Writing a Message Broker in GoLang

George Vanburgh

Supervised by Alvaro A.A. Fernandes

May 3, 2016

The University of Manchester

A report submitted in part fulfilment of the degree of
BSc (Hons) in Computer Science with Industrial Experience

Abstract

This report details an exercise in learning and evaluating the 'Go' programming language, by building a feature-rich application message broker using modern software engineering techniques and best practises. Some of the unique features of Go are presented, and the importance and relevance of these features in building simple, high-performance distributed software is explored. Discussion of the novel 'CSP-style' concurrency features in Go lead to an unorthodox design proposal for an enterprise-style messaging broker - George's Asynchronous Message Broker (gamq). The implementation of gamq is then is then described and critiqued. Finally, the process and results of the software testing are presented, and analysed.

Acknowledgements

Many thanks to my supervisor, Alvaro Fernandes, for his continuous support, guidance and patience - without which, this project would have been impossible. Thanks also go to my family and friends for their help and encouragement both during this project, and throughout my time at University.

Contents

Chapter 1

Introduction

1.1 Background

Many large, distributed applications (particularly in 'enterprise' environments) make extensive use of messaging brokers - software (or hardware[18]) responsible for passing layer-7[1] messages between services, over the network. Two examples of such software are TibcoEMS and RabbitMQ. Brokers improve the software development experience for engineers working on distributed applications, by abstracting away complexities involved with passing messages between software running on distributed, heterogeneous hardware - which may be separated by networks operated by multiple different administrators, and with suboptimal qualities[7]. These broker benefits are explored in-depth in Section 2.2.

GoLang is a modern, cross-platform, open-source programming language initially developed by Google, and designed primarily for systems programming, with a focus on writing performant, network-oriented software, such as service cluster managers, web-servers and monitoring software (such as Grafana[2]). More information on GoLang can be found in Section 2.4

1.2 Goals

This project centres around building an enterprise-style messaging broker, as an exercise in designing network-oriented distributed applications using a modern programming language (GoLang), whilst following software design, test and architecture best practises - such as unit-testing, Continuous Integration (CI), and built-in monitoring.

In broad strokes, my project has the following goals:

- Learn and evaluate the impact and usefulness of GoLang's fairly unique set of concurrency primitives in constructing and testing high performance software.
- Design and build a well-tested, stable message broker in GoLang.
- Evaluate the performance and feature-set of my implementation.

[1]`https://support.microsoft.com/en-us/kb/103884`

[2]Which is used to provide monitoring and dashboard-ing for this project - see Section 4.2

Chapter 2

Background

2.1 Messages

In today's interconnected world, computer programs rarely exist in a vacuum. Rather, they typically form one small part of a much larger Service Oriented Architecture (SoA) - consisting of multiple services, jobs and scripts, all exchanging information in the form of messages. These messages may adhere to a standard data-interchange format (for example, JavaScript Object Notation (JSON) or Extensible Markup Language (XML)). They may correspond to an agreed upon specification (for example, IEEE 1671-2010[11]). Or they may simply be blobs of binary information transmitted over the network - their meaning completely subject to the interpretation of the sender and receiver. At a fundamental level, though, a message is simply a collection of bytes, to be transmitted from point A, to point B - in order, and without data loss. By not enforcing a fixed schema, and treating messages as simply a collection of bytes - message brokers allow any network connected application to take advantage of the benefits brokered message passing provides (Section 2.2).

2.2 Message Brokers

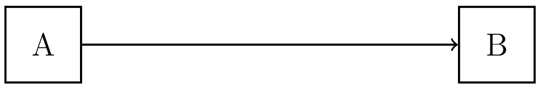


Figure 2.1: Two services, A and B, directly exchanging messages

To understand the role message brokers typically play in SoAs, we first examine the simplest method of sending messages between two applications - directly transmitting messages between two applications (shown in Figure 2.1).

In this example, the application 'A' wishes to transmit a simple message (Section 2.1) to application 'B', and does so in the simplest method possible. This could involve making an Remote Procedure Call (RPC), opening a Transmission Control Protocol (TCP) socket, or making a Hypertext Transfer Protocol (HTTP) web request. For the purposes of this illustration the exact mechanism by which bytes are transferred is unimportant, the fact that the transfer takes place *directly* between the two parties is all that matters.

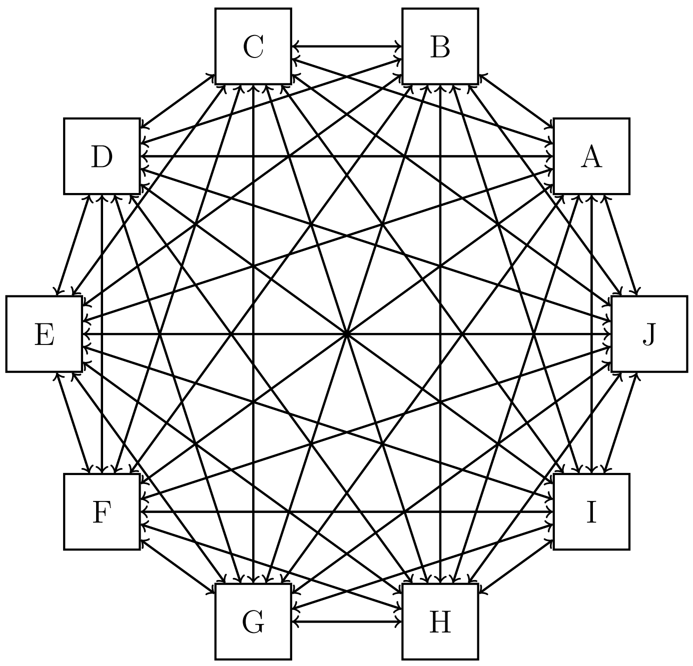


Figure 2.2: Ten services directly exchanging messages

This is a perfectly acceptable method of exchanging information between two services at a small scale. However, systems rarely exist in pairs. One of the biggest issues with simple application-to-application messaging is demonstrated in Figure 2.2 - as more nodes are added, the complexity of using direct connections increases exponentially (requiring $n^2 - n$ connections, where n is the number of services)[1]. Managing this complexity is difficult in a number of ways:

Service discovery

Firstly, each program requires some mechanism of discovering an endpoint for all the other services it wishes to communicate with. Typically, this is either provided via configuration shipped with the program, or is available from some central repository[2]. There are often problems involved with keeping this information up to date - as the services with which a client machine communicates may not always be available on the same endpoints all of the time (Possibly due to dynamic network configuration via DHCP, or the migration of a service between servers).

Number of connections

As the number of services a program communicates with increases, so do the number of connections said program is required to maintain. This could be a non-issue if a connectionless protocol such as User Datagram Protocol (UDP) is used (which has the distinct disadvantage of not having any deliverability guarantees). However, if a connection-oriented protocol such as TCP is used, there can be significant overhead associated with maintaining large numbers of connections[3]

[1]Assuming each service needs to talk to all other services

[2]Such as Apache Zookeeper

[3]Although this is less of a problem with modern hardware: `http://c10m.robertgraham.com/p/manifesto.html`

Readiness to receive

When a service sends a message across the network, the assumption is that the recipient of the message is ready to receive it. This is not always the case, however. The recipient may be performing some other task when the message is sent - or may even be busy processing the previous message it received. There are several programming techniques which can be employed to reduce the possibility of lost messages - relying on built-in congestion-control protocols, such as those present in TCP[14], and writing multithreaded or asynchronous (Section 2.4.1) code. However, more often than not, application developers end up having to write code to handle communicating with services that are (for whatever reason) unavailable.

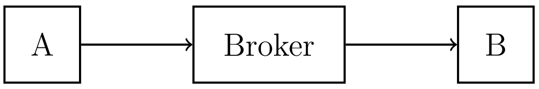


Figure 2.3: Two services exchanging messages via a broker

Message brokers offer a layer of abstraction to developers wishing to exchange messages between services. Rather than contacting the service directly, messages are relayed via the broker (Figure 2.3). This not only reduces the number of connections required for application to exchange messages (Figure 2.4), but also simplifies service discovery. Each service need only know the connection details for the message broker, and the name of the queue/topic (Sections 2.2.2 and 2.2.3) to send/receive messages on. The locations of the services sending/receiving messages through the broker no long matters - which removes the need for complex service discovery protocols. Additionally, by decoupling the sending and receiving of messages, brokers can act as a buffer between communicating processes, preventing the receiving process from becoming swamped by incoming messages (Denial of Service (DoS)), or losing messages in the event of an application crash - in both cases, messages simply buffer up on the message broker, until the receiving application is ready to consume them.

2.2.1 Pub/Sub

Publish-subscribe is a software pattern, which describes the exchange of messages between one-or-more producers (publishers), and one-or-more recipients (subscribers). In a typical implementation, publishers send messages (Section 2.1) to a broker - with a named destination (either a queue or topic), as shown in Figure 2.5. Subscribers register their interest ('subscribe') to these named destinations, and receive all messages sent to said destinations, in a manner dependent on whether the destination is a queue (see Section 2.2.2) or a topic (see Section 2.2.3). This pattern has several advantages due to its simplicity - pub/sub infrastructure can be easily reasoned about, and scaled to handle message routing for an entire datacenter, with relatively little complexity. These advantages are explored in detail in Section 2.2. The disadvantages of the pub/sub model are associated with its loose coupling - the model itself does not define a message format, leaving the creating and updating of message contracts to the application developer.[4].

[4]Although, as discussed in Section 2.1, this can also be an advantage, as it allows absolute freedom for application developers to freely define the format of their messages

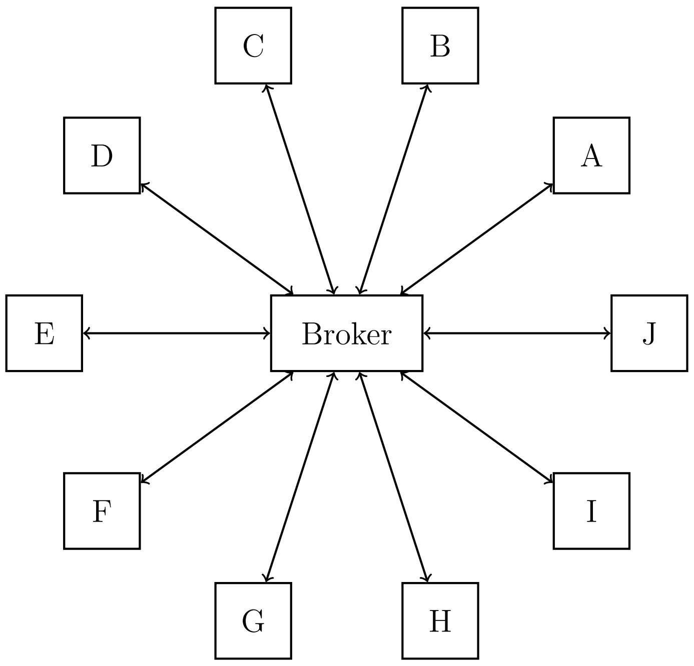


Figure 2.4: Ten services exchanging messages via a broker

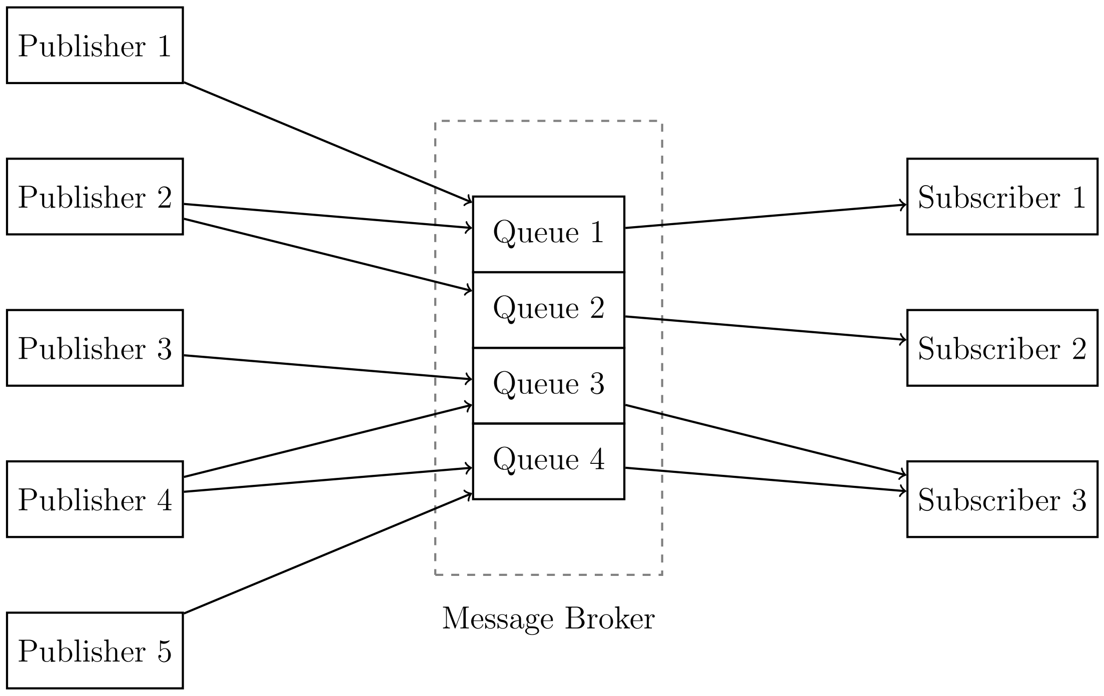


Figure 2.5: An example of several publishers and subscribers, interacting via a message broker

2.2.2 Queues

Queues are a standard feature of most programming languages and a core Computer Science construct[5]. Queues are a First-In-First-Out (FIFO) data-structure, in which messages are retrieved in the same order as they were placed into the queue. The other important features of queues (in terms of the pub/sub model), is that they can have multiple publishers, and multiple subscribers (Figure 2.5). In the event of a queue having multiple subscribers, each message is delivered to a single subscriber. The order in which messages are distributed amongst subscribers is typically a 'round-robin' pattern[6] - that is to say, the next subscriber to receive a message will be the one who is ready to receive a message, that has received a message *least recently.*

A typical use of a Message Queue may be the distribution of job information, where messages consist of configuration for jobs, and worker applications subscribe to the queue this job information is posted on. Say an application developer has a large set of images he/she wishes to crop to a certain size. These images are posted onto a message queue, which a number of servers subscribe to. Each server receives a image off the queue, crops it, and saves it to a fileserver - before receiving the next image. Each image is only ever received by a single worker. This is a scalable solution, as the rate at which images are processed increases with the number of servers subscribed to the queue (up to a point). In the event that a worker fails mid-way through processing an image, the broker may decide to redeliver the message to another subscriber, depending on how it was configured (Section 2.3).

