Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

CMQ - A lightweight, asynchronous high-performance messaging queue for the cloud

Joerg Fritsch[1*] and Coral Walker[2]

Abstract

In cloud computing environments guarantees, consistency mechanisms, (shared) state and transactions are frequently traded for robustness, scalability and performance. Based on this challenge we present CMQ, a UDP-based inherently asynchronous message queue to orchestrate messages, events and processes in the cloud. CMQ's inherently asynchronous design is shown to perform especially well in modern Layer 2 switches in data center networks, as well as in the presence of errors. CMQ's lightweight edge-to-edge design, which is somewhat similar to Unix Pipes, makes it very composable. By presenting our work, we hope to initiate discussion on how to implement lightweight messaging paradigms that are aligned with the overall architectures and goals of cloud computing.

Introduction

Cloud computing has two perspectives: first, an outward-looking perspective that embodies an elastic application executed in a secure container and accessible over the internet, as seen by developers and end users; Secondly, an inward-looking perspective that describes the large scale distributed cloud computing platform and its middleware as implemented and operated by the provider [1]. CMQ presents a message passing model (that is a middleware abstraction) implemented in Haskell that addresses the reality of both perspectives.

The inward-looking perspective

The physical cloud nodes in computing clouds are organized into "Points of Delivery"[a] and interconnected via equipment that either switches at line rate, or that uses lossless Ethernet fabric[b] technologies. In switched data center networks, all Ethernet (RFC 894) based networking protocols are switched without discrimination. Congestion and packet loss are extremely unlikely in such data center networks. The overhead of TCP/IP in 10Gbps data center fabrics has led to CPU performance issues ([2-4]) and has given rise to new connectionless Ethernet protocols, such as RDMA over converged Ethernet (RoCE) and the Internet Wide Area RDMA Protocol (iWARP). However, both protocols require specialized hardware (network cards, switching gear) that is not in line with the trend to build clouds from commodity hardware [5], and accept occasional failures rather than preventing failure at any cost [6].

In CMQ, UDP is used as the transport protocol for the following reasons:

- It is connectionless.
- It is the protocol that adds the least overhead to the ethernet network: it adds only 28 bytes overhead to every packet (20 byte IPv4[c] header + 8 byte UDP header).
- It is accessible to guest systems on hypervisors and clouds and is readily available to (Haskell) developers via standard modules.

Above all, the design of the UDP protocol fits the above-mentioned notion of cloud computing well, that is, to accept occasional failure and manage it, rather than struggling to prevent it.

The outward-looking perspective

The guest systems in clouds often have to cope with the suboptimal network conditions caused by software devices, a problem that the VEPA[d] standard tried to solve in 2009. The software devices, such as vswitches and

*Correspondence: J.Fritsch@cs.cardiff.ac.uk
[1]NATO CI Agency, Den Haag, Netherlands
Full list of author information is available at the end of the article

vrouters, are responsible for regulating network traffic inside the cloud nodes and are guest systems themselves. Depending on the virtualization ratio, one virtual switch could be responsible for up to 64 guest systems. Guest systems frequently have to cope with packet loss [7] that, when using TCP/IP costs many CPU cycles on systems that are themselves billed according to the available CPU cycles. Packet loss in TCP/IP can easily cause guest systems to grind to a halt.

Furthermore, ubiquitous computing is becoming increasingly important and prevalent: according to [8], "7 trillion wireless devices [will be] serving 7 billion people by 2017". Considering that packet loss is very common in radio transmission wireless networks, reliable network transmission protocols such as TCP suffer undesirable performance reduction due to the congestion avoidance algorithm used in TCP, while protocols based on UDP and the like, with optimized data transmission and performance advantages, are becoming more attractive for mobile devices that experience significant packet loss. Therefore, it is reasonable to assume that future (cloud) services, most of which will be dependent on one of the 7 trillion wireless devices, require protocols that are significantly better than TCP in the presence of errors.

Related work

According to [1], "the cloud demands obedience to [its] overarching design goals", and "failing to keep the broader principles in mind" leads to a disconnection of cloud computing research from real world computing clouds. Furthermore, scientists "seem to be guilty of fine-tuning specific solutions without adequately thinking about the context in which they are used and the real needs to which they respond". One overarching design goal however is to avoid strong synchronization provided by locking services. Wherever possible, all building blocks of a computing cloud should be inherently asynchronous. CMQ, being designed to meet the real needs of cloud computing, is strictly asynchronous and is the combined research result from many different research fields, including network- and data- center design, network protocols, message oriented middleware and functional programming languages.

UDP Protocol

The increasingly wide adaptation of the UDP protocol indicates the suitability of the UDP protocol as an efficient transport protocol for supporting distributed applications. For example, UDP is used for data transportation in Network File System (NFS) and for state and event transportation in Massive Multiplayer Online Games (MMOGs) [9,10]. EverQuest, City of Heroes, Asheron's Call, Ultima Online, Final Fantasy XI, etc. are among many MMOGs that use UDP as its transport protocol. The fact that MMOG applications are by nature of large, but elastic scale make them ideal customers for IaaS and PaaS offerings. By using cloud computing and storage facilities, not only cost and risks, that are usually linked to building new MMOGs, reduced [11], but also over-provisioning MMOG hardware to be on standby for peak times will be avoided [12]. However, whether computing clouds can fulfil the stringent real-time requirements of MMOGs is still an open issue [13].

In contrast to MMOGs that are largely event driven where the size of individual messages is expected to be small [14], NFS is data driven with larger packet sizes and higher throughput. NFS, according to [15], the most successful distributed application ever, has been using UDP as underlying transport protocol for more than two decades and was a stateless protocol up to NFSv3[e]. Compared with a large-scale cloud environment, NFS is arguably designed for a limited scale. The UDP based Data Transfer Protocol (UDT), described in [16], has showed the applicability of using the UDP-based UDT protocol for "cloud span applications" and won the bandwidth challenge at the International Conference for High Performance Computing, Networking, Storage, and Analysis 2009 (SC09). Furthermore, [17] also discovers that reliable transport protocols that outperform TCP transport protocols can be designed in the basis of UDP.

Message Oriented Middleware (MOM)

If we view computing clouds from the inward-looking perspective mentioned above, it can be seen that the cloud framework itself is a distributed application that in turn supports distributed guest applications, for the reasons listed below.

- Computing clouds have an inherent distributed character.
- The cloud framework enables elasticity, and parallelization (through distribution) of guest applications. The guest application should be distributable so that it has the flexibility to be distributed to other resources when the limits of the current available cloud resources are reached.

The relevance of message passing for computing clouds stems from the distributed programming model that is chosen to code either the cloud platform or the guest application. [18] sees the Actor concurrency model [19] as the foundation of cloud computing. The Actor model enables "asynchronous communication and control structures as patterns of passing messages" [20]. Two well-known implementations of the actor model are e.g. the functional programming language Erlang [21] and the Akka toolkit [22].

Whilst a UDP message queue for Actors is a new idea, UDP-based MOM (Message-Oriented Middleware) is not. The open source Light Weight Event System (LWES) [23] is a UDP-based MOM that is described as having a strong position in large scale, real-time systems that need to be non-blocking and is also described by Yahoo! as part of US Patent 2009/0094073 "Real Time Click (RTC) System and Methods". LWES is also described as being useful (for transporting large data to computing nodes) for parallel batch processing with Hadoop [24], which is an open source implementation of Google's Map Reduce [25]. In fact, MapReduce and Hadoop are posited, in [26], as the right cloud computing programming models.

