Scaling Instant Messaging Communication Services:

A Comparison of Blocking and Non-Blocking techniques

Leigh Griffin, Kieran Ryan, Eamonn de Leastar and Dmitri Botvich
Telecommunications Software and Systems Group
Waterford Institute of Technology
Waterford, Ireland
{lgriffin, kryan, edeleastar, dbotvich} @tssg.org

Abstract— **Designing innovative communications services that scale to facilitate potential new usage patterns can pose significant challenges. This is particularly the case if these services are to be delivered over existing protocols and interoperate with legacy services. This work explores design choices for such a service: large scale message delivery to existing Instant Messaging users. In particular we explore message throughput, accuracy and server load for several alternative implementation strategies. These strategies focus on approaches to concurrency, with best practice in current and emerging techniques thoroughly benchmarked. Specifically, a conventional Java Executor approach is compared with a functional approach realised through Scala and its Actors framework. These could be termed "blocking I/O" technology. A third approach has also been measured - a "non-blocking I/O" based on an alternative to Java Virtual Machine approaches - employing Node.js and Javascript. We believe that some of the results are startling.**

Keywords; Blocking IO; Instant Messaging; Non-Blocking IO; Scalability; XMPP

I. INTRODUCTION

Instant Messaging (IM) and presence services have become a mainstay of modern communications. Consumer messaging services such as Windows Live Messenger [1], Google Talk [2], and AOL Messenger [3] have become essential communication services for many organisations and enterprises. One particular protocol, XMPP (employed by the google talk service), is especially prevalent [4]. It is an IETF standard, is designed with extension in mind, and has a range of open source server and client implementations. These implementations provide a platform for customised implementations of XMPP, either to address security concerns, introduce new services based on the protocol or repurpose the protocol for unforeseen usage patterns. Additionally, the protocol supports federation, which enables custom servers to be linked to a broader network. Thus new services can be introduced into an existing network (and deployed clients).

This work explores techniques for building such service extensions, and in particular examines challenges associated with scaling messaging services beyond the levels for which they were originally architected. In particular we look at large scale delivery of individual messages, based on presence, to traditional Instant Messaging clients. Typically, IM systems assume that a users buddy list is scaled to human dimensions. So a buddy list (a roster) might typically have 50-100 contacts (buddies). However, in some circumstances it might be interesting to propose a usage pattern whereby a given user appears as a contact (buddy) on thousands, or tens of thousands of rosters. This could be for emergency services, direct marketing, customised alerts or other forms of usage that leverage presence of messaging on a large scale.

Such extensions will have to work with existing XMPP server implementations, and use custom plugins to provide these enhanced services. In this work we select the popular Openfire XMPP service [5], and build a set of plugins to implement a high volume messaging capability. In order to understand the limits associated with different approaches to scalability, we have constructed several variants of the plug-in, each taking a different approach to scalability. The first variant is built on the latest version of the Java Executor framework [6], a revision of the java concurrency support. The second is implemented in Scala [7]- a JVM compatible language - which implements an actor-based approach to concurrent programming. The third eschews Java completely, and implements the same functionality in Javascript. Moreover, the Javascript implementation exhibits a fundamentally different approach - it uses a "non-blocking I/O" pattern as facilitated by the Node.js [8] javascript platform.

This last approach (node.js) has achieved some surprising results recently, particularly in addressing the well known C10k problem [9]. Put succinctly, this C10k names a limitation of most web servers: they can handle at most 10,000 connections simultaneously. Node.js approaches are showing some interesting results when applied to this problem [10]. This work is not quite a replication of the C10k problem: we are more interested in a messaging and presence services than a plain HTTP service (which is the focus of most C10k experiments). However, we believe that we have conducted some interesting experiments in devising a hybrid environment where by some high volume processing is now possible, even in the context of interacting with a more traditional "blocking" service such as Openfire.

This paper is broken down into seven sections. This section, the first, serves as the general introduction. Section two examines the related work to this paper. The third section discusses approaches to concurrent programming. The fourth section presents the problem domain of instant messaging. The fifth section looks at plugin design. The sixth section presents our results. The seventh and final section is our conclusion.

