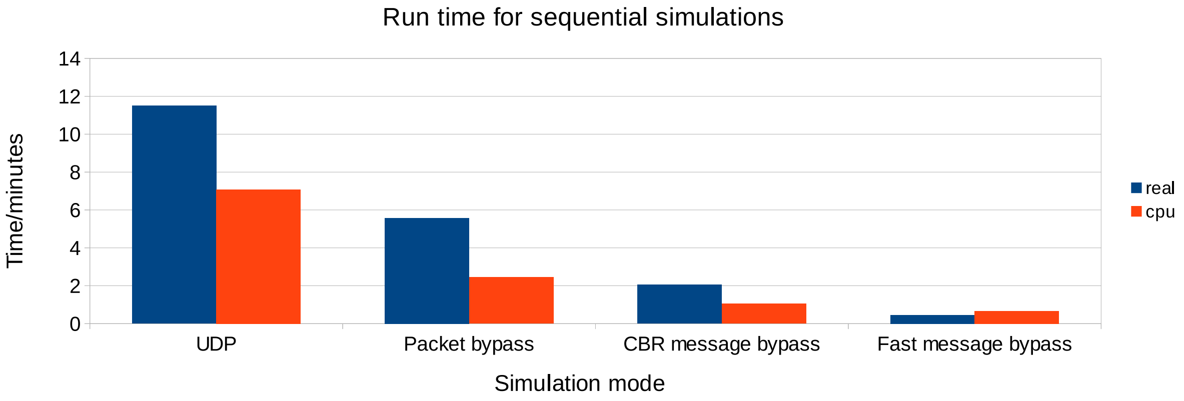

Figure 4.1: Time taken for a 100 node simulation run under each of the various simulation modes. The constant bit-rate message-level bypass has a bandwidth limit of 1 KB per second per connection, while the no bypass and packet-level bypass simulations have 12 KB overall shared between all peers of a node, and the fast bypass assumes infinite bandwidth.

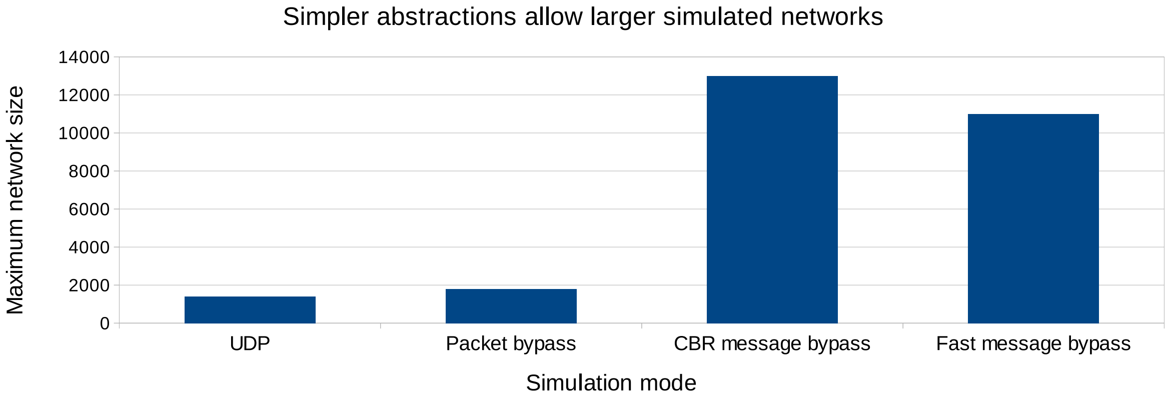

Figure 4.2: Maximum simulated network size on an AMD Phenom 2 X6 3.2GHz with 16GB of RAM

It also reduces the run-time, which is surprising since the bandwidth limits are the same. The discrepancy may be because the UDP code has a per-link congestion control window, but the packet-level bypass does not. No packets are lost, but the congestion window will start small for any given link, so on a large network it may take some time for every connection to reach full speed. Since this is an artefact of the relatively short simulation run-times, it is not worth the cost in performance and complexity.

Directly comparing the simulation modes may be slightly misleading in that the message-level bypass does not implement bandwidth limiting at all, the CBR bypass implements it per connection, while the UDP baseline and the packet bypass implement an overall bandwidth limit across all connections. However for sequential simulations this is mostly irrelevant.

4.1.3 Does Freenet routing scale?

Theoretical work[20] suggests that for a fixed node degree, the number of hops needed to find the data should increase with roughly $O(log^2 N)$ where N is the network size. The purpose of this project is to enable verifying such assumptions experimentally: the code may not implement the theory correctly. Figures 4.3 and 4.4 appear consistent with the expected scaling. However, the number of nodes a single request or insert will visit is limited by the Hops To Live counter (HTL), here set to 8. The "correct" value depends on the size of the network. As the number of hops needed to find the data increases, the success rate drops (Figures 4.5 and 4.6).

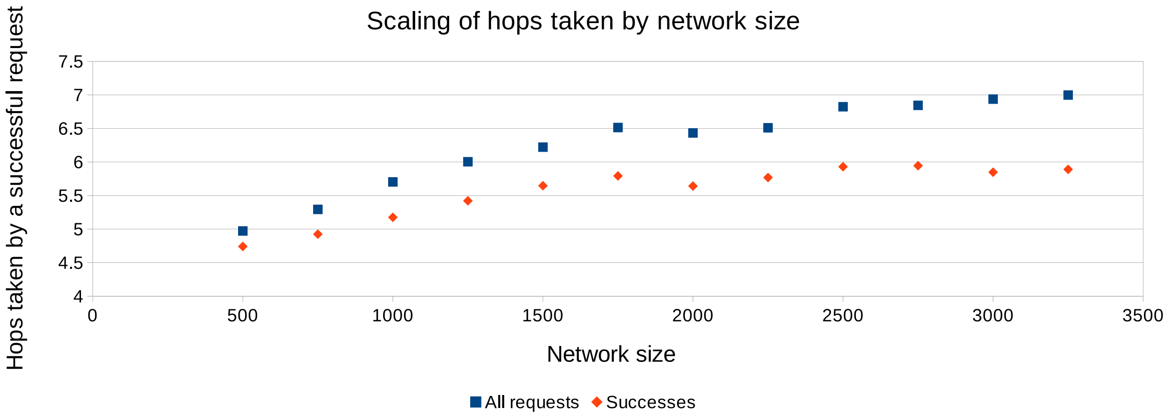

Figure 4.3: Number of hops taken by a request against the network size with maximum HTL of 8.

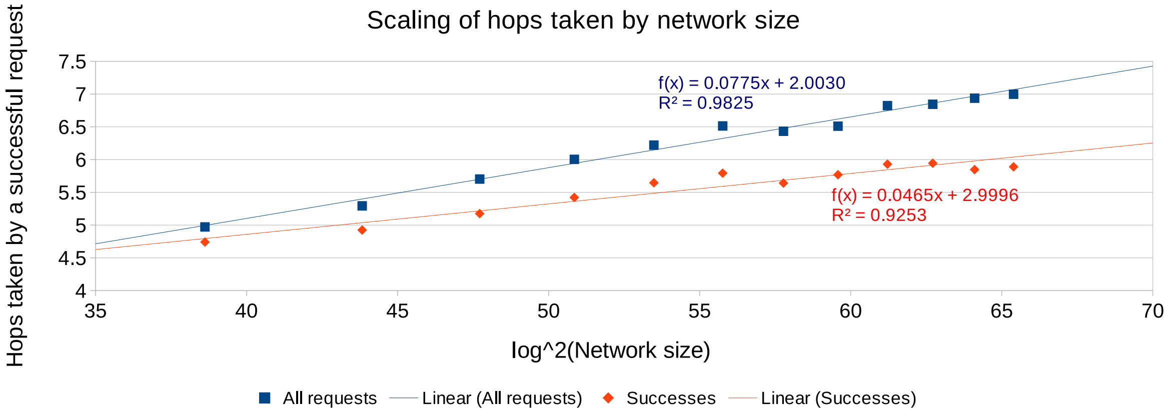

Figure 4.4: Number of hops taken by a request against the square of the natural logarithm of the network size with a maximum HTL of 8. Note that failed requests always go the full 8 hops, and the overall average is a closer fit than the hops taken by a successful request.