Functional programming languages

The functional programming language Haskell is chosen as the programming language to implement CMQ because of the following reasons:

- It is independent from any third-party platform or runtime (e.g., Clojure and Scala are built on top of JVM, F# on top of the .NET platform).
- It is being actively researched and has an ever increasing large research community. According to the popularity tracking website langpop.com [27], in 2011, it was ranked fifth out of the 32 most talked-about programming languages on the internet.
- It supports a wide range of concurrency paradigms [28].
- It is very powerful in list manipulation. Lists are used in CMQ to provide lightweight data structure that holds messages in sequence. List manipulation is useful to implement selective receives of messages with defined characteristics (rather than accepting messages in FIFO sequence) in an Erlang-like fashion.

With regard to the question whether functional programming languages are at all the right tool for the implementation of CMQ, we share the opinion stated in [29] that it is best to start with a programming language "whose computational fabric is by-default parallel" and that in the future "parallel programming will increasingly mean functional programming". Notably, MapReduce and Hadoop are frameworks that are eventually based on the functions map and fold (aka reduce) of functional programming languages.

Cloud Haskell, proposed in [30], aimed to further develop Haskell as a programming language for developing distributed applications. It was influenced by Erlang, and was intended to provide support for the actor model, message passing, and the mobility (with limitations) of functions with co-located data (closures). Coutts states [31] that protocols, as the centre of distributed systems, are playing a main part in the future development of Cloud Haskell. A proprietary protocol suite and protocol flexibility are among the considerations in the future of Cloud Haskell.

Although CMQ shares a similar goal to Cloud Haskell of providing a mechanism for distributed applications, it has adopted a different approach. Cloud Haskell, targeting language-level support for distributed applications, explores a lower, compiler-level implementation, while CMQ, is intented for a higher-level support. CMQ takes advantage of the current Haskell language to explore an implementation at the protocol level and above. The benefit of a higher-level implementation is the flexibility that the approach used in CMQ, since it is language-independent, can be easily adapted to other languages and environments, and thus serve the ultimate goal: finding appropriate communication approaches for cloud computing and the ubiquitous computing paradigm.

CMQ implementation

CMQ is a lightweight message queue implemented in Haskell. CMQ provides a polymorphic data type Cmq a where a is the content type of the queue. CMQ has currently three primitives: newRq (to initialize the queue and data structures), cwPush (to push a message into the queue), and cwPop (to pop a message from the queue). The code is published on github.com and available at https://github.com/viloocity/CMQ.

cwPush

Messages for remote processes are identified by a key tuple consisting of the IP address of the remote system and an integer which is reserved for future use, for example, it can be used to specify the PID of the remote process. When a message is pushed with cwPush two things happen:

- The key-tuple and the data are stored in a map, implemented using the Haskell library Data.Map (a dictionary that is implemented as a balanced binary tree).
- The key-tuple and the creation time are stored in a priority search queue (PSQ), implemented using the Haskell library Data.PSQueue [32] and used as a pointer to the corresponding binding in the map.

Figure 1 shows how CMQ works on the side of the sender. When cwPush is called to push a new message, a key-tuple k is built that consists of the IP address of the destination and a unique identifier (i.e. the PID). If the given key is not already a member of the PSQ, then a new binding (k, p) is inserted where the priority p is the creation-time of the binding. At the same time a new key-value pair (k, a) is inserted into the map, where a is a finite

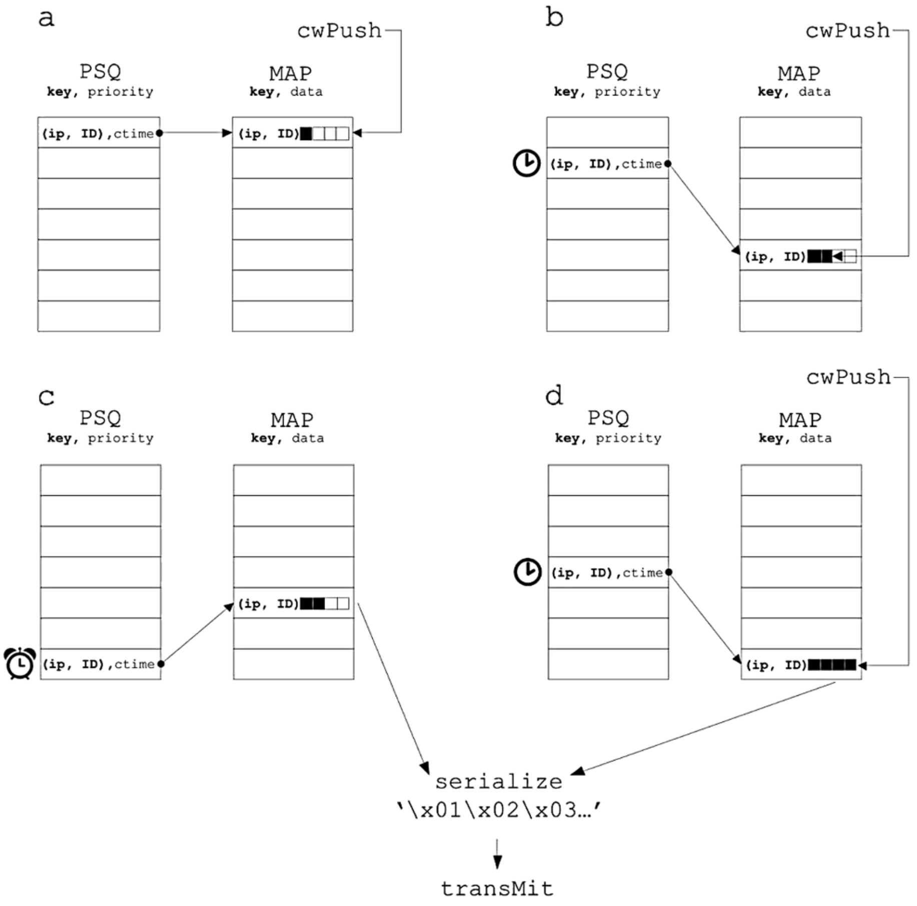

Figure 1 Map and PSQ pointer in CMQ. cwPush is called when the key for the recpient process is not present **(a)**, cwPush is called when the key for the recipient process is already present **(b)**, the timeout for a key has been reached **(c)**, cwPush is called when the key for the recipient process is present and the data length amounts to qthresh **(d)**.

list that contains the pushed messages (Figure 1(a)). Message queues are stored in the map structure and the map structure stores key-value pairs. The value of each key-value pair is a reference to a separate queue for a specific destination process.

If at the time when a new message is pushed its key k is already a member of the PSQ, the new message is appended to the end of the queue that corresponds to k (Figure 1(b)). When the total amount of messages in a queue (the gross length of all messages) for a specific key-tuple exceeds a set threshold (qthresh) then the whole queue will be serialized and transmitted to the recipient (Figure 1(d)). In order to ensure that messages only stay in the queue for a short time, a timeout threshold is used. No matter whether the data threshold qthresh is reached or not, once the timeout threshold is reached, all the messages in the queue will be serialized and sent once the timeout threshold is reached (Figure 1(c)). The function sendAllTo from the Haskell library Network.Socket.ByteString is used to bring the UDP datagrams onto the wire. The function sendAllTo guarantees that all data is successfully brought onto the wire and that there were no errors on the local network interface.