2.2.3 Topics

Similar to Queues (Section 2.2.2), topics are FIFO structures. The key difference between topics and queues, is that whist queues deliver each message to a single subscriber, topics deliver each message to **every** subscriber (Figure 2.6). Each subscriber therefore receives a copy of each message sent to the topic.

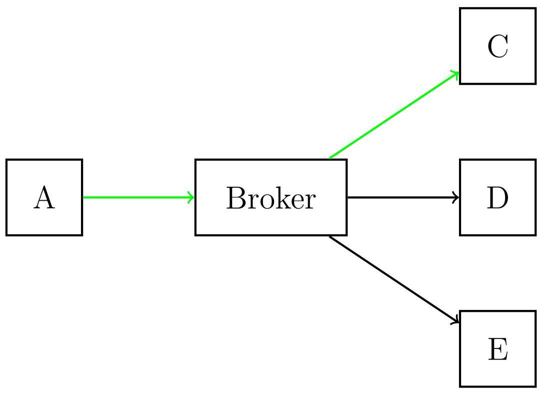
A common use of topics, is the updating of information in a distributed cache. Suppose a number of mobile phones maintain a cache of stock prices, and connect to pub/sub infrastructure to receive the latest prices. Obviously, a queue would be unsuitable for such a task, as pushing a new stock price onto a queue will only update the price for a **single** subscriber. A topic would ensure that each stock price published would be sent to every mobile phone - updating the price for all subscribers.

2.3 Broker Requirements

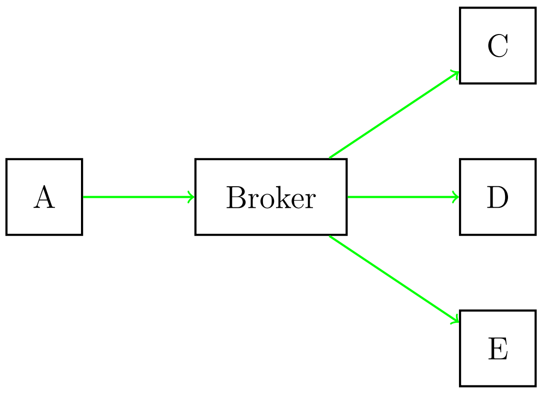
A message-broker is rather an unusual piece of software, due to the fact that is has very few functional requirements, they are simply required to deliver messages from point A, to point B. Nearly all of their requirements are non-functional, and fall into the following categories.

[5]As well as bring familiar to anyone who has ever been to a Post Office

[6]Although this is often configurable



(a) Queue delivery pattern with multiple subscribers



(b) Topic delivery pattern with multiple subscribers

Figure 2.6: Comparing queue and topic delivery patterns

2.3.1 Failure Handling

The issue with gracefully handling communications failure in distributed systems was illustrated in a 1988 paper by Xerox employees Andrew Birrell and Bruce Nelson[4]. They identified three different semantics with which RPCs could be executed:

Exactly once
: The ideal scenario is one in which messages are passed to their destination once, and exactly once. Typically, when failure does not occur during a message transfer this is trivial to assert - simply receiving an acknowledgement from the recipient of the message confirms this to be the case. Unfortunately, the same cannot be said when a response is not received from the recipient, for whatever reason (link failure/machine failure etc.). This is a typical illustration of the 'Two Generals Problem'[9] - and is extremely difficult to guard against. As a result - most messaging systems adopt one of the following behavioural models in the event of failure.

At most once
: In the event that a message is lost without acknowledgement - no attempt is made to redeliver the message. This is used in situations where duplicated messages pose a risk to overall system integrity - for example in most financial systems.

At least once
: In the event that a message is lost without acknowledgement, attempts to redeliver the message continue until successful receipt is acknowledged. This is typically used in situations where message delivery is deemed more important than message uniqueness/ordering. For example, if the unacknowledged message is intended to trigger a cache refresh in the recipient system - the fact that the refresh may occur multiple times may be insignificant next to the risk that the refresh does not happen at all.

Most brokers typically adhere to the *at least once* model when using TCP, and *at most once* when using UDP[7] - as these delivery models fit the characteristics of their respective delivery protocols.

2.3.2 Network Bandwidth Utilisation

In many situations (and in Internet of Things (IoT) devices in particular) network bandwidth is a limiting resource. Message compression can be used in certain situations to reduce overall bandwidth utilisation of the broker - at the cost of increased computational load on both the broker and consumer.

2.3.3 Network latency

Devices experiencing high degrees of network latency can have tremendous impact on the overall Round-trip time (RTT) of a given message. This can be compensated for through intelligent packet stuffing (such as Nagel's algorithm)/jumbo frames (if supported) by the network - all of which reduce the amount of overhead experienced by each message.

[7]Though again, this varies between implementations

2.3.4 Network packet loss

On networks experiencing a high rate of packet loss - error correcting protocols like TCP can help ensure packet delivery at the IP layer. Additionally, check-summing the messages can help guard against message corruption through lost packets.

2.3.5 Message power cost

Especially relevant for IoT and low-power devices - the amount of power consumed whilst exchanging messages (which is directly linked to the number of CPU cycles required to transmit/receive each message) can be an important consideration when designing an applications. Message attributes which can affect this include:

- Message compression
- Message size
- Internet Protocol (IP)
- Message protocol
- Network conditions (requiring re-transmits/computing checksums etc.)

2.3.6 Message throughput

Finally - the most obvious performance metric of a broker - the number of messages it can process per second. This is impacted by all of the above factors, and can be maximised through the use of small, cheap messages, with as little overhead as possible - and reducing the number of expensive operations performed when delivering messages (memory operations/network access' etc.). Network protocol can make a large difference here - with something like UDP being extremely cheap to send over the wire (if inherently unreliable) compared with TCP.

2.4 GoLang

As mentioned in the introduction (Section 1.1), GoLang is a compiled, statically-typed language developed in part out of the authors frustrations with using more traditional languages, such as Java and C++, to develop network applications[3]. Announced to the general public in 2009, GoLang was internally developed at Google to resolve the common criticisms with other popular languages at the time of development - a prime example being the long compilation times involved with working in languages such as C++[8]. One of the founding principals of Go was simplicity, and ease of maintainability[17] - even at the expense of performance in some areas[9]. As such, Go has a relatively light core feature-set compared with other '*network-oriented languages*', such as Java, .NET, Python and Ruby. Go also contains creature comforts not found in performance-oriented

[8]`http://stackoverflow.com/a/318440/1657432`

[9]Although GoLang does perform well in benchmarks, due to its compiled nature[2]

languages like C++, such as Garbage Collector (G.C.) - something which has typically been the sole preserve of languages with an interpreted runtime, such as Java, Ruby and Python. However, due to its nature as a compiled language, Go still outperforms these 'traditional languages' in a number of standard benchmarks[2] which, combined with the built-in '*CSP-style*'[10] concurrency features (Section 2.4.2) make Go an extremely compelling language for developing networked infrastructure software, such as a message broker.

2.4.1 Concurrency vs Parallelism

Confusion often exists as to the relationship between parallelism and concurrency, thanks in part to their often similar dictionary definitions:

Concurrent
: Going on side by side, as proceedings; occurring together, as events or circumstances;[15]

Parallelism
: Concurrent or contemporary; existing in the same period of time.[15]

However, as they relate to Computer Science, parallelism and concurrency are not at all the same. A concurrent task is one that has been designed to be independent of other tasks and which is interrupt-able. Concurrent tasks *may or may not* be run in parallel with other concurrent tasks, and have the potential to decrease the real-world-time taken to perform a set of actions, even if executed on single-threaded (non-parallel) hardware.

A real-world analogy - suppose the organisers of a draughts tournament wish to give 10 amateur players each the opportunity to play a game with a professional player, and wish to conduct these games in as time-efficient a manner as possible. There are multiple ways such a tournament could be run.

Sequentially

The professional player could sit down with each of the ten amateurs, and play him/her to completion - before moving on to the next amateur. This is synonymous with **sequential** code execution. Assuming each game takes 10 minutes to complete, and that the time taken to transition from one game to another is 6 seconds.

$$6 \text{ seconds} * 10 \approx 1 \text{ minute}$$

The time taken to complete the tournament is therefore 101 minutes (1 hour, 41 minutes). This is the worst case scenario.

Concurrently

An alternative method of running the tournament would be to get the professional to play all 10 games **concurrently**. In this scenario, she would play a move against one opponent, before immediately moving on to the next game - leaving the amateur player to make his move. Assume that the professional player takes 6 seconds to play each move and, as before, takes 6 seconds to transition between games. With 10 amateur players, it will take approximately 120 seconds before she returns to her starting seat. Now assume that each amateur player takes 45 seconds to take his/her turn. Based on the 10 minute timer from the serial game - each game will last for 11 rounds.

$$(10 * 60)/(45 + 6) \approx 11$$

The total time for the game is therefore

$$\begin{aligned}&(11 * \text{time taken per move}) + (11 * \text{transition time})\\ &\qquad = (11 * 51) + (11 * 60) \qquad (2.1)\\ &\qquad = 1221 \text{ seconds}\end{aligned}$$

This equates to 20 minutes, and 21 seconds - a dramatic improvement over the sequential method.

Parallel

Now we can look at the effect of simple **parallelism** on the tournament time. Assuming we return to the sequential tournament method, but invite a second professional player to the match. The effects of this are rather obviously - playing two games at once will half the time taken to complete all games, which will now take 50 minutes to carry out.

$$101/2 \approx 50$$

However, note that we not only require twice as many 'resources' (in this case, professional draughts players) as before, but took *longer* to complete the tournament than the concurrent method.

Concurrently Parallel

Introducing another professional player (as above) into a concurrent game again involves splitting the 10 amateur players into two groups of 5. Games are running in parallel across the two groups, but within the groups they are run in a concurrent manner.

Games in a concurrent group will complete in

$$\begin{aligned}&(11 * \text{time taken per move}) + (11 * \text{transition time across 5 players})\\ &\qquad = (11 * 51) + (11 * 30)\\ &\qquad = 600 + 330 = 930 \text{ seconds} \qquad (2.2)\end{aligned}$$

This equates to roughly 15 minutes 30 seconds, or the fastest time so far - and represents the approach GoLang takes to parallelism (concurrent functions, multiplexed across operating system threads - Section 2.4.2).

2.4.2 Concurrency in GoLang

GoLang has a rather unique set of concurrency features, which aim to make it simple to design concurrent functions, which can be multiplexed across multiple Operating System (OS) threads, to take advantage of hardware parallelism (with the benefits described in Section 2.4.1). The main two language features we'll be using are:

Goroutines Goroutines are, as mentioned above, concurrently executing functions - often described as 'lightweight threads' by the languages creators. Because they are managed by the Go runtime, Goroutines can be preempted by the Go runtime, as described in 2.4.3. This means that, Goroutines never block whilst waiting for IO - which is extremely important in networked environments (such as a messaging broker).

GoChannels Channels are a conduit for sending and receiving values, much like an internal queue (Section 2.2.2). These are typically (but not always) used for communicating safely between executing GoRoutines in an asynchronous manner. Suppose *goroutine1* wishes to communicate with *goroutine2* on system with a single OS thread. When *goroutine1* is scheduled for execution, it sends a message to the channel it shares with *goroutine2*. *goroutine1* then completes its current task, and is preempted by the scheduler. *goroutine2* has been asleep, waiting for a value on the channel - and is now placed on the scheduler queue, since it has a value to read from the channel. At some point in the future, *goroutine2* is scheduled for execution, and receives the value sent to it by *goroutine2*. This entire exchange is completely thread-safe, since no memory is shared between the two GoRoutines.

2.4.3 Why do we need concurrency?

A message broker operates in, what is an inherently parallel environment. As shown in Figure 2.5, a broker is expected to hold a great many 'conversations' with clients at any one moment in time. These clients will be sending new messages, expecting to receive published messages, creating/deleting queues and topics - all at the same time. As mentioned in Section 2.3, these client requests need to be serviced in a timely manner. Whilst a sequential approach to designing a broker may well work in simple environments, this approach simply does not scale to the levels of enterprise message brokers, which deal with hundreds of thousands of message per second, serving thousands of clients.

One approach to dealing with this environmental parallelism, is to use multi-threading. In a traditional language like Java, creating a separate thread for dealing with each client would improve the response times for individual clients, as well as allow the broker to take advantage of modern, multithreaded hardware. There exist, however, two big issues with this approach. In more traditional multi-threaded languages.