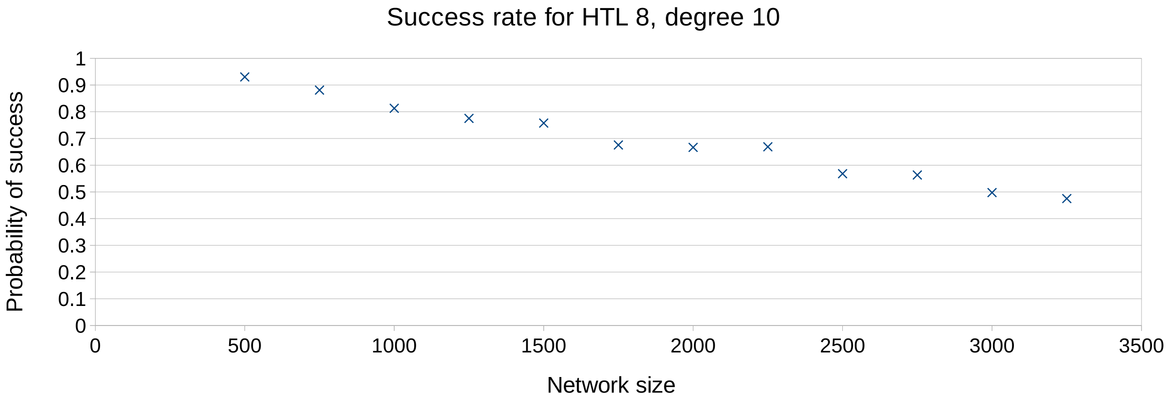

Figure 4.5: Success rate for increasing network size with a maximum HTL of 8

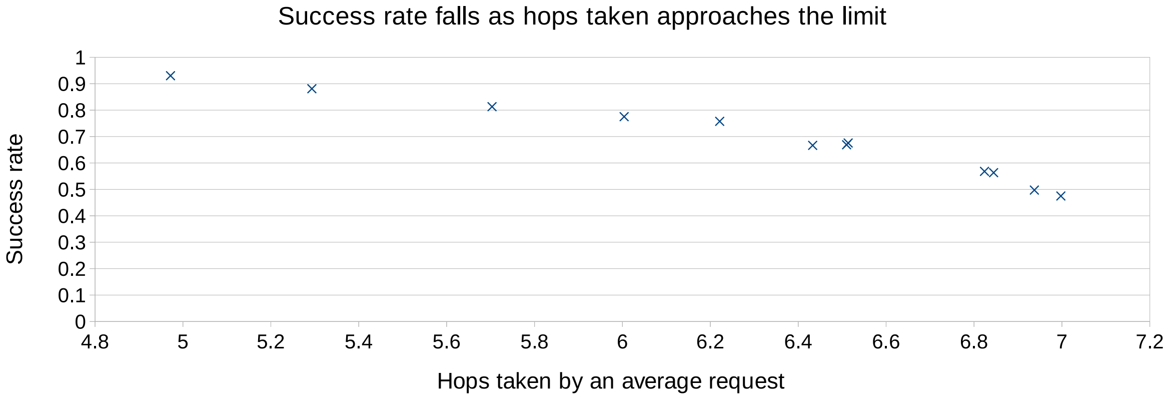

Figure 4.6: Success rate against average hops taken as the network grows with a maximum HTL of 8. As the network grows, the number of hops taken approaches the limit of 8 and the success rate falls.

4.2 Methodology for evaluating parallel simulations

4.2.1 Statistics and data collection

Unlike in the sequential case, simulations with many parallel requests are inherently non-deterministic, so the evaluation must be statistical. Section 3.5.5 describes the data collection for a single simulation run, which runs around 1000 requests. This is still a fairly noisy signal, for example, for one run with 25 nodes and 5 parallel requests:

Variable	Mean	Standard Deviation	Coefficient of variation
Hops taken	2.7	1.2	0.43
Success rate	0.84	0.37	0.44

The number of hops to a random key varies greatly on a small network, and this is inherent to the test environment, rather than the load management algorithms being tested. This makes inferences or comparisons difficult. Similarly request successes are a binomial variable which cannot be handled easily.

Hence each sample is repeated over 10 runs, using the same seeds (so the same network and the same series of keys), and the average and standard deviation are computed from this. This gives a much smaller standard deviation, mainly reflecting the remaining variability caused by concurrency issues, external load, random decisions in Freenet itself, and so on. This average will be approximately Gaussian (or more accurately follow a Student's t-distribution), so the error bars shown will have their usual meaning, and simpler statistical significance tests can be used.

4.2.2 Simulating different load management algorithms

In section 3.5.3, I explained that there are two kinds of parallel simulations. The first kind of simulator, evaluated in section 4.4.2, uses client-side load limiting in the same way as the real network, using closed-loop flow control based on feedback from the network.

On the other hand, the load management simulation without client-side load limiting attempts to keep a given target number of requests in flight at any given time. As the target load increases, requests are rejected and incorrectly routed, resulting in data not being found and requests failing. This is a good model of broken load limiting, caused by "cheating" nodes or by a buggy implementation of a new algorithm. In particular if a simulation bypass mode works with this simulation then it will work with any new load management algorithms to be tested. This is described in more detail in section 4.4.1.

4.2.3 Simulation modes for parallel simulations

As explained earlier, the message-level bypasses are not appropriate for parallel simulations. The simplest form of bypass delivers all messages immediately, so there are too few requests running at a time for a useful simulation of load management. The CBR bypass also breaks: since the message queue is bypassed, messages can get stuck behind block transfers, resulting in timeouts and disconnections. In practice this can be seen with 5 simultaneous requests on a 25-node network with the CBR bypass.

The packet-level bypass is more promising, providing fair queueing and reduced CPU usage relative to a full UDP-level simulation. Unfortunately the packet bypass diverges from the behaviour of the UDP code for high load levels, even though there is no actual shortage of resources on the simulation host. Hence the UDP simulations must be used when evaluating radical changes to load management. It may be possible to fix this with a better understanding of the issues raised below. However for the currently deployed load management algorithm, evaluated in section 4.4.2, load is kept well within the window in which the packet-level bypass works correctly.

4.3 Validation of the packet-level bypass for parallel simulations

Figure 4.7 compares the success rates for a UDP simulation to that with the packet bypass, with no client-side load limiting. Superficially the two series appear to be within each

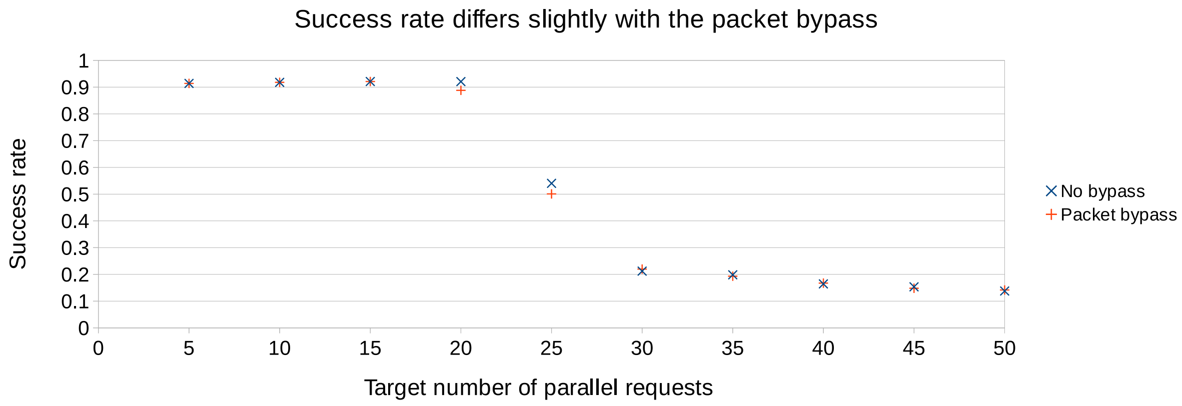

Figure 4.7: Comparison of success rates without client-side load limiting on a 25 node network for no bypass versus the packet-level bypass.

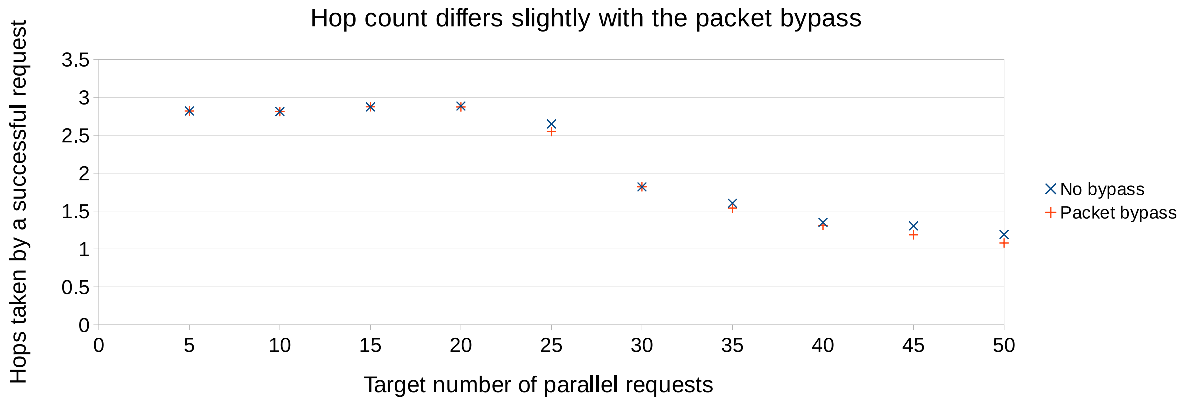

Figure 4.8: Comparison of hops taken by a successful request without client-side load limiting on a 25 node network for no bypass versus the packet-level bypass.

others' error bars (not shown for brevity's sake), but there is a larger gap between the two lines around 20-25 parallel requests, while the other samples are nearly indistinguishable.

This is reflected in the paired Student's t-test probability of 0.2, meaning there is a 1 in 5 chance that the difference between the two is due to noise which is independent of the input (Riley [19]). This is not considered significant, so may just be random variation. Figure 4.8 compares the number of hops taken. This appears to diverge systematically, with the packet bypass taking fewer hops at 25 and 35+ requests, which is reflected in the Student's t-test probability of 0.03, which is a significant difference between the two simulation modes.

The number of requests running is significantly different for the packet bypass, as shown in Figure 4.9 (p=0.006). As discussed in section 4.4.1, the actual number of requests running is equal to the target only up to a threshold (here 20), and after that it peaks and falls, because of inserts. However, this graph shows that the number of requests falls faster for the packet bypass than for the UDP simulations. The number of times the simulator stalls waiting for an insert to finish, discussed in detail in the next section, diverges less obviously (Figure 4.10, p=0.03).

