INTERNET, WIRELESS AND LEGACY INTEGRATION
Architectural Framework for Testing

Nenad Stankovic
Nokia, Burlington, MA 01803

Keywords: Internet, wireless, PSTN, testing, distribution

Abstract: Voice and data networks require performing components of highest quality. To achieve these goals software engineering employs testing. However, software performance and performance testing have been less studied and reported on. We present here the test-bed developed and used in performance and stability testing of the intelligent networks integration infrastructure. We used the Visper framework for rapid distributed application development to build our test-bed. We also report on common programming problems that have been identified in multiple applications written in Java, and on the experience with our test-bed. Our findings are based on the work and feedback provided by experienced professionals with a solid object-oriented background. Their experience with Java and J2EE was mixed, while the test-bed and the concepts of distributed programming were new to everyone involved.

1 INTRODUCTION

Over the last decade the Internet has become an everyday source of information, entertainment, messaging, and many business services for millions of users and companies around the world. Although omnipresent, the Internet has an important drawback due to its restriction to a location. Similar to the making of a Public Switched Telephone Network (PSTN) call, the lack of mobility restricts where those services can be accessed and content consumed. With the proliferation of wireless telephony, there is a widespread need for content to also go wireless and become accessible via mobile devices. The drive towards convergence of mobile devices and the Internet to accommodate anytime, anywhere access to multimedia communications is resulting in new services for the user and new business opportunities for the operator.

Each of the mentioned networks has its own characteristics and technologies as historically and physically their evolution was independent of the other domains, and their design was driven by different requirements. In the world of modern and integrated communications it is essential to deliver targeted and timely information and quality services, development of which is driven by customer needs. For seamless and reliable end-to-end services it is required to secure transparent, performing and high quality interconnect among these networks. In turn, interoperability between them and communication between end users across domain boundaries requires elegant solutions on the technical level. From an overall perspective, ultimately, it should appear as if there is only one unified network.

Although the separate domains of wireless, computer and wired telephony networks remain with their individual characteristics, recently they became integrated together. To enable such a seamless interoperability between key applications, network domains, and the user identity and addressing (Figure 1), we have developed a multidomain integration infrastructure based on open standards, technologies, and relevant initiatives. As a result, the user is not concerned with the underlying domain technologies and directly enjoys the richness of the supplied services and content irrespective of the access technology and methods as much as possible. In the same time, convergence between the same services regardless of their service providers has been progressing by addressing harmonization requirements for service technologies. These services and implementation should also not be tied to any proprietary or domain specific technology.

Stankovic N. (2004).
INTERNET, WIRELESS AND LEGACY INTEGRATION - Architectural Framework for Testing.
In *Proceedings of the First International Conference on E-Business and Telecommunication Networks*, pages 213-220
DOI: 10.5220/0001398602130220

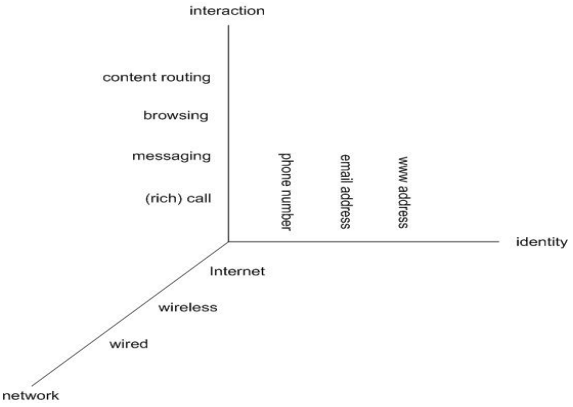

Figure 1: Networks Interoperability

The internetworking infrastructure, as dictated by the domains it integrates, is characterized by high availability, scalability, and reliability, while its usage pattern can change rapidly and dramatically in time. Therefore, it must be extremely dynamic, reactive, robust, and capable of handling thousands of different types of requests concurrently. Its performance must be predictable, stable under a wide range of loads, and ensure quality of service (QOS), all in the same time. To build this infrastructure is a formidable technical task which requires highly developed skills, and rapid development techniques through composability and component reuse. Subsequently, functional and high volume stability and performance testing is conducted. Testing is labour and resource intensive. On large software projects it often amounts to between 50 and 60 percent of total software development cost, even without factoring in the cost of purchasing equipment for testing. Without constantly performing quality testing throughout a whole software lifecycle the risk of the project being unable to meet the requirements within the given time and budget constraints is greatly increased.