Since CMQ is implemented in a pure functional programming language (Haskell), and pure functional data types are immutable, updating a node by writing directly to memory is not supported. The actual appending operation (++), which appends a new message to the end of a queue, does not update the tail node by changing its pointer so that it points to the new added message node, but recreates recursively each node in the queue, so that instead of writing a small node and a pointer to memory, the function returns, a complete new queue with the newly-added message returns [33,34]. This operation takes O(n) time.

It is observed that, while the appending operation has time complexity O(n), adding a new message to the head of a queue, by consing (cons :) the new message node directly to the head of queue, takes O(1) time. So an alternative method for the appending operation is to add a new message node to the head of a queue instead of the tail. The queue created using such a method maintains a reverse ordering of a FIFO queue. Before transmission of a particular queue, a reverse operation is performed on the queue to reverse the queue back to its normal FIFO form. The reverse operation takes O(n) time.

cwPush is implemented using two parallel threads, where thread1 enqueues messages and checks the total amount of messages; thread2 surveys whether the timeout for a particular queue is reached. By using time profiling (see section on Messaging passing performance) it was discovered that thread2 was very costly and could use up 70% of the CPU time. As a result, a function called threadDelay was introduced to control and limit the maximum number of times that the PSQ is checked.

From the above discussion, we see that CMQ can be tuned using two parameters: qthresh (the maximum amount of messages in bytes allowed in the queue) and the timeout (the maximum waiting time a message stays in the queue before it is sent).

All map queries that are used in CMQ, including insertion and deletion, have a complexity of O(log n). A function called findMin is used to check the PSQ for any queues that have exceeded the timeout threshold. The findMin function is implemented with a complexity of O(1), which is an attractive feature, considering it is one of the most frequently used functions.

An alternative design solution is to use a more conventional method. Such a method, instead of using a map data structure with a PSQ as pointer, uses a sequence [35] of tuples (creationtime, message-queue) with each sequence data structure being responsible for a specific destination of messages. However, this method does not scale well. Although it is possible to examine the right (viewR) and left (viewL) end of a sequence with O(1) complexity, all sequence data structures require identification and organization, which will increase the complexity of queries and insertions to up to O(n) time. Thus, this method becomes inefficient when the number of sequences become very large, which unfortunately is a common case in cloud or large scale computing environments. Aiming for better scalability, CMQ is implemented in a way such that the identification information is maintained in the key k that associates queues with their creation time (in the PSQ) and recipients with their specific queues (in the map). Using the identification information, the system can quickly identify the queue for a newly pushed message. Since all map related queries take O(log n) time and all PSQ related queries take O(1) time, comparing with a sequence based solution, CMQ demonstrates a clear advantage in terms of its efficiency and scalability.

cwPop

On the recipient the serialized data structure with all its messages is received, deserialized, and transferred onto a transactional channel (TChan). TChan is an unbounded FIFO channel implemented in Software Transactional Memory (STM, [36]). Once the messages are transferred into TChan, they are ready to be consumed. The function cwPop is used to pop an individual message from the queue. cwPop is a non-blocking function that examines the TChan to check whether there are messages before attempting to read messages from the TChan. If there are waiting messages they are returned having the type Maybe String whereas in a blocking implementation the returned messages would have the type String. The Maybe type in Haskell represents optional values making e.g. null pointers obsolete. In this case a String can be present in the queue or the queue can be empty.

Whilst in Erlang processes communicate with each other via mailboxes that are identified by the PID of the mailbox owner, in Haskell the preferred method for interprocess communication (IPC) are transactional channels TChan. TChan is created whenever it is needed. It has no dedicated owner and is not associated with any identifiers or addressing scheme. As a consequence, TChan is created by the developer and its identifier needs to be propagated. There have been some attempts to add additional layers of abstraction to TChan to make it work similar to Erlang mailboxes (e.g., Epass [37]) and more applicable to actor-based approaches. The majority of the attempts that are actually working and publicly available work only in local environment. Thus, they cannot send messages to a remote TChan. CMQ removes this limitation by allowing messages to be transmitted to a remote TChan via a CMQ queue.

The use of cwPush and cwPop

Figures 2 and 3 give a simple example that demonstrates how cwPush and cwPop are used in a real application. Figure 2 shows an example of a sender application which sends 10000 messages each of which contains a 4-byte string. The message type can be any Haskell data type that is a member of the Haskell serialize class. The application developer specifies the UPD port number (here UDP port 4711) to create the socket for UDP data transport. Figure 3 shows the example of the receiving application that uses cwPop to retrieve messages.

Where is the queue?

There are two queues involved for every recipient process: the queue stored in the map on the sender and the TChan, which is in fact a simple STM-based FIFO queue on the remote recipient host. However, the detailed implementation is completely hidden from the users, who can see the CMQ message queueing system as a single distributed

```
{-# LANGUAGE OverloadedStrings #-}

import System.CMQ
import Network . Socket hiding (send , sendTo , recv , recvFrom)
import Control . Monad

main = withSocketsDo $ do
        qs <- socket AFINET Datagram default Protocol
        hostAddr <- inetaddr "192.168.35.84"
        bindSocket qs (SockAddrInet 4711 hostAddr)
        (token) <- newRq qs 512 200--initializes the queue with the desired parameters
       --qlength = 512B and max delay time in the queue is 200ms (minimum is 40ms)
       --token is the queue identifier where messages are sent to or poped off
        forM_ [0..10000] $ \i -> do
            cwPush qs ("192.168.35.69" , 0) ("ping" :: String) token --send message "ping" to
           --ipv4 address 192.168.35.69 using the queue specified in token
```

Figure 2 Example of a sending application that uses cwPush.

queue with two functions cwPush and cwPop. The function cwPush is called when a message is needed to be sent to a recipient, and the function cwPop is called when the recipient process reads a message.

Zero Copy vs Functional Data Structures

TCP and UDP sockets need to copy received data from kernel space to the user space and vice versa. iWARP and RoCE address this and use zero copy implementations where the kernel shares buffers with the application rather than copying the data. Although this is not directly addressed in CMQ, there are two interesting aspects worth noticing: first, system calls are expensive and thus the number of send operations (which are systems calls) should be kept to a minimum; secondly, copying data, even in user space without involvement of system calls, is also far from optimal. Instead of invoking a send system call each time when a message is sent, CMQ reduces the number of send system calls by storing messages temporarily in a queue and sending the stored messages when either the total size of messages in the queue meets the size threshold, or the waiting time of the current oldest message in the queue meets the timeout limit. Accordingly, the number of receive system calls on the receiving end is also reduced.

```
import System.CMQ
import Network.Socket hiding (send , sendTo , recv , recvFrom)
import Control.Monad
import Data.Maybe

main = withSocketsDo $ do
        qs <- socket AF_INET Datagram defaultProtocol
        hostAddr <- inet_addr "192.168.35.69"
        bindSocket qs (SockAddrInet 4711 hostAddr)
        token <- newRq qs 512 200
        forever $ do
            msg <- cwPop token :: IO (Maybe String)
            print msg
```

Figure 3 Example of a receiving application that uses cwPop.