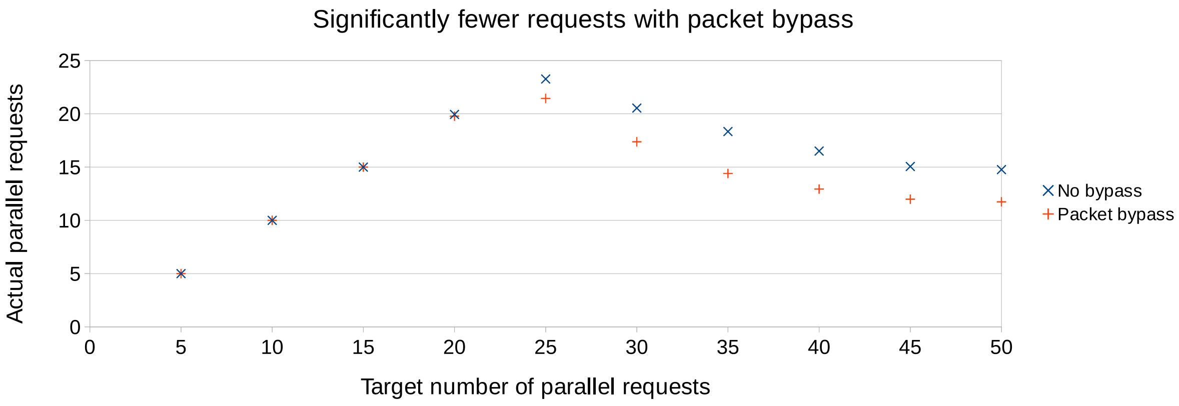

Figure 4.9: Comparison of actual requests against the target number of requests, for no bypass versus the packet-level bypass, on a 25 node network without client-side load limiting

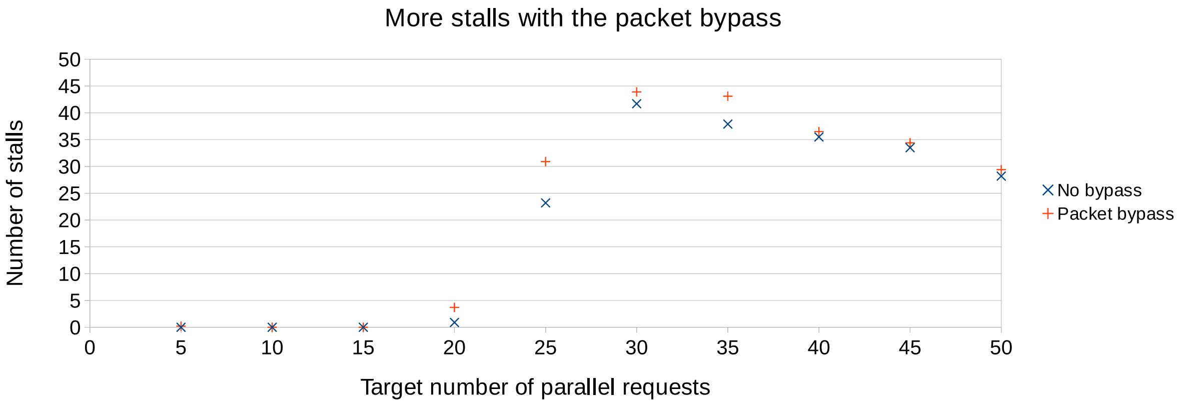

Figure 4.10: Comparison of stalls waiting for inserts against the target number of requests, for no bypass versus the packet-level bypass, on a 25 node network without client-side load limiting

I conclude that the packet-level bypass provides an accurate model only for the 5, 10 and 15 request cases here. Considering only this range, there is no significant difference between the two modes, with a p-value of 0.6 for success rates and 0.5 for hops, there are almost no stalls, and the actual number of request is within 0.1% of the target for both simulation modes. The packet-level bypass can be used as long as the number of actual requests matches the number of target requests, or the number of stalls is low. Neither of these criteria apply to more general load management algorithms where the number of requests is intentionally limited.

4.4 Load management simulations

4.4.1 Parallel requests without client-side load limiting

Inserts limit the maximum number of parallel requests

Every key requested must first be inserted, and the simulator starts the inserts well before the corresponding request will need the data (the data for request x is inserted while the request $x - 5n$ is being requested, for n parallel requests). Hopefully the insert will have completed by the time the key is needed. If not, the simulator must wait ("stall") for the insert to finish before starting the request.

Inserts are slower than requests, partly because they always continue until the Hops To Live counter is exhausted, while requests can terminate as soon as they find the data. Hence to keep the number of actual requests running close to the target, the simulator must avoid stalls by running more inserts than requests. For high load levels this overloads the network. This is expected, and should cause requests and inserts to be rejected by hop-by-hop load limiting, resulting in data not being found, being stored on the wrong nodes, and eventually in requests being unable to move past the originating node.

However the actual behaviour is surprising, revealing bugs that are not exposed at lower load levels. Some of these unexpected behaviours are different between the packet-level bypass and the full UDP simulations, as demonstrated in the previous section, which is why we cannot use the packet bypass to evaluate new, potentially broken load management algorithms. Note that there is no shortage of resources (CPU or memory) on the host computer.

Stalls and the distribution of insert times

The simulator originally ran all the inserts on a single node. The main loop would stall for long periods waiting for inserts to complete, so the average number of parallel requests fell dramatically, to around 13 on a 100 node network with a target of 350 parallel requests. Figure 4.11 shows the distribution of insert times. 31% of inserts complete within 40 seconds, but all the rest take over 5 minutes. Debugging shows this is due to block transfers taking a long time.

One theory I investigated was that the the difference between fast and slow inserts is caused by uneven distribution of load across the inserting node's neighbours. On a Kleinberg network, most of them will be close to the node's location and a few will be "long links", which most of the locally originated requests will start on. Routing randomly for the first hop ([33]) should result in more even load. But it does not dramatically change the distribution of insert times. It was worth implementing anyway as it improves anonymity and possibly performance on the real network.

Another theory was that the problem is caused by blocks from an insert being delayed for too long on rarely used connections due to a timeout being too large, which is not the case either. This is puzzling, and may indicate a bug in the message queueing code, but it is not relevant to the overall goals of this project. I have filed a bug report [23].

In the final version of the simulator, inserts are distributed in a round-robin manner across all the nodes other than the requesting node, which allows much higher load levels before the inserting nodes become a bottleneck.

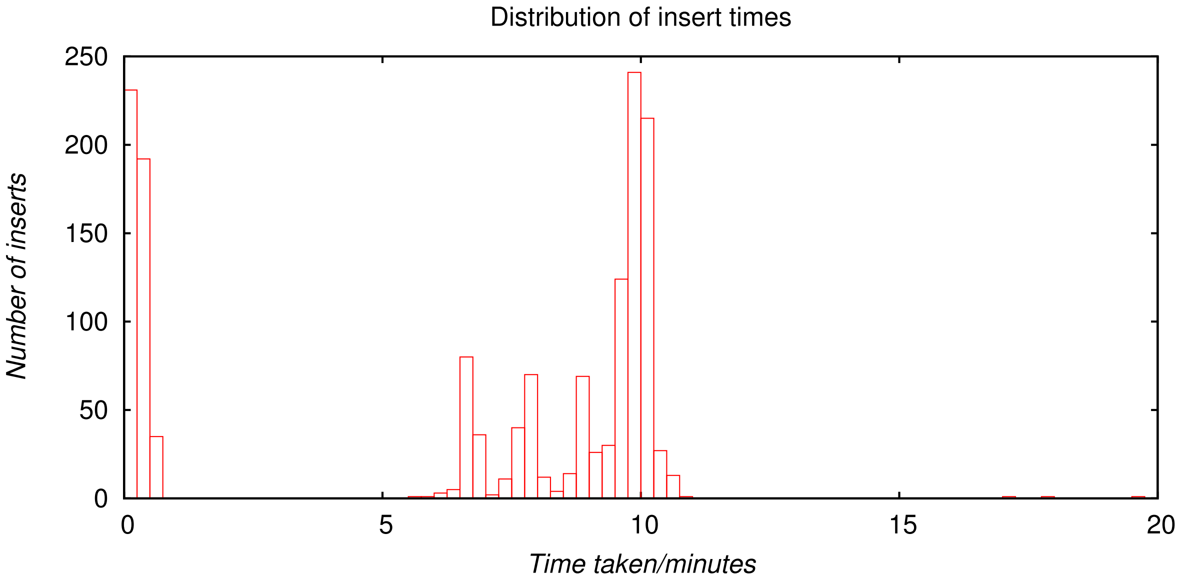

Figure 4.11: Distribution of time taken by inserts in a simulation of 100 nodes with a target of 50 parallel requests, with all inserts originated on the same node. Note that 69% of inserts take more than 5 minutes, while the remainder take less than 40 seconds!

Limiting factors in practice

On a 100 node network, for up to 80 parallel requests (60 with the packet-level bypass), distributing inserts across all non-requesting nodes works well. Requests do not have to wait for inserts, and therefore the number of requests running is within 1% of the target. Above 80 parallel requests, the simulator frequently stalls, unable to start the next request until the corresponding insert has completed (Figure 4.12), the actual number of requests diverges from the target (Figure 4.13), the success rate drops dramatically (Figure 4.14) and so does the number of hops taken by a successful request (Figure 4.15). The picture is similar with a 25 node network, which is used in most of the other simulations.

A bigger network distributes the load across more nodes, and thus allows more parallel requests before inserts become a limiting factor. A more realistic simulation, where some content is requested many times, or where many nodes are making requests, would allow for simulating more parallel requests. However, this depends on exactly what we want to test: only one node initiating requests allows us to see what happens to the AIMD request window more easily.