Memory footprint

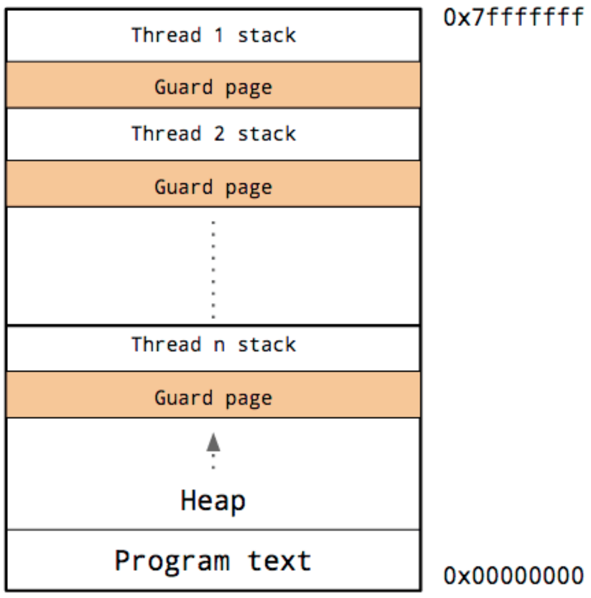


Figure 2.7: Traditional thread memory model [5]

Threads have a fairly large memory footprint. Java threads, for example, are allocated with roughly 512 KiB of stack memory. Each thread stack is bounded by a *guard page* - to prevent the stack from overwriting the heap, as shown in Figure 2.7, and which is typically 4 KiB in size. 512 KiB is a fairly large stack allocation, due to the fact that it is impossible to know at 'spawn time' how much stack memory a thread is going to use, and it is not possible to resize the stack at a later time. Assuming we were to use a single thread for each client, a broker with 1000 clients would consume roughly 512 MiB of memory *just* for the client threads.

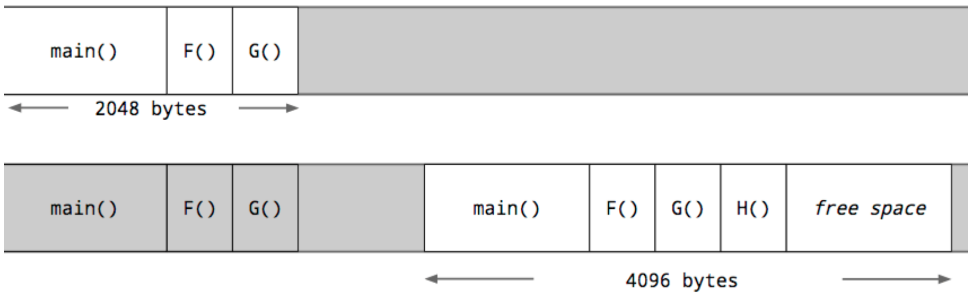


Figure 2.8: Goroutine memory model [5]

In place of a traditional multi-threaded approach, we can use a concurrent design in GoLang. A Goroutine starts life with a small stack, allocated from the heap memory (Figure 2.8). Rather than using a guard page to detect stack overflow, each time the function is called, a small piece of code tests that there is sufficient stack space for the function to run. In the event that there is insufficient space a new, bigger stack is allocated from the heap, copy the contents of the old stack over, and free the original stack memory. This resizing is possible because Go allocates stack space within the heap itself, rather than a separate chunk of memory. Because resizing is possible, Go can afford to allocate a much smaller initial stack for Goroutines, as the penalty for stack overflow is a (relatively mild) performance hit, rather than an application crash under the previous model (Figure 2.7). As of Go 1.4, the default starting size of a Goroutines stack is 2 KiB[10]. Using the same example from above, a 1000-client broker using Goroutines would require 2 MiB of memory, or a $\sim 250\%$ saving over the traditional thread model.

Context switching

Context switching occurs when a thread that is in the process of being executed, is replaced with a 'sleeping' thread by the operating system scheduler. If these two threads belong to separate processes, this can be a relatively expensive operation, as processes operate with entirely separate memory spaces. The operating system therefore needs to:

- Select the next thread to occupy the CPU from the queue of waiting threads, using a thread-scheduling algorithm (For example: FIFO, Least-Recently-Used (LRU))
- Store the current contents of the CPU registers, and restore the values used by the new process.
- (In most modern CPUs) flush the Translation Lookaside Buffer (TLB)[19]. This can impact the speed of subsequent memory accesses, as the cache of memory-address-to-page-mappings will be empty when the new thread starts running.

The time costs of context switching are hard to quantify, as they can vary wildly depending on the hardware platform, and OS. Performance numbers for modern hardware tend to be of the order of 10000's of switches per second[11].

Go moves the scheduling of Goroutines out of the operating system kernel, and into userspace. The Go runtime multiplexes Goroutines onto OS threads[12], and chooses which Goroutine to execute next independently of the OS. This allows the Go runtime scheduler to cooperatively schedule the switching of Goroutines at well-defined points in time. Some examples of this are[5]:

- Channel send and receive operations, if those operations would block (the channel is full).
- Blocking system calls (File and network operations).
- After being stopped for garbage collection.
- After adding a new Goroutine to the scheduler.

[10]`https://golang.org/doc/go1.4#runtime`

[11]Though again, this depends heavily on the memory access patterns of the running program

[12]The number of which can be set with the *GOMAXPROCS* environment variable

These can be broadly categorised as locations where a currently running Goroutine cannot continue its work until it has more data, or more space to put data. The fact that the scheduler behaviour is both deterministic, and context-aware means that it can make informed decisions about the best time to interrupt a currently running thread, whereas the system scheduler will preempt a running thread at any time (which might be midway through a calculation). This also means that Go can intelligently choose which registers are in use by the running Goroutine, and need to be saved as part of the context-switch, and achieve faster context switches as a result (by comparison, the OS scheduler will typically blindly save all registers during a context switch).

Chapter 3

Design

3.1 Project philosophy

One of the learning objectives for this project, was to discover and utilise Free and open-source software (FOSS) technologies wherever possible, as well as give back to the FOSS community. As such, all of the tooling used in the production of the project, as well as the project itself were open-source. The project itself is published under the MIT License, which allows anyone to "use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software", without restriction.

3.2 Presentation interface

As a (primarily) middleware project - coming up with a compelling presentation interface is crucial to allowing people to engage with the software, as well as providing the ability to monitor the broker, should it be deployed in a production environment. Broker metrics such as:

- Messages/second (broken down by topic)
- (Average) End-to-end latency for each message
- Memory/Disk usage
- CPU utilisation
- Pending messages
- Total messages processed

And client metrics such as:

- Messages sent
- Messages received
- Messages lost

Must be easily accessible, and visualised. Details on how this was implemented are given in Section 4.2.

3.3 Configuration

As with most pieces of software - there are situations where the default behaviour of a message broker requires adjustment to suit its operating environment. One example of this is the network port that the broker operates on, and which other applications will connect to in order to exchange messages. Whilst the default port (48879[1] in the case of this broker) may be suitable in most cases, since no other 'well-known' applications use that particular port[2] - there may be certain environments where it is necessary to change it. This can be achieved by editing the relevant gamq configuration.

3.3.1 Command-line configuration

A number of these configurable parameters make sense to expose as command-line parameters, specified at application start time. The available parameters can be exposed using the `--help` parameter, the output of which can be seen in Listing 1

```
$ ./gamq --help
Usage of ./gamq:
 -port int
     The port to listen on (default 48879)
 -profile
     Produce a pprof file
 -statsd string
     The StatsD endpoint to send metrics to
```

Listing 1: Output of running the broker with the –help flag

3.3.2 File-based configuration

Whilst specifying arguments on the command line gives flexibility, there are certain options that, whilst configurable, are either too numerous, or change too infrequently to justify command-line flags. One major example of this is the log configuration for the broker - an XML file specifying specifying the format, and location of messages logged using the Seelog library. An example log configuration can be seen in Listing 2, which defines the format and location of log messages written by gamq.

[1]A number chosen due to its memorable hexadecimal representation: `0xBEEF`

[2]`https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers`

```
<seelog minlevel="debug" type="asynctimer" asyncinterval="1000">
  <outputs formatid="common">
    <console/>
  </outputs>
  <formats>
    <format id="common"
      format="%Date(02-01-2006 15:04:05.000-0700) [%LEVEL] %Msg%n"/>
  </formats>
</seelog>
```

Listing 2: Example Seelog configuration file for gamq.

3.4 Code Structure

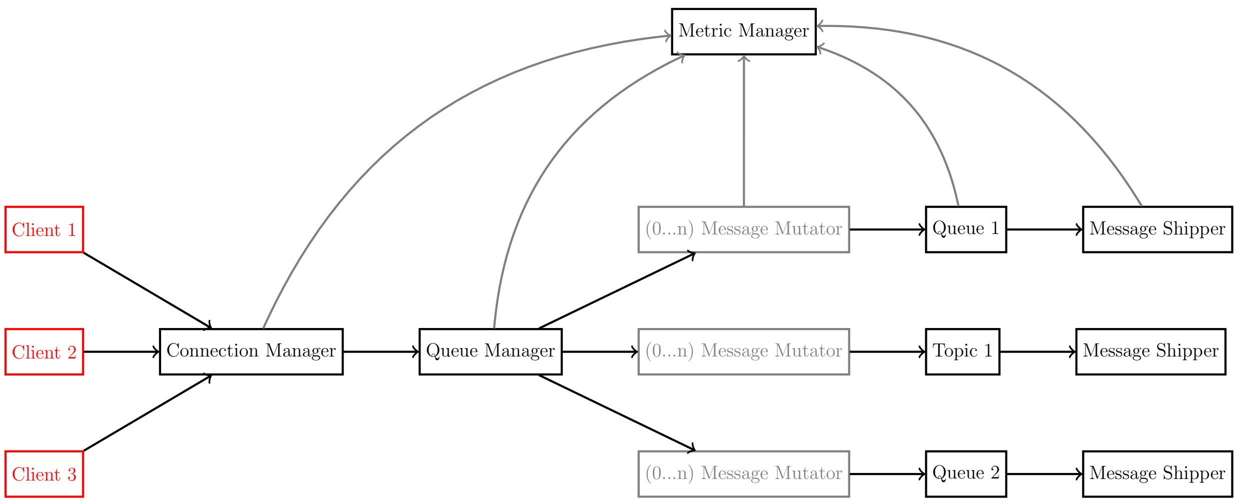


Figure 3.1: A block diagram of the system components, and their interactions

An overview of the project structure can be seen in Figure 3.1. The communications between each system block (shown in black) are handled via GoChannels (Section 2.4.2). The responsibilities of each block are as follows:

Connection Manager
: Handle incoming and open connections for each of the broker protocols, and maintain details of connected clients. The protocol used for client connections is abstracted inside ConnectionManager, so that a client can transparently use TCP, UDP - or some other protocol (such as HTTP, or Advanced Message Queuing Protocol (AMQP)). Connection Manager parses received commands (Section 3.5), and calls the relevant methods on QueueManager.

Queue Manager
: Maintains a list of active queues/topics, and handles the publishing of messages to said queues/topics, as well as the addition/removal of subscribers.

Message Mutators
: An optional module for each queue/topic, which allows simple mutations to be performed on messages as they pass through the broker. One example could be throwing away values below/above a certain threshold, or enriching data as it passes through the data, with information from an external database. Multiple message mutators can be chained together, should multiple mutations be required.

Queue/Topics
: The buffering datastructures designed to hold and deliver messages to consumers. Due to the semantic differences between queues and topics (Sections 2.2.2 and 2.2.3), subtly different datastructures are required for each - details of which are given in Section 4.1.2.

Message Shippers
: Responsible for managing the delivery of messages to subscribers.

Metric Manager
: Responsible for collecting metrics about broker performance from each of the broker components (latency, throughout, number of connected clients etc.), and asynchronously ship them to 4.2.1 for visualisation and analysis.

The implementation details for each of these components are explored in (Section 4).

3.5 Protocol Design

As a piece of network-connected software - a clear, well-defined protocol is essential to allow other networked applications to interact with the messaging broker. Although a simple protocol is used in the current implementation, the message decoding logic is designed to be somewhat modular, allowing for additional protocols to easily be supported. The NATS message protocol[1] served as an inspiration for gamq - as it showed it was possible for text-based protocols to be as performant as customised binary protocols, and had a slew of other benefits, such as human-readability, and ease of debugging.

The gamq protocol defines several commands, which can be seen in Table 3.1. These commands are all delimited by `\r\n`, which is the telnet standard. The two most frequently used commands (*PUB* and *SUB*) are intentionally short to reduce the overhead required to send them over the wire - which is especially important as message rate increases. Note that whilst all commands listed in the following sections are demonstrated using telnet, in reality they would be given pragmatically, by software wishing to send messages to the broker (An example of which can be seen in F)

Table 3.1: The gamq protocol commands

Command
`HELP\r\n`
`PUB <queue>\r\n <message>\r\n.\r\n`
`SUB <queue>\r\n`
`PINGREQ\r\n`
`DISCONNECT\r\n`
`SETACK <on/off>\r\n`

3.5.1 HELP

```
[mbax2gv2@kilburn ~]$ telnet E-C07KI1839.it.manchester.ac.uk 48879
Trying 130.88.194.143...
Connected to E-C07KI1839.it.manchester.ac.uk.
Escape character is '^]'.

help
Available commands:
        HELP: Prints this text
        PUB <queue> <message>: Publish <message> to <queue>
        SUB <queue>: Subscribe to messages on <queue>
        PINGREQ: Requests a PINGRESP from the server
        DISCONNECT: Disconnect from the server
        SETACK <on/off>: Enable/Disable message acknowledgement
```

Listing 3: Output from the gamq HELP command

Print the help message (as seen in Listing 3)

3.5.2 PUB

```
[mbax2gv2@kilburn ~]$ telnet E-C07KI1839.it.manchester.ac.uk 48879
Trying 130.88.194.143...
Connected to E-C07KI1839.it.manchester.ac.uk.
Escape character is '^]'.

pub abc 123
this is my message
it can span multiple lines
.
```

Listing 4: Publishing a message to gamq

Send the given message, to the given queue. In Listing 4, we are publishing a message to the queue 'abc'. Note that messages can span multiple lines, and are terminated with a single '.', on a line by itself.

3.5.3 SUB

```
[mbax2gv2@kilburn ~]$ telnet E-C07KI1839.it.manchester.ac.uk 48879
Trying 130.88.194.143...
Connected to E-C07KI1839.it.manchester.ac.uk.
Escape character is '^]'.

sub abc
this is my message
it can span multiple lines
```

Listing 5: Subscribing to a queue/topic in gamq

Subscribe to messages posted on the named queue (see Section 2.2.1). Note that in Listing 5, when we subscribe to the queue 'abc', we receive the message sent to that queue in Section 3.5.2.

