Daniel Brahneborg

Magisteruppsats, 30 hp

March 30, 2015

Pipelined Messaging Gateway

Abstract

The SMS gateway EMG uses separate threads for each connection. This requires a large number of locks to avoid data corruption, and causes problems when scaling to thousands of connections. The CPU load goes very high, and the amount of addressable memory per thread becomes low.

This report examines an alternative implementation, where each thread instead handles a separate part of the processing. The two architectures are compared based on throughput, resource utilisation, code complexity and more.

The results show what while throughput is about the same, the alternative implementation is better at keeping the CPU load within reasonable limits. However, it also requires more advanced data structure and algorithms to be efficient.

1. Introduction

1.1. EMG

EMG [1] is a massively multithreaded messaging gateway.

The primary use case is to forward short text messages between operators. The SMS center software that they use may come from different manufacturers, using different communication protocols. EMG sits in the middle, translating between these protocols. Nowadays it's also used for tunnelling messages between operators and SMS driven applications and email servers. Simply put, it keeps incoming and outgoing connections to other systems, receives messages, finds out where to forward it, and transmits it. All messages are deconstructed into their components when they are stored in the system, and then reassembled according to the right communication protocol on the way out.

There are possibilities to rewrite the sender and recipient address as well as the message contents, run plugins at various stages in the message lifecycle, and get everything logged both to file and database. Some of the development is done as consultancy efforts to customers that need specific additions, and some is done according to the combined view of the demands and wishes of current and future customers.

1.2. Problem Description

The current design of EMG has several problems.

Locks and caches

A large number of locks are required to ensure the consistency of the data, since there is much data that needs to be accessed from several threads. According to the analyser Callgrind [2], a significant proportion of CPU time is spent taking and releasing these locks over and over, often to access a value that has not changed. Finding these duplicate locks and caching the value within the thread has proven to be extremely difficult, as the cached value must be correctly invalidated.

It's sometimes said that computer science has two difficult problems: Cache invalidation, naming things, and off-by-one errors. With the concept that in EMG is called "connectors" being called "routes" in the rest of the SMS industry, with "routes" in EMG instead being the rules that decide where to send each message, EMG has all of those problems covered.

Race conditions

Further, race conditions occur occasionally, caused by a value not being correctly protected by a lock, the same lock being taken multiple times within the same thread, or multiple locks being taken in different order by different threads. In the first case the behaviour will be undefined at some point in the future, and in the two latter the resulting deadlock results in the program coming to a complete stop. Neither of these is acceptable. Four machines spend several hours each night stress testing the system to find problems like

these. As it's impossible to test all combinations of the available options, some of these bugs are found by customers. When new features needs to be added, they may have their own set of things they want to lock, so this is getting increasingly difficult.

Dependent on efficient threads

The program is highly dependent on the operating system being able to scale regardless of the number of running threads.

Large memory footprint

The memory footprint goes up to several GB when the number of threads increase up to several thousand, even though it is possible to limit the size of the stack for each thread. Most of it is virtual address space which is never mapped to physical memory, but the output from "ps" is still both misleading and confusing for the customers.

Large log files

The generated log files contain lots of redundancy, with the connectors logging "my queue is empty" in dozens of ways every second, requiring much disk space per time unit. The log files can be limited in size in a way similar to the Linux log rotate tool. This keeps a limit on the total disk space used by the log files, but it also means that it's sometimes impossible to find the core reason for an incorrectly sent message or a program crash, since the file containing the necessary information has been removed.

1.3. Purpose and goal

The purpose of this project is to recreate a minimal version of the existing application, to use as the baseline, and an alternate version using a pipeline architecture. The two versions will then be compared based on aspects such as performance, code size, code readability, code extensibility, scalability, testability, and more.

The goal is to find out if a pipeline architecture would be a better choice than the current one, making it easier and safer to add new functionality, and achieving better resource utilisation. For example, the current architecture uses one or two threads for each connection. When there are few connections, only some of the CPU cores are used, resulting in non-optimal throughput. With many connections, the CPU load goes up too high, making it difficult to run other programs on the same machine.

The hypothesis is that a pipeline architecture with a constant number of running threads will have a more consistent usage of the CPU cores.

1.4. Previous work

Dan Kegel [3] compares the different I/O models that exist in Unix, and the server side effects when handling thousands (or tens of thousand, hence the name of the essay) of clients. He is quite negative to use one thread per client, the method used by the EMG, as that

quickly can run out of addressable memory due to the large number of stacks.

2. Method

2.1. Environment

Both implementations will be done in C on a Linux system. Using C ensures the highest possible performance and control, and Linux is the operating system most often used by customers for running EMG. The compiler is gcc version 4.6.3. The machine used for testing has a 4 core Intel i5 at 2.67 GHz, and 4 GB of memory. The file system is jfs on an Intel SSD.

In order to minimise the total amount of code and make sure the comparison is fair, the base data structures and functions will be shared between the implementations whenever possible. For benchmarking the existing tools emgload and emgsink [4] will be used. They provide a minimal SMPP implementation, doing as little parsing as possible.

The setup is shown below. Emgload submits a message to the tested application, which sends a response back and puts it on a queue for outgoing messages. These are then sent to emgsink. The loops on the right and left halves are independent.

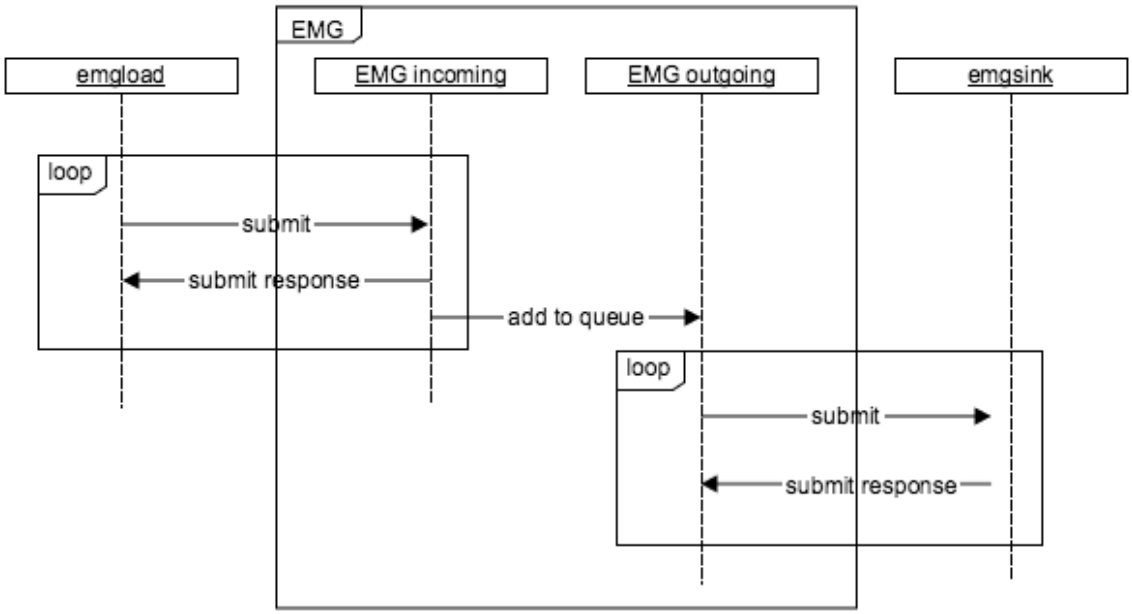

Fig 2.1, setup overview.

When connected to each other, emgload can push 100.000 - 140.000 messages per second to emgsink on the selected hardware. Since the full test setup will involve two transmissions, the theoretical maximum that can be reached using these tools is half of this.

2.1.1. Build environment

CMake [5] will be used as the build system. Some of the source files use preprocessing to enable certain fields and functions only to one of the two binaries, something which is easy to do using the set_target_properties() function. It also provides full dependency tracking, so everything is rebuilt if the compilation flags are changed, not only if source code files are

modified. CMake creates input files for the program “make”, which examines the timestamps of the source code files, and runs the compiler and linker where necessary.

The tool “git” [6] will be used for version control. This makes it easy review the latest changes, and make backups to another computer.

The files will also be compiled on Mac OS X, using the C compiler is Clang [7], a more modern compiler than the one from GNU. Primarily this will enable other warnings that are not supported by gcc. These warnings indicate potential bugs in the code that would be hard to find using testing or code reviews.

2.2. Concepts and data types

The most central concept is the connector. This data type represents a pool of incoming or outgoing connections. Each connector has an IP address and a port, and a maximum pool size. Each connection is called an “instance”. An incoming connector needs as many instances as the number of clients that will connect at the same time, while an outgoing connector should have as many instances as the number of allowed connections on the receiving end.

Each connector has a queue with messages that are waiting to be transmitted. This allows the connectors to operate independently and at different speeds. The structure used for a connector is called “connector_t”, and the one for an instance is called “connectorchild_t”.

The connectors communicate using different protocols, such as SMPP [8] or HTTP. Regardless of protocol, each message requires one piece of data for the message, and one for the reply. Each such piece of data is called a pdu, a Protocol Data Unit. In the illustration earlier, the “submit” and “submit response” arrows all represent the transmission of a pdu. The message itself is stored in the structure “QE”.

2.3. Log file analysis

EMG uses several types of log files, each with its own type of contents. Two types are especially important.

- A general log, containing detailed information about the system as a whole. This file contains startup and shutdown events, and entries from various functions to be able to track what each thread is doing.
- A pdu log file for each connector. These files contain details about each pdu that is written or read. This makes it possible to see exactly how each message pdu was encoded, whether a delivery report was requested, and which message id was returned. By examining the log file it’s also possible to calculate the latency for each operation.

Much effort is put into analysing these log files, after various test runs. This is a simple but effective way of finding out what the program is actually doing. Writing test code is good

when it is known what may go wrong, but simply browsing through a log file seems to utilise the brain's ability to detect patterns and deviations [9]. This browsing also detects when the same data is logged multiple times but from different functions, causing unnecessarily large log files and lower throughput. By logging a detailed timestamp on every line, unexpected pauses in the processing can be found.

Using a debugger is not possible in a system like this, since it's so inefficient trying to single step several threads in parallel. Instead, the key functions log when they are entered, printing some or all of their input parameters. In some cases they also log intermediate data, as well as their return value.