This paper is primarily concerned with performance testing of hardware and software required by this infrastructure. In principle, performance testing is conducted as end-to-end black box application testing where test cases are derived from the use cases and functional specification. We augment the tests used in functional testing by generating input load that is function of time, content and user actions. A problem with testing such a heterogeneous collection of applications is that we need many different tools in the process. Commercial tools are not only expensive, but different tools apply different techniques in setting up test cases. They often lack portability and generate results that are not easy to correlate and reproduce with other tools. More importantly, it may be hard to calibrate them and generally impossible to customize, and optimize their performance. For example, the capture and replay tool always picks up the attachment file from disk, rather than from an in memory cache. On the other hand, test programs are structured similarly and are relatively easy to code if not sophisticated.

Based on these premises we can derive a generic and extensible test-bed that facilitates development of test programs and setting up of test cases, and resolves the mentioned problems. By reusing proven software artefacts new projects and tasks do not have to reinvent and rediscover what was already accomplished before. The mentioned characteristics were successfully implemented and evaluated by building a large-scale test-bed based on Visper (Stankovic, Zhang, 2002). Visper is a distributed programming framework that provides constructs at programming language level that allow programmers to directly transfer architectural and domain specific decisions into an implementation. The architectural model of Visper has proven easy to adopt and suitable for rapid prototyping, and application deployment. Its runtime image is small in size, efficient in speed. The cost of overhead of its distributed and local services when integrated into an application is low and constrained mainly by the standard low level mechanisms provided by Java (Campione, et al., 2000), such as the java.io and java.net packages. Visper is scalable and extensible to support new applications and programming models that require the provided services and abstractions.

The paragraphs that follow first introduce the intelligent networks integration infrastructure. Section 3 motivates the test-bed and describes the main patterns that can be found in performance and stability testing. We also briefly describe Visper and the components that were used to accomplish this task. Section 4 presents a complex test case example that involves main infrastructure components, as well as much of the functionality built into the test-bed in order to drive such a performance and stability test. Section 5 summarizes our findings and experience with the infrastructure, Java, and test-bed, and Section 6 correlates our and similar research work. Section 7 concludes the paper.

2 INTELLIGENT NETWORKS INTEGRATION

As shown in Figure 2, the applications that support the seamless and intelligent integration of the Internet, and wireless and wired telephony implement a number of gateways (GW) and service