There appears to be a negative correlation between the simulator stalling waiting for inserts to finish and success rates. For a 25 node network with no bypass, the overall Pearson correlation coefficient is -0.88, but this is hardly surprising as both depend on the target number of requests. Taking correlation coefficients separately for each block of 10

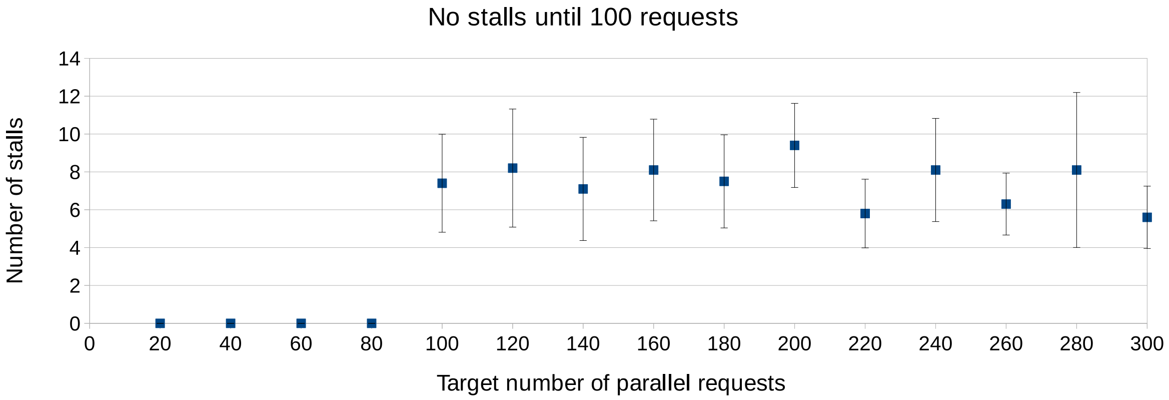

Figure 4.12: Number of times the simulator stalls waiting for inserts against the target number of parallel requests. For a UDP-level simulation of 100 nodes with degree 10, there are no stalls until 80 parallel requests. The number of stalls does not consistently increase with load, probably because several inserts can complete while the simulator waits for a single stall.

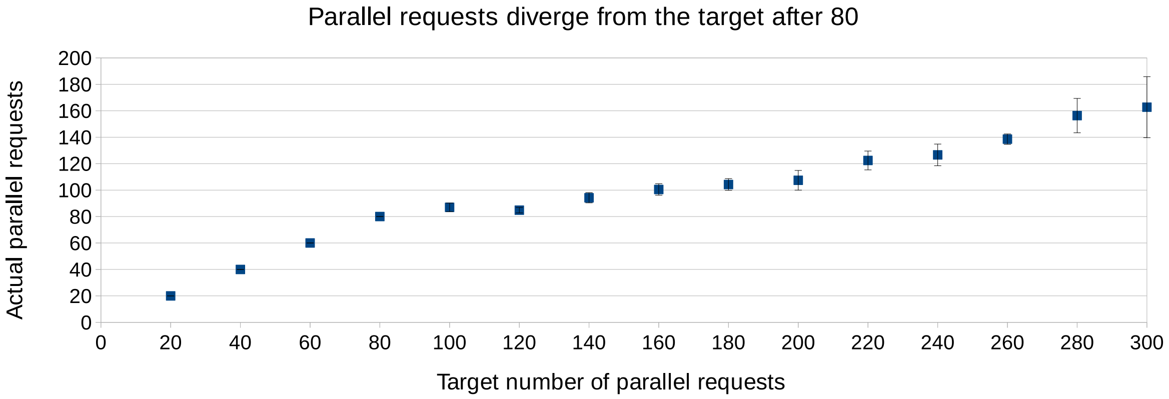

Figure 4.13: Actual number of parallel requests against the target number of parallel requests. The two are indistinguishable up to 80, but diverge after that.

simulation runs at the same number of parallel requests, there is a significant correlation for 20-40 requests, independent of the target number of parallel requests.

Exactly what is going on is not clear. It seems likely that when there are a large number of slow inserts blocking the network, we get both stalling and request rerouting causing low success rates, and odd behaviour by inserts. The key point is that the success rate drops dramatically above a threshold number of actual requests, therefore the goal of load limiting should be to keep the number of requests below that level, but high enough to maximise throughput.

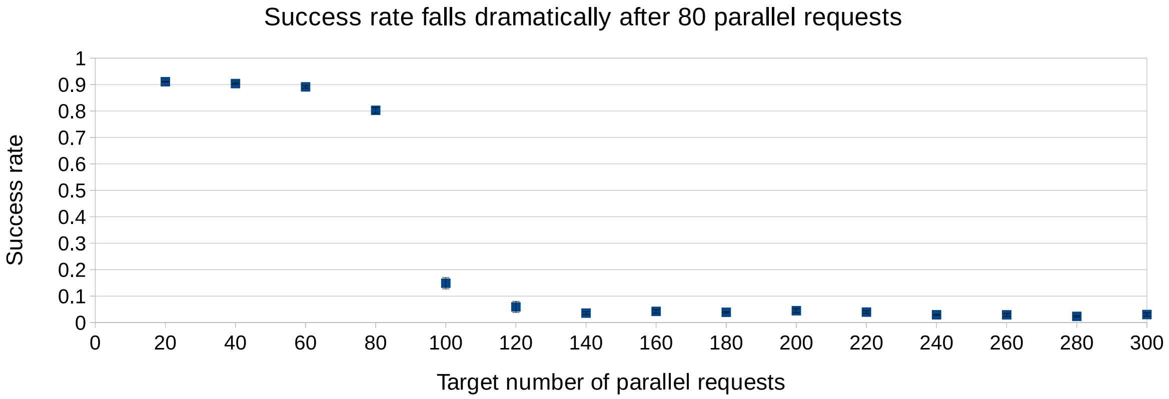

Figure 4.14: Request success rate against the target number of parallel requests. Collapses after 80 parallel requests. This is likely due to requests being rejected by hop-by-hop load limiting when there are too many inserts running on a connection.

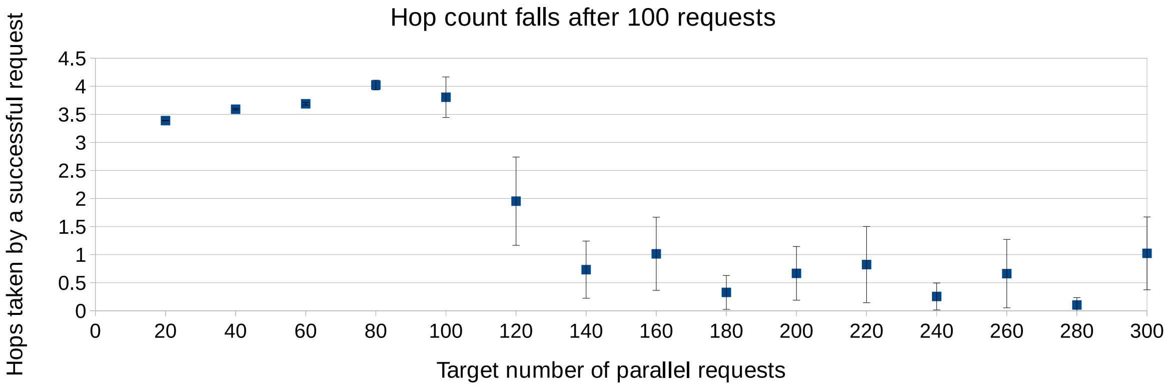

Figure 4.15: Number of hops taken by a request against the target number of parallel requests. Collapses after 100 parallel requests. I suspect this is due to requests being blocked close to the originator.

4.4.2 Parallel requests with client-side load limiting

Figure 4.16 shows the average number of actual requests running on a UDP-level simulation with full client-side load limiting as described earlier. Requests are queued by the simulator, and actually started on the network by the request starter thread, which limits the number of requests running according to the AIMD request window, based on feedback from the network. The graph shows that the number of actual requests stabilises at slightly under 9 parallel requests. This is well within the limit of 15 or so we established earlier for this network, so we can reasonably expect it to be similar for the packet bypass case, but I did not have time to compare the two modes directly in this case. Once this stabilises at around 30 requests queued, there is no significant correlation (Pearson correlation coefficient of 0.17) between the target number of requests and the number actually running.

I expected the success rate to initially decrease and then stabilise as the number of target requests increases: increasing the number of target requests should only increase the length of the queue, and this should have no effect on the request window size, or the success rate. Figure 4.17 is consistent with this expectation. Once the request window stabilises at around 30 requests, there is no statistically significant correlation between the number of target requests and the success rate: the Pearson correlation coefficient is $r = -0.33$, which gives a z-score of 1.2 and a probability of 0.21 that there is no correlation ($\rho = 0$), which is not considered significant.

Figures 4.18 and 4.19 compare the success rate and actual number of requests running on the originating node with client-side load limiting enabled with the corresponding results with it disabled. This shows that, on a small network and in the absence of "cheating", load limiting works.

However, comparing the two figures, the success rate only starts to drop off for the no-load-limiting case after 20 requests, while the number of parallel requests started by the simulation with full client-side load limiting stabilises at less than 10. This suggests that the equilibrium reached is suboptimal, and the algorithm is too conservative: the node could send more than twice as many requests into the network with only marginal impact on success rates. There was insufficient time to do the same comparison on larger networks, to see what happens to this gap, but it should be a priority for future work.

More data, over a wider range, would be interesting given the success rate appears to oscillate. If there is a trend it would be reasonable to expect the success rate to be much lower at a much higher number of requests: Provisional results at 200 parallel requests target give a success rate of 0.92 and 8.5 request window, which is within the range shown here.

I conclude that simulating load management is practical and useful conclusions can be drawn from a relatively small network. In particular, Freenet's current load limiting algorithm works, but is sub-optimal. More tentatively, the packet bypass appears to work as long as load management works, but unfortunately the converse is not true. Given the poor performance of the packet-level bypass, it makes more sense to use the full UDP level simulations for simulating changes to load management.