2.4. Simplifications

Whenever possible, existing code from EMG will be reused. Since the primary goal is to measure performance, the code will be simplified as much as possible, removing the parts that are not relevant.

- The user authentication.
- The configuration handling. Instead, the configuration is either hard coded or can be set at startup through command line options.
- Delivery reports. When a message has been successfully delivered or failed for some reason, a delivery report is sent back. The code for this is basically just a few additional hash table operations, so the performance when testing such two way traffic ends up being half of when just sending normal messages.
- The application level error handling. The only relevant errors are outgoing connection failures. Authentication failures can not happen, and since messages are never refused by the receiver, there is no need for logic to dealing with resending of messages.
- Rewriting addresses and message contents.
- All protocols except SMPP. The other protocols supported by EMG require just slightly different amount of work by the CPU and consist of code of equal complexity, so including them would have little value.
- In the original EMG the code for splitting the message into one or more pdus depending on the message body length, handling different character encodings, the UDH data and other things is quite complex. In this project, all messages become a single pdu, and the message body is taken as is. The same function is used by both architectures. The protocol encoding function itself is taken unchanged from the original code.

In addition to this, the original code for connecting and disconnecting is quite complex, so both new implementations use a simpler version, based on the reference counting of the connections.

2.5. Existing architecture

2.5.1. Connectivity flow

The way incoming connections are handled is both simple and safe. When the program starts, each connector creates a socket for receiving incoming connections. One thread is created for each instance. All threads start by trying to take the “listening” lock belonging to their connector. The one that gets the lock, then hangs in the poll() system call, waiting for a connection. When the new connection has been handled by accept() and received a new socket descriptor, the “listening” lock is released. At this point, another thread takes the lock, and starts to wait for a new incoming connection.

Meanwhile, the original thread keeps on running, handling the I/O on its socket. When it’s done, it closes the socket, returns to the starting point, and tries to take the “listening” lock again. This way, only one thread at a time will accept new connections. This method also enforces the connection limit. When the maximum number of connections has been reached, there are no more threads accepting additional connections. This is illustrated in the figure below.

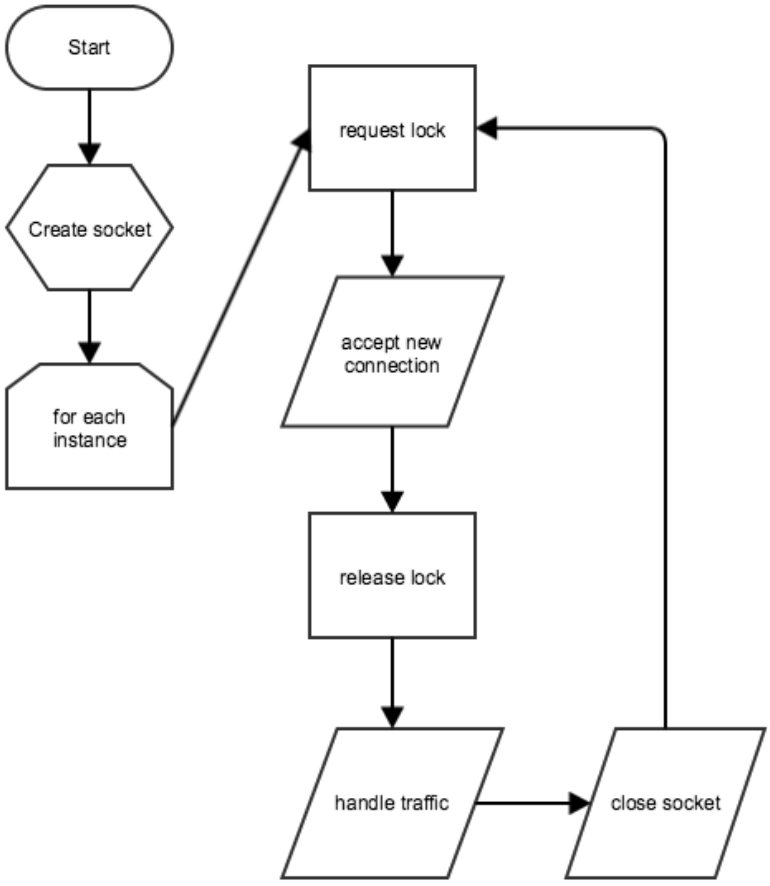

Fig 2.2, connectivity flow for incoming connections

2.5.2. Single vs double threads

There are two groups of protocol implementations in the existing EMG. The first group consists of the simple, synchronous protocols such as HTTP and SMTP. They wait for data, act on it, send a reply, and then wait for more data. Because of this, each connection is handled by a single thread, which performs both reads and writes. When there is no traffic, it spends its time waiting for new data.

The second group contains the asynchronous SMS protocols such as SMPP and UCP. Except for the first dialog containing the authentication details and the corresponding response, dialogs may be initiated by either side at any time. For these protocols, two threads are used for each connection. One is responsible for reading new data, and the other one performs all the writes. Just as for the synchronous protocols, the reader thread mostly hangs waiting for new data on the socket. If the reader reads data that represents a new message, that message is saved in a QE, and a reply is posted to the writer thread.

For the writer, it is more complicated. First, it must check if there are any new messages in its queue. If so, they are encoded into a pdu according to the selected protocol. This pdu is added to a list of outstanding operations. When the response for this operation comes back to the reader thread, this record is extracted. This way EMG knows which message has been sent. If it was successful, or failed with a permanent error, it is removed from the queue. For outgoing connectors, checking for new messages is where the activity starts when it is not connected. It finds a new message, initiates a connection if it is not open already, and after doing the login dialog, sends the message.

Second, the writer must check if there are any replies created by the reader thread. If so, they are extracted and written to the socket. The third responsibility is to send heartbeats. Many operators require the client to send a ping with regular intervals, or they will be disconnected. For this, the writer checks if a certain amount of time has passed since the last pdu was transmitted. If so, a ping is sent. The response to this will also be read by the reader thread.

When the connection has been idle for too long, it should be closed. Some protocols require a logout message to be sent before the connection is closed, so this procedure must be executed in a controlled maner. This is also the responsibility of the writer thread.

2.6. Pipeline architecture

2.6.1. Fundamental idea

The new architecture is inspired by a modern CPU, using a pipeline with small and very specialised worker threads which hand off data to each other. Each step is handled by a separate thread. This is an improvement from the existing architecture, since that means that the number of threads remain constant regardless of the number of connections. By moving connectors and messages between the workers, each representing a specific state, the number of checks that has to be done to make sure the data flows do not not disturb each other is minimised.

For example, before a connector instance has actually connected, there is no need to see if it has timed out. Also, if a message is added to the connector queue, causing it to connect, there is no need to check for new messages until the connection has been fully established, and any login procedure has been completed. Since the data sent between the different stages of the pipeline is different, each one will get a slightly different “hand off” function, depending on what type of data it wants to pass on. The idea is that this will avoid any unnecessary data conversions.

The more details about the current state that is encoded in the state, the higher number of states, and thus threads, will be needed. However, the more stages a message must go through in order to be completely handled, the more locks will have to be taken per message, which will decrease the throughput. A compromise is made by having the connection state as different workers, and application states as state machines in the protocol drivers.

2.6.2. Workers

2.6.2.1. Overview

The following worker threads are used.

1. A listener. This thread listens for new incoming connections from clients.
2. A reader. This thread waits for data to arrive on any of the open sockets, and reads it.
3. A parser. Each data that has been read is parsed, according to the selected protocol for that connection.
4. A pdu logger. All pdus are logged to a connector specific file, with the data separated into the protocol specific fields.
5. A builder. When the received data has been parsed, it is converted into an internal format common for all protocols, and stored in a QE.
6. A writer. All outgoing messages and replies to incoming requests are written to the sockets by this thread.
7. A dispatcher. This thread checks for new outgoing messages on the connector queues, and hands them off to available connector instances.
8. A connecter. Any needed connections are initiated in parallel by this thread.
9. An encoder. This thread is the inverse of the builder, converting back the data in a QE to the protocol specific format.
10. A pdu builder. This thread is the inverse of the parser, combining the data created by the encoder with the proper surrounding data for the transmission layer.

The image below shows the flow for incoming connectors. Arrows with solid lines show control flow, and dashed lines show object flow. The client here is emgload, and the server is emgsink.

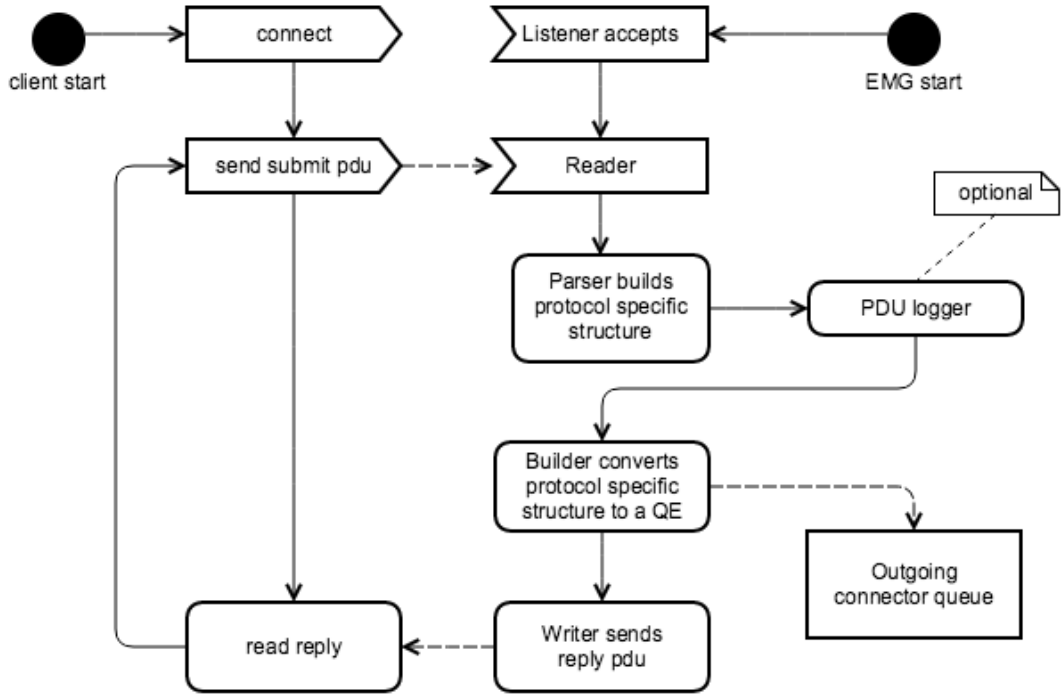

Fig 2.3, flow for incoming connectors.