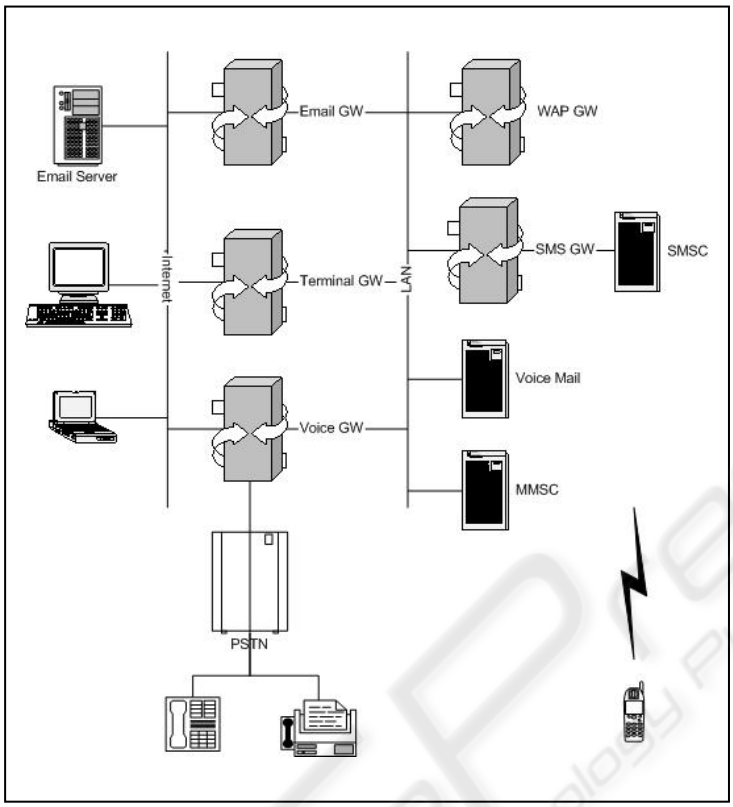

Figure 2: Intelligent Networks Integration

centers. The reason is that historically these domains have evolved independently, and are based on standards that are not compatible, not needed or not found in the other domains. While the basic infrastructure requirements are familiar (e.g. billing, content encoding and conversions, transient and persistent message repository, notifications), the number of specialized conversion services (e.g. JPG, GIF, AIF, BMP, WBMP, WAV, AVI, MOV) and protocols (e.g. HTTP, HTTPS, SMTP, SS7 (SS7), T1 (T1), TCP, WAP (WAP)) required is rather large, which has dictated their separation into multiple applications that may be enabled or disabled as necessary.

Therefore:

- The MMSC (Multi-Media Service Center) is the point of connectivity into and from the wireless domain. Its main role is to receive multimedia messages, provide temporary storage and to attempt to forward the messages to the receivers
- Multimedia Message Service (MMS) delivery to an email address requires that the MMSC communicates with an existing email server. The multimedia Email GW provides the connectivity between these end points, together with the routing of messages based on content and identity
- The Terminal Gateway enables delivery of MMS messages to legacy terminals. It stores MMS message content in its own local storage, and sends an SMS to the receiver, informing the user of a Web address where the content can be viewed via a Web browser. The Terminal GW also provides users with an album for messages and images. These images can be used to create a new MMS message either on the phone or in the browser
- The Voice GW supports unified messaging and intelligent call routing in all three domains, such as Voice over Internet Protocol (VoIP), PSTN voice, and advanced speech recognition. Instead of receiving a textual notification, the voice message can be encapsulated as an MMS and sent directly to the phone
- Content conversion is important when an image has a format that is not supported by the recipient. The MMSC routs the MMS to a content converting application, after which the new message is sent forward
- The WAP Gateway receives a MMS from the wireless device via a WAP Post, and transmits the MMS from the MMSC to the wireless devices via a WAP Get

The Email, Terminal and Voice GWs are known as external applications to the MMSC. They have been developed in Java and J2EE (J2EE, 2003), while the other applications are based on C and C++ with significant embedding into the architecture for performance reasons.

3 TEST-BED

The development work on these applications has been conducted in parallel, as has been the testing work performed both by development teams and QA. This program organization requires not only dedicated computing resources for development and testing, but also requires special tools to conduct the testing. Even without such a large suite of products, the cost of commercial testing tools and hardware would be significant and most likely unacceptable if unrestricted access is granted to everyone involved.

On the other hand, the software developers are required, while coding, to produce test programs and test their classes for functional correctness, stability and performance. Initially there was no standard for designing, writing, executing, and managing test artifacts that would facilitate reuse and controlled experimentation. Although programmers spent a good portion of their time on writing test programs, generally that part of their work was not reused by their colleagues or shared. Test programs have been developed and executed locally on a PC, due to unavailability of dedicated UNIX workstations for individual testing and development work.

3.1 Requirements

To resolve these shortcomings in the testing process, a number of possibilities were explored, and Visper was selected due to the following perceived advantages:

- Simple yet powerful model that can be adopted quickly, with low level classes that support process control and configuration for rapid definition of test cases and test profiles for performance and stability testing
- Does not require significant additional programming skills. Test scripts are organized as text files, with attribute-value pairs, and they support nondestructive attribute redefinition. Reuse of software is mandatory
- Easy and safe sharing of computing resources between programmers. Since programs in Visper are executed on dedicated sessions, users are shielded from any adverse impact that other users can exert upon them
- Dynamic configurability and transparent reassignment of resources. Test machines have been a scarce resource, and often reallocated and reconfigured. It is not acceptable to constantly update and rebuild test programs after each change in the laboratory setup
- Does not require special configuration files and multiple copies of the same process, that unnecessary overloads the testing machine. A cold restart only affects that session, and deployment and redeployment is dynamic from multiple network nodes
- Scalability that allows executing thousands of user profiles and multitude of test cases concurrently, without a programming intervention
- Test artifacts are organized in a directory structure on a per test case basis and under a versioning control, as to ensure reusability and reproducibility of experiments

Programmers were introduced to the test-bed in a 1 day get started course in which they were thought about the relevant concepts, tools, components, how to code and execute test programs, and prepare and configure tests in a distributed environment.