However, functional data structures are by definition immutable. When bindings in a Map or PSQ are inserted, deleted or modified, strictly speaking, the returned data structure is not the original data structure but a data structure that is identical with the previous data structure but containing the alteration. The Map and PSQ data structures used in CMQ are pure functional data structures that are immutable, so their insert, update and delete operations involve some degree of copying as opposed to typical mutable data structures where changes are written directly to the memory. To be more precise, a Map or PSQ insertion involves the copy of O(log n) amount of data for a data structure with n elements [38] plus some additional logarithmic overhead [39]. It remains to be shown by future research whether or not a pure lazy language (e.g., Haskell) and its data types can retain the same asymptotic memory use as an impure strict one (e.g., Erlang) in all situations. However, in return functional data structures make it easier to keep multiple modified versions of the same data structure without storing whole copies.

By reducing the number of send and receive system calls, the data copying between kernel and user space is also reduced. Pure functional data structures may on the one hand be a slight drawback in terms of performance, but on the other hand give low cost access to data that needs to be replayed.

UDP as a message passing paradigm

The Haskell benchmarking library Criterion [40] is used for all the tests and a garbage collection was performed after every test. The CMQ testbed was setup in a client-server model where at first the client and the server would alternately send and receive messages similar to the ping pong test of the INTEL MPI benchmarks (IMB) [41]. In order to investigate the benefits of asynchronous message exchange (fire-and-forget messaging) and queuing, CMQ itself was allowed to use asynchronous non-blocking send operations (which means CMQ was allowed to send the next message before a reply to the previous message had been received) similar to the IMB ping ping test.

The testbed consisted of a cloudstack [42] POD implemented on data centre grade hardware (listed in Table 1) analogous to commercial computing clouds. Figure 4 shows a logical diagram of two cloud computing nodes from our POD. The guest virtual machines (VMs) used for CMQ testing were resident on two separate computing nodes. Direct networking based on VLAN tagging is configured between the VMs and the physical networking gear. The VMs on our POD communicate via VLAN ID 2012, which is part of a VLAN trunk terminated on the computing nodes. The cloudstack (CS) virtual router is used to provide DHCP functionality and provide IP addressing to the VMs but, in this configuration, does not actually take part in routing and forwarding of packets.

The code that was used for benchmarking is published on github.com and is available at https://github.com/viloocity/Haskell-IPC-Benchmarks.

Message passing performance

MessagePack and 0MQ

Rather than doing strictly competitive benchmarks, it would be more beneficial to investigate and compare several paradigms. For this reason, MessagePack [43] and 0MQ [44] are chosen in the CMQ performance evaluation. MessagePack and 0MQ are two IPC systems that provide Haskell bindings. MessagePack is a library that is based on RPC and focuses on object serialization. 0MQ provides a framework that focuses solely on message passing and queuing.

MessagePack uses RPC to transfer messages which in fact are all serialized objects, and was initially described as IPC to "pass serialized objects across network connections" [45]. Although the most recent descriptions of MessagePack focus mainly on its outstanding object serialization capabilities, it serves also as a general message passing mechanism. Since every message has to be serialized before it is sent, providing an effective serialization method is one of the major concerns involving message passing.

In CMQ a message queue that consists of multiple messages is serialized before it is sent. The messages used in the tests are composed of only 8-bit ASCII characters and the Haskell library Data.Bytestring.Char8 is used for the serialization. For messages or objects with other character encodings, the Haskell Data.Bytestring library or even MessagePack may be used for the serialization[f].

Mean performance

Figure 5 presents the mean performance of UDP Sockets, MessagePack, 0MQ and CMQ for exchanging 1000 messages with message sizes between 4 B and 16 KB. UDP Sockets are not included in Figure 6 for the reason that the benchmarking application used for UDP sockets supports only synchronous operations in lossless environments. The test results show that when message sizes are less than 1 KB UDP sockets perform comparable to TCP-based messaging queues; when message sizes are larger than 1 KB, UDP sockets outperform all tested TCP queuing methods. As for CMQ, it, in general, outperforms all other messaging queuing methods. CMQ demonstrates a clear advantage for small to medium sized messages up to 4KB. It shows a speed increase of up to 100 times for the transmission of small messages such as integers (e.g. error codes), flags or applications that need only a single request - response [46], since TCP messaging requires the establishment of a TCP connection which would incur a 60% overhead for a small sized message. From the tests it was discovered that CMQ achieves its best performance with a qthresh of 512 B (a value that is also used by the DNS protocol) and a queue timeout threshold of 200ms.

Performance in the presence of errors

TCP is a reliable protocol that provides reliable, connection-oriented delivery of data. It detects for example packet loss, delay, congestion and replays lost packets when required. However, the reliability causes a significant overhead, especially for messages of small sizes. This is one of the disadvantages of using TCP. More seriously, when delay or packet loss are detected, TCP assumes congestion and slows down the rate of outgoing data [46-48] propose formulae to calculate the effective bandwidth of TCP connections in the presence of errors where, for example, a 0.2% packet loss eventually slows down and limits the effective connection speed to 52.2 Mbps irrespective of the nominal bandwidth. In practice, retransmitting packets is very costly since it also involves queuing and reordering packets that arrive until the retransmit is complete, thus stopping time-sensitive data

Table 1 CMQ testbed setup

	Hardware	Software
2 Server	HP DL 580 G5 4 quad core Intel Xeon E5450 @ 3GHz 32GB RAM	Citrix XEN 5.6
2 Linux VMS on separate XEN Server	ARCH Linux 2011.8, Haskell GHC 7.0.4	2 vCPUs @ 2GHz 4GB RAM
2 Fabric extenders	Cisco Nexus 2248TP-E	
1 Core switch	Cisco Nexus 7000	NXOS 5.1(2)

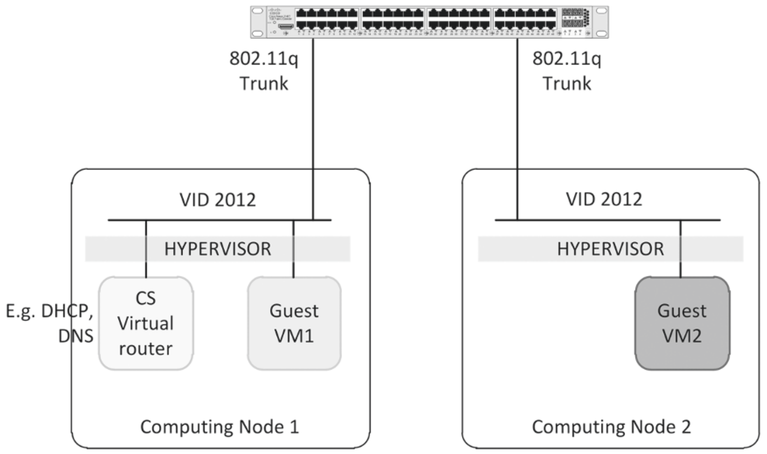

Figure 4 Logical diagram of the testbed configuration.

from going through in the meantime [49]. Furthermore CPU usage spikes when TCP retransmissions are needed and applications frequently become unresponsive.