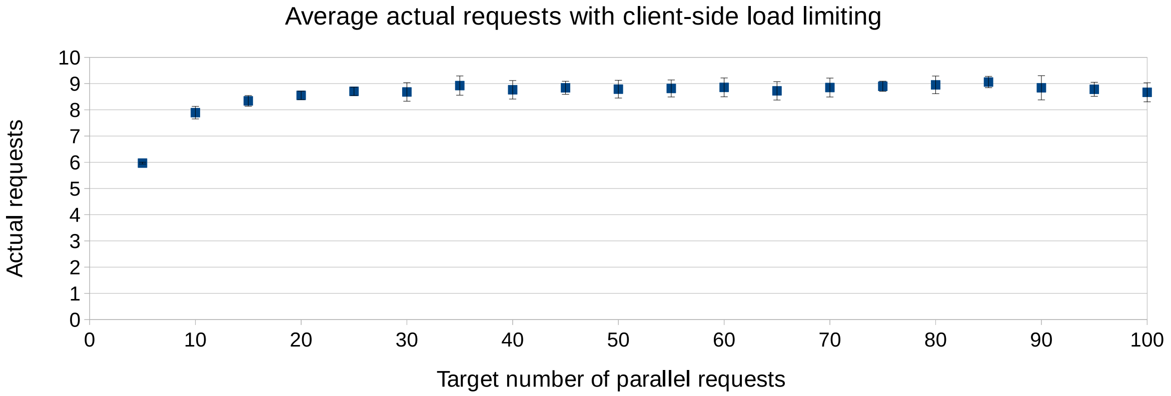

Figure 4.16: Actual number of requests running against the target number of parallel requests for a 25 node network with client-side load limiting, node degree 5 and HTL 5. This appears to stabilise at around 9 parallel requests.

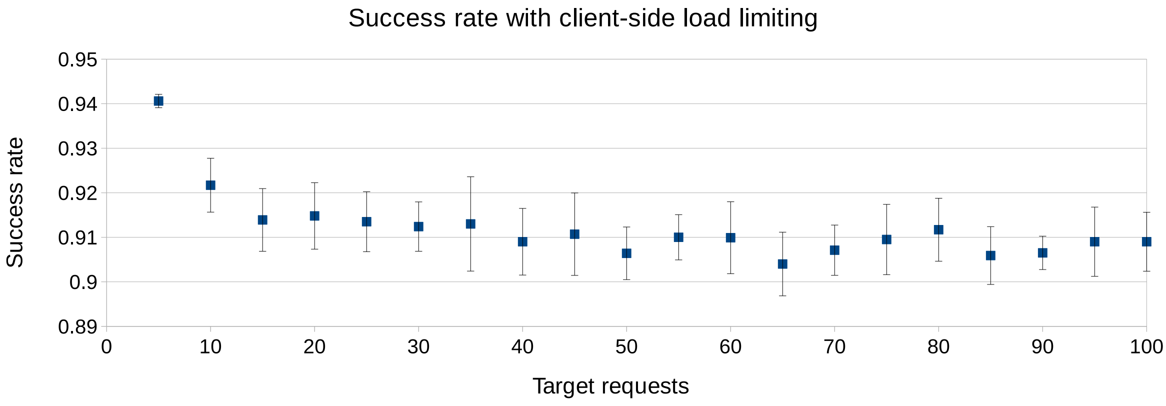

Figure 4.17: Success rate against the target number of requests for a 25 node network with client-side load limiting. Nodes have degree 5 and 5 HTL. The success rate initially falls quickly, and then appears to stabilise.

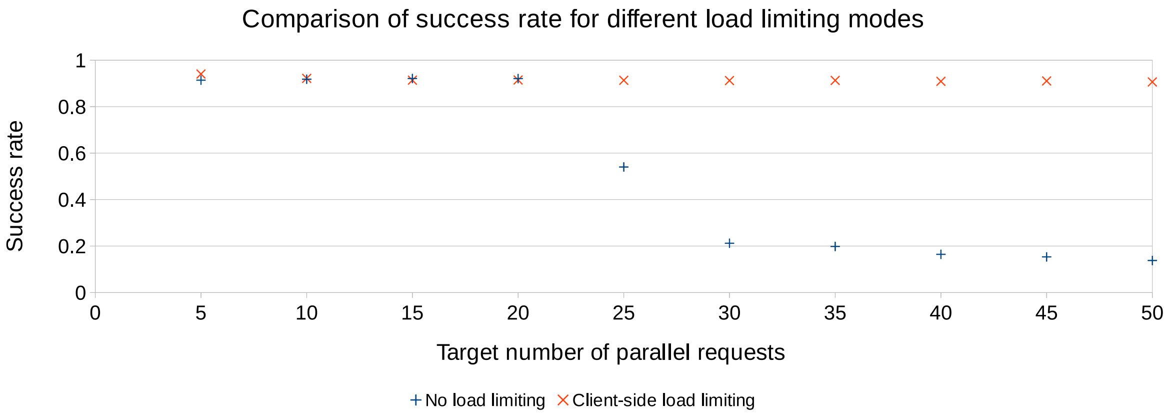

Figure 4.18: Success rate against the target number of parallel requests for a 25 node network with and without client-side load limiting

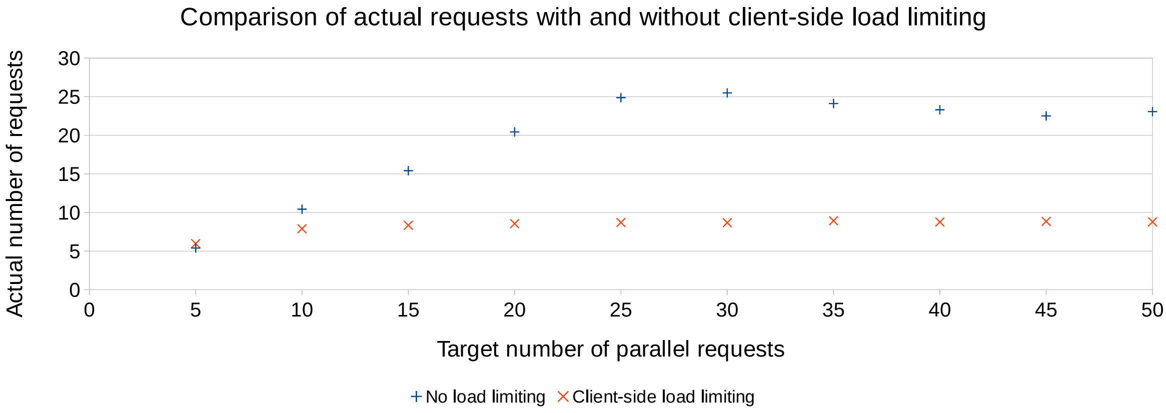

Figure 4.19: Actual requests against target requests for a 25 node network with and without client-side load limiting

4.4.3 Scalability, goals and future work

Simulating a small darknet network with a single requesting node appears to be sufficient for simple comparisons between different load limiting algorithms. It also simplifies evaluation because only one AIMD request window evolves over time. This may be enough for sanity checking changes to load limiting before deployment. Section 4.4.1 explains limitations on the number of simultaneous requests that can usefully be simulated, and a larger network would not significantly improve accuracy unless accompanied by a more complex workload model, for example with multiple request sources.

A more realistic model with better traffic generation and multiple sources of requests would be useful, but there was insufficient time to implement this. It would likely be harder to evaluate, so the lower-level model implemented here is a necessary precursor. So this is left for future work.

There is a tradeoff between run-time and CPU usage by changing the bandwidth limits, and it is easily possible to simulate parallel requests on 100 nodes, in UDP mode, using a relatively old system (an AMD Phenom 2 X6 with 16GB of RAM), so it should be possible to simulate large busy networks. Hence while the packet bypass does not currently provide accurate results for parallel simulations of load management, useful simulations of load management can be implemented without it.

A bypass at the level of block transfers might allow using the message-level bypass for parallel requests. That would allow much bigger networks, as shown in the sequential simulation section. However, it is possible that fairness between messages sent by different requests is necessary even then, since requests typically involve a number of messages other than the actual data transfer. More importantly, the limiting factor on a deployed Freenet network is bandwidth, and the CBR bypass does not model this accurately unless load is uniformly distributed across each node's connections. Optimising the packet scheduler (e.g. [14]) would improve CPU usage for the packet bypass, provided that the issues identified in section 4.4.1 can be addressed.

I conclude that more complex simulations are certainly possible on modest hardware, but the work done in this project is sufficient for basic comparisons and sanity checks, in particular demonstrating exactly why load management is necessary and that the current load limiting code works for simple cases.

4.4.4 Earlier results

Some of the early simulations include simulations of a new load management algorithm. These are worth including here even as a proof of concept, although the results should not be relied upon since the bypass code used at the time had some serious bugs, and this was before the investigations in section 4.4.1. Also these results only include the most basic statistics of the success rate and number of hops taken, meaning it is not possible to tell what the actual number of requests in flight was, which is crucial to performance because it determines throughput. All these simulations model 100 node networks with the packet-level bypass enabled, and started all inserts on a single node, unlike the current code.

Proposed enhancement to load limiting

With the current load limiting, nodes only reduce their AIMD window size after a request is rejected. Hence in equilibrium many nodes are backed off at any given time, causing requests to be routed incorrectly even when nodes are not severely overloaded. Ian Clarke proposed to change this so that nodes send the signal to reduce the number of requests being sent when the node is close to its capacity but has not yet exceeded it, before it actually starts rejecting requests, hence hopefully only nodes which are very slow or overloaded by external factors will be backed off, and fewer requests will be routed incorrectly. This is implemented by my pull request [29].

Figure 4.20 shows the success rate for increasing numbers of requests with this patch. Figure 4.21 compares the success rate for increasing numbers of requests between the current client-side load limiting and the new patch. Performance appears to improve with load with this patch, which is unexpected, and might indicate that the window size is reduced too quickly. Figure 4.22 compares all three modes: no load limiting, standard AIMDs for client-side load limiting, and the early slow down patch. The curve with no load limiting is different to that given in section 4.4.1, partly because in these older simulations all inserts were started on a single node.

This change to load management should not be deployed without re-running the simulations with the current simulator code, and checking that the latest code produces the same results. Also, it is important to get a graph of the number of actual requests running: if this is much less than with the current code, the improvement in success rates may not be worth the reduction in throughput. Unfortunately this was only added after these simulations, and there was insufficient time to re-run them. However it is an interesting proof of concept: a real alternative load limiting algorithm evaluated using the new simulation framework.