The flow for outgoing connectors is shown below.

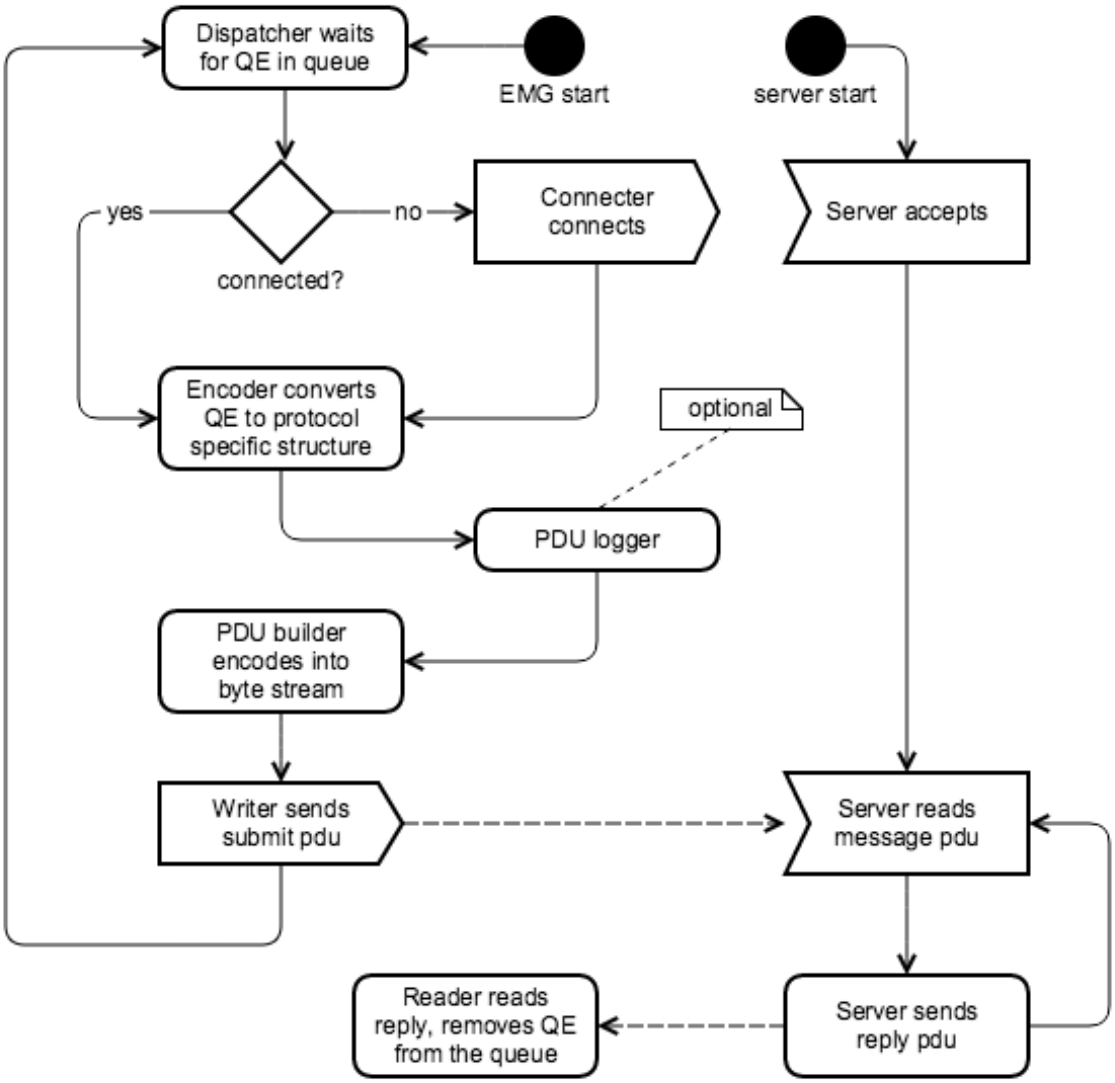

Fig 2.4, flow for outgoing connectors.

2.6.2.2. Listener

The “listener” thread iterates over all connectors that are designated for incoming traffic, creates an initial socket for each of them, and then listens on all these sockets using a single poll() system call. When a request comes in on a socket, the new connection is accepted using the accept() call, one of the connectorchild_t instances is extracted from the pool of the corresponding connector, given the new socket descriptor, and handed off to the “reader” worker.

2.6.2.3. Reader

The “reader” thread is similar to the “listener” thread, as it also uses poll(). A call to poll() is done for all connectorchild_t instances with an active socket. To avoid opening up to a denial-of-service attack where one client keeps sending data without waiting for replies, only a small amount of data is read from each socket each time. The data that is read is pushed to

the parser used for the protocol configured for the connector, along with the connector instance. This allows the parser to know where the data came from.

The same poll() call is also used when a new connector instance is received from the "listener" worker. Michael Kerrisk [10] calls it "the self-pipe trick". When the reader worker thread starts, an internal pipe is created. In addition to waiting for data to read for the connector instances, it also waits for something to read from this pipe. When a new connector instance is added to the readers list, a single byte is written to this pipe. This causes the poll() call to terminate, all available data is read from it, and then the loop starts from the beginning again. This time the new connector instance is included in the watched set. The result is that after a client has connected to the application, data is read almost immediately.

The "self-pipe trick" is also used by the dispatcher and connecter workers.

2.6.2.3.1. Reference counting

Before a socket can be closed, EMG must first make sure that no worker will try to use it during the current session. The problem is doing this safely. The selected solution uses reference counting. All workers later in the chain that may need to write data to the existing connection, increment a counter when the job is given to them. When the job has been handled, this counter is decremented. If the job should be handed over to another worker, this must be done before decrementing the counter. This ensures that it never reaches 0 as long as the connection may be used by a job anywhere.

If the client side closes the connection all future write operations will of course fail, but that does not really matter. Any processing done by these workers may be of no use, but since each task is small, so is their cost. The important thing is that the file descriptor used by the socket is not reused by a new connection all of a sudden.

When the logout request has been seen, a reply is posted, and then a flag is set on the data structure that holds the reference counter. This ensures that as long as the connection is still open by the remote side, the logout reply will be sent before the connection is closed. This is a problem in the original EMG, where the connection is sometimes closed before this pdu has had a chance to be sent. The logout flag is then examined at the end of the loop in the "reader" worker. If it is set, the connection is removed from the list of connections to listen to, and the reference counter is decremented one extra step. This has the effect that when the last worker has released the connection, the application can be sure that no thread will read from that connection again. The socket itself is at that point closed, and can be reused by the next connection.

It is important that only one thread checks this logout flag. If multiple threads check it and decrement the counter an extra step, it may end up as a negative value. This means that some thread is accessing data even after the reference counter has reached 0, resulting in memory corruption. It could also lead to the socket being reused by a new connection before data is written to it, resulting in data being sent to the wrong client.

2.6.2.4. Parser

Each piece of data that has been read successfully by the reader worker, regardless of its size, is added to the end of the job queue for the “parser” worker. The parser worker then extracts these one at a time, examines which connector instance the data belongs to, and parses the data according to the protocol that the connector uses. However, first the data needs to be preprocessed a little bit.

A pdu may be written with several write operations from the client, resulting in multiple IP packets in the network. These can be further split and joined together by the operating systems and network nodes between the sending client and EMG, so the data stream read by the reader has no relationship to the pdu boundaries. As the first step of parsing the data stream, the incoming data is repartitioned back.

For SMPP every pdu starts with a 16 byte header. Those bytes contain the operation number and the amount of additional bytes in the pdu. For a connector that uses SMPP, there is no point in parsing the data until all these 16 bytes has arrived. For UCP, the corresponding header is 7 bytes. For HTTP and SMTP, the parser would instead have to keep collecting data until it reached an end-of-line character.

The main function in the parser worker collects data until it has enough, the protocol parser function is called, and then the worker moves forward in the data in the job entry. The protocol parser uses a state machine to know if it should parse the protocol header or payload. As a result, the logic for collecting the right amount of data can be common for all protocols, and the protocol parsers can be kept simple since they always get as much as they need.

The parser extracts the individual fields values in the data, and these values are stored in a protocol specific data structure. This raw data structure is then passed on to the “builder” worker, optionally via the “pdu logger” worker.

In a production system, there would have to be some sort of timeout handling, printing an error and prompting a closing of the connection if not enough data arrive within the selected timeout interval.

2.6.2.5. Pdu logger

The pdu logs were added rather late, when EMG had existed for several years. In order to disturb the existing code as little as possible, it was implemented at two different places. For incoming traffic, the data is first parsed and then processed. The pdu log file is updated between those two steps, using the intermediate data structure created by the parser. For outgoing data, there is no such data structure. Instead, the outgoing data is assembled, and then parsed to get the data for the pdu log. This means that the data must be parsed, an operation that can be quite expensive.

For this project, outgoing traffic is logged a bit differently. Instead of creating the outgoing data directly, the same data structure is used as when parsing incoming data.

Building the response now takes a little more work by the CPU, but that is more than compensated by not having to parse it. The data is logged, and finally used to assemble the data that should be sent.

2.6.2.6. Builder

The parsed protocol data is processed by the “builder” worker. In most cases, this results in the creation of a new message object, which is then posted to the “router” worker. Any replies that should be sent back to the client, are posted to the “writer” worker. This worker should probably be called “processor” instead, since that better explains what the worker is doing. However, that name would be even more confusing.

2.6.2.7. Writer

In the current setup, outgoing data is always created by the same worker. For an incoming connector the response is created by the “builder”, and for outgoing connectors the data is created by the “encoder”. The “pdu builder” adds the transport layer, and the “writer” worker writes the data to the socket. Having this task in a separate worker has several benefits, even though it is not strictly necessary right now.

