Scaling Instant Messaging Communication Services: 
A Comparison of Blocking and Non-Blocking Techniques

Leigh Griffin, Waterford Institute of Technology, Ireland
Kieran Ryan, Waterford Institute of Technology, Ireland
Eamonn de Leastar, Waterford Institute of Technology, Ireland
Dmitri Botvich, Waterford Institute of Technology, Ireland

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 the authors 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 Input/Output, eXtensible Messaging and Presence Protocol (XMPP), Instant Messaging, Non-Blocking Input/Output, Scalability

1. INTRODUCTION
Instant Messaging (IM) and presence services have become a mainstay of modern communications. Consumer messaging services such as Windows Live Messenger (n.d.), Google Talk (n.d.), and AOL Messenger (http://www.aim.com) have become essential communication services for many organisations and enterprises. One particular protocol, XMPP (employed by the google talk service), is especially prevalent (XMPP, n.d.). 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

DOI: 10.4018/jaci.2012070101
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 user’s 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 (Ignite Realtime, n.d.), 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 (Kim & Wellings, 2010), a revision of the java concurrency support. The second is implemented in Scala (Oliveira & Gibbons, 2008)- 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 javascript platform (http://nodejs.org/).

This last approach (node.js) has achieved some surprising results recently, particularly in addressing the well known C10k problem (Kegel, 2011). Put succinctly, this C10k problem 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 (Salihfendic, 2010). 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 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 eight sections. This section, the first, serves as the general introduction. Section 2 examines the related work to this paper. Section 3 discusses approaches to concurrent programming. Section 4 presents the problem domain of instant messaging. Section 5 looks at plugin design. Section 6 presents our results. Section 7 examines applying the lessons learned within a broader domain. Section 8 is the final section and presents our conclusion.

2. RELATED WORK

The authors of Liu and Deters (2008) 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 in order to
Societal Implications of Current and Emerging “Smart” Technologies
www.igi-global.com/chapter/societal-implications-of-current-and-emerging-smart-technologies/183452?camid=4v1a

An Efficient Kinetic Range Query for One Dimensional Axis Parallel Segments
www.igi-global.com/article/efficient-kinetic-range-query-one/190654?camid=4v1a