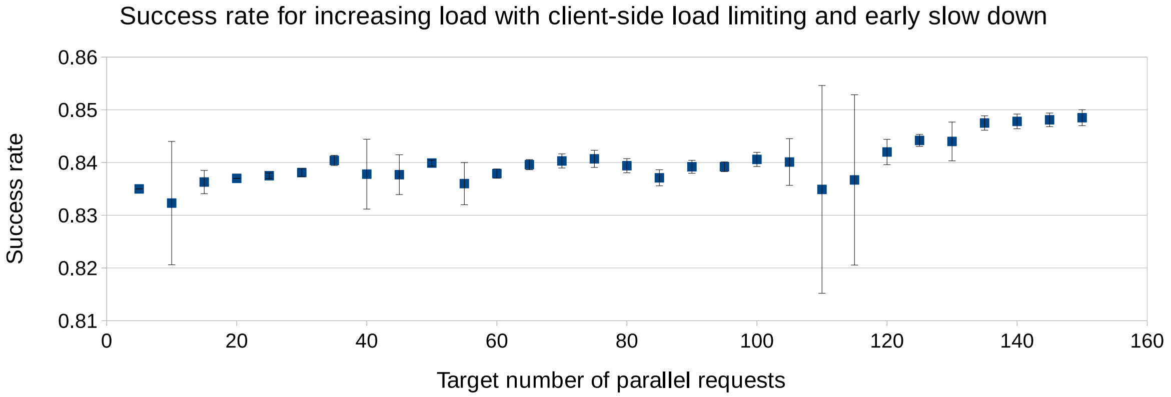

Figure 4.20: Success rate against the target number of parallel requests with the enhanced client-side load limiting algorithm. Based on older code so may not be accurate. Note the large error bars for a few points!

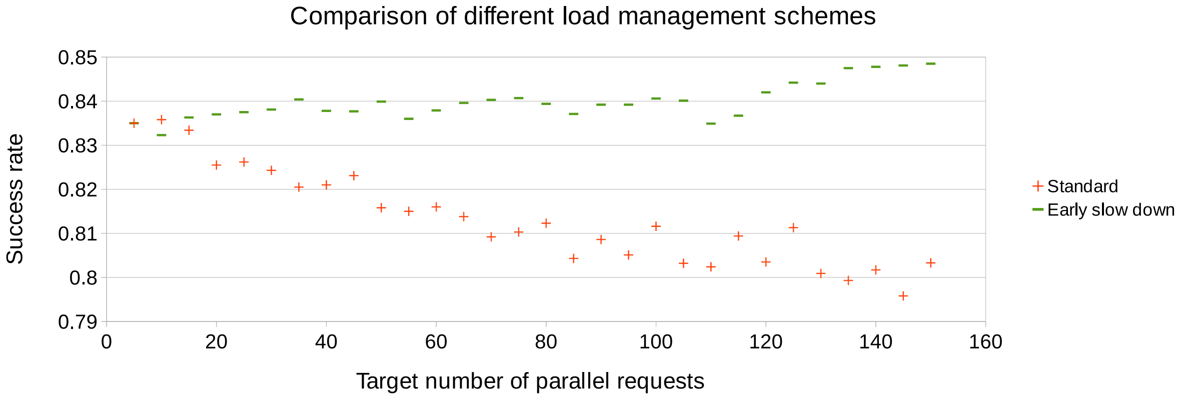

Figure 4.21: Success rate against the target number of parallel requests with standard AIMDs versus the enhanced client-side load limiting algorithm, on a 100 node network, using the packet-level bypass. Based on older code so may not be accurate.

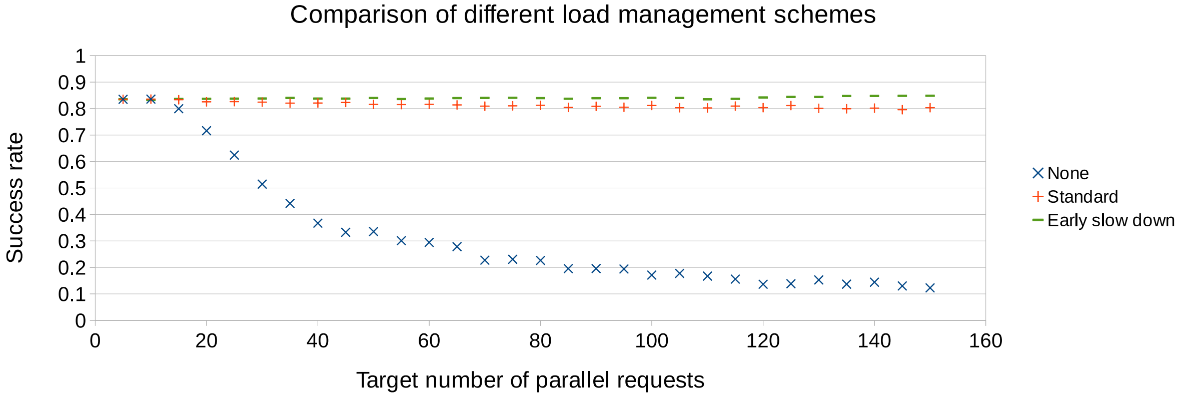

Figure 4.22: Success rate against the target number of parallel requests with various load limiting schemes, on a 100 node network, using the packet-level bypass. Based on older code so may not be accurate.

Chapter 5

Conclusion

The first goal of the project was to improve the performance and reliability of sequential (one request at a time) simulations, by implementing various bypass paths which allow faster and more efficient communication between multiple simulated Freenet nodes in a single Java Virtual Machine. This has been a clear success: The simulation framework now works reliably, provides fully deterministic results which are identical for all bypass modes, and the “fast message-level bypass” improves performance by an order of magnitude. This has enabled me to add new integration tests for routing to the standard test suite, which verify that there are no major regressions in routing or request handling. The fastest test, using the message-level bypass on a 25 node network, runs during the standard build-time test suite, while the more involved tests only run when extensive testing is requested.

It also allows simulating networks larger than the currently deployed Freenet on my relatively low-specification home server (6 cores and 16 GB of RAM). I have also used this to confirm that an important theoretical result about Freenet’s scalability is in fact correct with the current Freenet source code. The packet-level bypass achieves smaller performance gains, but can be used in a wider range of tests. While the message-level bypass is limited by CPU usage, the packet-level bypass appears to be limited by memory usage, which can likely be improved by turning off more unnecessary code in simulation.

The second goal was to implement a framework for testing changes to load management, by simulating many simultaneous requests. My implementation supports both simulations with the current client-side load limiting algorithm and those without it. The latter is important because the Freenet developers need to be able to simulate new load management algorithms and implementations which may not correctly limit load. For very high load levels inserting data quickly enough to keep up with the requests becomes a bottleneck and the packet-level bypass does not work correctly, and there is some interesting but not yet understood bimodal behaviour with inserts, which may indicate a bug in Freenet. When load limiting is turned on, the number of requests is kept within the range where the packet-level bypass works correctly. However when simulating changes to load management, the UDP-level simulations should be used, and as I have demonstrated this is easily practical on modest hardware.

In the process of investigating these issues I have added extensive instrumentation, recording not only the success rate but also the exact path taken by every request across the network, the actual number of requests running, the time taken by inserts and requests and so on. This required adding various hooks to Freenet which have a negligible performance impact on standard Freenet nodes outside of simulation, and I expect the code to improve maintainability, particularly by improving testing, so it should be merged upstream shortly. I have also fixed numerous bugs in Freenet, especially in the first stage of the project where various problems initially prevented reliable, reproducible simulations.

The workload model is relatively crude in that every block is inserted once and requested once. This is the worst-case scenario for long-term content availability, which arguably determines the usefulness of the Freenet network. In reality data is requested many times, resulting in improved performance. Also, in the simulation, a single node starts all the requests for data, although it must share resources with inserts started on other nodes, which makes it easier to compare load management algorithms and measure the effect on the request window (the number of simultaneous requests sent into the network by a node). The code given here can easily be extended to a more realistic model, which would avoid some of the problems with inserts, but might require more computational resources. In any case the current simulations are sufficient for simple evaluation and comparison of load management algorithms.

I have verified experimentally that the existing client-side load limiting works, at least for a small network. The number of requests and the success rate stabilise, however the equilibrium reached appears to be somewhat below the capacity of the network: the request window could be almost doubled with the same success rate. I have also evaluated a proposed variant on the current load limiting protocol, which seems to substantially improve performance, but the simulations were based on an early, buggy version of the code and there was insufficient time to re-run and re-evaluate them.

I have achieved all of the original success criteria, except that the packet-level bypass is only applicable for a limited range of input loads. The original sub-projects were slightly modified: the global datastore optimisation was not needed for either type of simulation, and while traffic generation is currently not very realistic, it does model a useful point in the design space, which is of legitimate interest and which much work has been spent on over the last decade: the "long tail" of data that is not often accessed, but makes up the majority of available content.

While I have modified the detail of the original project goals somewhat, I have achieved the two fundamental goals: to efficiently test routing, and to enable and demonstrate modelling of changes to load management. This is reasonable and consistent with modern software engineering practice, which is often iterative, especially for a project with a significant experimental component as here. I have also confirmed two key assumptions about Freenet's behaviour, discovered and fixed several bugs in Freenet, and revealed some new questions.

Bibliography

[1] Content hash key. `"https://wiki.freenetproject.org/index.php?title=Content_Hash_Key&oldid=2615"`. [Freenet wiki article, accessed 19/04/2016].

[2] The freenet project. `"https://freenetproject.org/"`. [Open-source project homepage].

[3] Freenet source code. `"https://github.com/freenet/fred"`. [Git repository].

[4] Mockito. `"http://mockito.org/"`. [Online; accessed 07-May-2016.

[5] Ns-3. `"https://www.nsnam.org"`. [Open-source project homepage].

[6] Signed subspace key. `"https://wiki.freenetproject.org/index.php?title=Signed_Subspace_Key&oldid=2613"`. [Freenet wiki article, accessed 19/04/2016].

[7] Brian Bailey and Grant Martin. *ESL Models and their Application*. Springer, 2010.