II. Related Work

The authors of [11] investigated the reverse C10k (RC10k) problem. Supporting 10,000 outbound HTTP requests presents a different challenge to handling an equal number of inbound requests. The authors present a discussion on concurrency and the design issues that arise out of handling so many connections. An external component and a thread pool were deployed to manage client connections coming close to, but not achieving the goal of solving the RC10k problem. The authors recommended using a language that is lightweight with no memory sharing. Languages such as Erlang and Scala possess these characteristics.

Tilkov recently published [12] an article on using Node.js to build high performance network programs. The Javascript event based model of utilising callbacks provides a more efficient, controlled and scalable environment for developers. Node.js is presented as an emerging ecosystem supported by Javascript in the key roles of front-end and back-end, thus creating a low friction environment ideal for tackling issues of scalability within a demanding environment.

Xiao [13] documented the traffic characteristics of Instant Messaging in a previous study. This work reinforces the view that presence and roster sizes present a serious scalability bottleneck. When operating under heavy load XMPP drops packets, including messages, in order to preserve the integrity of the roster management. Understanding the traffic profile of an XMPP server and the workload generated is essential for the efficient design of extensions,

Griffin [14] examined the management capabilities of XMPP when deployed to solve the communication needs of a large scale healthcare scenario. A care group formed to meet the needs of a patient is an ever evolving group, with a dynamic membership base that needs to be current and accurately replicated on all participants' devices. When the group size scaled up the cascading effect of roster updates as members joined and left the group, overwhelmed the XMPP server and showed that the roster design is a scalability bottleneck.

III. Concurrency Programming Paradigms

A. *Traditional Approaches*

Diverse approaches to programatically "coping" with concurrency have long been a source of contention among software developers. The evolution of the various approaches to concurrency are well illustrated in the C like languages, particularly Java. Although Java was designed with thread based concurrency in mind (unlike C & C++), its currency support has evolved significantly since its inception, with adjustments made to the core syntax, the libraries and the recommended approaches. The fundamental mechanism (synchronised keyword to serialise method access), has been supplemented with concurrent data structures, more expressive annotations, and an extensive rework of the concurrency model in Java 5 [15] to incorporate a new "executor" framework. However, concurrent programming in Java is still regarded as complex and error prone, with non-determinism an ever present worry, even for systems long deployed in the field.

The java concurrency module is founded on the shared state semantics of a single multi-threaded process, whereby threads can share resources and memory, but with locks associated with specific data structures. Alternatives to this model have gained some ground. The actors model rules out any shared data structures (and their resource hungry locks), with concurrency achieved by message passing between autonomous threads - each thread (an actor) has exclusive access to its own data structures. In functional languages derived from Java (Scala, clojure), immutability itself is elevated to be the default programming model. This requires wholesale adoption of functional approaches (or object-functions hybrids in the case of Scala), with the consequent profound change in programming style and heritage. With all of these approaches there is one common characteristic. Separate threads are created, with their own stacks and program counters. Although the opportunities for inter-thread synchronization vary, such synchronization must occur at some stage, with consequent overhead associated with task switching, memory usage and general processor load.

B. *An Alternative Approach*

There is an alternative, which has its origins in an era that predates the general acceptance of multi threaded infrastructure. Evolved to meet the requirements for responsive I/O in single processor systems, it sometimes takes the term "Non Blocking I/O", although this term has also been applied to threaded designs. Originally devised as a set of interrupts and associated daisy chained interrupt handlers, in the modern sense (if we can call it that), non-blocking I/O implies an extensive use of callbacks in API design and usage. In this context, all opportunities for blocking are replaced by passing a callback parameter, to be invoked on completion of the deferred task or I/O request. A somewhat counter-intuitive programming style, it has been criticised for its verbosity and general awkwardness.

In certain programming languages it is indeed verbose - Java in particular is encumbered with a high-ceremony anonymous inner class syntax which make callbacks quite difficult to orchestrate. Also, in Java and other languages of that generation, the callbacks are limited in scope and place severe restrictions around the context they can access. What they lack is a "closure" capability - essentially a form of delegate/callback/function handle - which also carries (encloses) a well defined context that can be safely accessed when it is activated. Closures have become a hot topic in programming language recently, and Java itself is slated to this capability in future versions. JVM derived languages such as Scala and Groovy [16] have this capability, as does Closjure via its Lisp [17] heritage. In fact the term closure originates from these functional languages.