3.2 Patterns

Test programs are often not complex to code, but still require a considerable amount of time to prepare if sophisticated. This, however, does not address refinements, potential distribution and scalability issues. Without guidance from an existing body of work, the test case implementer must engineer a solution from scratch, and on a case by case basis. Architecture plays a central role in software engineering by developing high level views of the system, and defining models, components and guidelines for composing systems. Further, software engineering has recognized patterns as a means to capture common and proven design concepts, to protect and reuse them (Bruegge, Dutoit, 2004). Patterns can be adapted before being used, either by wrapping new objects around them or through inheritance. When designing a test program, we can identify the following common patterns:

- Driver that performs a task
- Profile that characterizes a task
- Scheduler that provides typical modes of task invocation
- Configurator that reads textual configuration files that allow easy test setup without a need for recompilation
- Logger that provides the methods to create standardized log files

– TestBed is the factory that creates a test case based on a configuration

An important pattern developed for the test-bed is the Scheduler pattern. It provides three modes of task invocation: uninterruptible, timer driven, and counter driven. The uninterruptible mode simply lets a task run forever. The timer driven mode stops a task after a predefined time period. In both cases task invocations can be time sliced, either randomly or within a fixed time interval. The counter mode stops a task after a given number of invocations. Generally, the scheduler is used as is, and requires no additional programming. Distributed schedulers can coordinate their activities by joining a group. A distributed scheduler can also be used when no distribution is immediately required, providing the group it joins is unique.

The Profile pattern defines a virtual user of an application. Profile is an abstract class that is closely tied to a driver and a test case. A Profile comprises the attributes and actions required to set up a test case. A test case can define any number of profiles. Each profile is assigned a thread to run on by the scheduler, and a driver by the TestBed. Different profiles may use different scheduling policy and drivers. However, this is more a matter of convenience when testing multiple applications in the same time, rather than a necessity when measuring the performance of a single application in a single end-to-end test case.

The Driver pattern defines the standard interface to execute a task in our test-bed. Tasks require establishing the communication channel when simulating an external entity or user, or simply invoking a method for white or black box testing. For example, when testing the Email GW, after composing the email message with or without attachments in a profile, an SMTP channel is established, and the message is sent to an Email GW. A number of standard drivers and profiles have been implemented for application, database and J2EE bean testing.

The Configurator pattern provides the methods to read in, and parse configuration files and scripts. These are text files with attribute value pairs, one pair per line. The same attribute can be nondestructively redefined multiple times, so that multiple profiles can be defined by reusing the same attribute name. Some test cases require a more complicated mechanism than just sending a message. Often, interaction with a sequence of actions is required between a virtual user and a service provider, such as browsing an album of messages. Also a man-to-machine dialogue can take place, where the dialogue is based on a grammar. For example, when leaving a message on the phone a hierarchy of menus that lead the user to a goal must be followed. In Voice GW we use VoiceXML (VoiceXML) to define the grammar. Rather than programming multiple profiles for such a use case, Configurator provides the methods to parse and prepare a dialogue script for execution. Test scripts, configuration files, and content files (e.g. message body, attachments) can be accessed from multiple locations in a network via a file server.

The Logger pattern defines the methods to create standardized log files suitable for analyzing in a spreadsheet. When a test case executes, generated data can be saved to file or redirected to the console for online monitoring (see Section 3.3).