In the CMQ testbed, where we test CMQ in the presence of data loss, in order to simulate packet data loss, iptables [50] (see command listed below) is used on one of the Linux VMs to drop incoming packets with 1.4% probability:

```
iptables -A INPUT -m statistic –mode random –probability 0.014 -j DROP
```

and it was found that CMQ is largely unaffected and produces the same performance results (within the standard deviations) as if there were no errors on the network. All other benchmarked queuing methods show an overall delay of approximately a factor of 4. It was also discovered that in the presence of errors the benchmark results

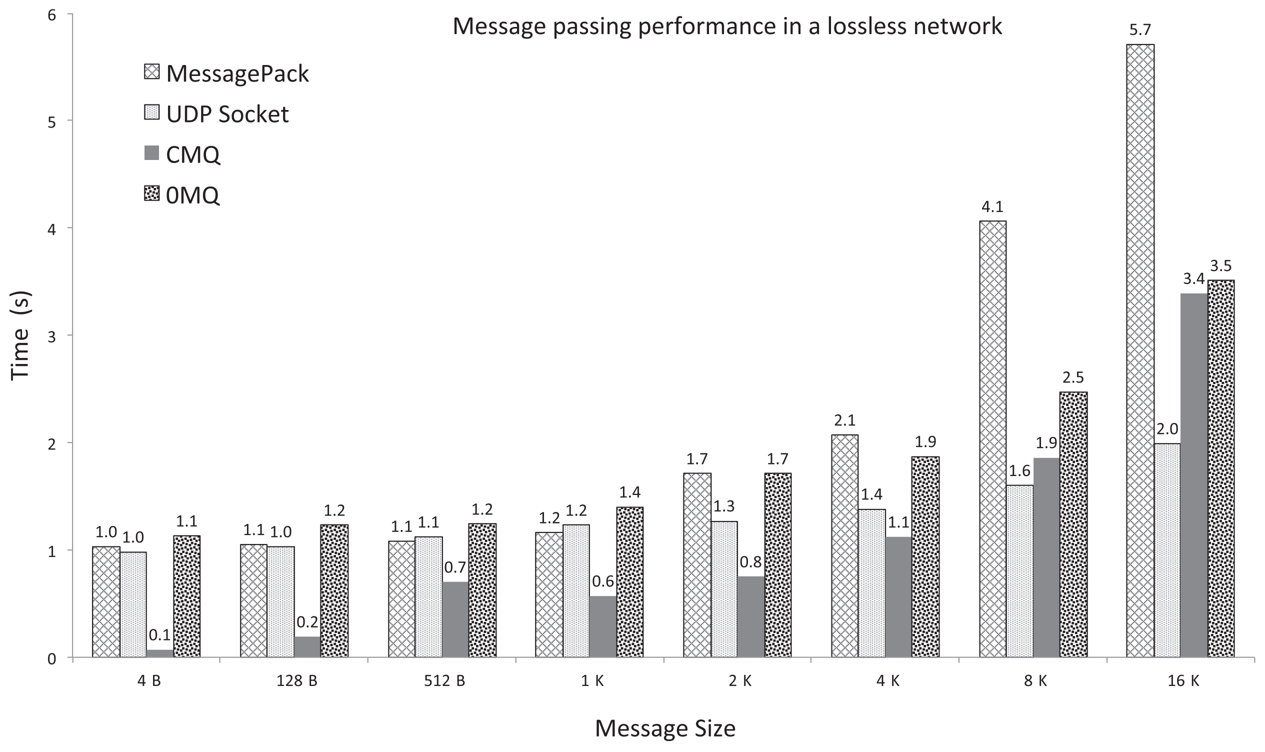

Figure 5 Message passing performance on a lossless data center network.

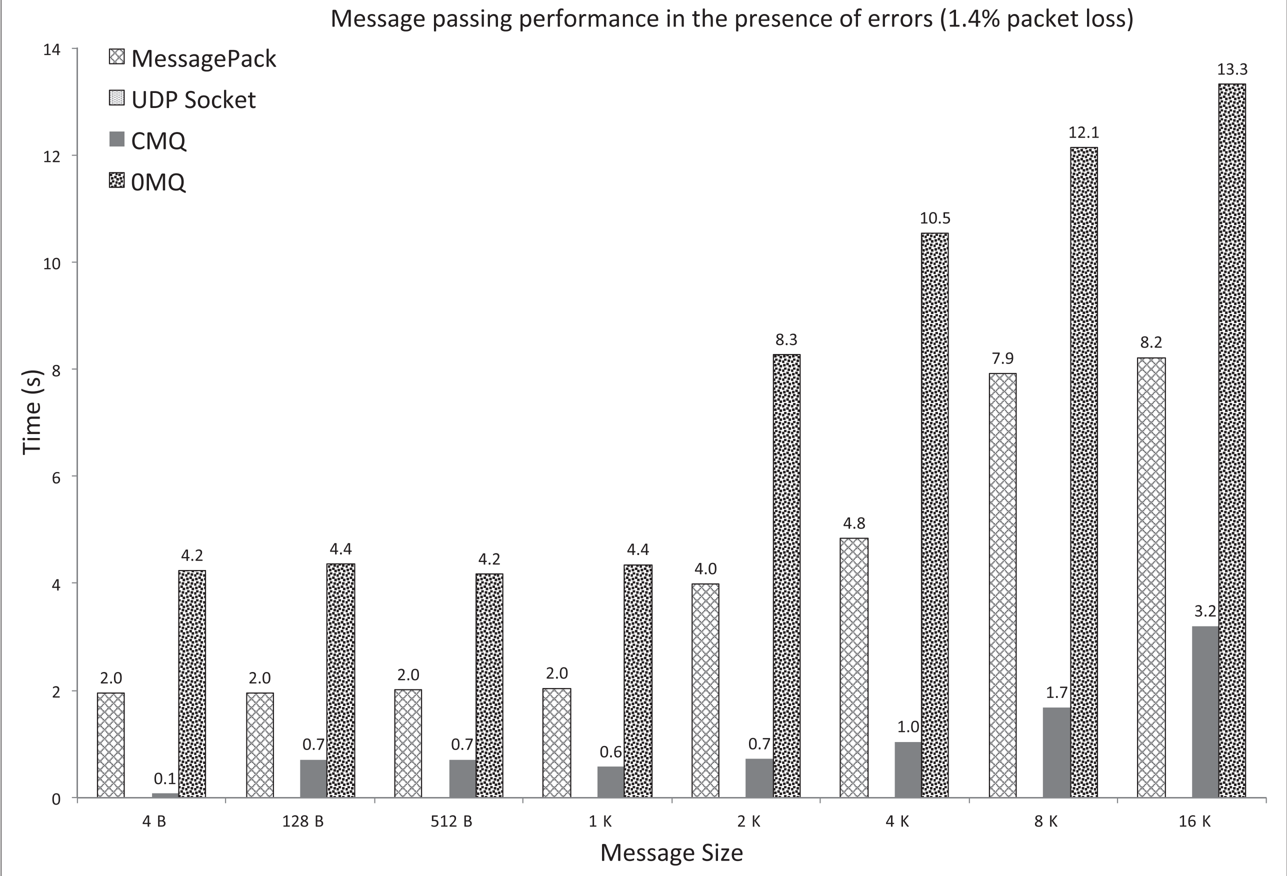

Figure 6 Message passing performance in the presence of errors.

of CMQ still show a narrow standard deviation. For example, for a message of 512 Bytes, the standard deviation of CMQ stayed at 52 ms whilst the standard deviation of MessagePack increased from 54 ms (with no simulated data loss) to 820 ms.