C. The Node.js Movement

This approach though has received a new lease of life recently from an unexpected quarter. Javascript might just be the most widely deployed programming environment in history (every web browser in existence). Initially regarded as a very limited language, its true nature and power has only recently been appreciated in any depth, and a major move is now underway to apply this language in new and fascinating contexts. In particular, its prototypical inheritance mechanism, innate support for closures, and its highly expressive and efficient object literal notation (JSON) provide a foundation for fresh perspectives on performance, concurrency and efficiency. The node.js initiative - and associated satellite projects - is at the heart of this movement, generating impressive initial results and contributing to a rethinking of many of the fundamental patterns for achieving highly scalable services and applications.

Google required fast Javascript so that its services like Gmail and Google Calendar would work well under load. To do this, Google developed the V8 [18] Javascript engine, which compiles Javascript into highly optimised machine code on the fly. The open-source V8 engine was adapted by the community for cloud computing. The cloud computing version of V8 is known as Node.js, a high performance Javascript environment for the server. Node.js wishes to provide an easy way to build scalable network programs. The API of Node.js is non-blocking, either because the task is not blocking or when it is Node.js prevents blocking allowing a callback to be registered. Every call to the Node.js API is an opportunity for the engine to change the request and execute any pending callback waiting. The result is running requests are gradually executed in parallel. The additional functionality that node brings can allow an elegant solution be engineered for traditional scalability problems that might benefit from non-blocking I/O.

IV. The Problem Domain

A. Instant Messaging: Reliability and Scalability

Instant Messaging has inherent advantages over other text based delivery platforms such as email and SMS because it is almost instantaneous, usually includes a built-in subscription mechanism, and provides an indication of a users availability and context via presence. Harnessing Instant Messaging for mass message delivery is therefore a compelling use-case; It provides the message publisher with a receptive and available audience, and the recipient with instantaneous information when and where they are available to receive it. However, scaling up is problematic as various problems are evident under load. Load conditions include high numbers of simultaneous online users, frequent messaging and presence updates and increasing roster size. The eXtensible Messaging and Presence Protocol (XMPP), the protocol of choice for driving Instant Messaging has roster size as a known issue for XMPP Servers [19]. Unlike other non-presence based messaging systems such as email, XMPP Servers must maintain a "roster". The roster provides a dynamic record of the presence relationships between a user and his buddies. Presence updates become more expensive to maintain as the number of entries in the roster grows. For example a user with 20,000 buddies in roster would trigger 20,000 presence stanzas each time he/she changes presence from "Away" to "Available". This results in an increase in processing load on the XMPP server as well as increased network traffic both across federated links and to instant messaging end-users. In addition to this, the XMPP server must receive and process presence stanzas from all of its online contacts.

In functional terms, reliable, scalable short message broadcasting means being able to account for every message sent, and to be able to deliver messages quickly and efficiently, even as the number of users attached to the XMPP roster increases. It could be argued that this use-case for XMPP was not envisaged, i.e. the normal use-case, is a private individual with a few hundred or less contacts on his/her roster where reliability and scalability are not crucial. As with many other technologies, actual use-cases cannot always be predicted. Mass presence handling and message broadcasting built on top of XMPP is attractive for both publisher and consumer. Implementing this use-case seems possible, but is certainly not trivial.

B. System Overview and Experimental Approach

For the purposes of our investigation, an XMPP server was required. Several options were considered for the XMPP server, with Openfire, eJabberd [20] and Prosody [21] examined. Openfire, an open source Java based XMPP server was chosen for the tests. The extensible nature of the server, delivered in the form of plugins and components, along with the availability of its core API was desirable. The service scenarios of interest such as direct marketing campaigns or emergency communication had the potential to involve tens of thousands of users. The messages sent would be time sensitive and only distributed to those in a position to receive the message. Consulting a roster with thousands of users was not possible to implement within existing XMPP servers without the aid of a plugin. Rosters of that size are unwieldy and have the potential to cripple the performance of the server. Additionally, no guarantee is provided that the message sent was received by the server and processed for delivery. The approach taken saw the authors design three plugins to be tested with the openfire server and compared to a fourth legacy plugin. The XMPP plugin extends the functionality of the Openfire XMPP Server, interacting with the server, the roster and the end-user in the form of a buddy.