Currently, incoming connectors only handle incoming messages. To be able to transmit messages on an incoming connection, an operation called “deliver” in the SMS world, a new set of workers would have to be added. This would run in parallel with the handling of incoming connections, and using a common writer worker ensures that the data do not get mixed up, but instead get nicely serialised on the socket.

If delivery reports were supported, the handling of this would belong in a separate worker, which would create data to be sent back to the client. This has the same problems as “deliver” above.

Having the writing step isolated also makes it possible to parallelise the writing, similarly to what the “reader” is doing. By using poll() with the POLLOUT flag before writing, a slow reader at the other end would only pause traffic on that socket, all other data could still be sent.

2.6.2.8. Dispatcher

Just as the “listener” worker is the starting point for incoming connectors, the “dispatcher” is the starting point for outgoing connectors. Similarly to the “listener” worker, this worker is also centered around poll(). Each connector has a dedicated pipe for this worker, making it possible to react immediately when something has changed on a certain connector, while not having to examine all the other connectors.

Before starting the worker loop, one pointer to a “connectorthread_t” structure for each instance is added to a connector specific list of available instances. At this point none of these instances are connected.

When a connector instance has successfully connected or a new message is added to the outgoing queue, a single byte is written on the pipe. When poll() returns that there is available data for a connector, the worker reads all available data, and then examines the connector queue. If there are no messages in the queue, the processing stops, and the worker hangs in poll() again.

When a message has been found, it must check if the connector instance is connected already. This note safest way to do that is to simply try to fetch the reference counted socket structure. If the connection is notis notis not open, this pointer will be null, and the message is then passed to the “connecter” worker. Otherwise, it’s passed directly to the “encoder” worker.

When the “connecter” worker has successfully connected, it notifies the “dispatcher” worker. Now the window size for the connector is examined. If it is higher than 1, the additional number of instances is added to the beginning of the instance list for the connector. By putting the connector instances at the beginning of the list, connected instances are prioritised before disconnected ones. The connector is then signalled. The original message that initiated the connection is encoded and transmitted. At the same time any additional messages are extracted and used to fill the transmission window.

By letting connector instances occur multiple times in these lists, the transmission window size is automatically enforced. Each occurrence of an instance represents one outstanding operation, so with a window size is 5, the dispatcher will dispatch up to 5 messages to each instance before a reply must be received.

When a reply has arrived and been processed, the number of outstanding operations decrease to 4. The connector instance is returned to the connector’s list of connector instances, and the “dispatcher” pipe is signalled. A new message is dispatched and sent, bringing the number of outstanding operations up to 5 again. There is no explicit operation counter, instead it’s an effect of having multiple occurrences of the same instance.

When the reference counter gets down to 0, which wo not happen until all outstanding operations has gotten their replies, the instance is disconnected. It is removed entirely from the list, and put back just once.

The algorithm has the following characteristics.

1. New messages are handled immediately, if there are available instances.
2. Connected instances are reused before unconnected ones.
3. Transmission windows are supported, and are kept as full as possible.
4. All operations are in O(1) time.

2.6.2.9. Connecter

The “connecter” worker contains an array of connector instances that are not yet

connected, each one given to it by the dispatcher worker. In order to connect them all in parallel, the worker iterates over this array. Each iteration has five phases.

1. A socket is created for each instance, it is set to be nonblocking, and the connection is initiated.
2. All instances where a socket was successfully created, either in this iteration or previously, is moved to the beginning of the array.
3. An array of “struct pollfd” structures are created for the sockets, and populated with the socket descriptor and the POLLOUT flag. This flag tells poll() that the application wants to know when a connection has been fully established. An extra pollfd entry is created for a control pipe. This pipe is used to interrupt the poll() call when a new connector instance is added to the array, so a new iteration can be started immediately.
4. The actual poll() call is performed, which will block for up to one second, waiting for any of the connections to complete.
5. The returned data from poll() is examined. Connections which resulted in an error are thrown away, and a new attempt is made 10 seconds later. This allows the system to automatically resume traffic. In a production system, this delay would have to be configurable. If the connection has succeeded, the instance is given to the “reader” worker so it can starting listening for incoming traffic. The message that initiated the connection is sent to the “encoder” along with the connector instance, just as if the connection was already open when the “dispatcher” worker extracted it.

2.6.2.10. Encoder

The “encoder” worker takes an outgoing message, and encodes it for the protocol used by the connector instance on which it should be sent out. It builds the same protocol specific data structure that the parser creates for incoming pdus. When this is done, this structure is passed on to the “pdu builder”, via the “pdu logger”.

If splitting messages into multiple pdus were to be supported, this is where that would be performed.

2.6.2.11. Pdu builder

The “pdu builder” adds the transport layer data, which is the last step before the message can be sent. This data can consist of a transaction number, the name or number of the operation, and either a start and stop marker or the length of the payload. Once this is done, the resulting byte stream is pushed to the “writer” worker.

2.6.2.12. Production workers

In a production system, there would be a few additional workers.

1. A router. The routing logic may be quite complex, involving checking the first word

of the message body (used by voting on TV shows, for example), checking the destination number prefix, etc. This would justify having a separate worker.

2. A plugin runner. Some connectors need to execute customer specific code at various times in the message lifecycle.
3. Something handling authentication. Before accepting a new connection, the credentials must be checked. This could be done by a memory lookup or one or more database requests.
4. A persister. Adding a message to a connector queue would include persisting the message to disk, and perhaps updating a database.
5. An idle watcher. The connection should be closed if there is no traffic on an outgoing connection within a certain amount of time.

2.6.3. Difficulties

A number of tasks were more difficult than anticipated.

2.6.3.1. Reference counting

In a small run with just 40 messages, the log file contained a surprisingly large number of calls to the reference counter functions. The total number of lines in the log file was 3369 lines, and these calls comprised 1792 of them. This is almost 45 modifications of the reference counter per message, which is clearly too many. The theoretical lower bound is 4, consisting of get and release for read and write, respectively.

By extracting the thread id, the culprit turned out to be the reader worker. All other workers increment the reference counter when the job is added to their queue, and then decrement it again when the job has been handled. The reader worker instead incremented and decremented the counter for all connections on every iteration. The reader worker is the only one that is allowed to do the extra decrement call that closes the connection from the application side, so it can just keep using the handle it already has.

After correcting this, the number of reference counter modifications decreased from 1792 to 1282, or about 32 per message. All accesses to the reference counters are protected by a mutex lock, so it's important to keep this number low. See also section "2.6.4.1 - Small or large workers" on page 24 for more discussion about the locks. The number of locks needed to be lowered further, but at least this was a step in the right direction.

Even though the reference counting for the connections in theory prevents the threads from accessing a dead socket, separating knowing that the connection should be closed, signalling this fact, and actually closing the socket, was far from obvious.

The implemented solution, by allowing only the reader worker to close the socket, highlighted the importance of having a clear ownership of all data. If all workers always decrement the counter as much as that they incremented it, and only one worker can decrement it one extra step, the counter can never go below 0, which would case access to deallocated memory.

Furthermore, this solution makes it safe to use this data without any further locks at all. While the worker has a pointer to the data, it knows that the reference counter is above 0, and that the data is still valid.

2.6.3.2. Incoming vs outgoing connectors

There is a large difference in complexity between incoming and outgoing connectors.

The incoming connectors, i.e. those accepting socket connections from clients, are very straightforward. They have a loop to poll for new connections, a call to accept(), and then they are handed off to the reader worker. This reader reads whatever data arrives on the

socket, parses it, and processes it.

Outgoing connectors sound even simpler, in theory. If there is a message in the queue, make a connection and send it. However, there are several things to consider.

- Connections should be kept open, so they can be reused.
- If there are not any connections available, but more connections are allowed, another one can be opened.
- Since it make take some time to fully establish a connection, other connections must not be affected during that time span.
- The protocol may require some sort of initial authentication dialog.
- The program must not check for new messages on an empty queue incessantly, as that would be a waste of resources.
- It must be possible to limit the number of outstanding operations. The reason for this is discussed in section 3.2.4 - Window size on page 31.
- There may be hundreds of connectors, each with hundreds of instances, and multiple queues with thousands of messages, so the algorithms must be scalable.

2.6.3.3. Tracking data

In the existing EMG, almost all data is created, used and destroyed within the same thread. When looking at the log files, the primary question was therefore what the thread is doing.

As discussed in more detail in 3.2.7 - Memory allocations on page 33, in the pipeline version more data was sent between the threads. A more common question was therefore about the ownership of a certain piece of data. Who had created it, where was it now, and who will eventually release it? Knowing this was crucial to avoid memory leaks.

A large portion of the log files therefore consisted of information about these events, including the address to the relevant memory area. While this information certainly helped, knowing exactly when each memory area could be released was sometimes unexpectedly difficult.

2.6.3.4. Dual perspectives

Using the original EMG source code as the starting point was both a curse and a blessing. A lot of time was saved by not having to reinvent dynamic string buffers and other fundamental data types. Having working code for parsing SMPP pdus was also very useful. Paradoxically, the existing SMPP code caused the most problems.

In the original EMG, incoming SMPP data was handled in a very straightforward way. The data was parsed, processed, and a reply was posted back. If the pdu consisted of a

message, one was created and added to the right outgoing queue. Incoming pdu's were added to the pdu log, and before any replies were written to the socket they were parsed in order to be able to add them to the pdu log. The processing of the data, which meant dealing with login requests and message submissions, was intertwined with the creation of the pdu that should be sent back as the reply. This in itself, worked fine, and was easy to understand.

For the pipeline version, this was no good at all. The pdu log is written to by one worker as a separate step, and so is the processing and the creation of the final reply pdu. In order to be able to log the outgoing replies, this entire section had to be split into maximally decoupled parts.

Letting the "process" code build the response by updating fields in a second pdu data structure, allowed the classic version to still behave almost exactly as before. At the same time, this new data structure could be passed on to the next worker, keeping the steps separate. This also led to the creation of a new function which was the exact inverse of the parsing function, taking the protocol data structure and building a buffer with the formatted values. Making it easy to see exactly how the reply is created for each operation was such an improvement, it may be worth it to merge it back to the original EMG.