These components were built on top of Visper, either by refining existing classes (e.g. Logger, Scheduler), or by adding new classes (e.g. Configurator, Driver, Profile). The reuse ratio was about 90% excluding the new classes such as Profile and Driver that are test-bed specific. In terms of programming work, drivers were the most demanding, due to their domain specific and case-by-case nature, and the number of different protocols and systems employed by the infrastructure. However, most drivers are simple to write, as they can reuse existing code or call standard Java APIs such as JavaMail (JavaMail), or similar. This has, nevertheless, caused problems initially because multiple programmers had to code the same specific drivers, before a comprehensive database of drivers was established.

3.3 Visper

Visper is a novel object-oriented framework that identifies and enhances common services and programming primitives, and implements a generic set of classes applicable to multiple programming models in a distributed environment. It emphasizes separation of concerns in the design, and hierarchy of layers in the implementation. Component reuse facilitates easy customizability and adaptability to different environments and application needs. Visper also features grouping of distributed collaborating objects, and agent-based (Genesereth, Ketchpel, 1994) distributed system management.

Visper has been designed and implemented in Java, with a number of utilities that facilitate portability and visual program development. Pertinent to this task, it implements a container in which remote threads execute, collaborate and communicate with each other if necessary. A remote thread is the basic unit of computation controlled by the host container. Remote threads are autonomous computing elements that can be dynamically instantiated and can migrate from one machine onto another. The Loader class implements a class loader

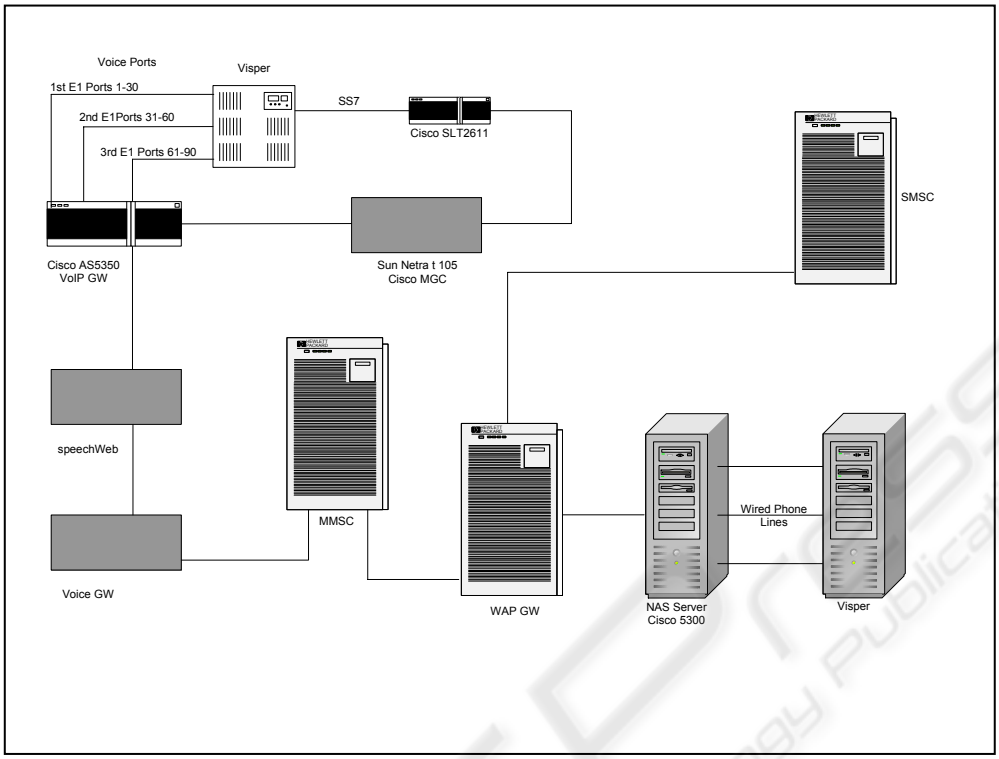

Figure 3: Voice GW Testing

that supports multiple access modes to Java .class files. The Console package implements a GUI for user-to-system interaction and system-to-user feedback. Specifically, following the master-slave scenario, console is the master that spawns and controls the remote threads on behalf of the user.