3.5.4 PINGREQ

```
[mbax2gv2@kilburn ~]$ telnet E-C07KI1839.it.manchester.ac.uk 48879
Trying 130.88.194.143...
Connected to E-C07KI1839.it.manchester.ac.uk.
Escape character is '^]'.

pingreq
PINGRESP
```

Listing 6: Pinging the broker through telnet

Ask the broker to return a 'PINGRESP' message. This can be used to check that the connection to the broker is active, and the broker is responding to messages. Clients should send this at regular intervals.

3.5.5 DISCONNECT

```
[mbax2gv2@kilburn ~]$ telnet E-C07KI1839.it.manchester.ac.uk 48879
Trying 130.88.194.143...
Connected to E-C07KI1839.it.manchester.ac.uk.
Escape character is '^]'.

disconnect
telnet> Connection closed.
[mbax2gv2@kilburn ~]$
```

Listing 7: Closing a connection to the broker

Unsubscribe from all subscribed queues, and close the connection to the broker. Whilst the client can achieve the same effect by simply closing its side of a TCP connection, since UDP is a connectionless protocol, it requires an explicit disconnect message (as in Listing 7)

3.5.6 SETACK

```
[mbax2gv2@kilburn ~]$ telnet E-C07KI1839.it.manchester.ac.uk 48879
Trying 130.88.194.143...
Connected to E-C07KI1839.it.manchester.ac.uk.
Escape character is '^]'.

pub abc
this message will not be acknowledged
.

setack on

this message will be acknowledged
.
PUBACK
```

Listing 8: Demonstrating the effect of the SUBACK command

Turn the explicit acknowledgement of messages on/off (As demonstrated in Listing 8). Clients can use the acknowledgement to detect potentially lost messages using a timeout, and optionally retransmit the lost message if at-least-once semantics (Section 2.3.1) are required. Explicitly acknowledging messages (as opposed to relying on built in TCP error correction) can have a significant overhead, but can also dramatically reduce the number of lost messages when using a protocol with no built-in error correction (UDP).

Chapter 4

Implementation

4.1 Broker Implementation

A system-block diagram, showing the interactions between the various gamq submodules, can be seen in Figure 3.1.

4.1.1 Connection Manager

As mentioned in Section 3.4, the Connection Manager has two main functions, accepting incoming connections, and parsing incoming arguments in compliance with the protocol (Section 3.5).

The acceptance of incoming connections requires the specifics of each protocol (TCP, UDP, HTTP etc.) to be abstracted away using a standard interface through which messages can be sent to clients, the 'writer' interface in this case. An example of the UDP writer interface can be seen in Appendix C. Once connected, each client is serviced by a separate GoRoutine in an *Event Loop*-style pattern - the main Connection Manager thread performs the minimal amount of work necessary before farming a client out to a background GoRoutine, in order to prevent one connecting client from blocking another.

The parsing of each client message is simply a matter of tokenisation and pattern matching, as seen in Appendix D. The biggest performance gains to be exploited in a message broker are in ensuring efficient serialisation and serialisation of messages - so this was a focus of the protocol parsing logic, as explored in Section 6.1.2.

4.1.2 Message Storage

In the event that, for whatever reason, a client is not ready to accept a message when it arrives - this message must be buffered up in a datastructure of some kind, until delivery is possible. There are a number of different datastructures that could be used - each with their own pros and cons.

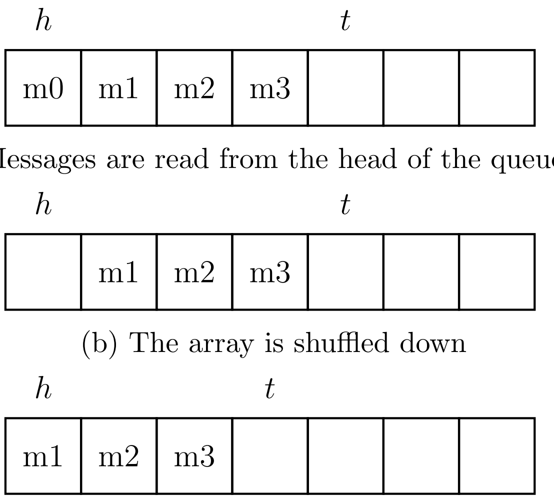
As mentioned in Section 2.2.1, message brokers typically support a number of different delivery patterns, the two most common of which are Queues (Section 2.2.2) and Topics (Section 2.2.3). The backing datastructures used for each are explored below.

Queues require that messages are delivered in the order they are pushed onto the queue, and that each message is delivered to a single consumer - so any backing datastructure must support this. Queues typically support two main operations:

enqueue(m)
The message m is placed onto the queue.

dequeue()
Returns the message at the front of the queue (optionally blocks whilst the queue is empty)

The simplest method of representing a queue of messages, would be an array. New messages are inserted into the first unoccupied slot in the array, and messages are read front-to-back, in a FIFO fashion.



(a) Messages are read from the head of the queue (h)

(b) The array is shuffled down

(c) The next message is now ready to be read

Figure 4.1: Reading a message from an array-based queue

However, there are several disadvantages of this approach, for both the enqueue and dequeue operations.

enqueue(m)
Arrays in GoLang have a fixed length - but there is no limit on the number of messages a queue may be asked to buffer. We can compensate for this by using 'slices'[1] instead of arrays - which are expandable using the `append()` command (Listing 9). The message m is stored into the first available array index, which will typically complete in $O(1)$ time. However, in order to create the illusion of an 'expandable' array, slices use a fixed-length array under the hood - which is expanded when full. This expansion involves creating a new array, which is twice the size of the original. Each message in the underlying array is then copied to the new array, which is used for all slice actions going forward. Therefore, whilst the

[1]`https://blog.golang.org/go-slices-usage-and-internals`

vast majority of messages will be stored in $O(1)$ time, a message that triggers an expansion of the underlying array will require $O(n)$ time to store (where n is the number of messages in the queue).

dequeue()

Pulling the first message in the array requires that the array be shuffled after each `dequeue()` operation, so that the next message to be delivered always resides in array position 0 (see in Figure 4.1). An array implementation would be extremely inefficient, requiring $n-1$ items to moved down an array position. However, because a slice is simply a 'view' onto an array, this operation can be made extremely efficient - by simply creating a new slice, with all but the first element in common with the original. It is therefore possible to complete this operation in $O(1)$ time.

```go
func main() {
        var test = make([]int, 1) // Create an integer slice, of length 1
        fmt.Println(test)         // Printing the array yeilds: [0]
        test = append(test, 1)    // Append the number '1' to the array
        fmt.Println(test)         // Printing the array yeilds: [0 1]
}
```

Listing 9: An example of appending to a GoLang slice

In addition to the disadvantages given above, slices are quite inefficient when it comes to memory utilisation. At any moment in time, up to 50% of the underlying array will be empty. Whilst these array indexes do not contain messages, they still occupy space in memory, meaning that (in the worst case scenario) $O(2n)$ space is required to store n messages. This is not ideal.

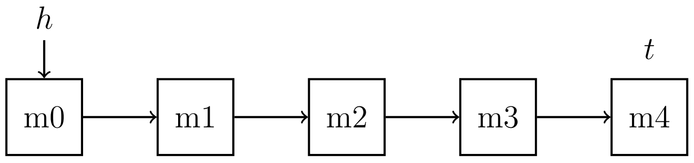
An alternative method of representing an in-memory-queue, is a linked list (Figure 4.2). This has numerous advantages over an array - with both enqueue and dequeue operations completing in $O(1)$ time. Linked lists also improve on the space complexity of the queue, as the datastructure grows and shrinks in response to the number of messages - meaning that we require $O(n)$ memory to store n messages.

enqueue(m)

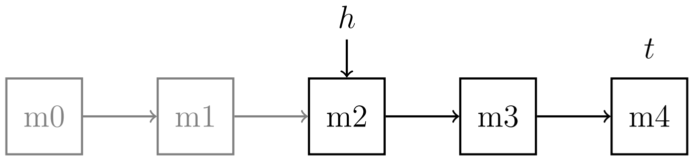
The item at the tail of the queue (t) - seen in Figure 4.2a - is simply updated with a pointer to m. This is an $O(1)$ operation, and simple to perform.

dequeue()

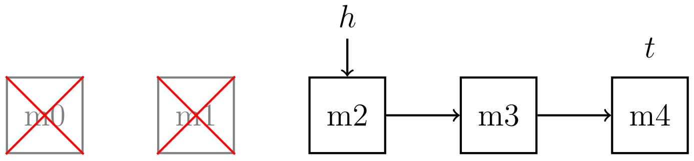
The item at the head of the queue (h) is returned. The head pointer is then updated to point to the next message to be read, as in Figure 4.2b.



(a) Messages are read from the head of the queue (h)



(b) Head moves, and read nodes are marked for garbage collection



(c) At some point in the future (depending on memory pressure), the G.C. will automatically clean up the read messages

Figure 4.2: Reading a message from an array-based queue

A full implementation of a linked list queue can be found in Appendix B.

When it comes to representing topics, the situation is similar. As mentioned in Section 2.2.3, the main difference between queues and topics, is the fact that queues deliver each message to a single consumer, whereas topics deliver each message to *all* consumers. This requires a subtly different datastructure to hold the messages - as consumers may read messages at different rates. If we use an array-based datastructure, similar to that in Figure 4.1, we would be required to maintain a separate queue for each subscriber to read from - as shown in Figure 4.3. A topic with s subscribers, and n pending messages would therefore require at *least* $O(n * s)$ memory.

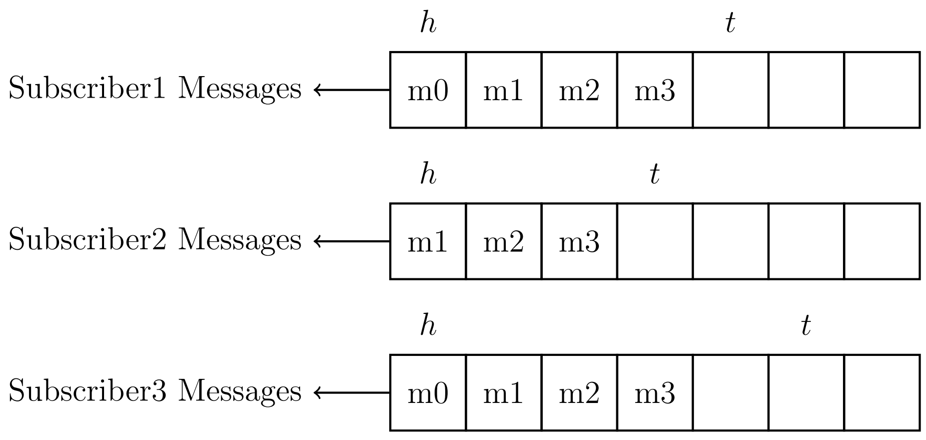


Figure 4.3: An array implementation of a topic

A linked-list implementation, however, has the potential to be highly efficient when it comes to storage space. Rather than storing s messages, we can simply keep multiple 'head' pointers on the same linked-list, one for each subscriber. When a subscriber consumes a message - only the head pointer belonging to it is moved up the list, the rest remain static (Figure 4.4). Similar to the queue implementation, the G.C. will dispose of messages once all consumers have read them (all of the 'subscriber' pointers have passed that message in the chain - Figure 4.2). This method only requires $O(s + m)$ memory, for a topic with s subscribers, and m messages. The linked-list implementation of a topic therefore has an even bigger efficiency saving over the corresponding array-based implementation, than in the queue examples given above, as seen in Figure 4.5.

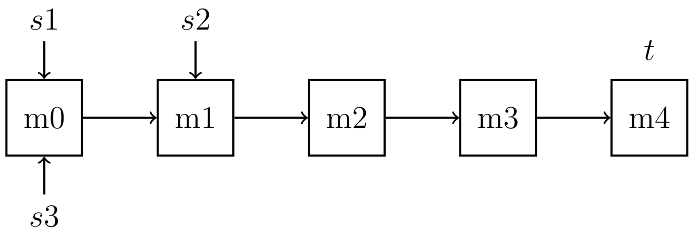


Figure 4.4: A linked-list implementation of a topic

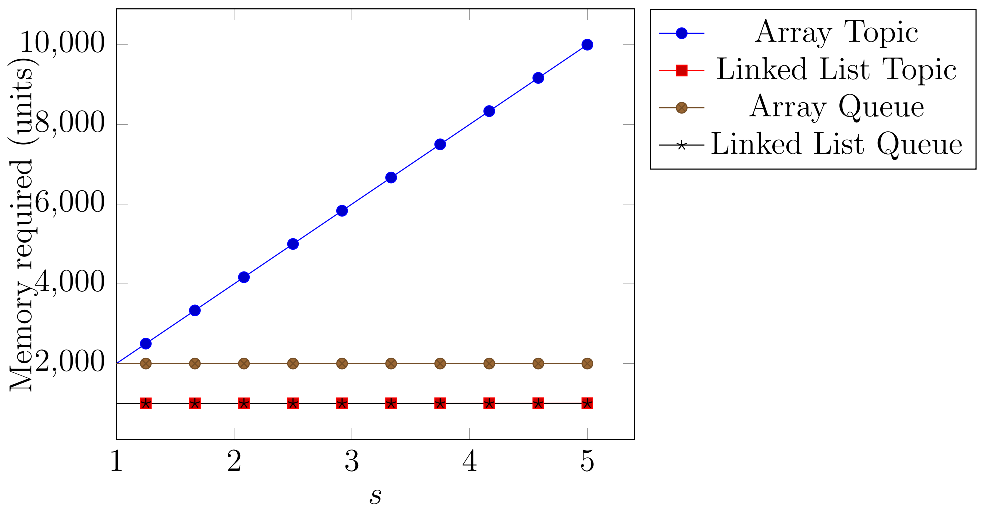


Figure 4.5: A comparison of the space complexities of various queue/topic implementations, as the number of subscribers (s) increases

4.1.3 Message Shipper

The Message Shipper is responsible for pulling messages off an internal gamq queue/topic (Section 4.1.2), and sending them to consumers. Thanks to the blocking characteristic of GoChannels (Section 2.4.2), this is as simple as listening to the GoChannel attached to the subscribers queue/topic, encoding messages as they are pulled off the queue and sending the bytes to the 'writer' object created by the Connection Manager (Section 4.1.1) - as seen in Appendix E.