Achieving this, while keeping the original flow mainly unchanged, took a long time. It may had been easier to duplicate this code to be able to restructure it as needed for the pipeline version, but the selected solution is better.

2.6.3.5. Sliding windows

In the original EMG, supporting transmission windows is done by checking the number of outstanding operations before checking for the next message to encode and send. When the maximum level is reached, the thread can hang on a semaphore, waiting to be woken up when a response arrives. Since the same thread handles all connections in the pipeline version, letting the encoder or writer pause for a while also delays the transmission of outgoing messages, which means that the queue will never shrink. Keeping multiple pointers to the same connector instance neatly reduces the problem to whether there are any entries at all in the dispatcher's list.

2.6.4. New possibilities

2.6.4.1. Small or large workers

One of the goals of a multithreaded application is to keep all CPU cores as busy as possible, in order to achieve maximum throughput. When the cores need to exchange data, they must use some sort of locking mechanism to avoid corrupting any data. This has two implications for the implementation.

1. The time spent holding a lock must be as short as possible, as the other threads interesting in the same lock can not move forward during this time.
2. Every lock and unlock operation requires some communication between the CPU cores, making them quite expensive. Because of this, the number of locks must be kept at a minimum.

The first aspect is addressed by each worker holding their lock only while actually adding entries to or removing them from the job queue. While doing the real work, they know that they are alone in accessing the data belonging to each job, and can thus proceed at full speed.

It might be more efficient to extract several jobs into a temporary list while holding the lock. This would allow for even more work being done for each lock, which addresses the second aspect. However, this could possibly cause a bottleneck, and limit the amount of work for workers later in the chain. They would be spending much time waiting for new data, lowering the CPU usage and therefore also the total throughput.

To keep all CPU cores fully active, it's better to have many small workers than a few large ones. Small workers can share the same core, keeping the total CPU utilisation high. One CPU core can run multiple workers, but each worker can only run on one core at a time. By having many small workers, the number of workers are kept higher than the number of cores, ensuring that no core stays idle.

An early test showed that the bottleneck was with the parser worker, which initially also processed the data and built the reply. Since it was not possible to simply use more threads on that task, it was divided into a parser part and a builder part. After this change the load between the workers became more even. The chosen set of workers is meant to be a compromise between the two conflicting requirements above.

2.6.4.2. Dynamic workers

Some workers will inevitably spend more time than others. Either their task is simply harder and requires more processing, or they spend time waiting for results from an external source such as a database. In order to circumvent this bottleneck, multiple instances of this worker can be started. This allows their job to be handled by CPU cores which would otherwise be idle. Even if the total time used for each job does not change, the number of processed jobs per time unit will increase. This reduces the idle time for the later workers,

and therefore increases the total throughput.

For maximum throughput, the number of instances of each worker could be adjusted dynamically. A new “manager” worker can examine the queue sizes for each worker, and if one of them is significantly higher than the others, it could be assigned an extra thread. Workers with small queues and multiple threads would get one of the threads stopped by the manager, to keep the total number of threads down, and lowering the risk of threads being idle. In the existing architecture, where each thread performs all tasks, this kind of adjustment is impossible.

There are some important considerations here. Since accept() is synchronous, there can only be one thread listening on each socket. Otherwise, multiple threads may try to accept the same incoming connection, causing the others to stop. To support multiple listeners, the connectors would thus have to be divided between the listener threads. The same goes for the “reader” worker. If multiple threads read data from the same socket, the parser may get the data in the wrong order, so the list of open sockets must be divided among the threads here as well.

Splitting the “parser” is even more difficult. If incoming data is split between multiple packets, they may end up with different parser threads. At that point, the second packet might be handled before the first part has been completely processed. This would cause data to be interpreted incorrectly. A solution could be to let each reader thread have a dedicated parser.

The “writer” would also require special handling. If there are multiple packets posted for the same connector instance, they must be written to the socket in the right order. If two or more workers would extract jobs for the same instance, scheduling issues could lead to them ending up on the socket in the wrong order. This could be solved by statically dividing the instances between the writer workers.

A simple solution to these problems would be to let each connector instance request one worker of each type when the connection is initiated, and then release the workers when disconnecting. In the default case, all instances would get the same worker thread. By using reference counting, the manager could know when a worker was no longer needed, so it could be removed. The system would not be able to react right away if a certain worker had a queue that was much longer than the others, but there would be no problem with data being out of order.

2.6.4.3. Protocol specific workers

As explained above, it’s not possible to add parallelism by simply using multiple threads for the same worker. However, by letting the workers handle only connectors with a certain protocol, a similar effect can be achieved. Dividing the connectors and connector instances becomes trivial, so no dynamic management is needed. With enough cpu cores, some more work can be done without having to increase the window size, using more instances, etc. The total amount of work can be divided between more workers, possibly increasing the cpu

usage, but without introducing any extra steps for each message.

2.6.4.4. Read using epoll

The classic version for waiting on new data in a Unix system is to use select(). This function uses a bitmask to tell the kernel which file descriptors should be examined. The data structure used for this bitmask has a fixed size, putting a limit on the value of the file descriptors that can be watched. On many systems it's as low as 1024. With a few hundred customers, that limit is quickly reached. Because of this, since a number of years back, EMG uses poll() instead.

The poll() function sends the list of file descriptors as an array of integers, and some flags indicating which events should be reported. This makes it independent on the value of the file descriptor, and since each thread in EMG only listens to one socket at a time, it's very fast and efficient.

For the pipe architecture however, using poll() is problematic. Data about all descriptors that the program wants to wait for, needs to be initialised and sent to the kernel on each invocation. When it returns, the application must once again loop through all descriptors to see which ones have new data. With hundreds of descriptors, the overhead for the system call can be significant. If there is activity on only a few of the descriptors, the application loop is also a waste of resources.

To solve this problem, Linux has epoll. This group of functions keeps the list of watched descriptors in kernel space, and when the corresponding wait function returns, the application only gets the ones with activity. This makes it scale much better.

There is another advantage to using epoll instead of poll(). With the latter, the worker needs to keep a list of active sockets, which is then used to build the parameter array on each iteration. Items can be added to this list from other workers, so it needs to be protected by a lock. With epoll, this list is not needed any more, so the lock can be removed. This also makes the code somewhat smaller.

Even after changing just the reader worker to use epoll, performance improved sightly. Other workers using poll() could probably benefit from this change as well.

A downside with epoll is that it's Linux specific. The same problem is solved in other ways in other operating systems, so there exists separate libraries such as libev [11] which act as a frontend. The solution that is available in the current operating system is used, and if none such solution exists, poll() is used instead. The cost for this kind of frontend library is often a higher number of memory allocations, and it's not obvious that the advantage given by epoll is worth the cost for the extra allocations.

Somewhat unexpectedly, using 64 connections actually showed a small decrease in performance, or at best stayed the same as for 16 connections. This could be explained by the fact that epoll probably is most useful in a more realistic situation, when there is traffic on

only some of the connections. In the benchmarks used here, all clients sent data at full speed. This means that there was little or no extra cost to examine the returned events from poll().

2.6.4.5. Transport vs payload

In the existing architecture, each protocol is responsible for both reading data, parsing the transport layer, and parsing the payload. This works well in most situations.

However, HTTP is used as the transport layer with lots of protocols. SMTP is used not only for normal email communication, it's also used for some MMS protocols. In the latter case, the exact same payload encoding is used as for another protocol that uses HTTP. One customer even used HTTP as transport and UCP for the payload.

With the current architecture, separating these two layers is difficult, since the transport layer parser drives the payload parser. In the pipeline version, it would be easy to push data from the reader to the transport layer parser, and then take the payload and handle that with another parser.

3. Comparison

3.1. Measuring performance

The Linux machine used for the measurements, occasionally was busy with other tasks. To compensate for this, each test was run for between 5 and 10 minutes. Every time a message (a QE) was processed, a counter was incremented. Each time the second changed, a new counter was used, and the average for the last 16 seconds was printed. The performance value for the test run was taken as the highest value of these averages.

The rationale was that at least 16 seconds of these 5-10 minutes would be mostly free of other processes doing anything, and getting rid of occasional peaks when there was little incoming traffic due to the queue size limitation which just would measure the performance of the outgoing workers. When the tests were run multiple times, spread over several days, the resulting performance numbers stayed stable, indicating that the values were representative.

3.2. Factors affecting performance

There are several factors that affect the behaviour and performance when using an application like this. Testing all combinations of all possible options (some of them described below) can not be done within a reasonable amount of time, so focus has been given to the aspects that have shown to be most significant.

3.2.1. Locks

The reason it's so expensive to take a lock in a multithreaded environment is that the CPU caches must be invalidated, and synchronisation must occur between the cores. It's better if the threads can run at their own pace as much as possible. This motivated doing a log file analysis collecting statistics for the lock calls.

The classic architecture was found to use between 40 and 45 locks per message. Meanwhile, the pipeline architecture used between 60 and 70, so something had to be changed. Each handoff from one worker to another requires two locks, one when adding the job to the queue and one when extracting it, so the lower bound is about 20 locks per message. The difference between 20 and 60 had to be investigated. The following sections describe the steps used to improve this. In comparison, the real EMG uses between 150 and 200 locks per message.

3.2.1.1. Extract the first item

The function that used the highest number of locks, was list_extractfirst(). This function, as the name suggests, extracts the first item in a list. The reason for the many locks was that several of the workers used a loop with the following steps.

1. Check if the program should terminate.

2. Extract the first item in the job queue.
3. If nothing is found, wait for the “somebody has added something to the list” event. Then start over at step 1.
4. Process and release the found item.

All functions on this type of list use a lock, since most of them require several operations. As long as there are jobs in the queue each job only requires one lock. This is unavoidable, and therefore has to be tolerated. However, as soon as the queue becomes empty, in addition to that first lock, it also needs a lock during the wait call, and then a third one when extracting the new job. The way that wait function is implemented, it uses three more locks than in the first case. If the worker is just slightly faster than any of the previous ones, this will happen for every message being processed.

To improve this, a new list_timed_extractfirst() function was created. It combines the second and third step. If the queue has any entries, one lock is used just as before. If it needs to wait, only a single additional lock call is used. The same “extract or wait” pattern was found in the original EMG, so this new function was quickly added there as well.