[8] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD), January 2005.

[9] Tony F Chan, Gene H Golub, and Randall J LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

[10] I Clarke, O Sandberg, B Wiley, and TW Hong. Freenet: A distributed anonymous information storage and retrieval system. freenet white paper, 1999.

[11] Ian Clarke, Oskar Sandberg, Matthew Toseland, and Vilhelm Verendel. Private communication through a network of trusted connections: The dark freenet. `"https://freenetproject.org/papers/freenet-0.7.5-paper.pdf"`, 2007. [Online; accessed 13-October-2015].

[12] Vincent Driessen. A successful git branching model. `http://nvie.com/posts/a-successful-git-branching-model/`, 2010. [Online; accessed 01-April-2016].

[13] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[14] Eleriseth. Eleriseth's pending keys patch. `"https://github.com/freenet/fred/pull/404"`. [Git pull request for Freenet].

[15] Erik Hjelmvik. China's man-on-the-side attack on github. `http://www.netresec.com/?page=Blog&month=2015-03&post=China%27s-Man-on-the-Side-Attack-on-GitHub`, 2015. [Online; accessed 09-May-2016].

[16] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[17] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO87*, pages 369–378. Springer, 1987.

[18] Prateek Mittal, Matthew Wright, and Nikita Borisov. Pisces: Anonymous communication using social networks. *arXiv preprint arXiv:1208.6326*, 2012.

[19] Kenneth Franklin Riley, Michael Paul Hobson, and Stephen John Bence. *Mathematical methods for physics and engineering: a comprehensive guide*. Cambridge University Press, 2006.

[20] Oskar Sandberg. *Searching in a small world*. PhD thesis, Chalmers tekniska högskola, 2006.

[21] Oskar Sandberg and Ian Clarke. The evolution of navigable small-world networks. *CoRR*, abs/cs/0607025, 2006.

[22] TheSeeker, yd, and oo. Let it burn. `"https://github.com/freenet/fred/pull/243"`. [Git pull request for Freenet].

[23] Matthew Toseland. 0006833: Some inserts take less than 40 seconds and all the rest take more than 5min in simulation. `"https://bugs.freenetproject.org/view.php?id=6833"`. [Freenet bug report].

[24] Matthew Toseland. Auto-update improvements. `"https://github.com/freenet/fred/pull/475"`. [Git pull request for Freenet].

[25] Matthew Toseland. Cooldown fixes. `"https://github.com/freenet/fred/pull/525"`. [Git pull request for Freenet].

[26] Matthew Toseland. Fix keep-alive packets not being sent. `"https://github.com/freenet/fred/pull/398"`. [Git pull request for Freenet].

[27] Matthew Toseland. Lazy start for datastore checker thread. `"https://github.com/freenet/fred/pull/526"`. [Git pull request for Freenet].

[28] Matthew Toseland. Lazy start for requeststarters. `"https://github.com/freenet/fred/pull/524"`. [Git pull request for Freenet].

[29] Matthew Toseland. Load management send slow down when close to limit. `"https://github.com/freenet/fred/pull/388"`. [Git pull request for Freenet].

[30] Matthew Toseland. Make it possible to entirely disable plugins. "https://github.com/freenet/fred/pull/527". [Git pull request for Freenet].

[31] Matthew Toseland. No path folding at high htl. "https://github.com/freenet/fred/pull/490". [Git pull request for Freenet].

[32] Matthew Toseland. Random delay for opennet ack when not path folding. "https://github.com/freenet/fred/pull/491". [Git pull request for Freenet].

[33] Matthew Toseland. Random initial routing. "https://github.com/freenet/fred/pull/529". [Git pull request for Freenet].

[34] Matthew Toseland. Remove load limiting high-level token buckets. "https://github.com/freenet/fred/pull/523". [Git pull request for Freenet].

[35] Matthew Toseland. Ssk inserts: commit before reply and collisions. "https://github.com/freenet/fred/pull/485". [Git pull request for Freenet].

Appendix A

Project Proposal

Computer Science Project Proposal

Efficient simulation of an anonymous peer-to-peer network

Matthew Toseland (mjt92), Wolfson College

Originator: Matthew Toseland

20 October 2015

Project Supervisor: Stephan Kollmann

Director of Studies: Dr Chris Town

Project Overseers: Prof Jon Crowcroft & Dr Thomas Sauerwald

A.1 Introduction

I propose to improve the performance of simulations of Freenet, using the current official Freenet source code ([3]) for the higher layers but replacing the lower layers with more efficient bypass paths. This will allow testing proposed changes to Freenet with a controllable tradeoff between accuracy and efficiency. I will validate these approximate simulations against the real code, and demonstrate which is appropriate for two use cases (routing and load management). I will then investigate Freenet's load management, possibly including mechanism design issues with the current algorithms.

A.1.1 Freenet

Freenet ([2], [11]) is a distributed, anonymous data-store designed to prevent censorship. Unlike most anonymity tools, it supports both “opennet” (peer-to-peer) and “darknet” (friend-to-friend) mode operation, the latter using existing social trust to provide stronger security. In both cases this results in a small-world network which allows for greedy routing according to the node identifiers, see [20]. It supports publishing content including websites, larger files, a forums system, and a micro-blogging system.

Content is divided into fixed-sized blocks, identified by a key. Uploaded data blocks, and requests for data, are routed through a path converging on the nodes with the closest identifiers to the key. Data is cached on (nearly) every node and stored more permanently on the closest nodes. Load management is similar to TCP, with the originator deciding how many requests to send based on feedback. Exponential backoff is used to deal with temporarily overloaded nodes.

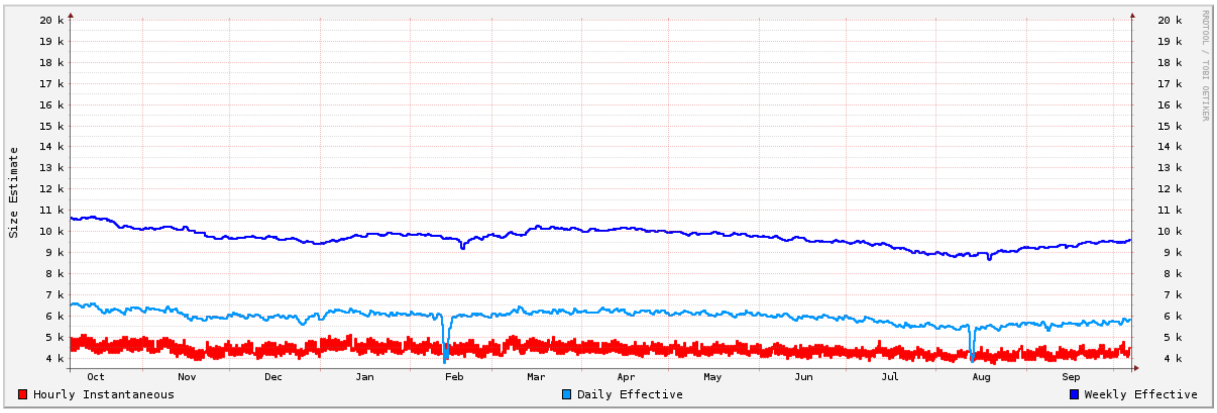

Figure A.1: Freenet active users in the year to October 2015. Thanks Steve Dougherty.

In the past the developers would try out changes to load management or routing on the main network. This was usually inconclusive and sometimes disastrous! Furthermore, the current load management is very vulnerable to “cheating”, and there is a widely distributed patch which appears to exploit this.

A.1.2 Starting Point

Freenet is written largely in Java. The network protocols are still evolving, in spite of 15 years work, hence the “0.x” versioning. I am familiar with the vast majority of the codebase as I was chief developer from 2002 to 2013.

Freenet includes several simulated tests that create many Freenet nodes within a single Java Virtual Machine, using the real code. However this is too slow to simulate large networks, or even to form part of the automated test suite. Most of the current simulations run one request at a time, which greatly reduces the CPU cost, but is not usable for testing load management, where we need to see the effect of many simultaneous requests from different nodes.

These simulations are in “darknet” mode, i.e. fixed connections on an idealised Kleinberg network, with limited bandwidth, but no latency. This models a real darknet, which we assume to be a small-world network of friend-to-friend connections. The real network is mostly opennet but the goal is to move towards more darknet. It also assumes that all nodes are constantly online, which is clearly an unrealistic starting point in terms of long-term behaviour.

Also, various ad hoc simulators have been written over the years for Freenet, usually for routing or the transport layer. None of these provide an acceptable model of load management, and they are not suitable for evaluating potential changes to the source code before deployment.

A.1.3 Proposal: A faster simulator

Figure A.2 shows Freenet's layering. I propose to bypass a configurable set of the lower layers to boost performance, while still using the current source code for the higher layers. Hence the simulator is always up to date, which is ideal for trying out changes before deployment.

For more abstract design work, a simpler, higher level, event-driven simulator might perform better. Conversely, modelling the underlying IP network accurately would be another project, possibly based on NS-3 ([5]). Here I am interested in testing and modelling the upper layers, especially any difficulties with mechanism design.

Over the summer I added a branch to bypass the message queue and below: Messages queued on one node are delivered instantly to the target node, bypassing all the lower layers. For e.g. testing routing, this should suffice, but for load management we need a more accurate, more expensive model.

A.2 Major sub-projects

1. An improved message level bypass, including per-connection bandwidth limits and latency. Messages will be delivered to other nodes in the same VM, but with a calculated delay. Note that the real code shares bandwidth fairly between all its peers depending on demand, so this is not an accurate model. However, the packet scheduler is complex and costly. If this is significantly faster than the packet-level simulator, it may be interesting to look at approximations.

2. Mid-level simulations using a packet level bypass. That is, I will use efficient internal data structures to represent packets rather than encoding actual encrypted UDP packets. This will accurately model sharing bandwidth between a node's peers, but use more CPU time, allowing fewer simulated nodes.