4 EXAMPLE

The configuration in Figure 3 has been used in testing the Voice GW for performance and stability. This configuration is rather complex, since it involves many distributed software and hardware components. It has multiple input points for load generation, such as the wired telephony, the VoIP access, and wireless messaging. In terms of development work, this was the most demanding task taking almost 3 man months to complete the VoIP and wired drivers. The complexity was due to the usage of native libraries to support SS7 and to simulate incoming wired phone lines that had to be linked into the Java Virtual Machine (JVM) (Lindholm, Yellin, 1999) via a Java Native Interface (JNI). Also, software for voice activity detection had to be incorporated.

Test-bed sends MMS messages to the MMSC, simulates callers dialing Voice GW subscribers, reroutes the calls to the Voice GW, and plays voice messages for recording. Voice GW plays voice, text and fax messages to the subscribers. It sends the MMS message to the subscriber multimedia terminal via a WAP GW.

As mentioned before, the standard handling of configuration and script files had to be improved to support dialogues with speech and DTMF grammars. For example, the legacy user dials a mobile phone number, and the call gets rerouted to a call center that starts playing a menu. The test script must be able to select a menu option, detect when there is no speech activity on the channel, wait for a tone, record the message, save it or even modify it before a save option is selected. Also, an option to verify that the right prompt was played by the Voice GW could be used in a dialogue.

5 EXPERIENCE

Performance testing is based on a number of repetitive tasks, with a multitude of user profiles. With application testing, the number of test cases is defined by the number of use cases which is often

low. However, for any realistic testing the number of user profiles per test case should be large to simulate closely real world. This is particularly important for applications written in Java to understand, among others, the garbage collector (GC) behavior and optimize the code as to minimize its impact on performance. The impact of garbage collection due to bad memory management can be severe, and could slow down the system by 50% or more. These implementation problems have been common and widespread, because the engineers either did not understand the consequences of their decisions, or did not feel confident to explore alternative solutions.

Not all of these deficiencies were easily resolved and when uncovered through testing and subsequent code analyses were sometimes ignored due to lack of time for improvement. Initial code and design reviews could easily notice this, but, being performed by peers, they generally failed to do so for the same reason. We found that peer inspections tend to degenerate into comments on style and first order semantic issues. This stands in contrast to the findings reported by other empirical studies (e.g. Boehm, Basili, 2001), even though we believe that these were representative teams for the semidetached and embedded mode projects both in size, age, and experience (60 engineers in total).

Occasionally, we ran a short noncompulsory online survey to identify problems with the test-bed or gather opinions on the approach as more practical experience was acquired over time. The test-bed implementation proved stable and did not cause any major disruption. Product support was organized such that problems were resolved with highest priority. Initially, the main source of problems was in fact introduced in the introduction course. The course was focused on explaining the techniques and classes to write test programs for the test-bed. As such, little attention was paid to explaining the overall distributed environment concepts and programming. It was assumed that programmers would eventually pick up those through experimenting, if needed.

However, early on more than half of the programmers actively using the test-bed were admittedly confused by the distributed nature of this environment, and the dynamic program distribution. The other half *understood in principle actions and reactions* as they have been encountered, and they were comfortable with the tool. As schedule was tight, there was little or no attempt to investigate beyond the effort to make the test program work. The questions asked and problems reported have shown that concepts from J2EE were blindly applied here without any discrimination.

One of the reasons to use Visper as a base for the test-bed was in its ability to easily allocate and use machines for more processing power, without any programming intervention. However, as programs were being written, most of them have been executed on the same machine. Also, rather than using more machines to increase the test load, more Java threads were started via the Scheduler, and log was sent to the console. All those practices caused performance problems. Due to lack of time, no additional group training has been undertaken to rectify and explain the observed misconceptions, but most of them were resolved in a person-to-person conversation, via email, online FAQ and How To pages.

Overall, the satisfaction with programming in test-bed was at about 30% initially, but gradually increased and stayed at about 65%. The main advantage, as perceived, was that within this framework it was much easier to write, understand and share test cases, and more importantly the drivers, and consequently the system under development. The situation that many programmers who worked on the development did not know how to use the system, or being familiar only with their own code, and had little or no knowledge of other parts of the system that have been developed by other programmers or groups was rectified. Also, quality of test programs was improved, so that each developer could perform not only functional testing, but also stability and performance testing as code was written and product was being developed. The task of writing test programs became very much mechanical, and while the time required to program a test case was reduced, the test results were improved, and became more comprehensive.