3.2.1.2. Builtin atomics

Another big user of locks was the reference counting code. Since there can be several threads both increasing and decreasing the counter, it has to be protected. However, most workers need to update this counter both when the job is added to their queue and when it is removed, requiring lots of costly lock calls.

For the counter itself, a feature in the compiler gcc called “builtin atomics” [12] was used. This is a group of functions providing an interface to the assembler instructions used to implement locks. They are all of the form “fetch and perform operation”, which means they perform some operation on the data, and then return the previous value. All this is done atomically. A regular mutex can be implemented using two of these as a pair, for example “add” and “sub”.

The reference counter used in this project could be implemented using only one of these functions per operation, cutting the number of these atomic operations in half.

3.2.1.3. Faster with fewer locks

After these changes, the number of locks had decreased to about 35 per message. With a small number of connections, performance increased 10-20%. With more connections, the difference was smaller, suggesting that the bottleneck was somewhere else.

3.2.1.4. Lock free data structures

What about the so called “lock free” data structures? Could it be possible to decrease the lock count even more using these?

As it turns out, there is no such thing as lock free data structures. The implementations and algorithms that exist all use the CPU instruction Test And Set, Compare And Swap, or some other version of the same concept. This means that they still invalidate the CPU cache, and therefore cause all threads to slow down for a moment.

A test with these functions, as they are provided by gcc (described previously in the section about 3.2.1.2 - Builtin atomics on page 29), showed that they are about twice as fast as normal Posix locks. This means that a performance gain may be achieved by using them.

The “lock free” aspect simply means that the thread will not stop entirely until the locked data is available. Instead the function will return immediately, making it possible to do other work until it’s time to make another attempt. A better name for them is thus block free, not lock free.

3.2.1.5. Block free data structures

A simple version of a block free list can be written by protecting the existing linked list with the gcc builtin atomics. When worker A wants to push a job on to worker B, a normal mutex is currently used to prevent list corruption. A reasonable algorithm for doing this in a block free way would be as follows.

Worker A uses “fetch and set” to set a lock byte to 1. If it was 0, the current thread owns the data. The job can then be added to the list.

If it was not 0, worker B is at that moment extracting data from the list. Only worker A pushes new jobs to the list, so instead it adds the job to an array. When the next job is about to be pushed, it is added to this temporary array, and a new “fetch and set” is used. New jobs are added to the array until 0 is returned. At that point the entire array is pushed to the job list.

If A got the list, a 0 is written to the lock byte.

Worker B starts the same way, using “fetch and set” to request ownership of the list. The first element in the list is extracted, the lock byte is cleared, and those jobs are then handled. The extracted element is an array created by A.

The problem is the signalling, both when the lock byte is busy and when the list is empty. Just trying the same thing again results in an expensive spin lock, so that’s a bad idea. Using a fixed delay sets a hard ceiling on the performance, unless there is a very high number of connections all using a large window size. The third alternative is to utilise the Posix conditionals with “wait” and “broadcast” used with the normal locks, but then the entire point of using these new functions for speed is moot.

The net effect of having more complex code requiring more allocations of data on the heap (for the job array), was lower performance and/or much higher message latency. In theory what should have been faster methods, only created new problems.

3.2.2. Logging

All connection and login attempts in both directions, all messages and their lifecycle as well other internal processes must be logged. Some of this can and should be logged to a database, but for this project only logging to files is supported. During development and troubleshooting, a high log level should be used, to let the program record every step in detail. In production the log level is usually decreased a bit, only saving the data that is used for billing. This saves disk space and increases performance.

To support this, the configured log level is examined at runtime. Since every log call includes the level for the call, as well as a bit representing the current module, it's possible to selectively enable just some of the log calls. This is useful when debugging, to keep the amount of data down. The real EMG as well as the two minimal versions used in this project differ in performance with a factor of about 4 between logging nothing and everything.

3.2.3. Connection count

Since the classic EMG uses one or two separate threads for each connection, and the number of running threads affect the risk for lock conflicts, the CPU load etc, all tests were done with a wide range of connection counts. There is a limit to the number of threads that can be created on a single system. In Linux this value can be found in the /proc/sys/kernel/threads-max file. There is also a limit to the number of threads that can be handled by the emgload and emgsink tools, since they were not designed to handle more than about 100 threads.

Using a logarithmic scale gave the most meaningful values, especially with some extra focus on low values to match the number of CPU cores. The selected values are 1, 2, 4, 16 and 64.

3.2.4. Window size

A common problem with network traffic is latency. If the client is forced to wait for the response for each operation before sending the next, it will spend lots of time doing nothing. It takes time for the message to travel from the client to the application, to be parsed and processed by the application, and for the reply to travel back to the client. While the message travels in the network, neither side can do anything but wait. The SMS protocols all support windowing as a remedy of this problem.

The clients can send as many operations as given by their configured window size, each tagged with a transaction number. To allow the client to match responses to the sent pdus, all replies include the same transaction number as the original pdu.

Windowing is still useful even though all network traffic in this project is within the same machine. From the moment a message has been received, it takes some time until it has been parsed, stored, and a reply has been constructed and sent back. If the window size is larger

than 1, the client is able to send more messages during this time.

A higher window size induces a higher risk. If the connection dies before all replies have come back, the client has no way of knowing whether the messages were received and processed. The higher the window size is, the more messages will have to be put back into the outgoing queue, in order to be resent once the connection has been reestablished. If the messages had already been received, this means that they would be sent twice, ending up twice at the recipient, at twice the cost for the owner of the application using the higher window size. Because of this, the EMG licensees are advised to use a window size that is as small as possible. By default, a window size of 1 is used.

Two window sizes were used. First the basic case with 1, and then with 10, since tests with higher values gave no further significant performance increase.

3.2.5. Message latency

Even though one of the most important performance aspects of a messaging gateway is the average throughput, it's also interesting to examine the latency caused by the application. Some of it can be managed by using a larger window size on the client, but as mentioned earlier, that also means a higher risk.

With the classic architecture, the latency for a message stayed at a few hundred microseconds for up to 16 connections. With 64 connections, the cpu load went up to 40, and the latency increased to about 1 millisecond. Clearly all threads could not run at full speed at that point, causing all messages to run slightly slower through the system.

For the pipeline architecture there was an entirely different pattern. For between 1 and 4 connections, the latency was about 200 microseconds, roughly the same as for the classic architecture. The cpu had 4 cores, so most likely each core ran one worker each, all processing a different message. Each worker can only run at the full speed of a single core, and they are not affected by what the other cores and workers are doing.

With a higher number of connections, there are more messages than workers, so the messages end up waiting in the worker queues for a longer time. Because of this, the latency for 64 connections was almost exactly four times as high as for 16 connections. This can be explained by 16 connections being enough to keep all workers busy all the time, but the additional connections only led to longer queues.

More workers, for example for advanced routing, additional logging, making plugin calls etc, running on a machine with more cpu cores would mean more messages could be processed in parallel. More workers would mean more processing per message, increasing the total latency, but as long as there were available cpu cores, the time spent processing each message would stay constant. This would also increase the threshold for the number of connections where neither throughput nor latency is affected.

3.2.6. Queue sizes

There is an important issue that must be handled when measuring the performance of this type of application. If the incoming side is faster than the outgoing, there will be an infinitely growing queue with messages that should be sent. Measuring the speed only from the sending side would then give a value that is too high. On the other hand, if a limited number of messages was sent in order to limit the size of that queue, the rate measured on the outgoing side would also be incorrect. When there is no more incoming data to handle, there would be more cpu power to use for sending, giving a value that is not representative.

As a way to resolve this, the queue size for outgoing messages on each connector has a limit of 50.000 messages. When this limit is reached, the response back to the sending client is slightly delayed. Since the client slows down automatically to respect its window size, this gives the sending side a chance to catch up, shrinking the queue. Still, the queue size stays within the same interval, indicating that all components are in balance, making the throughput from the sending side representative of the average throughput of the application.

3.2.7. Memory allocations

Running both versions of the program within Callgrind made it obvious that there were big differences in how the cpu was utilised. With the pipe version, the top positions consisted of memory allocation functions. With the classic version, these were hardly within the top 10.

The program Valgrind can not only verify memory accesses, it can also print the number of memory allocations that the program performs. This showed that the classic version used 37 memory allocations per message, while the pipe version used 74. Since the performance of the classic version was almost 50% higher in some configurations, it seemed like a reasonable idea to try to bring that latter number down.

Some of the allocations were for string buffers used by outgoing SMPP operations. After adjusting the implementation to use the same memory area for both the operation structure and the encoded version, the allocation numbers shrunk to 15 for the classic version, and 47 for the pipe version.

Another group of culprits was the queues used by most of the workers. Each job required one memory allocation, and then another one for the linked list entry containing it. Since the queue size is limited, the linked list was replaced by a dynamically resized array with a rotating “read at” and “write at” index pointers. Normally, just a few simple instructions would be required, protected by a mutex. When the array gets full, a new larger array would be allocated, and the pointers moved there. With the new queues, the number of allocations shrunk to 36.

In the classic version, keeping data areas on the stack, which is far cheaper than using allocations, was easy. Most data processing is done in functions that were called by the owner of the data, so many sections has code on the form “initialise data, process data, release data”.

In the pipeline version, these three parts are sometimes done by completely separate threads. Identifying data that could live on the stack was therefore not feasible. Instead, focus was given to reusing existing data areas, and allocating multiple areas with the same function call.

3.2.7.1. Job entries

Every time one of the workers needed to pass an operation to the next one, a new "job" structure was allocated. After being processed and passed on to the next worker, it could safely be released again. This was an easy solution to ensure that there was no memory leaks.

The downside of this was that every step in the pipeline required one memory allocation. To keep the reference counting correct, each step also had to both increment and decrement the counter. Memory allocations and atomic operations are both expensive, so there was room for improvement.

Many of the fields used in some of the workers were identical, so instead of passing some of the fields and creating a new job object, the previous one was simply reused. Step by step the handovers were changed, finally resulting in the builder, the connecter, the encoder, the pdu builder and the pdu logger all using the same job_t structure.

The reader, parser and writer kept their own data structures, since they were too different from the other ones. For example, the data that the reader sends to the parser, does not have a one-to-one relationship to a message and therefore not to the job objects used by the workers later in the chain.