CMQ scalability at large

Birman and Chockler [1] state that currently "Not enough is known about stability of large-scale event notification platforms, management technologies, or other cloud computing solutions" and identifies the development of testing methods that can validate the relevance and demonstrate the scalability of any new solution "... without working at some company [e.g. Yahoo, Google, Amazon] that operates a massive but proprietary infrastructure" as an item on the cloud computing research agenda.

In the absence of established "cloud scale" testing methods, conventional tests and checks were used to examine CMQ and to demonstrate that nothing obvious is limiting its scalability:

- Haskell Program Coverage (HPC) [51] was used to determine which areas of the source code and boolean controls were actually exposed to testing.
- Space (Heap) Profiling was used to investigate whether any cost centre of the application may consume too much memory because of excessive laziness (also called "space leak"). Time profiling was used to identify functions that are possible CPU hogs.
- The criterion test suite was used to do the actual testing and produce additional data about the accuracy and repeatability of tests.

Although the areas of code that executed are dependent on the parameter settings for timeout and qthresh, overall all areas of code were exposed to the testing. The heap profile Figure 7 showed that the memory consumption of 80K is very moderate and we find that the memory consumption related to serialization cost (the band labelled as "PINNED") is the most prominent feature. Although for the performance tests we send messages composed of 8-bit ASCII characters, CMQ is internally built with poly-

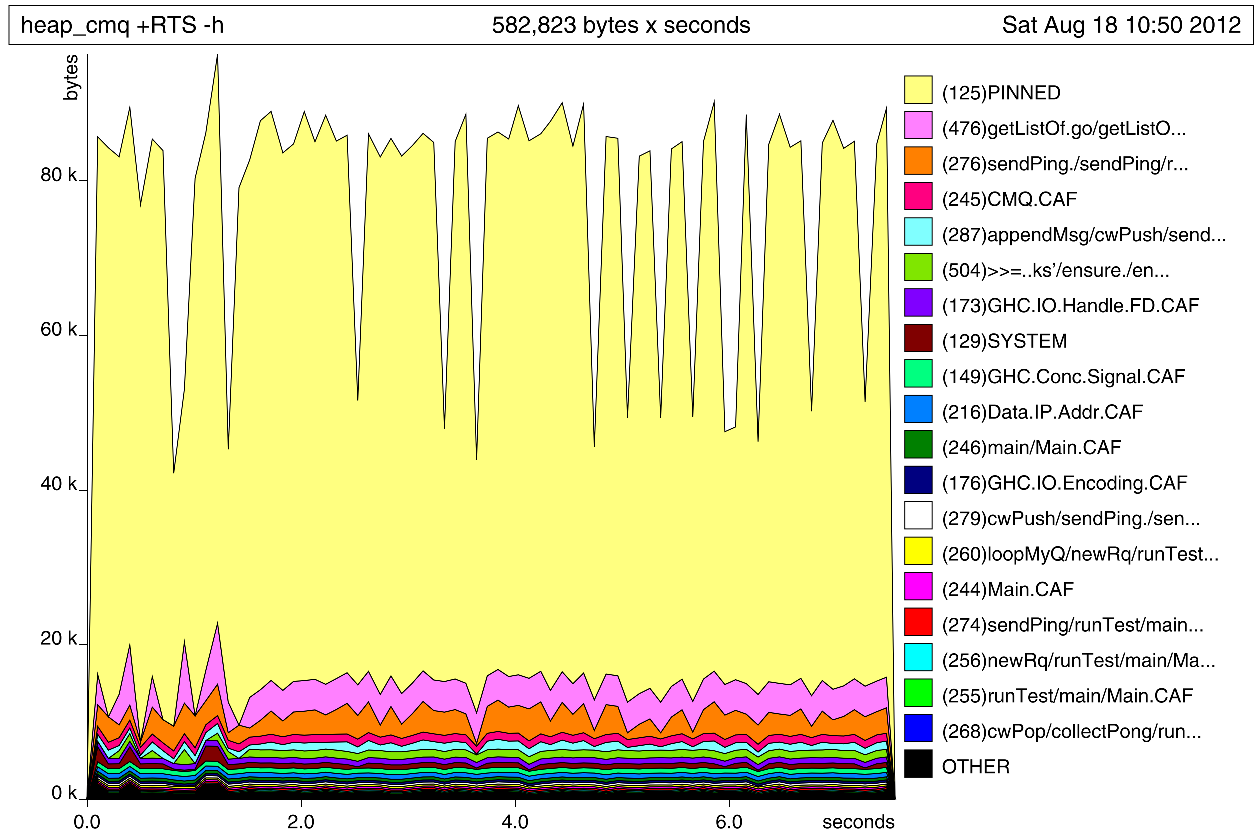

Figure 7 Graph for memory usage on the heap. The heap is split into the 20 most prominent cost centres as inserted by the compiler. qthresh was set to 512K and timeout to 200ms. The maximum length tested was five hours where the pattern could be sustained.

morphic functions and can transfer arbitrary Haskell Data Types under the condition that they can be serialized. In order to achieve polymorphism, CMQ must compare the queue length to qthresh when the queue is serialized, since functions that can determine the length of an ASCII based queue do not fire any more under these circumstances. Thus, more serialization activities are necessary compared to an implementation that would be limited to the data type String. Overall, there was no evidence of space leaks.

In the time profile we observe that half of the time is consumed in the benchmarking application that pops the ping messages from the queue after they have been received - this is a wrapper application for the non-blocking cwPop function. Having cwPop non-blocking, in the performance testing application amounts to using a continuous loop to pop incoming messages from the queue as they arrive. It seems natural to ask whether this overhead could be reduced. However, the key is that CMQ is architected in depth to retain “the core principle of [cloud]scale: decoupling” and avoid even minor blocking operations [1]. Whilst in our testing application this feature might not appear to be an advantage, we are convinced that asynchronous operation is required in large scale computing clouds.

Based on the findings above, we conclude that nothing is hindering the scalability of CMQ.

Message passing strategies.

Replace vs Replay

Although message loss is very rare in switched data center networks (no message loss is detected in our testbed), the question of how to deal with message loss is always present. Although in switched data center networks it is unlikely that packets are lost in transit, there are still conditions existing where UDP packets will be discarded, for example, the exhaustion of the internal buffers or (un)-bounded message queues.

When using IPC that is based on UDP, the application needs to detect lost or partially ordered messages and, if required, will need to deal with it. One possibility is to replace lost messages rather than to replay them. This paradigm requires the application to replace a lost packet if necessary with either a message with the same data or with new data. The application would need to resubmit it to CMQ where it gets queued and delivered as every other new message without holding up any time critical messages that need to be sent at the same time.