6 BACKGROUND

As pointed out by Weyuker and Vokolos (Weyuker, Vokolos, 2000), not many research papers have been published in the domain of software performance testing. Interestingly, their findings are also rooted in telephony, but their contribution is mainly in the domain of performance testing per se rather than testing tool development. While they made recommendation on how to approach software performance problems, the reported findings are not based on a whole software lifecycle. They experimented with an existing gateway system that was being redeployed on a new platform. Nevertheless, we find their conclusion that *performance testing differs in many ways from functional testing* correct.

Recently, Gorton and Liu (Gorton, Liu, 2002) investigated different J2EE compliant COTS middleware for large-scale applications and found that their adoption and use is not straightforward, which complies with our findings. Also, when selecting a middleware technology, it is important that it provides and sustains the necessary peak performance. As one would expect, the main middleware vendors (i.e. BEA, Borland, and IBM) yielded similar performance results. The more interesting question remains, as to how much of that raw performance can be exploited and retained when business specific code is added. The problem is not defined only as how computationally expensive is each use case, what is the background noise, but also how to identify and mitigate the bad design and implementation decisions.

7 CONCLUSION

Thus far, the answer to building a large commercial system has been in server centric 2-to-n-tier architecture and middleware to support it. Among the most recent addition to this family of products has been the J2EE middleware. As information exchange and availability became more and more ubiquitous, so did the demand for system integration become paramount. We find that the main contribution of J2EE is in providing a comprehensive platform for developing such systems. J2EE relieves programmers from thread and pool management, as they do not have to deal with these issues explicitly. On the other hand Java beans have introduced a new level of behavioral complexity and configuration problems that has to be understood and managed. Otherwise, unexpected problems may occur that are hard for a novice to resolve. Typical performance problems found in Java applications and techniques to overcome them have been extensively studied and published, but their broad adoption and understanding by (less experienced) programmers is still questionable.

In this paper we also focused on evaluating Visper in a production environment, where it served as a basis for building a comprehensive set of testing tools and programs. The easy configurability and simple programming model have proven valuable in an environment where the pressure of change is constant and activities evolve quickly. Even though the tool was new to the programmers, it was generally quickly adopted and used without major problems or dissatisfaction. Test-bed gave the programmers a chance to interact with the application as it was being developed, and to understand and resolve problems as they appeared. As a result, quality was significantly improved, and the number of problems discovered and reported by QA was proportionally reduced.

REFERENCES

Boehm, B., and Basili, V.R., 2001. Software Defect Reduction Top 10 List, IEEE Computer, 34(1), pp. 135-137.

Bruegge, B., and Dutoit, A.H., 2004. *Object-Oriented Software Engineering*, Pearson Education, Inc. Upper Saddle River, NJ 07458.

Campione, M., Walrath, K., and Huml, A., 2000. *The Java Tutorial: A Short Course on the Basics*, Addison-Wesley, Reading, MA, 3rd edition.

Genesereth, M.R., and Ketchpel, S.P., 1994. Software Agents, *Communications of the ACM*, 37(7), pp.48-53.

Gorton, I., and Liu, A., 2002. Software Component Quality Assessment in Practice: Success and Practical Impediments, ICSE, pp. 555-558.

J2EE, 2003. java.sun.com/j2ee/1.4/docs/index.html

JavaMail. java.sun.com/products/javamail/index.jsp

Lindholm, T., and Yellin, F., 1999. *The Java Virtual Machine Specification*, Addison-Wesley, Reading, MA, 2nd edition.

SS7. www.iec.org/online/tutorials/ss7

Stankovic, N., and Zhang, K., 2002. A Distributed Parallel Programming Framework, *IEEE Transactions on Software Engineering*, 28(5), pp.478-493.

T1. www.t1-t3-dsl-line.com

VoiceXML. www.w3.org/Voice/Guide

VoIP. www.fcc.gov/voip

WAP. www.wapforum.org

Weyuker, E.J., and Vokolos, F.I., 2000. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study, *IEEE TSE*, 26(12), pp. 1147-1156.