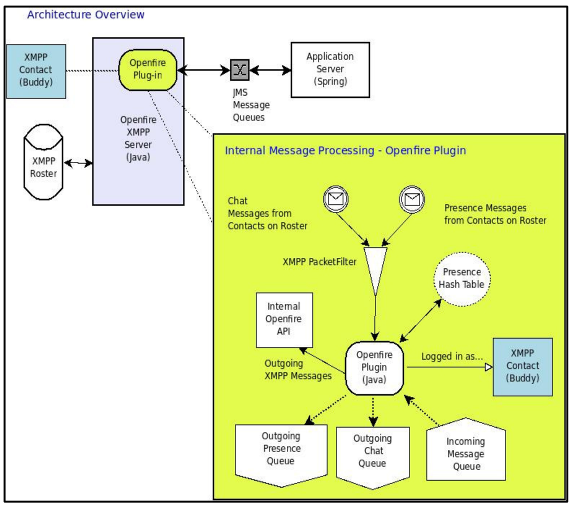

Figure 1. Overview of XMPP Plugin Internals

The role of each plugin, was to accept presence messages from contacts on the servers roster, record the presence state and deliver chat messages to designated recipients who were online and available to receive them. Messages to users with a presence indication that they were not in a position to chat would not be sent. Figure 1 above shows the internal structure of the XMPP plugins.

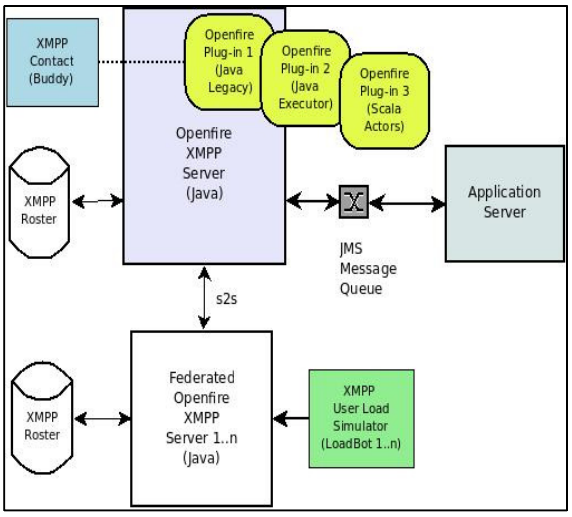
XMPP Contact (Buddy)
Openfire Plug-in 1 (Java Legacy)
Openfire Plug-in 2 (Java Executor)
Openfire Plug-in 3 (Scala Actors)
Openfire XMPP Server (Java)
XMPP Roster
JMS Message Queue
Application Server
s2s
XMPP Roster
Federated Openfire XMPP Server 1..n (Java)
XMPP User Load Simulator (LoadBot 1..n)

Figure 2. Experimental Setup

Figure 2 shows the architecture used for the scalability and reliability experiments. It shows the Openfire XMPP plug-ins (shaded in yellow) and the role they play within the architecture. The plugins interacts with the XMPP server, the XMPP Roster, the end-user in the form of a buddy, and with a Java Message Service (JMS) Message Broker [22]. The application server represents an external service that requires mass message delivery to online and available contacts. This service maintains groups of JIDs with the end user capable of sending a message to a specific group or groups. For the purposes of our experiments the application server's message load and recipient list is generated by our simulator and fed to the JMS message queue. JMS Messages can also be created by the plug-in if required, such as when presence changes occur. These messages are checked to measure accuracy i.e. correctness of the plugin's behavior.

V. Plugin Design

A. *Legacy Plugin*

The Legacy Java plugin was not developed with a realisation of the concurrency issues that would come into play, especially under load conditions. The approach employed is to wait for an event on the XMPP or JMS (message queue) interfaces and to process the event to completion on the event thread. The main class employed was not thread-safe due to access to a shared hash-map and the approach was monolithic rather than decomposed into tasks. This plugin is included for completeness and as a point of reference for one set of experiments. The plugin could be classified as non-blocking by virtue of not using threads, but no attempts are made to optimise the performance.