Request - Response

(RR) message exchange patterns (MEP), for example as described in the OASIS SOAP over UDP standard [52], are an application-level means to detect packet loss and to act on it. Using this paradigm the application would maintain its own store to record sent requests and wait for a matching answer before a predefined timeout occurs. If a response is received the store is updated; if no response is received and the timeout is met, either the request is resent or the user is alerted of a failure. However, already the SOAP envelope that wraps the actual message is approximately 512 B in size, thus making SOAP envelopes less effective for smaller message sizes.

Multicast IPC

The publish/subscriber paradigm, where subscribers are notified when a message that is interesting to them has arrived, is a very popular means to achieve asynchronous message passing communication, although the underlying protocols (e.g. TCP or RPC) are strictly synchronous. Using conventional message queues, asynchronous communication is usually supported by a broker to decouple the sender and receiver and by maintaining publish/subscriber channels. However, because of the inherent asynchronous nature of CMQ, Multicast Messaging is easily supported without the need to maintain publish/subscriber channels and additional infrastructure for the broker.

Use cases for multicast IPC are connectionless servers that propagate information to the (local) network [53] e.g. simultaneous updates of databases (replication), the propagation of intermediate results in grids, multiplayer games or realtime news [54]. Multicast messaging has also been found applicable to Map Reduce where it can be used to propagate tasks and results [55].

To illustrate, conventional message queues are frequently deployed to enable parallelism by using a broker to decouple the sender and receiver. The publish/subscriber paradigm where subscribers are notified when a message that is interesting to them has arrived is a very popular means to achieve asynchronous message passing communication although the underlying protocols (e.g. TCP or RPC) are strictly synchronous. CMQ instead is inherently asynchronous and offers multicast Messaging without the need to maintain publish/subscriber channels.

Future type message passing

As described by [56], is a further message passing strategy that fits the nature of CMQ. Future Type Message Passing utilizes future objects that should behave like a queue similar to TChan. Analogue to concurrent programming with explicit futures [57], the future object is looked at by the time it is required. In case of CMQ, if cwPop returns Nothing by the time the result is required, the original message might be lost, and accordingly, the recipient process may have become unavailable or may have failed, and thus trigger some remedial action. The remedial action can be either replacing the missing message, re-sending it with a new future, or restarting the receiving process. A future object can be either a result of computation represented by an actor or as simple as a flag such as received? With CMQ, Future Type Message Passing can be easily implemented using the primitive newRq that creates a queue and a TChan where the TChan is subsequently used to represent the future message.

Conclusion and future work

CMQ is a lightweight message queue in Haskell. The concept to use UDP instead of TCP is motivated by our understanding that, in Cloud Computing, omnipresent off-the-shelf technologies (both in hard- and software) are encouraged, and if preventing errors from occurring becomes too costly, dealing with the errors may be a better solution. This paper has demonstrated the capability of using UDP for message queuing in the presence of errors, and has shown the stability of UDP messaging in such conditions. Methods that deal with packet loss at the application level are also discussed. The implementation of CMQ is a Haskell Module that utilizes pure functional data structures. The implementation of CMQ is available as module System.CMQ from the hackageDB at http://hackage.haskell.org/packages/hackage.html, and also it can be installed automatically via the Haskell package manager cabal on every Haskell Platform.

Although CMQ is a message queue oriented communication approach, CMQ is different than the conventional MOM approach because it challenges a number of assumptions under which conventional MOM is built. For instance, in conventional MOM, messages are “always” delivered, routed, queued and frequently follow the publish/subscriber paradigm. It is often accepted that this requires an additional layer of infrastructure and software where logic is split form the application and configured in the additional layer. On the contrary, CMQ does just enough. It does not offer guarantees, thus is very light weight with low overhead and fast speed. Although it does not offer guarantees, it appears to be stable in the presence of errors.

CMQ is a starting point for future research on distributed applications. It will serve as a message queuing mechanism for a lightweight cloud computing framework named CWMWL that we are currently developing. One of the main goals of CMQ is to make the use of pervasive asynchronous parallelism easy and with minimal effort. Furthermore, we are also interested in developing an optional reliability layer that supports reliable virtual connections, protocol-driven and/or application driven security mechanisms, and "persistent" templates for workloads that pre-distributes code and data to the target systems to enhance the performance and reduce the run-time distribution cost.

Endnotes

[a] physical unit of scale in a cloud, e.g. a standardized rack of interconnected servers

[b] e.g. IEEE 802.3 × PAUSE frames or vendor specific technologies

[c] IPv6 would be 40 bytes

[d] IEEE 802.1Qbg

[e] NFSv4 preserves state and is no longer built to deal with packet loss, thus requires the TCP protocol.

[f] An interesting detour would be to investigate whether MessagePack can better support
serialization of arbitrary functions and closures in order to transmit data, code and state over the wire.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
JF had the initial vision of CMQ as part of an inherently asynchronous cloud computing framework that would follow the overall notion of remediating error rather than preventing it. Joerg Fritsch developed the Haskell code, carried out the laboratory experiments and wrote the first draft version of the paper. CW As the co-author of the paper, Dr. Walker contributed actively to the concept of the CMQ system, as well as providing guidance and support to the implementation of the system. She played a critical role in revising the draft manuscript and in ensuring a high standard of presentation. Both authors read and approved the final manuscript.

Author details
[1]NATO CI Agency, Den Haag, Netherlands. [2]School of Computer Science and Informatics, Cardiff University, Queen's Buildings, 5 The Parade, Roath, Cardiff CF24 3AA, UK.