4.1.4 Metrics Manager

As shown in Figure 3.1, all other blocks in the system produce performance metrics, to be formatted as StatsD (Section 4.2.1) messages, for graphing and visualisation. These metrics are aggregated by the Metrics Manager, before being sent to InfluxDB (Section 4.2.2). Whilst metrics are important, sending them should never impact the performance of the broker itself - so the metrics manager uses a '*buffered GoChannel*'[13]. Unlike a normal GoChannel, putting a message on a buffered channel will not block in the event that the GoRoutine on the other end is not currently listening[2]. Once the Metrics Manager has received a metric from the channel, it adds the value to a local buffer - which it flushes once a second, formatting all buffered metrics as lightweight StatsD UDP packets (Section 4.2.1) - which are then sent to the configured StatsD network address (Section 3.3).

4.2 Presentation Interface

The presentation interface consists of the following components.

4.2.1 StatsD

StatsD is an open-source statistics daemon, written by developers at Etsy in 2010[8], and which has seen widespread adoption by infrastructure teams ever since. The project aims to define a simple protocol (Listing 10) for sending statistics over the network, and to make it simple to aggregate and store those metrics for real-time analysis. StatsD can differentiate between metrics of different types:

```
metricname:1|c
```

Listing 10: A simple counter metric, exactly as it would appear inside a network packet

Counters A simple counter that can be incremented or decremented. An example use case could be the number of messages sent through a broker.

Timing The amount of time taken to complete a task. An example could be the amount of time taken to process a message.

Gauges An arbitrary value, stored until it is superseded by a new value. An example of this could be the number of connections open to the broker at a single point in time.

These packets can be sent over the network as either TCP or UDP packets. Packets can contain multiple metrics, separated with newline characters (Listing 11), which can greatly improve the efficiency of the transfer, as the size of each metric (the order of a few bytes) is far below the maximum capacity of a single packet - whilst the typical Maximum Transmission Unit (MTU) of Ethernet networks is 1500 bytes[12]. gamq uses the UDP protocol by default (although this is configurable, see Section 3.3 for details), as the cost of sending UDP packets is very low cost when compared to TCP, and we do not require

[2]Assuming the buffer is not full

the deliverability guarantees of TCP (metrics are non-critical). Metrics are typically sent over a Local Area Network (LAN), where packet loss is usually quite low. In the event that packet loss exists, but is relatively uniform, the loss of some metrics should not affect the overall trend of the data too adversely.

```
gorets:1|c\nglork:320|ms\ngaugor:333|g\nuniques:765|s
```

Listing 11: Multiple metrics in a single packet

4.2.2 InfluxDB

InfluxDB is an open-source, NoSQL database, optimised for the storage and retrieval of Time-Series Data. Influx is part of a family of databases designed to store (amongst other things) a large amount of metrics data at very high speed (the order of millions of events per second), which also includes Graphite and OpenTSDB. InfluxDB was chosen for this project due to its maturity (compared to OpenTSDB) and performance (compared to Graphite/Carbon).

4.2.3 Grafana

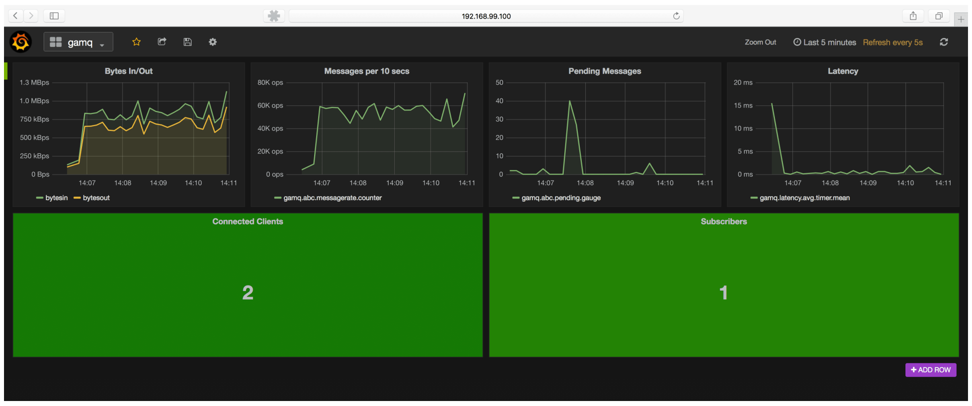


Figure 4.6: A screenshot of gamq metrics viewed in the Grafana visualisation tool

Grafana is an open-source visualisation tool for Time-Series Data, typically that which is produced by Internet infrastructure and application analytics. Developed by a consortium of Internet companies as a modern replacement for the previously popular Graphite-Web project. Grafana can visualise Time-Series Data from a variety of data-sources, such as the ElasticSearch, the aforementioned Graphite and, most importantly, InfluxDB.

4.2.4 Docker

As mentioned in Section 4.2.1, gamq is designed to send StatsD messages (Section 4.2.1) across the network. In a production environment, these would be sent to the above software stack, running on a separate server in the datacenter. In a development environment, the metrics stack could be run on a development server, or the developers local machine.

The configured monitoring stack should therefore be as portable as possible, to make development as simple as possible. Docker allows environments to be defined in code, and built into container images - which can be run in a manner similar to that of a lightweight virtual machine. The Docker container gamq sends metrics to is available on Github.

Chapter 5

Testing and Evaluation

5.1 Unit Tests

As mentioned in Section 2.4, Go was originally designed to be a modern alternative to C++. One of the major issues the original designers had with C++, was the reliance on community tools[1] - some of which were high-quality, and many of which were not. As a result, when developing GoLang, they wanted all the requisite tooling to be built into the language, and authored by the language creators themselves, to ensure a certain quality. As such, Go has support for unit tests out of the box, without the need to consume third party libraries (more on that in Section 5.1.1). Tests can be invoked using `go test ./...`[2], which will look for any files ending in *_test.go*, and run the tests they contain. Tests belong to the same package as the software under test - meaning they optionally have access to all private methods and variables, and can perform black-and-whitebox testing. Files ending in *_test.go* are excluded from standard compilation, however - so tests do not end up in the final compiled binary.

An example of a test in Go is shown in (Listing 12). A test function is simply one which begins with the word *Test*, and which takes in a test controller (*t* in the given example). The test then performs some actions (in this case, calling a method on the ConnectionManager), and optionally fails the test by calling the relevant method on the test controller (`t.Fail()`). Note that whilst the example given in Listing 12 is a unit test, integration tests are written using the same formula. The output from the full test suite can be seen in the appendix (Section G), as well as viewed online (Section 5.1.3).

5.1.1 Asynchronous Testing

Whilst the built in Go unit testing fulfils the needs of most projects, several third-party libraries have been written to improve the experience of testing asynchronous code (Section 2.4.1). Blocks of code which use GoRoutines/GoChannels (Section 2.4.2) will behave in a non-deterministic manner, which can make testing them challenging. For example, a block of code that receives a value sent by another GoRoutine, has no knowledge of when the value will be received - as the GoRoutine it receives the value from is non-deterministically scheduled by the runtime. Therefore, when testing asynchronous

[1]debuggers, profilers, test libraries etc.

[2]Runs all unit tests in the current project

```go
func TestConnectionManager_parseClientCommand_helpMessageReturns(t *testing.T) {
        underTest := ConnectionManager{}

        buf := new(bytes.Buffer)
        bufWriter := bufio.NewWriter(buf)
        mockClient := Client{Name: "Mock", Writer: bufWriter}

        var emptyMessage []byte

        underTest.parseClientCommand([]string{"HELP"}, &emptyMessage, &mockClient)

        if buf.String() == unrecognisedCommandText {
                t.Fail()
        }
}
```

Listing 12: Testing the gamq connectionmanager

code, an assertion that a particular parameter has been set correctly must take place after the thread that sets the value of said parameter has written it. A naive method of achieving this, would be to simply wait a certain length of time after starting the value-setting thread - and *hope* that the value is set when we come to make our assertion. An example of this strategy can be seen in Listing 13.

```go
// We'll inspect this value
var valueWillBeWrittenHere int

// Kick off the thread that sets the value
go methodThatSetsValue(*valueWillBeWrittenHere)

// Wait 5 seconds
time.Sleep(5 * time.Second)

// Assert the variable holds the expected value
if (valueWillBeWrittenHere != expectedValue) {
  t.Fail()
}
```

Listing 13: Sleeping to test asynchronous code

There are two troublesome scenarios this test creates, however. First, from an efficiency standpoint, tests like those in Listing 13 can lead to test suites that take un-necessarily long to run. On average, it may only take 0.5 seconds for `methodThatSetsValue()` to complete - meaning that this test wastes 4.5 seconds, holding up the execution of other tests whilst doing so. In a sufficiently large code-base, this can lead to test suits that take hours to run, and which are run less frequently as a result - which reduces development agility.

Second, and more seriously - the thread that sets `valueWillBeWrittenHere` may be scheduled to run after 5.5 seconds, by which time the test will have already been failed. This is known as a *false negative*, tests that report failure where none exists. These may not appear serious, however in large software projects they have a similar effect to 'crying wolf' - they can create a lack of faith in a test suite, which could potentially lead to the ignoring of a serious bug.

A much better method of asynchronous testing is provided by the *gomega* testing package - which provides methods to regularly poll the values of variables, terminating either after the pass criteria has been met, or a configurable timeout has been exceeded. Because the variable is polled at regular intervals (by default, every 10msec), the test will pass 10msec after the variable has been set to the correct value (in the worst case). An example of using Gomega to test asynchronous code, can be seen in Listing 14.

```
// Setup gomega
gomega.RegisterTestingT(t)

// We'll inspect this value
var valueWillBeWrittenHere int

// Kick off the thread that sets the value
go methodThatSetsValue(*valueWillBeWrittenHere)

// Poll the variable every POLLING_INTERVAL milliseconds
// Terminate either when valueWillBeWrittenHere == expectedValue (test passes),
// or after a TIMEOUT second timeout (test fails)
gomega.Eventually(func() int {
  return valueWillBeWrittenHere
}, TIMEOUT, POLLING_INTERVAL).Should(gomega.Equal(expectedValue))
```

Listing 14: Using gomega to test asynchronous code

5.1.2 Benchmarks

As well as ensuring the correctness of the code checked in, monitoring changes in performance for certain critical sections of code is also a concern, when building a system where speed is required. In order to do this, the built-in GoLang test framework's benchmark support was used[3] to record stable execution times for critical functions.

For example, Listing 15 shows a benchmark function to evaluate the execution time of the `Push()` function for the gamq MessageQueue implementation. When run as part of the CI build, the following output (Listing 16) is given for each benchmark. These results can then be compared across run, with the option of failing the build in the event that a major increase in average execution time for certain methods is detected, when compared to the previous build (i.e there has been a performance regression).

[3]Details about which can be found in the testing package documentation

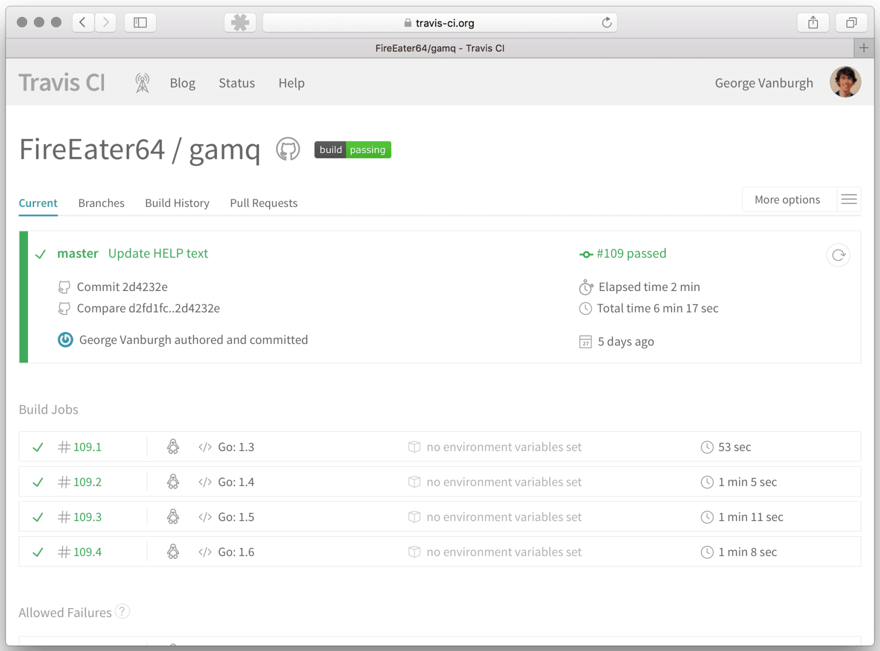
```
func BenchmarkMessageQueue_Push(t *testing.B) {
        underTest := NewMessageQueue()
        testString := "test"

        for i := 0; i < t.N; i++ {
                underTest.Push(&testString)
        }
}
```

Listing 15: An example of a benchmark in Go

```
BenchmarkMessageQueue_Push-2 # Benchmark Name
5000000 # Number of times loop executed to get a stable average
239 ns/op # Average execution time across all runs
```

Listing 16: Example benchmark output

5.1.3 TravisCI



FireEater64 / gamq build passing

master Update HELP text
Commit 2d4232e
Compare d2fd1fc..2d4232e
George Vanburgh authored and committed
#109 passed
Elapsed time 2 min
Total time 6 min 17 sec
5 days ago