Every time a worker was modified in this way, decreasing the number of memory allocations, performance improved.

3.2.7.2. calloc vs malloc

Callgrind showed that memset() was frequently called, every time by calloc(). Some of the job structures were allocated using calloc(), even though all of their fields were set immediately afterwards. Some of them also included a data buffer used by a vbuf_t, in order to avoid an extra memory allocation. When such a structure was allocated with calloc(), it resulted in memset() being called for a 4KB data area, which would never be accessed before being written to anyway.

Workers based on poll() used an array of struct pollfd entries. Not only were these arrays reallocated on each iteration, they were also allocated using calloc(). These workers were all rewritten to use malloc(), to reuse the area as long as possible, and to reallocate the area only when it needed to grow.

3.2.7.3. Combined effect

The above mentioned changes pushed the performance for the pipe version up almost 20%. While the classic version was still 15% faster for up to 4 connections, the pipe version

was slightly faster when using 64. However, the classic version pushed the cpu load up to 45, while the pipe version stayed at 5 for achieving the same performance. At this point the number of allocations in the pipe version was just 19 per message, almost the same as for the classic architecture. According to Callgrind, this put malloc() at position 4 and free() at position 7. The function that appends data to the data buffer structure, had moved up to position 2.

In the top 12, there is actually only two functions doing any real work. One is the function that appended data, and the other one extracts fields from the incoming SMPP pdu's, at position 12. The other 10 are various sorts of memory management: allocations, initialisations, and locks. This shows how much effort was still required for administration.

A reasonable assumption is that optimal performance is reached when the program does as little as possible except deconstructing and reassembling pdu's, using a minimal number of calls to functions that are as efficient as possible. After years of tweaking, the functions are quite efficient now. The best performance can therefore be achieved by designing the system to keep them at the top. More specifically, the system should definitely not spend a majority of the resources managing memory and locks.

3.3. Throttling

The pdu builder is responsible for making sure that the outgoing connector queues do not become too large. The idea is that all clients will use some sort of limitation of their transmission windows. By delaying the responses, it's possible to both enforce a certain submission rate, and get clients to slow down even more when the queues grow too much. With less incoming traffic, there are more resources left for the outgoing connectors, letting them process the entries in the queue in peace. When the queue shrinks, more incoming traffic can be allowed again.

In the original version, doing this throttling is simple. The thread just needs to sleep for a little while before continuing. This delays the response without affecting any other connection.

For the pipeline version, that method does not work. The same threads are used for both incoming and outgoing traffic, so there is no place to put such a pause without also slowing down the outgoing traffic.

Instead, the response object has an additional field "send_after". When needed, this field is set to the next second. When building the responses, this field is examined. If the set time is not reached yet, the job entry is put back at the end of the queue. This allows any other entries to be considered before this one is tried again.

The same worker is used for pushing both outgoing messages and replies to incoming messages, so the job queue would not be empty at this point. Because of this, the same entry wo not be tried again and again. The only thing that could make this queue empty is if there are no outgoing messages on the connector. However, in that case there would be no reason

to delay the response.

3.4. Code size and complexity

There are surprisingly many similarities between the workers in the pipeline version. If this project would be expanded to a real product, it may be a good idea to use C++ instead to be able to take advantage of these similarities, for example by using base classes for the workers.

1. All workers need an “init” and a “shutdown” function, to create and release, respectively, the data structures they needed.
2. Within their main function, they all start by checking if the program is about to terminate. If not, they call a function which does the actual processing.
3. Both the “reader” and the “connecter” use an internal socket to be notified of new entries. This was natural, since they both hang in poll().
4. All other workers use Posix conditionals to wait for new events.

The classic version always waits in poll() on a single socket, so it has a common helper function which initiates the input parameters and handles any return values which indicates that the call should be retried. Similar helpers exist for read() and write(). The pipeline version uses more advanced algorithms and a more fine grained notification system, so each worker performs their own call to these functions.

3.5. Tracing a thread

For both architectures, it was necessary to be able to follow the execution of each thread, in order to verify that no thread neither paused for too long or started spinning uncontrollably. With the classic architecture, this was easy. Since each thread corresponds to a specific connector instance, most functions print the name of the connector and the instance number as part of the log message.

For the pipeline architecture, each line instead contains the thread id, as returned by pthread_self(). The connector instance information is still printed as well, making it possible to trace both threads and connector instances in the same log file.

The classic version benefits from this addition as well, solving two problems with the log file. This proved to be so useful, it quickly got backported to the original EMG.

1. The protocols that use two threads per instance can not be separated in the log file.
2. Some functions deep within the I/O layer do not have a reference to the connector instance at all, making those log entries difficult to pair with the correct instance.

3.6. Testability

A system which is constantly being developed, adding new functionality and refactoring

existing code, requires having automated test suites. Two levels of testing are currently used.

3.6.1. Unit tests

For a function to be testable, it should be as independent as possible. In the simplest case, it makes no outgoing function calls of its own. In this case, the test program only has to create the data to be sent to this function as parameters, and then examine the returned result. Outgoing calls to simple functions for dealing with strings, hash tables and so on, seldom cause any problems.

In the original architecture, with one thread per connection, the natural solution to the required program logic resulted in rather large functions. When receiving a new message, it first had to read the data from the operating system, which with buffering, timeout handling etc, became a rather complicated and deep function call stack. Testing this part separately is therefore almost impossible. All protocol implementations do their own reading, but even if the actual calls to poll() and read() are refactored to common helper functions, there are still error codes and timeouts to deal with. Any adjustments in the I/O code result in all protocols having to be updated.

In the pipeline architecture, not only are all calls to the operating system in the same place, just as before, but the reader worker can also handle all return codes itself. The protocol drivers themselves are only given new data when it has been read. Because of this, testing a protocol driver no longer requires writing mock versions of the data reader, but can instead be done by simply feeding the driver the necessary data, and checking the resulting message object and the data it wants to return.

Likewise, there is no need to involve the protocol drivers when stress testing the connectivity code.

3.6.2. System tests using Valgrind

Some tests are best performed on a full system, since they are easiest to describe as “if the configuration includes setting X, and it receives a message containing A, the outgoing message should contain B”. The entire application is started and shut down for each test, to ensure that the functions for opening and closing connections work as expected.

Not only should the output be correct, there must also not be any memory corruption issues. Since the application is written in C, there is a high risk of memory leaks, memory which is accessed after it has been released or before it has been written to, etc. The tool Valgrind, only available in Linux, is used to check for these errors.

Valgrind traps every memory access, printing a warning for each error. This includes a full stack trace of where the error occurred. When accessing released memory, the stack trace for that operation is also shown, making it easy to find the root cause for the bug.

Valgrind not only traps the memory accesses, it also simulates the threads. However,

there is a limit of the number of threads it can handle, set by default to 50. The limit can be increased by modifying the source code, but it does not really scale too high, so the Valgrind authors discourage that. One of the test setups for the original EMG make use of almost twice that amount, which makes it cumbersome to run with Valgrind. With the pipeline architecture, this would not be a problem since the number of threads remain the same regardless of the number of connections.

3.7. Graphs

3.7.1. Description

In the graphs below, “EMG” means the classic architecture, and “PMG” the pipe version. For each of them, two configurations were selected. The “info” level is with minimal logging and window size 10, giving best possible performance. The “debug2” level is the development level, with full logging including pdu logs, and window size 1, giving worst possible performance. Any other log level and window size would fall between these two extremes.

3.7.2. Raw performance

The “Performance” graph shows the raw performance, or how many messages per second was processed. With few connections, the classic architecture wins by a large margin. However, when the number of connections increases, this difference disappears.

The test with connecting emgload directly to emgsink showed a theoretical maximum of about 60.000 messages per second, which is the level reached here. The difference of about 10%, both for 64 connections and for 2-4 connections using EMG, can be explained by other processes running on the machine, and the fact that Linux is not a realtime operating system.

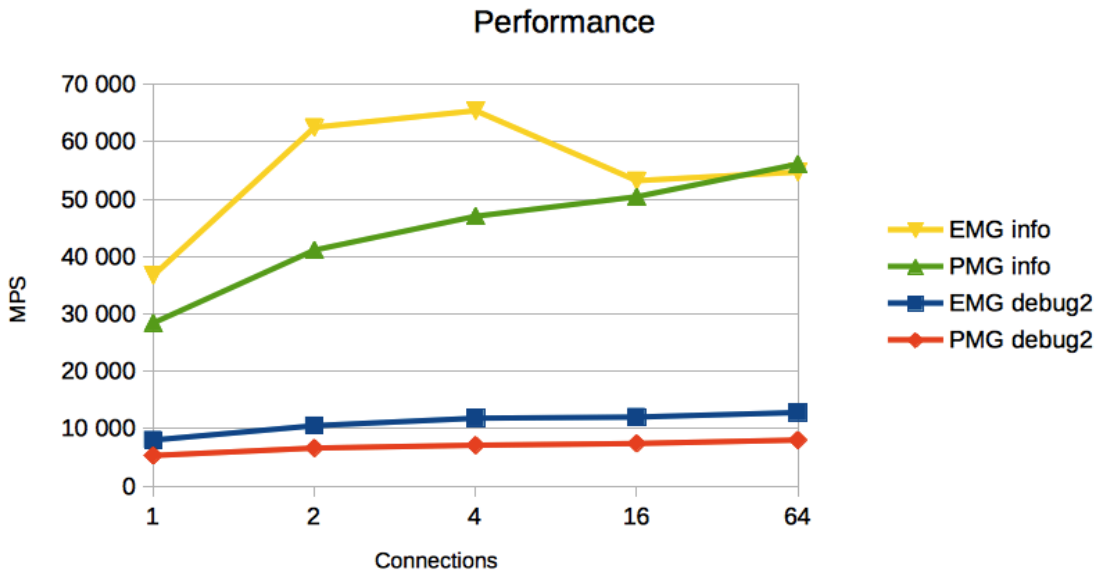

Fig 3.1, throughput

3.7.3. Performance per cpu load

The “MPS / load” graph shows the performance divided by the cpu load reported by the operating system. This shows clearly how much better the pipe architecture scales.