B. *Java Plugin*

The Java plug-in uses a fixed size thread pool with a tunable thread parameter encapsulated by a custom demultiplexer abstraction. Each type of event is modelled as a Task which performs a discrete unit of work, or calls on other Tasks to perform work. In each case the Task is submitted into an Executor for queueing and execution by the next available thread from the thread pool. This plug-in uses the Java Executor framework. There are two such thread pools employed in this plug-in, one for JMS events produced by the application server and the other for XMPP events. Each one may produce new tasks for the other. For example, a JMS Message request from the application server will produce an XMPP message event to an XMPP end-user. Reliable access to shared data was identified as a problem for the system in this study owing to the use of a shared presence map. With thread safety a crucial requirement for the plug-in, it was necessary to implement a reliable thread safe data structure. The state of the art way to do this is to use Java's concurrent collections [23]. The existing non thread-safe HashMap implementation was replaced by the java.util.concurrent ConcurrentHashMap. This implementation employs its own thread-safe concurrency mechanisms, is highly efficient and is already thoroughly tested.

C. *Scala Plugin*

The third plugin uses the Scala language (version 2.8) and Scala Actors. Five actors are employed: a

PresenceActor, MessageActor, ControlActor, JMSActor and an XMPPActor. The MessageActor routes XMPP chat messages to the outgoing JMS queues via the JMS Actor. The ControlActor processes requests from the application server, as well as XMPP Query packets, and routes outgoing messages to the XMPP interface. The JMS Actor sends control, chat and presence messages over the JMS queues to the application server, and the XMPP Actor sends XMPP packets out via the XMPP server. Each Actor uses the "react()" method rather than "receive()" method which is well suited to event-based applications, and fine-grained tasks where the work scheduler can employ "work-stealing" techniques [24]. The PresenceActor provides guaranteed thread-safe presence lookups on a Scala Map. This Presence facility was an important aspect of the design of the plug-in. Since the data contained in this map is shared between objects, the default choice for a Java developer would be a synchronised or concurrent HashMap. This choice was deliberately avoided in favour of an ordinary Scala HashMap free of any locking and simultaneous access. The consequence of this choice is that the lookup becomes asynchronous. The only practical way for the calling actor to know which presence result belongs to which message, and without maintaining state information in the calling Actor, is to pass the message along with the lookup (see Figure 3) and to have the Presence Actor return it along with the result. The possible disadvantage here is the additional data transferred between the entities but since Instant Messages are generally short the trade-off seems acceptable. The use of Case- Classes and pattern matching keeps the code short and easy to read.

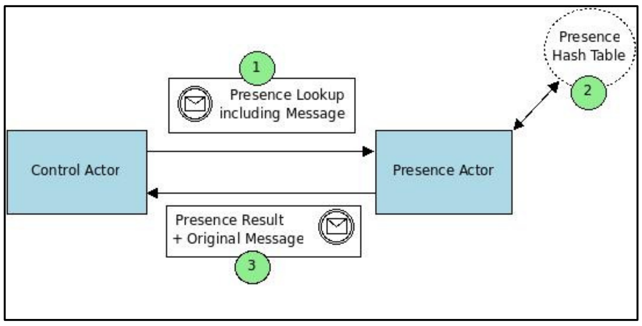
Presence Hash Table
1
2
3
Presence Lookup including Message
Control Actor
Presence Actor
Presence Result + Original Message

Figure 3. Asynchronous Presence Lookup

D. *Node.js Component*

A different approach was taken to create the non blocking plugin. The open source community developed xmpp.js [25], a node.js library that would allow you to connect to an XMPP server as a component. An XMPP component [26] behaves in the same manner as a plugin, implementing new features but with the added benefit of not being tied to a specific server implementation, thus making it portable and reusable. The component binds to the XMPP server domain and becomes a part of the server in the same manner as a plugin. It is addressable and has it's own JID in the form of a domain name making it accessible e.g. component1.myserver.com. All incoming stanzas addressed to that domain or to entities on that domain e.g. buddy@component1.myserver.com will be routed to the xmpp.js base code where they will be subsequently delivered. Outgoing stanzas can be sent on behalf of any user on the domain giving the component full control of message delivery and allowing the design of innovative services such as [27]. The authors used the power of this library to create a simple component which would deliver the same functionality as the Java and Scala plugin discussed earlier. This non-blocking design resulted in the component having two functions, an onPresence and an onMessage function which would be used for callbacks to handle presence and message events respectively.