3. A global datastore and a bypass for block transfers, while still accounting for the data being transferred. This avoids both storing and copying data, so should save lots of CPU as well as RAM and on-disk storage. One possible extension would be to have procedurally generated blocks (e.g. from a random seed), which would save even more memory, but need further changes.

4. Test scripts and settings. Because this is a real-time rather than event-driven simulator, CPU overload is a potential problem if simulating a large network. We need to detect this and abort the simulation, or work around it, rather than allowing it to affect load management directly.

5. A realistic traffic generator for load management simulations.

A.3 Evaluation

It is not realistic to model the real network at the moment. Gathering data is restricted by privacy concerns, and the network is too complex and probably too large to directly compare numerical measures to an equally large simulation. Hence the best way to validate the simulation is to compare performance between different levels of approximate simulations. The baseline level uses the same code as the real network. The main performance metrics are the number of hops needed for a request and the success probability. I will compare these for the same network, and then repeat with different networks for a statistically significant comparison.

There are 2 interesting test cases, which likely require different approximation levels:

1. Routing-related tests where we can do sequential requests on a large network relatively cheaply. I expect that the message queue bypass with instant delivery implemented this summer will have almost identical routing metrics to the full real-code simulation, while allowing dramatically more nodes in simulation.

2. Load management tests where we need to simulate many simultaneous requests. Hopefully one of the new simulations will give an acceptable approximation of the real network (i.e. similar routing metrics), while allowing many more nodes than the real-code simulation.

In summary, the evaluation goals are to:

1. Compare accuracy (routing metrics) and performance (number of nodes on the same hardware) between models for the same network, for both sequential routing tests and load management.

2. Demonstrate the effect of load given the current load management algorithm.

A.4 Possible Extensions

There are numerous interesting extensions, including:

- Measure the performance of "honest" nodes as a function of the proportion of "cheating" nodes using the performance patch.

- Implement and evaluate some of the proposed alternative load management algorithms.

- Clean up the simulator API so that a useful simulation can be constructed quickly.
- Implement low-level simulations bypassing encryption and UDP, with artificial delays simulating inter-node latency.
- Further optimisations driven by profiling.
- Investigate multi-host simulations.
- Investigate the effect of variable uptime on darknet.
- Implement and evaluate friend-of-a-friend connections for darknet.
- Model an opennet network, with realistic churn.
- Compare routing performance on an ideal Kleinberg network to that on a darknet based on real-world social topology data.
- Implement a full event driven simulation, using a virtual clock, but still using the real code for the higher layers.
- Implement high-level integration tests using the simulator.

The schedule below incorporates only the core sub-projects and evaluation. Hence it is a ”reasonable worst-case” estimate. I hope to get into some of the extensions, some of which are arguably more interesting technically.

A.5 Timetable and Milestones

Start date is Monday 26 October 2015 (middle of week 3).

1. **Michaelmas weeks 3–4** Preparation of equipment (e.g. get cloud hosting working). Reading related papers and relevant course content.
2. **Michaelmas weeks 5–6** Further research and setup. Investigate other simulator frameworks e.g. NS-3.
3. **Michaelmas weeks 7–8** Re-implement and test message-level bypass. Implement statistical hooks (e.g. path length).
4. **Michaelmas vacation (4 weeks)** Implement packet-level bypass. Sort out issues around CPU usage. Evaluate performance.
5. **Lent weeks 1–2** Validate simulation. Write progress report.
6. **Lent weeks 3–4** Implement global datastore and block transfer bypassing.
7. **Lent weeks 5–6** Implement traffic generation. Measure performance of the simulator.

8. **Lent weeks 7–8** Demonstrate the effect of the current load management code. Start writing up.

9. **Easter vacation (4 weeks)** Write up.

10. **Easter weeks 1-3** Finish writing up and aim to submit in week 3 at the latest.

A.6 Success Criteria

The project will be a success if:

- The mid-level and high-level simulators are functional.

- All the simulators, including real-code, achieve similar results for simple simulations, e.g. average path length and success probability.

- For more complex simulations, at least one of the new simulation options provides a reasonable approximation of the results from the real-code simulation.

- The new simulators use significantly less resources than the full-code simulation.

- There are quantitative results from simulating load management, possibly including testing the “cheating” patch.

A.7 Resources Required

I will use my own systems for development and smaller scale simulations. I have a quad-core laptop with 12GB of RAM and a small backup laptop. Both connect to my home server which has 16GB of RAM, mirrored disks and 6 cores. I accept full responsibility for these machines and I have made contingency plans to protect myself against hardware and/or software failure. I can however move to MCS if necessary. I will use Git revision control and backups on my home server, MCS space and Github.

For larger scale simulations, I will purchase CPU time from Amazon EC2 (or failing that another cloud provider). I don’t believe that CONDOR is appropriate as it is an idle-time system. I will apply for £300 from college but I will fund it myself if necessary.

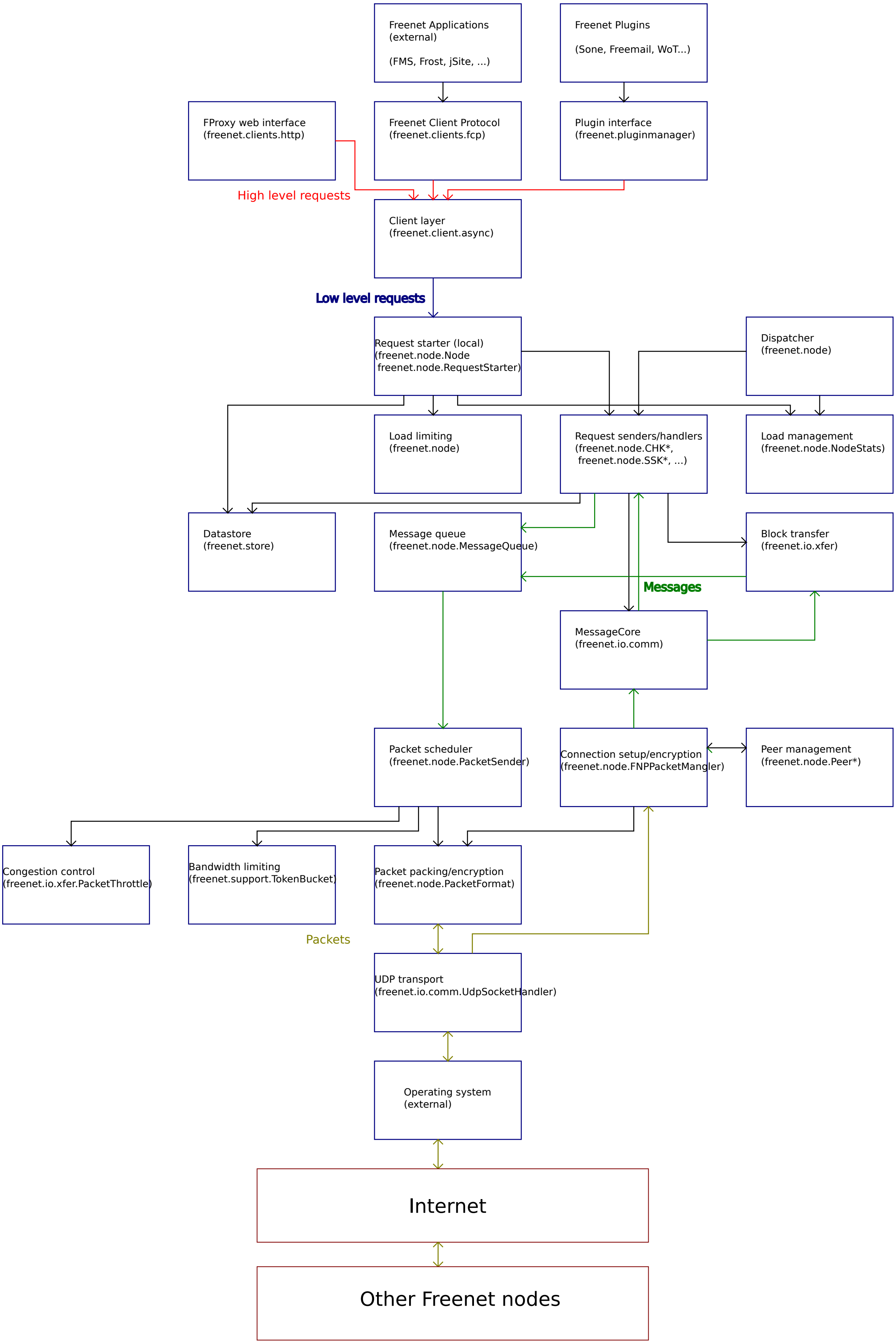
Freenet Applications (external)
(FMS, Frost, jSite, ...)
Freenet Plugins
(Sone, Freemail, WoT...)
FProxy web interface (freenet.clients.http)
Freenet Client Protocol (freenet.clients.fcp)
Plugin interface (freenet.pluginmanager)
High level requests
Client layer (freenet.client.async)
Low level requests
Request starter (local) (freenet.node.Node freenet.node.RequestStarter)
Dispatcher (freenet.node)
Load limiting (freenet.node)
Request senders/handlers (freenet.node.CHK*, freenet.node.SSK*, ...)
Load management (freenet.node.NodeStats)
Datastore (freenet.store)
Message queue (freenet.node.MessageQueue)
Block transfer (freenet.io.xfer)
Messages
MessageCore (freenet.io.comm)
Packet scheduler (freenet.node.PacketSender)
Connection setup/encryption (freenet.node.FNPPacketMangler)
Peer management (freenet.node.Peer*)
Congestion control (freenet.io.xfer.PacketThrottle)
Bandwidth limiting (freenet.support.TokenBucket)
Packet packing/encryption (freenet.node.PacketFormat)
Packets
UDP transport (freenet.io.comm.UdpSocketHandler)
Operating system (external)
Internet
Other Freenet nodes

Figure A.2: Diagram of Freenet layers relevant to this project