Received: 30 May 2012 Accepted: 3 October 2012
Published: 11 October 2012

References

1. Birman K, Chockler G, van Renesse R (2009) Toward a cloud computing research agenda. SIGACT News 40(2): 68–80. http://doi.acm.org/10.1145/1556154.1556172
2. Rashti M, Grant R, Afsahi A, Balaji P (2010) iwarp redefined: Scalable connectionless communication over high-speed ethernet. In: High Performance Computing (HiPC), 2010 International Conference on. pp 1–10
3. Feng W, Balaji P, Baron C, Bhuyan L, Panda D (2005) Performance characterization of a 10-gigabit ethernet toe. In: High Performance Interconnects, 2005. Proceedings. 13th Symposium on. pp 58–63
4. Regnier G, Makineni S, Illikkal I, Iyer R, Minturn D, Huggahalli R, Newell D, Cline L, Foong A (2004) TCP onloading for data center servers. Computer 37(11): 48–58
5. Barroso L, Dean J, Holzle U (2003) Web search for a planet: The Google cluster architecture. IEEE Micro 23(2): 22–28
6. Vogels W (2008) Keynote: Uncertainty, The Next Web Conference. https://vimeo.com/1386054. Accessed Oct 2012
7. Wang GWG, Ng T (2010) The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In: IEEE INFOCOM. Proceedings. pp 1–9, Mar. 2010
8. Uusitalo MA (2006) Global vision for the future wireless world from the wwrf. Vehicular Technol Mag, IEEE 1(2): 4–8
9. Wu C-C, Chen K-T, Chen C-M, Huang P, Lei C-L (2009) On the challenge and design of transport protocols for MMORPGs. Multimedia Tools and Appl 45(1-3): 7–32
10. Networking for Game Programmers. http://gafferongames.com/networking-for-game-programmers/. Accessed May 2012
11. Sung M (2010) From Cloud Computing To Cloud Computing. Network and Syst Support for Games (NetGames) 9th Annual Workshop: 1–48
12. Marzolla M, Ferretti S, D'Angelo G (2010) Dynamic scalability for next generation gaming infrastructures. Proc. 4th ACM/ICST Int Conference on Simul Tools and Techn (SIMUTools 2011): 1–8
13. Nae V, Prodan R, Fahringer T, Iosup A (2009) The impact of virtualization on the performance of Massively Multiplayer Online Games. Network and Systems Support for Games (NetGames), 2009 8th Annual Workshop on: 1–6
14. Henning M (2004) Massively multiplayer middleware. Queue 1(10): 38
15. Waldo J, Wyant G, Wollrath A, Kendall S (1997) A note on distributed computing. In: J. Vitek and C. Tschudin (eds) Mobile Object Systems Towards the Programmable Internet, ser. Lecture Notes in Computer Science. vol. 1222. Springer, Berlin / Heidelberg, pp 49–64
16. Gu Y, Grossman R (2007) UDT: UDP-based data transfer for high-speed wide area networks. Comput Networks 51(7): 1777–1799
17. Ren Y, Tang H, Li J, Qian H (2009) Performance comparison of UDP-based protocols over fast long distance network. Inf Technol J 8(4): 600–604
18. Hewitt C (2010) Actor model for discretionary, adaptive concurrency. CoRR abs/1008.1459
19. Agha G (1986) Actors: A Model of Concurrent Computation in Distributed Systems (Mit Press Series in Artificial Intelligence). The MIT Press, Cambridge, Massachusetts
20. Hewitt C (1977) Viewing control structures as patterns of passing messages. Artif Intelligence 8(3): 323–364. http://www.sciencedirect.com/science/article/pii/0004370277900339
21. Armstrong J, Virding R, Williams M (1993) Concurrent Programming in Erlang. Prentice Hall, Englewood Clics, N.J.
22. Akka (toolkit and runtime for building highly concurrent, distributed and fault tolerant event-driven applications on the jvm). http://akka.io Accessed May 2012
23. Light Weight Event System. http://www.lwes.org/ Accessed May 2012
24. Bialecki A, Cafarella M, Cutting D (2012) Hadoop: a framework for running applications on large clusters built of commodity hardware. http://wiki.apache.org/hadoop/. Accessed May 2012
25. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1): 107–113. http://doi.acm.org/10.1145/1327452.1327492
26. Foster I, Zhao Y, Raicu I, Lu S (2008) Cloud computing and grid computing 360-degree compared. In: Grid Computing Environments Workshop, 2008. GCE '08. pp 1–10
27. Programming Language Popularity. http://www.langpop.com. Accessed May 2012
28. Marlow S (2012) Parallel and concurrent programming in haskell. In: Central European Functional Programming School, ser. Lecture Notes in Computer Science. vol. 7241. Springer, Berlin / Heidelberg, pp 339–401
29. Peyton-Jones S (2011) Parallel = Functional - The way of the future. http://research.microsoft.com/en-us/people/simonpj/. Accessed Oct 2012
30. Epstein J, Black AP, Peyton-Jones S (2011) Towards haskell in the cloud. In: Proceedings of the 4th ACM symposium on Haskell, ser. Haskell. ACM, New York, NY, USA, pp 118–129. http://doi.acm.org/10.1145/2034675.2034690
31. Coutts D (2012) Cloud Haskell. In: Fun in The Afternoon, FiTA. Well-Typed LLP, Oxford
32. Hinze R (2001) A simple implementation technique for priority search queues. In: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming, ser. ICFP '01. ACM, New York, NY, USA, pp 110–121. http://doi.acm.org/10.1145/507635.507650
33. Stackoverflow. http://stackoverflow.com/questions/1435359/why-can-you-only-prepend-to-lists-in-functional-languages. Accessed May 2012
34. Lipovača M (2011) Learn You a Haskell for Great Good!, ser. A Beginner's Guide. No Starch Pr, San Francisco
35. Data.Sequence. http://hackage.haskell.org/packages/archive/containers/0.4.2.1/doc/html/Data-Sequence.html. Accessed May 2012

36. Harris T, Marlow S, Jones SP, Herlihy M (2008) Composable memory transactions. Commun ACM 51(8): 91–100. http://doi.acm.org/10.1145/1378704.1378725
37. The epass package. http://hackage.haskell.org/package/epass. Accessed May 2012
38. Okasaki C (1999) Purely Functional Data Structures. Cambridge Univ Pr, Cambridge
39. Bird R, Jones G, De Moor O (1997) More haste, less speed: lazy versus eager evaluation. J Funct Programming 7(05): 541–547
40. O'Sullivan B The criterion package. http://hackage.haskell.org/package/criterion. Accessed May 2012
41. INTEL GmbH (2006) Intel® MPI Benchmarks, Users Guide and Methodology Description. Germany
42. cloudstack. http://cloudstack.org. Accessed May 2012
43. MessagePack. http://msgpack.org/ Accessed May 2012
44. 0MQ. http://www.zeromq.org/. Accessed May 2012
45. MessagePack Blog. http://msgpack.wordpress.com/. Accessed May 2012
46. Pessach Y (2006) UDP DELIVERS: take total control of your networking with .NET And UDP. Microsoft MSDN Mag: 56–65
47. Mathis M, Semke J, Mahdavi J, Ott T (1997) The macroscopic behavior of the TCP congestion avoidance algorithm. ACM SIGCOMM Comput Commun Rev 27(3): 67–82
48. Seveik P, Wetzel R (2008) Improving Effective WAN Throughput for Large Data Flows, NetForecast Report 5095. pp 1–8. http://www.netforecast.com/reports/
49. Fiedler G Reliability and Flow Control. http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/. Accessed May 2012
50. The netfilter.org project. http://www.netfilter.org/. Accessed May 2012
51. Gill A, Runciman C (2007) Haskell program coverage. In: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, ser. Haskell '07. ACM, New York, pp 1–12. http://doi.acm.org/10.1145/1291201.1291203
52. Jeyaraman R (2009) Soap-over-udp version 1.1. OASIS WS-DD TC
53. Leffler S, Fabry R, Joy W, Lapsley P, Miller S, Torek C (1986) An advanced 4.3 BSD interprocess communication tutorial. University of California, Berkeley, California
54. Larrabeiti D (2002) Multicast in IPv6, Global IPv6 Summit 2002. http://www.ipv6-es.com/02/docs/david_larrabeiti.pdf. Accessed May 2012
55. Lee K, Boykin DPO, Figueiredo DRJ (2009) Multicast Tree Map-Reduce: Self-organizing Resource Discovery and Monitoring using Structured P2P Systems. In: NSF CAC Semiannual Meeting. University of Florida
56. Romanovsky A, Xu J, Randell B (1998) Exception handling in object-oriented real-time distributed systems. Object-Oriented Real-time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on. pp 32–42
57. Sabel D, Schmidt-Schauß M (2011) A contextual semantics for concurrent haskell with futures. In: Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming, ser. PPDP '11. ACM, New York, pp. 101–112. http://doi.acm.org/10.1145/2003476.2003492

doi:10.1186/2192-113X-1-20
Cite this article as: Fritsch and Walker: **CMQ - A lightweight, asynchronous high-performance messaging queue for the cloud.** *Journal of Cloud Computing: Advances, Systems and Applications* 2012 **1**:20.