VI. THE SCALABILITY OF BLOCKING I/O VS NON BLOCKING I/O

A customised Botz library [28] was designed by the authors to enable us to rapidly create user accounts and authenticate with the system. The experiments performed saw a load generator assume the role of the application server. The load generators role was to login 10,000 users, and produce 10,000 messages to be distributed to the user base, a single message per user. These messages would be fed to the JMS and subsequently handled by the plugins. A second set of tests was also devised to investigate how the plugins throughput would be affected by a heavy presence load. A second load generator was set up using the Botz library with 5000 users set to login and logout rapidly, thus producing "Available" and "Unavailable" presence statuses upon connecting and disconnecting with the server. This kind of rapid flooding of presence messages is designed to replicate a busy XMPP server and provide a more realistic performance evaluator of each plugin as they attempted to deal with the messages sent from the original generator. For each series of tests, the plugins were attached to the same server independent of each other and every effort was made by the authors to ensure accuracy and independence in the results gathered. The machine used for all tests was a 2.13GHz Intel Xeon powered 8.04 Ubuntu Server with 2Gb of RAM. Openfire version 3.6.4 and JVM version 1.6.020 were also used. Tests were run 20 times and the results gathered for analysis are presented below

A. *Message Throughput*

Table I shows the single load generator results. All figures below are in terms of messages per second that the plugin dealt with.

TABLE I. SINGLE LOAD GENERATOR RESULTS

Plugin	Min	Mean	Max
Java Exec	109	270	322
Scala	214	249	267
Node.js	1360	1485	1608

The Java Plugin showed a trend of decreased throughput most noticeable as the thread pool size increased. The Scala plugins performance was comparable with the Java plugin but only at the higher end thread-pool settings. For the smaller thread pool settings the Java plugin was on average 15% faster. The Node.js component non blocking approach saw the lowest throughput to be one order of magnitude more then it's Java equivalent. The average throughput was considerably higher when threads are removed from the scenario. Table II shows the results of the dual load generator with the Legacy plugin included as a baseline comparison

TABLE II. DUAL LOAD GENERATOR RESULTS

Plugin	Min	Mean	Max
Legacy	8	21	69
Java Exec	14	76	151
Scala	29	82	108
Node.js	518	621	745

The Legacy plugin was used in this series of tests to provide a base figure for how a standard openfire server would perform while trying to deliver messages in an environment with a lot of background presence noise generated by the second load generator. As to be expected the plugin performed poorly, dropping to as low as 8 messages per second. The Java Exec plugin had the highest peak throughput of the non blocking I/O based plugins but performance was somewhat erratic. The more controlled nature of the Scala plugin led to more predictable results with a marginal improvement on average throughput. Figure 4 below shows the three non-blocking I/O plugins performance.

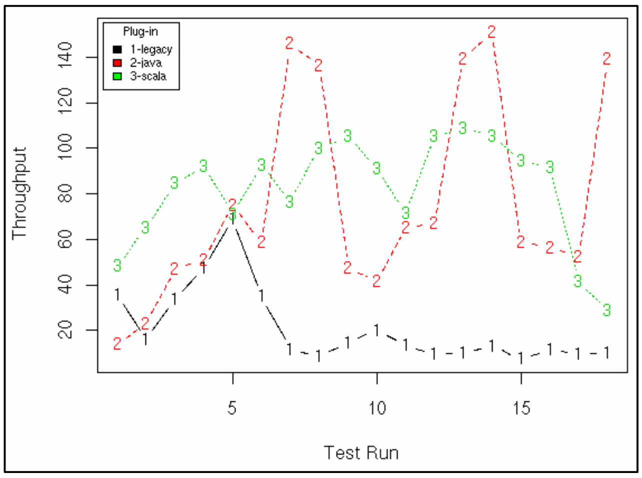

Figure 4. Dual Throughput performance of blocking I/O plugins

The non blocking I/O node.js component did not suffer the same percentage drop in average throughput when it was faced with the noise of the rapid user logins and presence updates. Figure 5 shows the results gathered over the 20 runs