Build Jobs

```
✓ #109.1   Go: 1.3   no environment variables set   53 sec
✓ #109.2   Go: 1.4   no environment variables set   1 min 5 sec
✓ #109.3   Go: 1.5   no environment variables set   1 min 11 sec
✓ #109.4   Go: 1.6   no environment variables set   1 min 8 sec
```

Allowed Failures

Figure 5.1: Travis.CI

Despite the initial plan being to set up and self-host most of the CI infrastructure, it became clear after an initial trial that it would be far easier for to ensure build repeatability/speed if a cloud-based CI service was used. The popular and excellent Travis CI was selected - which provides a free tier for open-source projects like gamq. Setup was as simple as adding a '.travis.yml' file to the project, and enabling the gamq GitHub repo for builds on the Travis CI admin panel. The initial .travis.yml file can be seen in Listing 17, and the latest version is available on GitHub. There are a multitude of configurable options available[4], but the initial configuration in Listing 17 consisted of only three:

```
language: go

go:
  - 1.3
  - 1.4
  - tip

script: go test -v ./... -bench=.
```

Listing 17: Initial .travis.yml

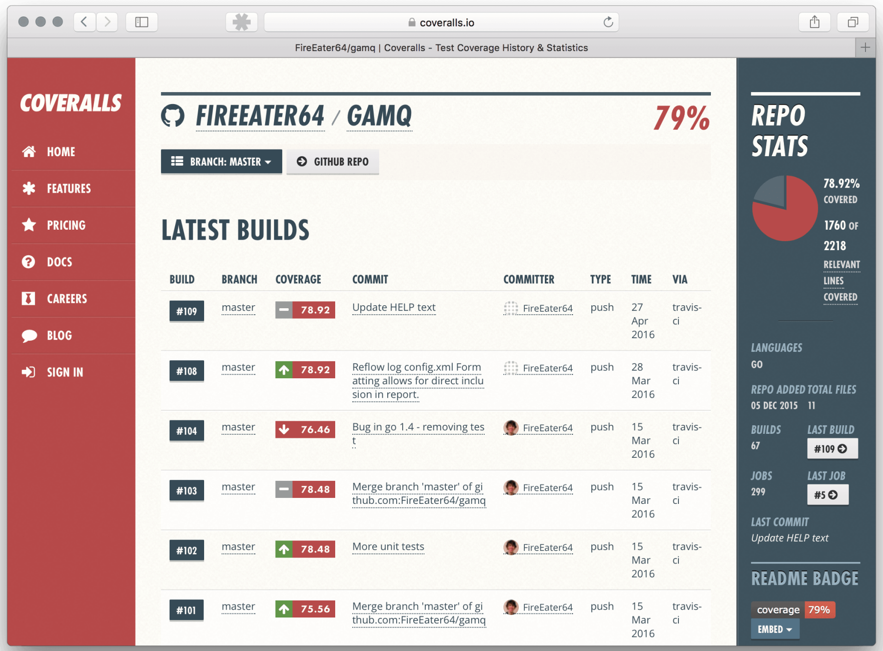
Language Defines the language of the project being built - in this case GoLang. TravisCI builds (usually) take place inside Docker containers, with the 'language' section of configuration dictating which pre-built container is used for this particular build. In this case, a language value of 'go' will ensure that the build container contains all of the binaries and environment variables required for building and running GoLang projects.

Go This section defines different versions of the go compiler the project is built using. When code is checked in - Travis will spin up a separate Docker container (Section 4.2.4) for each version specified, and run your complete test suite inside each container in parallel. This feature is known as the 'build matrix', and is *incredibly* useful for ensuring software consistency on multiple different compilers/runtimes (especially useful for interpreted languages such as Java), and is something which would be hard to replicate outside of a containerised build environment (i.e. if using a self-hosted CI environment).

Script Any custom commands to be run as part of the build. Travis contains[5] a standard build script for most languages (which are selected via the 'language' configuration detailed above), however builds invariably require additional scripts/commands to be run as well - some example use cases could be to run additional tests, or deploy build binaries to an artifact repository. In this case, the commands specified execute both the unit/integration test suite, and the benchmark suite (Section 5.1.2).

[4]More information on the contents of .travis.yml files can be found in their excellent documentation.
[5]Pun intended

5.1.4 Coveralls



LATEST BUILDS

BUILD	BRANCH	COVERAGE	COMMIT	COMMITTER	TYPE	TIME	VIA
#109	master	78.92	Update HELP text	FireEater64	push	27 Apr 2016	travis-ci
#108	master	78.92	Reflow log config.xml Formatting allows for direct inclusion in report.	FireEater64	push	28 Mar 2016	travis-ci
#104	master	76.46	Bug in go 1.4 - removing test	FireEater64	push	15 Mar 2016	travis-ci
#103	master	78.48	Merge branch 'master' of github.com:FireEater64/gamq	FireEater64	push	15 Mar 2016	travis-ci
#102	master	78.48	More unit tests	FireEater64	push	15 Mar 2016	travis-ci
#101	master	75.56	Merge branch 'master' of github.com:FireEater64/gamq	FireEater64	push	15 Mar 2016	travis-ci

REPO STATS

78.92% COVERED

1760 OF 2218 RELEVANT LINES COVERED

LANGUAGES	
GO	
REPO ADDED	TOTAL FILES
05 DEC 2015	11
BUILDS	LAST BUILD
67	#109
JOBS	LAST JOB
299	#5

LAST COMMIT
Update HELP text

README BADGE
coverage 79%
EMBED

Figure 5.2: Coveralls

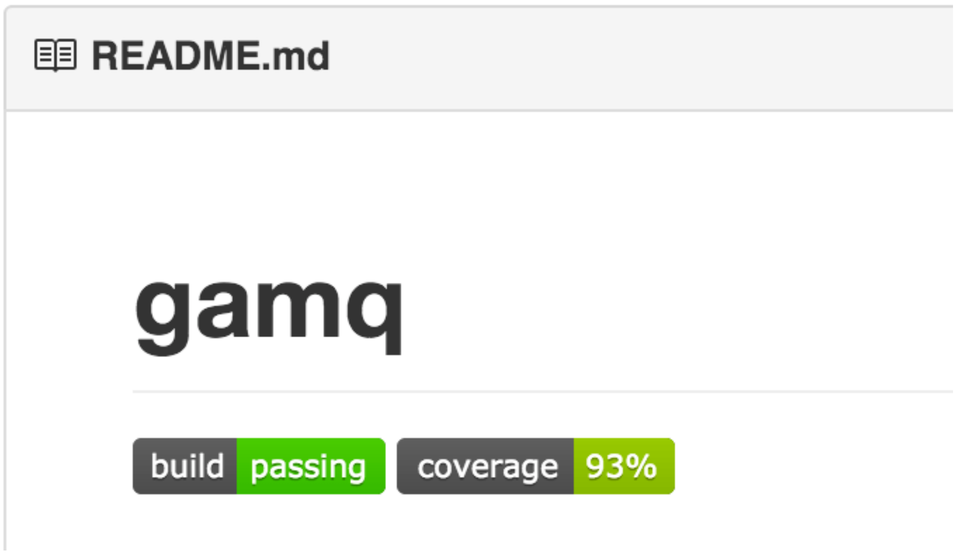
One important (though not definitive) metric associated with unit/integration tests is *code coverage*, which is the total percentage of the code-base that is executed when running unit tests. This is a useful metric to keep an eye on, as it helps[6] indicate which sections of code are susceptible to bugs as a result of not being tested. Go's built in test runner is capable of producing code coverage reports during test runs, however the decision was taken to use an open source tool called 'gocov', as it allows the code quality metrics produced by the build to be sent to another online service, coveralls. The reasons behind doing this, rather than relying on the build-in HTML report were:

Visibility Publishing code quality metrics in an easily accessibly, publicly visible website (as well as the front page of my GitHub project), rather than hiding them away in build logs helps to 'keep developers honest', as well as '*gamify*' the process of driving up coverage.

Monitoring Coveralls can set 'thresholds' for coverage metrics, and will fail the build if these are not adhered to. For example, the gamq build will fail if the coverage of the checked in code is even 0.1% less than that of the previous successful build.

The code quality metrics for gamq are available on Coveralls, as well as being summarised in the README for gamq on GitHub (Figure 5.3).

[6]Though doesn't always


README.md

gamq

build passing
coverage 93%

Figure 5.3: Code status badges in README.md

5.2 Environmental Testing

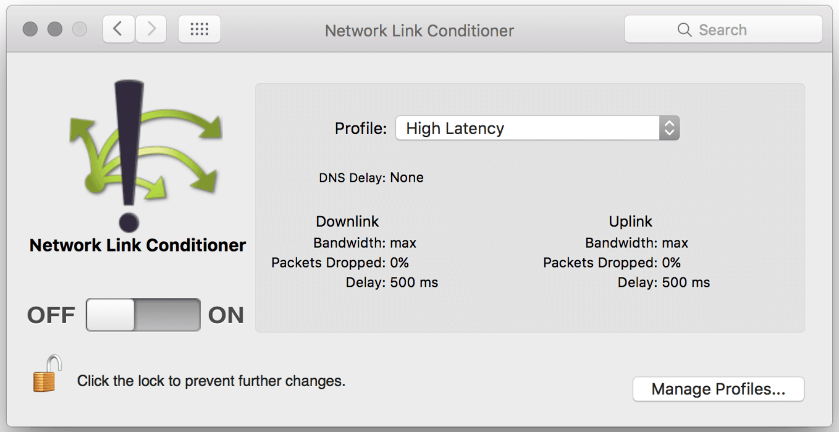
In-place, or *environmental* testing helps ensure that software/systems are capable of full operation under sub-optimal circumstances. It is an important step when writing software, as there are many design and implementation failings that only become apparent under these suboptimal conditions. For example, a message broker using UDP may appear to perform perfectly in a development environment, but if the production environment spans multiple datacenters - the high probability of packet loss on inter-datacenter links may cause problems. This is one of the reasons development infrastructure should resemble that of production as closely as possible, warts and all.

5.2.1 Tools

Using the built in real-time performance metrics tools discussed in Section 4.2, and a set of Python benchmarks, like the one listed in Appendix F - rough performance figures for gamq could be measures. The Network Link Conditioning Tool (Figure 5.4) was also used to simulate suboptimal networking conditions (latency, packet loss etc.) in a reproducable fashion, to validate the operation of the broker in less-than-ideal environments. Gamq can also produce performance profile information in the standardised 'pprof' format using a command line flag (Section 3.3.1), which can be visualised to produce 'hotspot' graphs, that details the amount of CPU time spent in each function. An example graph can be seen in Appendix H.

5.3 System Performance

Nominal performance of gamq using TCP messages without acknowledgement messages, was around 15,000 messages/second on a second-generation Intel Core i7. UDP messages produced roughly double this figure - 30,000 messages per second, but also produced a



Profile: High Latency

DNS Delay: None

Downlink
Bandwidth: max
Packets Dropped: 0%
Delay: 500 ms

Uplink
Bandwidth: max
Packets Dropped: 0%
Delay: 500 ms

Click the lock to prevent further changes.

Figure 5.4: Screenshot of the network link conditioner

very large number of lost messages, due to the lack of built in flow control in the UDP protocol. Enabling message acknowledgements brought the TCP message rates down to 10,000 messages/second, and actually increased the UDP performance slightly - to 34,000 messages/second, by reducing the number of lost messages.

Chapter 6

Conclusion

Overall, the project was an excellent learning experience. It provided an opportunity to learn a new programming language and explore modern, open-source development techniques - as well as learn about the intricacies involved with building infrastructure software, the complexities of which are often overlooked.

6.1 Project Goals

The goal of this project has changed multiple times over the last 8 months. Initial plans, were to build a very simple message broker - to be used as a vehicle for exploring autonomic computing principals, and create a 'self-healing', 'self-improving' message broker. These goals were made with no knowledge of the challenges associated with learning a new language like Go, and in dealing with the complexities of parallelism at the scale used in gamq. During a mid-project review it was decided that, whilst it was theoretically possible that the complexity of the broker could be scaled down, and the original project goals still met - the goal of the project should be refocused towards producing a mature, high-performance, well-tested messaging broker. This reflected not only the reality of the situation at the time, but a change in the authors interests. These new goals, the ones outlined in Chapter 1 have, as far as the author is concerned, all been met - and the decision to change the project focus was, in hindsight, the correct one.

6.1.1 Experience with GoLang

After completing the authors first substantial piece of work with the Go programming language, the power of CSP-style concurrency, combined with a clean, modern development infrastructure - demonstrate why Go is now one of the top 20 most popular languages on GitHub[16], despite its young age. Whilst it is arguable that a broker with similar feature-set could have been produced in a more familiar language[1] in less time, the performance and scalability of the implementation would have almost certainly lagged behind that of gamq.

[1]Such as Python, or Java

6.1.2 Performance

Whilst the performance of gamq (Section 5.3) is considerably higher than originally hoped (original estimates predicted somewhere in the region of 5,000 messages/second), there are still numerous potential performance increases that could be explored. The biggest of these (according to the CPU performance profile in Appendix H) lies in reducing the amount of garbage created by the application (and consequently, the amount of time spent performing garbage collection), through the reuse of message objects, and the use of zero-allocation buffers [6] when receiving data.