For EMG, the cpu load is almost the same as the number of connections. All threads want to run, but their work gets lost in the overhead. For PMG, the load never goes above 5.

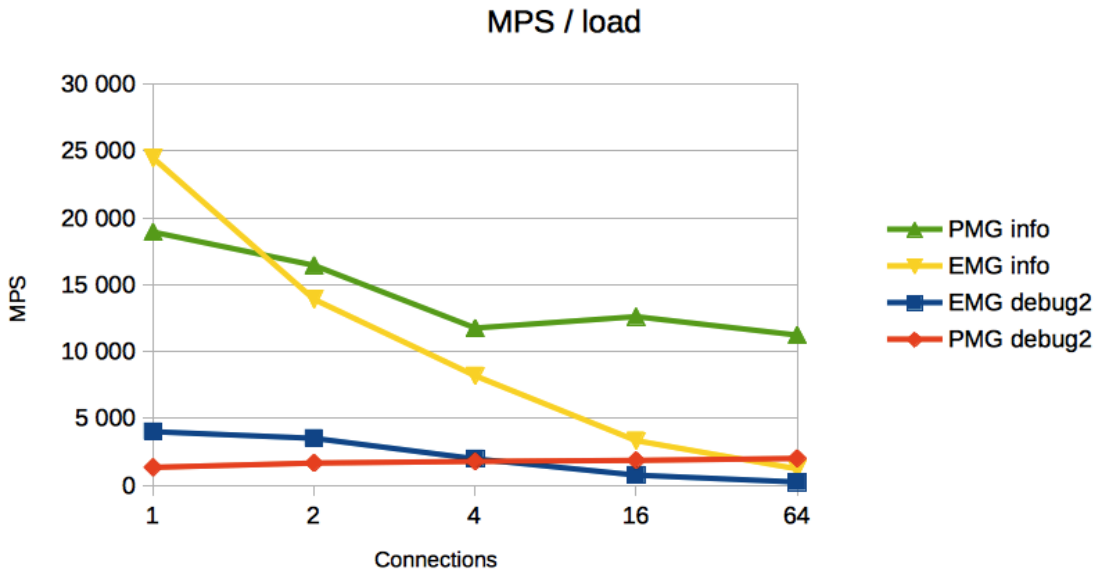

Fig 3.2, throughput divided by cpu load

3.7.4. Performance per cpu usage

The “MPS / cpu usage” graph shows the performance divided by the cpu usage by the application. For EMG, the lower values can be explained by a larger overhead for the higher number of threads. For PMG, the higher number of connections help keeping all workers busy, which increases efficiency. Since the amount of actual work per message is the same for the two versions, it’s unlikely that the graph would continue much higher given even more connections.

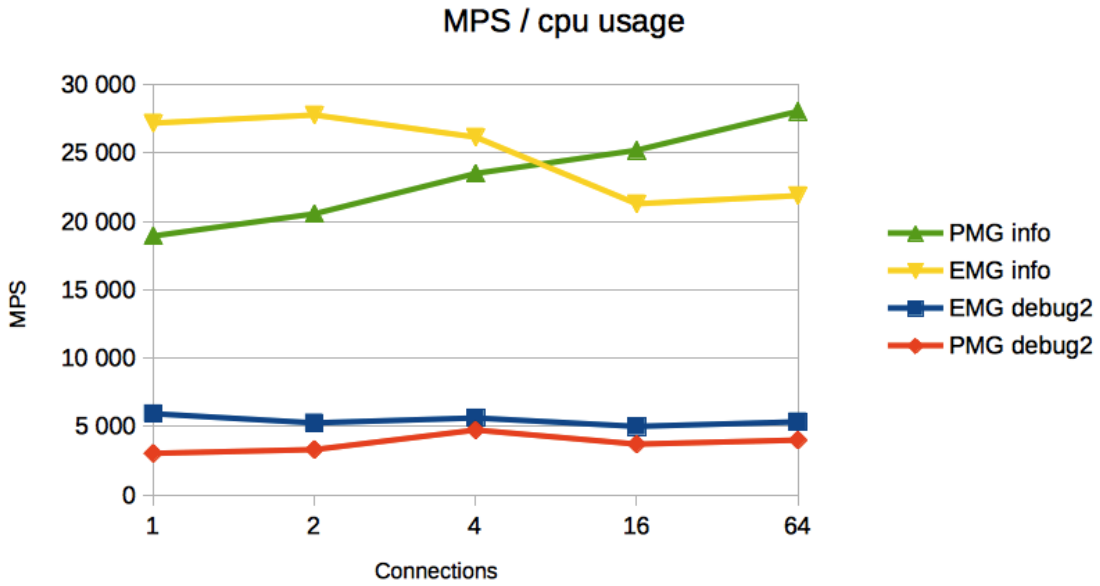

Fig 3.3, throughput divided by cpu usage

4. Discussion

4.1. Limitations

A number of additional tests were considered, but skipped in order to keep the project from growing too much.

1. Running emgload and emgsink on a different machine to spread the load, letting the tested application run at full speed.
2. Using Solaris or Mac OS X.
3. Implementing other protocols than SMPP, to see if they would behave differently.
4. Adding workers which performed database accesses.
5. Testing communication on several parallel connectors.

Since much of the source code is taken directly from EMG, it's not possible to make the source code for this project publicly available.

4.2. Real world considerations

To avoid denial of service attacks such as Slowloris [13], the connector instance can be removed from the list of sockets to poll until the previous data has been parsed. Unfortunately it would mean having to take and release at least one extra mutex for each read operation, slowing things down. It's probably better to keep reading all available data, taking advantage of the sliding window used by the client.

4.3. Future work

There are a number of further improvements that can be done at this point. First, of course, the items mentioned in "4.1 - Limitations". It would also be interesting to see the structure of the source code that would result by implementing the two versions in another language, and what performance they would have.

Even if it is possible to use a common parser for all HTTP based protocols, and perhaps even handle both HTTP and SMTP by the same code, this is not as easy with binary protocols such as SMPP. If may be an interesting experiment to devise heuristics for this. In reality, the users of a gateway application like this will know which protocol each client will use, and it's easy to make the use separate ports for different protocols.

There is a small delay between when a reply has been read and the next message is sent, since both versions wait for that reply before they try to encode and send another message. The pipe version could possibly be improved here, by letting it extract and encode an extra message and let it pause just before the pdu logger worker. This could improve the throughput when using a small window size.

Letting the pdu builder handle throttling is not optimal, because the entry will be logged

to the pdu log long before it's actually encoded and sent out. There are no other workers involved from the point where the program knows that it has gotten a message and where it should be routed, until the response is encoded and written. Fixing this would require adding another worker before the pdu logger. This would allow a more complex solution, perhaps using a balanced binary search tree sorted on the "send_after" value, only for those entries where it was actually set. For a production system it may be worth it, since it gives more exact control. For this project it's enough to be able to limit the queue sizes, and the current solution is good enough for that.

The message queue is currently stored in a balanced binary search tree, more specifically an AVL tree [14]. This allows messages to be sorted on priority, delivery time, etc. It could be worth investigating other data structures, since in reality, the list is mostly linear. With large queues, the cost for finding the next item is always O(lg n), while it really should be O(1). Eric Domaine calls this "Sequential access property". If a connection to an SMSC goes down while a client performs a bulk submission of lots of messages, the queue can contain up to a million messages. The extra "lg" cost then becomes significant.

Some of the solutions used in this project, for example the reference counting and building a protocol object for the replies and not just their encoded version, may be used to guide further development of EMG.

4.4. Conclusions

The differences between the two versions turned out to be even larger than anticipated. At the same time, the amount of log data was almost identical.

The classic version got simpler code for many tasks, since it could just keep working in the same thread until it was done. The pipeline version instead had to create good handoff points between the workers, which required a more detailed analysis of what actually had to be done. As could be seen with the "dispatcher" worker, the different architecture also required a larger number of complex algorithms to simultaneously handle data belonging to multiple connections.

The large number of locks and related issues in the original EMG is most likely due to the extensive functionality, since in this stripped down version, only a few locks remained. The pipeline version needed more locks since there were more modules that needed to synchronise their work. Most locks only involved two or three threads, so it may be possible to replace them with other, more efficient, techniques. Problems with a limited number of actors tend to have simpler and faster solutions than the generic case.

Using multiple threads for processing incoming messages apparently is not necessarily faster than doing them all in a single thread, one by one. A reasonable change to EMG could therefore be to unify the reader threads the way the pipeline architecture did, while keeping the writer threads as is. This would cut the number of required threads in half, and simplify the handling of logouts and disconnects.

As expected, the cpu load was almost linear with the number of connections in the classic version, but maxed out at 5 for the pipeline version. This validates the hypothesis that the constant number of running threads will result in a more consistent CPU usage.

The optimal version is probably the pipeline architecture but with fewer workers, minimising the overhead for each message. The results are summarised in the table below. The throughput is measured as the number of messages per second.

		Baseline	Pipeline
Few (2) connections	CPU load	low	low
	Throughput	44900	27100
Many (64) connections	CPU load	high	medium
	Throughput	46100	40400
Code complexity		medium	high
Testability		low	medium
Overhead per message	Memory allocations	15	19
	Number of locks	40	35

Table 4.1, summarised results.

1. EMG, http://www.nordicmessaging.se/products/emg/enterprise-messaging-gateway.html
2. Valgrind and Callgrind, http://valgrind.org/
3. Dan Kegel, The C10K problem, http://www.kegel.com/c10k.html
4. Emgload and emgsink, http://www.smpp.com/smpp-benchmarking.html
5. CMake, http://www.cmake.org
6. Git, http://git-scm.com/
7. Clang, http://clang.llvm.org/
8. SMPP, http://docs.nimta.com/smppv50.pdf
9. Pattern recognition, http://en.wikipedia.org/wiki/Pattern_recognition_%28psychology%29
10. Michael Kerrisk, *The Linux Programming Interface (http://man7.org/tlpi)*, 2010
11. Libev, http://software.schmorp.de/pkg/libev.html
12. GCC Atomic Builtins, https://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/Atomic-Builtins.html
13. Slowloris, http://ha.ckers.org/slowloris/
14. AVL trees, Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 458–475 of section 6.2.3: Balanced Trees.