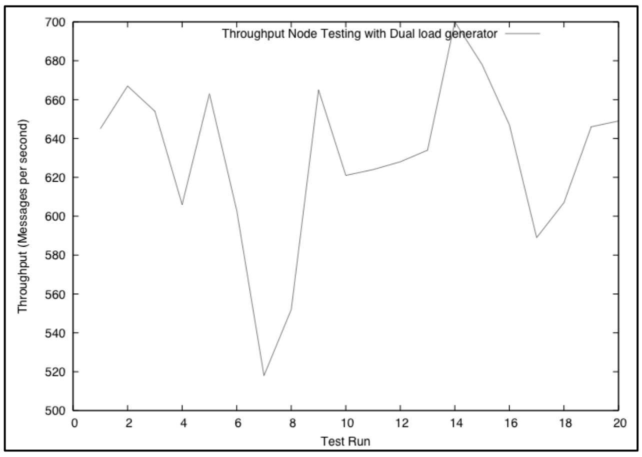

Figure 5. Dual Throughput performance of blocking I/O plugins

B. *Memory footprint*

Memory usage was relatively light for all four plugins with some noticeable differences depending on the load scenarios. For the single load generator tests, the Scala profile was less then 50% of the Java profile. This situation was reversed when the second load generator became active. Scalas memory footprint increased from an average of 20Mb to over 180Mb. This was to be expected in an environment where shared memory is kept to an absolute minimum. The memory overhead was eventually enough to cause throughput degradation and eventually heap space errors on the Openfire JVM. The Legacy, Java and Node plugins had roughly the same memory footprint of around 70Mb across all tests.

C. *Message Accuracy*

Table III shows the message and presence delivery accuracy of the plugins within the dual load generator tests.

TABLE III. ACCURACY FOR PRESENCE AND MESSAGE PACKETS

	Legacy	Java	Scala	Node.js
Presence	100%	100%	100%	100%
Messaging	70%	90%	100%	100%

The legacy plugin became overwhelmed quiet quickly and the resulting loss of 30% represents a serious QoS problem for using a default XMPP server within a high load scenario. The Java plugin was a marked improvement in terms of accuracy but at times of high contention the concurrency issues were reflected by the number of messages lost. The Scala and Node.js plugins resulted in 100% message delivery. It is interesting to note the priortisation of presence, which is directly related to roster management. The 100% presence delivery is required to guarantee the accuracy and integrity of the roster.

D. Observation

The CPU utilization for the tests was also recorded. The blocking I/O based plugins rarely troubled the CPU and did not consume many CPU cycles. The Node.js based plugin however consumed 100% of available CPU resources when run on the single load generator tests. The multiple callbacks to handle events and deliver messages required a lot of CPU usage but delivered a far superior throughput for this trade off. On the dual load generator tests the throughput of the node.js plugin was directly related to the available CPU. The average CPU usage for the node.js process was 68% with the min and max results outlined earlier having a corresponding CPU usage of 54% and 79% respectively. The chance to take more CPU cycles was denied by the prioritsation of the roster updates by the openfire server. This costly, but necessary action limited the potential of the node.js component. Running the node process on a separate machine to the openfire server would increase the performance but was not within the scope of this paper due to the nature of the other plugins developed.

VII. Future Work and Conclusion

Traditionally Instant Messaging systems involved point to point communication. Part of our future work vision involves users actively participating in groups. A group based communication service when coupled with a richer multimedia service presents significant challenges. The stricter QoS metrics and variable network conditions, particularly for mobile consumers, will make this a difficult environment to work in. The desirable qualities of XMPP, which would be required in such a scenario also bring with them associated problems such as roster management. The authors have already considered this in previous work and a novel solution to managing groups of services [29] and groups within XMPP [14] for an emerging context of interest has already been proposed. The work presented here could be extended from sending one message to thousands of users, to sending many messages to thousands of groups, each potentially containing thousands of users. This level of scalability, plus the additional group management overheads warrants further investigation.

The move towards cloud computing has brought about a change in attitudes towards scalability. The traditional approaches of sinking capital into powerful machines has given way to more innovative design patterns, providing better optimization, throughput and consequently lower costs. The emergence of the app store model [30] has strengthened the position of the cloud, with services consumed on the move. Innovative group based communication services, living in the cloud and consumed on end users devices are under consideration for future work.

This paper examined the characteristics of blocking (both conventional and functional) and non-blocking I/O, taking a popular service as a domain for comparison. The authors developed two approaches to the design of a blocking I/O plugin to try and guarantee the delivery of messages to a mass number of users. It was shown that a simple change of programmatic style can result in a more stable and controlled approach, guaranteeing a baseline QoS for operators. The power of the non-blocking approach, championed by the emerging Node.js, showed that mass message delivery can not only be accurate, but timely as well.