6.2 Future Work

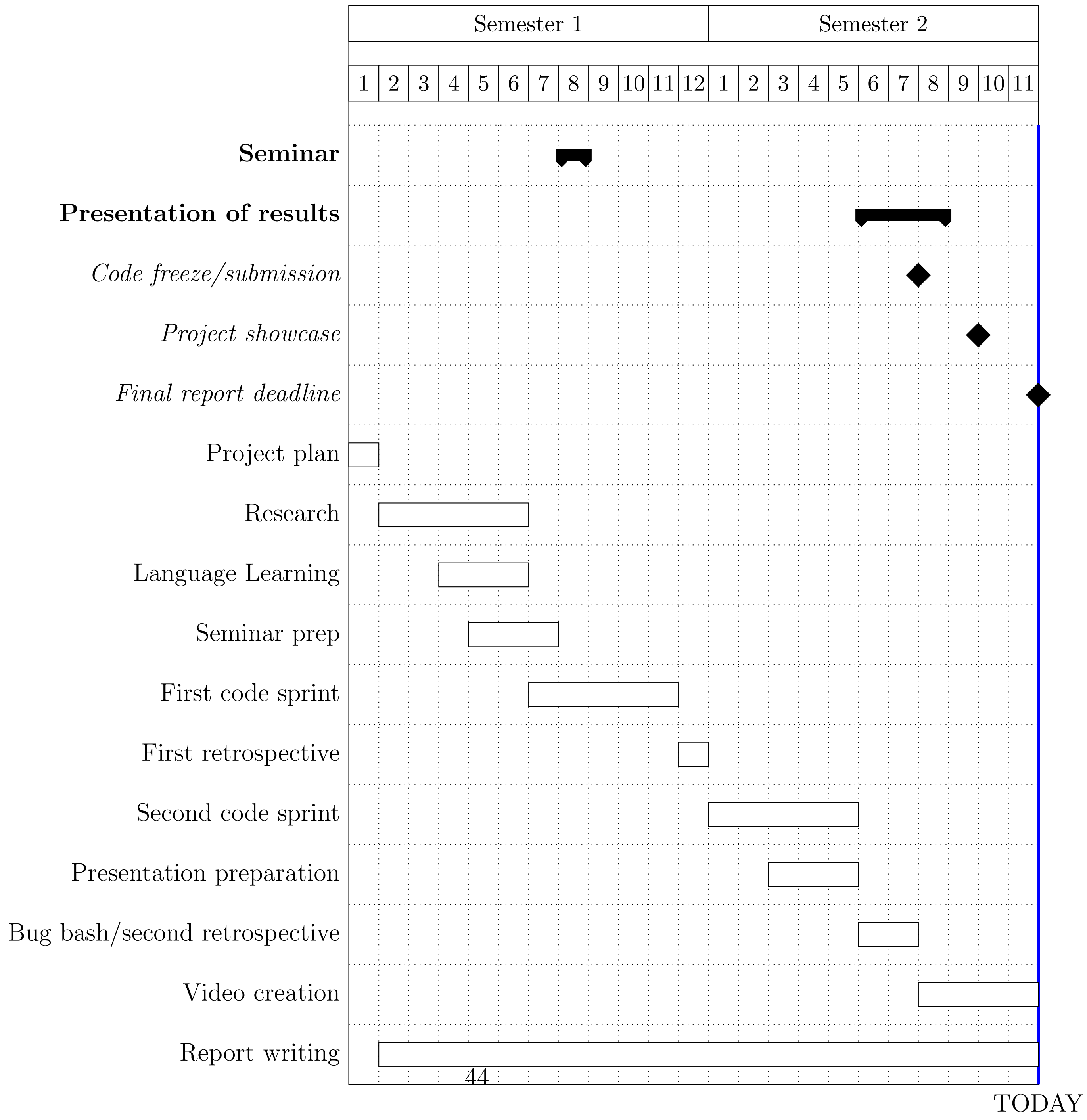
Going forward, it is hoped that some of the knowledge (and possibly code!) gained from this project continue to see use in some form or another. Towards the end of the project, the 'professional open-source' message broker NATS[2] was examined for similarities/differences with gamq. Whilst the size and scope of NATS far outstrips that of gamq, it is hoped that it will be possible to transfer some of the knowledge gained during this project in the form of open-source contributions[3] to other projects like NATS.

[2]`https://nats.io/`

[3]`https://github.com/nats-io/gnatsd/pulls`

Appendix A

Project Gantt Chart



Appendix B

Queue Code

```
package queue

import (
	"github.com/FireEater64/gamq/message"
)

// Queue represents an in-memory queue, using a slice-basedgit stat
 ↪  data structure, and
// channel io
type Queue struct {
	Name          string
	InputChannel  chan *message.Message
	OutputChannel chan *message.Message
	head          *queueMessage
	tail          *queueMessage
	length        int
}

// NewQueue create a new queue, with given name
func NewQueue(givenName string) *Queue {
	q := Queue{}
	q.InputChannel = make(chan *message.Message)
	q.OutputChannel = make(chan *message.Message)

	go q.pump()

	return &q
}

// Run in a goroutine - receives and transmits messages on the channels
func (q *Queue) pump() {
pump:
	for {
```

```
                // If we have no messages - block until we receive one
                if q.head == nil {
                        newMessage, ok := <-q.InputChannel
                        newQueueMessage := newQueueMessage(newMessage)
                        if !ok {
                                break pump // Someone closed our input
 ↪  channel
                        }
                        q.head = newQueueMessage
                        q.tail = newQueueMessage
                        q.length++
                }

                select {

                case newMessage, ok := <-q.InputChannel:
                        if !ok {
                                break pump // Someone closed our input
 ↪  channel - we're shutting down the pipeline
                        }
                        newQueueMessage := newQueueMessage(newMessage)
                        q.tail.next = newQueueMessage
                        q.tail = newQueueMessage
                        q.length++

                case q.OutputChannel <- q.head.data:
                        q.head = q.head.next
                        q.length--
                }
        }

        // Finish sending remaining values
        nextMessageToSend := q.head
        for nextMessageToSend != nil {
                q.OutputChannel <- nextMessageToSend.data
                nextMessageToSend = nextMessageToSend.next
        }

        close(q.OutputChannel)
}

func (q *Queue) PendingMessages() int {
        return q.length
}
```

Appendix C

UdpWriter

```
package udp

import (
        "net"
)

// Writer is a simple implementation of the writer interface, for
↪  sending
// UDP messages
type Writer struct {
        address    *net.UDPAddr
        connection *net.UDPConn
}

// NewUDPWriter returns a new UDP Writer
func NewUDPWriter(givenConnection *net.UDPConn, givenAddress
↪  *net.UDPAddr) *Writer {
        return &Writer{address: givenAddress, connection:
↪  givenConnection}
}

// Write is an implementation of the standard io.Writer interface
func (udpWriter *Writer) Write(givenBytes []byte) (int, error) {
        return udpWriter.connection.WriteToUDP(givenBytes,
↪  udpWriter.address)
}
```

Appendix D

Parse Client Command

```
                return
        }

        commandTokens[0] = strings.ToUpper(commandTokens[0])

        switch commandTokens[0] {
        case "HELP":
                manager.sendStringToClient(helpString, client)
        case "PUB":
                // TODO: Handle headers
                message := message.NewHeaderlessMessage(messageBody)
                manager.qm.Publish(commandTokens[1], message)
                if client.AckRequested {
                        manager.sendStringToClient("PUBACK\n", client)
                }
        case "SUB":
                manager.qm.Subscribe(commandTokens[1], client)
        case "DISCONNECT":
                *client.Closed <- true
        case "PINGREQ":
                manager.sendStringToClient("PINGRESP", client)
        case "SETACK":
                if strings.ToUpper(commandTokens[1]) == "ON" {
                        client.AckRequested = true
                } else {
                        client.AckRequested = false
                }
        case "CLOSE":
                manager.qm.CloseQueue(commandTokens[1])
        default:
                manager.sendStringToClient(unrecognisedCommandText,
↪   client)
        }
```

```
}

func (manager *ConnectionManager) sendStringToClient(toSend string,
 ↪  client *Client) {
```

Appendix E

Send Messages To Client

```
func (shipper *messageShipper) forwardMessageToClient() {
        for {
                select {
                case message, more := <-shipper.messageChannel:
                        if more {
                                _, err :=
 ↪  shipper.subscriber.Writer.Write(*message.Body)
                                if err != nil {
                                        log.Errorf("Error whilst sending
 ↪  message to consumer: %s", err)
                                }

                                // Write end runes
                                shipper.subscriber.Writer.Write(shipper.⌋
 ↪  endBytes)
                                shipper.subscriber.Writer.Flush()

                                // Bit of a hack - but stops an
 ↪  infinite loop
                                if shipper.queueName != metricsQueueName
 ↪  {
                                        // Calculate and log the
 ↪  latency for the sent message
                                        shipper.metricsChannel <-
 ↪  NewMetric("latency", "timing",
 ↪  time.Now().Sub(message.ReceivedAt).Nanoseconds()/1000000)
                                        // Log the number of bytes
 ↪  received
                                        shipper.metricsChannel <-
 ↪  NewMetric("bytesout", "counter",
 ↪  int64(len(*message.Body)+len(shipper.endBytes)))
                                }
```

```
                    } else {
                            return
                    }
            case closing := <-shipper.CloseChannel:
                    if closing {
                            log.Debugf("Message shipper for %s
 ↪  closing down", shipper.ClientName)
                            return
                    }
            }
    }
}
```

Appendix F

Benchmark Code

```
#!/usr/bin/env python

# Python benchmark for gamq

import time
import socket
import threading

# Global variables
HostAddress = "localhost"
HostPort = 48879
Protocol = ""
AckMessages = False
NumberOfMessages = 0

# Helper function to check if a number is valid
def isNumber(givenObject):
    try:
        int(givenObject)
        return True
    except:
        return False


def getSocket(protocol):
    if protocol == "tcp":
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    elif protocol == "udp":
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    else:
        print "Invalid protocol: {}".format(protocol)
        exit(-1)
    s.connect((HostAddress, HostPort))
    return s


def writeThread():
    s = getSocket(Protocol)

    if AckMessages:
        s.sendall("setack on\n")

    startTime = time.clock()
    for i in range(0, int(NumberOfMessages), 1):
        s.sendall("pub abc\n")
        s.sendall("{}\n".format(i))
        s.sendall(".\r\n")
        if AckMessages:
            response = s.recv(8)
            if response[:6] != "PUBACK":
                print "Error whilst publishing {}, got response:
↪   {}".format(i, response)
    endTime = time.clock()
    s.close()
    print "Took {} seconds to write {} messages".format((endTime -
↪   startTime), NumberOfMessages)


def readThread():
    s = getSocket("tcp")
    startTime = time.clock()
    s.sendall("sub abc\n")

    for i in range(0, int(NumberOfMessages), 1):
        response = ""
        while response[-3:] != ".\r\n":
            response += s.recv(1)
        response = response.translate(None, ".\r\n")
        if int() != int(i):
            print "Expected {}, got {}".format(i, response)
    endTime = time.clock()
    s.close()
    print "Took {} seconds to read {} messages".format((endTime -
↪   startTime), NumberOfMessages)


def readConfig():
    global AckMessages, NumberOfMessages, HostAddress, HostPort, Protocol
    # Get benchmark parameters
    protocol = raw_input("Protocol to use (tcp/udp): ")

    if protocol not in ["tcp", "udp"]:
        print "Invalid protocol"
        exit(-1)
    else:
        Protocol = protocol

    numberOfMessages = raw_input("Number of messages to send: ")

    if not isNumber(numberOfMessages):
        print "Invalid number"
        exit(-1)
    else:
        NumberOfMessages = int(numberOfMessages)

    ackMessages = raw_input("Ack messages (y/n): ")

    AckMessages = (ackMessages == "y")

    hostAddress = raw_input("Host to connect to: ")

    if hostAddress == "":
        print "Defaulting to localhost"
    else:
        HostAddress = hostAddress

    hostPort = raw_input("Port to connect to: ")

    if hostPort == "":
        print "Defaulting to 48879"
    elif isNumber(hostPort):
        HostPort = hostPort
    else:
        print "Invalid number"
        exit(-1)

readConfig()
writeThread = threading.Thread(target=writeThread)
readThread = threading.Thread(target=readThread)
readThread.daemon = True
writeThread.daemon = True
writeThread.start()
readThread.start()
```

```
while threading.active_count() > 1:
    time.sleep(1)
```