Acknowledgment

The authors would like to acknowledge funding support from the Irish HEA PRTLI Cycle 4 FutureComm programme and by Science Foundation Ireland via grant 08/SRC/I1403 (“Federated, Autonomic Management of End-to-End Communications Services”). The work in this paper builds upon research and applications developed as part of the IMPRUVE and ZIMBIE projects which the authors would also like to acknowledge.

References

[1] Windows Live Messenger [online]. Available from htttp://explore.live.com/windows-live-messenger Accessed on 07-FEB-11

[2] Google Talk [online]. Available from http://www.google.com/talk/ Accessed on 07-FEB-11

[3] AOL Messenger [online]. Available from http://www.aim.com/ Accessed on 07-FEB-11

[4] XMPP Standards Foundation [online]. Available from http://xmpp.org/ Accessed on 07-FEB-11

[5] Openfire XMPP Server version 3.6.4 [online]. Available from http://www.igniterealtime.org/projects/openfire/ Accessed on 07-FEB-11

[6] Kim, MinSeong., Wellings., Andy. Using the executor framework to implement asynchronous event handling in the RTSJ. Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, 2010.

[7] Oliveira, Bruno., Gibbons., J. Scala for generic programmers. Proceedings of the ACM SIGPLAN workshop on Generic programming, 2008.

[8] Node.js [online]. Available from http://nodejs.org/ Accessed on 07-FEB-11

[9] Kegel, D., The C10k problem [online]. Available from http://www.kegel.com/c10k.html Accessed on 07-FEB-11

[10] Salihefendic, A., Plurk: Instant conversations using Comet [online]. Available from http://amix.dk/blog/post/19490 Accessed on 07-FEB-11

[11] Liu, D., Deters, R. The Reverse C10k Problem for Server-Side Mashups. Proceedings of the International Conference on Service Oriented Computing, 2008

[12] Tilkov, S., Vinoski, S. Node.js: Using Javascript to Build High-Performance Network Programs. Internet Computing, IEEE, 2010

[13] Xiao, Z., Guo, L., Tracey, J. Understanding Instant Messaging Traffic Characteristics. 27th International Conference on Distributed Computing Systems, 2007.

[14] Griffin, L., de Leastar, E., Botvich, D. The Management of Dynamic Shared Groups within XMPP: An Investigation. IEEE International Symposium on Integrated Network Management, 2011.

[15] Long, B., Long, B.W. Formal specification of Java concurrency to assist software verification. Parallel and Distributed Processing Symposium, 2003.

[16] Koenig, D., Gloer, A., King, P., Laforge, G., Skeet, J. Groovy in Action, Manning, 2007.

[17] Allen, J. Anatomy of Lisp. McGraw-Hill, 1978.

[18] V8 JavaScript Engine [online]. Available from http://code.google.com/p/v8/ Accessed on 07-FEB-11

[19] Saint-André, P., Smith, K., and Tronçon, R. XMPP: The Definitive Guide: Building Real-time Applications with Jabber Technologies. Farnham: O'Reilly, 2009.

[20] Ejabbered: The erlang Jabber/XMPP daemon [online]. Available from http://www.ejabberd.im/ Accessed on 07-FEB-11

[21] Prosody IM [online]. Available from http://prosody.im/ Accessed on 07-FEB-11

[22] Chappell, D., Monson-Haefel, R., Java Message Service. O'Reilly, 2000.

[23] Bloch, J. Effective Java (2nd Edition). Prentice Hall, 2008.

[24] Haller, P., Odersky, M. Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science, 2008.

[25] XMPP.js [online]. Available from https://github.com/mwild1/xmppjs Accessed on 07-FEB-11

[26] Saint-André, P. XEP-0144: Jabber Component Protocol [online] Available from http://xmpp.org/extensions/xep-0114.html Accessed on 07-FEB-11

[27] Zimbie: The online Marketing Tool for Time Sensitive Products [online]. Available from http://www.zimbie.com/ Accessed on 07-FEB-11

[28] Botz: Internal Bot Library for Openfire [online]. Available from http://community.igniterealtime.org/docs/DOC-1130 Accessed on 07-FEB-11

[29] Foley, C., Power, G., Griffin, L., Chen, C., Donnelly, N., de Leastar, E., Service Group Management facilitated by DSL driven Policies in embedded Middleware. International Symposium on Computers and Commnunications, 2010.

[30] Chrome web store [online]. Available from https://chrome.google.com/webstore Accessed on 07-FEB-11