Appendix G

Test Output

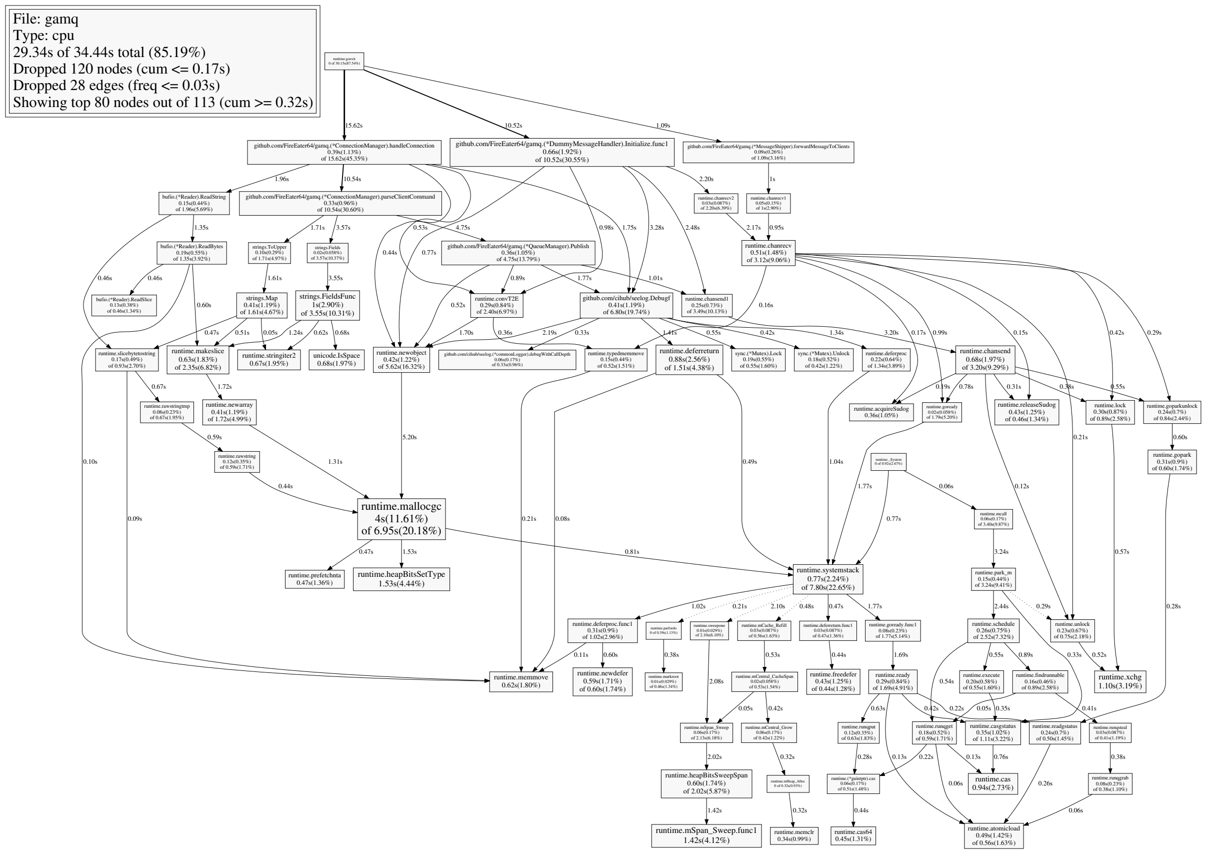
```
$ go test -v ./... -bench=.
=== RUN   TestNewClient_ReturnsValidClient
--- PASS: TestNewClient_ReturnsValidClient (0.00s)
=== RUN   TestConnectionManager_parseClientCommand_helpMessageReturns
--- PASS: TestConnectionManager_parseClientCommand_helpMessageReturns
 ↪  (0.00s)
=== RUN   TestConnectionManager_parseClientCommand_isCaseInsensitive
--- PASS: TestConnectionManager_parseClientCommand_isCaseInsensitive
 ↪  (0.00s)
=== RUN   TestConnectionManager_setAck_setsClientAckFlag
--- PASS: TestConnectionManager_setAck_setsClientAckFlag (0.00s)
=== RUN   TestConnectionManager_parseEmptyCommand_doesNotCrash
--- PASS: TestConnectionManager_parseEmptyCommand_doesNotCrash (0.00s)
=== RUN
 ↪  TestConnectionManager_subscribeToQueue_addsClientToListOfSubscribers
1461795441110701930 [Debug] Initialized QueueManager
1461795441110795546 [Info] Mock subscribed to test
--- PASS:
 ↪  TestConnectionManager_subscribeToQueue_addsClientToListOfSubscribers
 ↪  (0.00s)
=== RUN   TestConnectionManager_disconnectCommand_removesClient
--- PASS: TestConnectionManager_disconnectCommand_removesClient (0.00s)
        connectionmanager_test.go:114: Disconnecting
=== RUN
 ↪  TestConnectionManager_parseClientCommand_invalidCommandProcessedCorrectly
--- PASS:
 ↪  TestConnectionManager_parseClientCommand_invalidCommandProcessedCorrectly
 ↪  (0.00s)
=== RUN   TestConnectionManager_emptyClientCommand_handledGracefully
--- PASS: TestConnectionManager_emptyClientCommand_handledGracefully
 ↪  (0.00s)
=== RUN
 ↪  TestConnectionManager_whenInitialized_acceptsConnectionsCorrectly
```

```
1461795441110898515 [Debug] Mock unsubscribed from test
1461795441110918247 [Debug] No subscribers left on queue test - closing
1461795441110922431 [Debug] Closing test
1461795441110930018 [Debug] Removing test from active queues
1461795441111042636 [Debug] Message shipper for Mock closing down
1461795441111237828 [Debug] Initialized QueueManager
1461795441111721481 [Debug] Received metric: test.subscribers - 0
1461795441111866985 [Debug] Received metric: test.pending - 0
1461795441112068864 [Debug] A new TCP connection was opened.
--- PASS:
 ↪ TestConnectionManager_whenInitialized_acceptsConnectionsCorrectly
 ↪ (0.00s)
=== RUN   TestQueue_sendMessage_messageReceivedSuccessfully
1461795441112169657 [Debug] Received metric: clients.tcp - 1
--- PASS: TestQueue_sendMessage_messageReceivedSuccessfully (0.00s)
=== RUN   TestQueue_sendMessage_generatesMetrics
1461795441112451500 [Debug] Received metric: bytesin.tcp - 8
1461795441112483027 [Debug] 7444913517715839379 closed connection
--- PASS: TestQueue_sendMessage_generatesMetrics (1.01s)
=== RUN
 ↪ TestQueue_sendMessageAfterUnsubscribe_messageReceivedSuccessfully
1461795442121403743 [Debug] Test1 unsubscribed from TestQueue
1461795442121441451 [Debug] Message shipper for Test1 closing down
--- PASS:
 ↪ TestQueue_sendMessageAfterUnsubscribe_messageReceivedSuccessfully
 ↪ (0.02s)
=== RUN   TestQueue_xPendingMetrics_producesCorrectMetric
--- PASS: TestQueue_xPendingMetrics_producesCorrectMetric (1.02s)
=== RUN   TestQueue_initialize_completesSuccessfully
--- PASS: TestQueue_initialize_completesSuccessfully (0.00s)
=== RUN   TestMessageShipper_SuccessfullyForwardsMessages
--- PASS: TestMessageShipper_SuccessfullyForwardsMessages (0.00s)
=== RUN
 ↪ TestMetricsManager_ReceivesBasicMetric_PublishesDownstreamAndSendsToStatsD
1461795443163371686 [Debug] Initialized StatsD - sending metrics to:
 ↪ localhost:49038
1461795443163392465 [Debug] Initialized QueueManager
1461795443163402523 [Info] Test subscribed to metrics
1461795443163595829 [Debug] Received metric: test - 123
1461795443163652406 [Debug] Logging metrics
1461795444173968597 [Debug] Received metric: test - 123
1461795444173986585 [Debug] Logging metrics
1461795445167617084 [Debug] Received metric: test - 123
1461795445167635142 [Debug] Logging metrics
```

```
--- PASS:
 ↪  TestMetricsManager_ReceivesBasicMetric_PublishesDownstreamAndSendsToStatsD
 ↪  (3.01s)
=== RUN   TestQueueManager_queuesClosed_removedFromMap
1461795446168997177 [Debug] Initialized QueueManager
1461795446169024372 [Info] Dummy subscribed to testing
1461795446169087402 [Debug] Dummy unsubscribed from testing
1461795446169102974 [Debug] No subscribers left on queue testing -
 ↪  closing
1461795446169106598 [Debug] Closing testing
--- PASS: TestQueueManager_queuesClosed_removedFromMap (0.00s)
PASS
ok          github.com/FireEater64/gamq        5.063s
?           github.com/FireEater64/gamq/main        [no test files]
=== RUN   TestNewMessage_CreatesSuccessfully
--- PASS: TestNewMessage_CreatesSuccessfully (0.00s)
=== RUN   TestNewHeaderlessMessage_CreatesSuccessfully
--- PASS: TestNewHeaderlessMessage_CreatesSuccessfully (0.00s)
PASS
BenchmarkMessageCreation-2        10000000               263 ns/op
ok          github.com/FireEater64/gamq/message        2.885s
=== RUN   TestQueue_CanSendAndReceiveBasicMessages
--- PASS: TestQueue_CanSendAndReceiveBasicMessages (0.00s)
=== RUN   TestQueue_ReceiveBeforeSend_ReturnsExpectedResult
--- PASS: TestQueue_ReceiveBeforeSend_ReturnsExpectedResult (0.02s)
=== RUN   TestQueue_CloseQueueImmediately_ThrowsNoErrors
--- PASS: TestQueue_CloseQueueImmediately_ThrowsNoErrors (0.00s)
=== RUN   TestQueue_EvenNumberOfPushesAndPops_GivesZeroFinalLength
--- PASS: TestQueue_EvenNumberOfPushesAndPops_GivesZeroFinalLength
 ↪  (0.01s)
PASS
BenchmarkQueueSendRecv-2         1000000              1918 ns/op
ok          github.com/FireEater64/gamq/queue        1.980s
?           github.com/FireEater64/gamq/udp        [no test files]
```

Appendix H

Profile Results


File: gamq
Type: cpu
29.34s of 34.44s total (85.19%)
Dropped 120 nodes (cum <= 0.17s)
Dropped 28 edges (freq <= 0.03s)
Showing top 80 nodes out of 113 (cum >= 0.32s)
github.com/FireEater64/gamq.(*ConnectionManager).handleConnection
github.com/FireEater64/gamq.(*DummyMessageHandler).Initialize.func1
github.com/FireEater64/gamq.(*ConnectionManager).parseClientCommand
github.com/FireEater64/gamq.(*QueueManager).Publish
github.com/cihub/seelog.Debugf
runtime.mallocgc
4s(11.61%)
of 6.95s(20.18%)
runtime.systemstack
0.77s(2.24%)
of 7.80s(22.65%)
runtime.chanrecv
runtime.chansend
runtime.newobject
runtime.memmove
runtime.heapBitsSetType
runtime.mSpan_Sweep.func1
runtime.xchg
runtime.atomicload

Glossary

false negative A test result indicates that a condition failed, while it actually was successful.. 35

gamify The concept of applying game mechanics and game design techniques to engage and motivate people to achieve their goals.. 38

GoChannel Channels are the *pipes* that connect concurrent goroutines.. 29

GoRoutine A GoRoutine is a lightweight thread managed by the Go runtime.. 25, 30, 33

preempted In computing - the act of temporarily interrupting a task being carried out by a computer system. 14

Time-Series Data A sequence of data-points recorded over a continuous time-interval. 31

Acronyms

AMQP Advanced Message Queuing Protocol. 20

CI Continuous Integration. 3, 35–37

DoS Denial of Service. 6

FIFO First-In-First-Out. 8, 16, 26

FOSS Free and open-source software. 18

G.C. Garbage Collector. 12, 28, 29

gamq George's Asynchronous Message Broker. 1, 19, 21, 22, 25, 29–32, 35, 38, 39, 41

HTTP Hypertext Transfer Protocol. 4, 20, 25

IoT Internet of Things. 10, 11

IP Internet Protocol. 11

JSON JavaScript Object Notation. 4

LAN Local Area Network. 31

LRU Least-Recently-Used. 16

MTU Maximum Transmission Unit. 30

OS Operating System. 14, 16, 17

RPC Remote Procedure Call. 4, 10

RTT Round-trip time. 10

SoA Service Oriented Architecture. 4

TCP Transmission Control Protocol. 4, 6, 10, 11, 20, 24, 25

TLB Translation Lookaside Buffer. 16

UDP User Datagram Protocol. 5, 10, 11, 20, 24, 25, 30

XML Extensible Markup Language. 4, 19

Bibliography

[1] Apcera. *NATS protocol*. 2015. URL: http://nats.io/documentation/internals/nats-protocol/ (visited on 04/27/2016).

[2] Doug Bagley. *Go programs versus Java*. 2000. URL: https://benchmarksgame.alioth.debian.org/u64q/go.html (visited on 04/22/2016).

[3] Andrew Binstock. *Interview with Ken Thompson*. 2011. URL: http://www.drdobbs.com/open-source/interview-with-ken-thompson/229502480 (visited on 04/03/2016).

[4] A. D. Birrell and B. J. Nelson. "Innovations in Internetworking". In: ed. by C. Partridge. Norwood, MA, USA: Artech House, Inc., 1988. Chap. Implementing Remote Procedure Calls, pp. 345–365. ISBN: 0-89006-337-0. URL: http://dl.acm.org/citation.cfm?id=59309.59336.

[5] Dave Cheney. *Performance without the event loop*. 2015. URL: http://dave.cheney.net/2015/08/08/performance-without-the-event-loop (visited on 04/15/2016).

[6] Derek Collison. *GopherCon 2014 High Performance Systems in Go by Derek Collison*. Youtube. 2014. URL: https://youtu.be/ylRKac5kSOk?t=10m46s.

[7] Peter Deutsch. *The Eight Fallacies of Distributed Computing*. URL: https://blogs.oracle.com/jag/resource/Fallacies.html (visited on 05/03/2016).

[8] Etsy. *StatsD*. 2010. URL: https://github.com/etsy/statsd (visited on 04/20/2016).

[9] Jim Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978, pp. 393–481. ISBN: 3-540-08755-9. URL: http://dl.acm.org/citation.cfm?id=647433.723863.

[10] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: http://doi.acm.org/10.1145/359576.359585.

[11] "IEEE Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Equipment and Test Information via XML". In: *IEEE Std 1671-2010 (Revision of IEEE Std 1671-2006)* (2011), pp. 1–388. DOI: 10.1109/IEEESTD.2011.5706290.

[12] S. Deering J. Mogul. *Path MTU Discovery*. RFC 2581. The Internet Engineering Task Force, 1990, pp. 1–14. URL: https://tools.ietf.org/html/rfc1191.

[13] William Kennedy. *The Nature Of Channels In Go*. 2014. URL: https://www.goinggo.net/2014/02/the-nature-of-channels-in-go.html (visited on 05/02/2016).

[14] V. Paxson M. Allman and W. Stevens. *TCP Congestion Control*. RFC 2581. The Internet Engineering Task Force, 1999, pp. 1–14. URL: https://tools.ietf.org/html/rfc1654.

[15] OED. *Oxford English Dictionary Online, 2nd edition*. http://www.oed.com/. 1989.

[16] Stephen O'Grady. *The RedMonk Programming Language Rankings: January 2016*. 2016. URL: http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/ (visited on 05/03/2016).

[17] Rob Pike. *Less is exponentially more*. 2012. URL: http://commandcenter.blogspot.co.uk/2012/06/less-is-exponentially-more.html (visited on 04/03/2016).

[18] Solace Systems. 2015. URL: http://www.solacesystems.com/products/message-router-appliances.

[19] Boston University. *Tagged TLBs and Context Switching*. 2011. URL: http://blogs.bu.edu/md/2011/12/06/tagged-tlbs-and-context-switching/ (visited on 04/19/2016).