I-RMI: Performance Isolation in Information Flow Applications

Mohamed Mansour and Karsten Schwan

College of Computing, Georgia Institute of Technology,
Atlanta, GA 30332-0280
{mansour, schwan}@cc.gatech.edu

Abstract. A problem with many distributed applications is their behavior in lieu of unpredictable variations in user request volumes or in available resources. This paper explores a performance isolation-based approach to creating robust distributed applications. For each application, the approach is to (1) understand the performance dependencies that pervade it and then (2) provide mechanisms for imposing constraints on the possible 'spread' of such dependencies through the application. Concrete results are attained for J2EE middleware, for which we identify sample performance dependencies: in the application layer during request execution and in the middleware layer during request de-fragmentation and during return parameter marshalling. *Isolation points* are the novel software abstraction used to capture performance dependencies and represent solutions for dealing with them, and they are used to create (2) I(solation)-RMI, which is a version of RMI-IIOP implemented in the WebSphere service infrastructure enhanced with isolation points. Initial results show the approach's ability to detect and filter ill-behaving messages that can cause an up to a 85% drop in performance for the Trade3 benchmark, and to eliminate up to a 56% drop in performance due to misbehaving clients.

1 Introduction

Modern middleware and programming technologies are making it ever easier to rapidly develop complex distributed applications for heterogeneous computing and communication systems. Typical software platforms are Microsoft's .NET, Sun Microsystems' Java 2 Enterprise Edition (J2EE) specification, and vendor implementations of these specifications like IBM's WebSphere, BEA's WebLogic, and open source efforts like JBoss. Businesses use these platforms to link different enterprise components across the wide spectrum of hardware and applications that are part of their daily operation. Science and engineering applications benefit from their rich functionality to capture data from remote sensors and instruments, access shared information repositories, and create remote data and collaboration services.

The software platforms identified above are mapped to hardware infrastructures in which end clients are concerned with data capture or presentation (Tier 1), supported by two server-level tiers that implement application and storage services, respectively. The J2EE architecture follows this 3-tier model by defining three container types to host each of the tiers, where containers offer sets of standard services to cover non-functional requirements like transactions, messaging, and security. The goal is for developers to be able to focus on business logic and processes rather than having to deal with dependencies on client or server hardware and software systems.

A barrier to creating the system-independent services envisioned by application development platforms is the level of performance robustness of the distributed applications created with them, in lieu of unpredictable variations in user behavior or in the resources available for satisfying user requests. Recognition of this fact has resulted in a multiplicity of techniques for dealing with behaviors like bursty request volumes, including dynamic load balancing and migration, server replication, and similar runtime methods [2, 6, 26]. For media-rich or data-intensive applications, bursty loads can be combated by reducing the fidelity of media content, skipping media frames, or using application-specific techniques for reducing computation and communication loads [36].

Our interest is to use application- or environment-specific techniques like those listed above to create more performance-robust distributed applications. The goal is to better *isolate* applications from each other with respect to their performance behaviors. The consequent technical contributions of this paper are the following. First, experimental evidence demonstrates the importance of performance isolation toward creating well-behaved distributed applications. Specifically, we show that the unusual behavior of even a single client can substantially diminish a data-intensive J2EE server's ability to provide suitable levels of service to its other clients. Second, we propose an approach to achieving performance isolation that (1) exposes system resource information to the middleware layer, (2) enriches the middleware layer with methods for analyzing and adapting application behavior, *isolation points* and *adaptation modules*, (3) permits the middleware layer to execute these solutions when or if necessary, the latter based on (4) user-defined SLAs (Service Level Agreements). A final contribution is the description of a general architecture for performance-isolated messaging both for J2EE applications and for the popular publish/subscribe programming model.

The concrete artifact produced by and evaluated in this research is I(solation)-RMI, a version of RMI-IIOP enhanced with functionality that enables applications to detect and react in meaningful ways to violations of performance isolation SLAs. Our initial results attained with I-RMI are encouraging. For the well-known Trade benchmark, for example, we are able to sustain high throughput in the presence of resource-intensive requests (a 85% improvement over traditional RMI-IIOP). We also report the complete elimination of side-effects (an up to 56% drop in throughput) resulting from slow clients. These results are achieved by using a sliding window algorithm at two different isolation points.

In summary, the idea of performance isolation is to understand the causes of performance dependencies in distributed applications and then provide middleware-based solutions that prevent their 'spread' through the distributed client/server system. In the remainder of this paper, the next section elaborates on the motivation behind our work as well as gives a detailed overview of I-RMI design and implementation. In Section 3 we list the software platforms and applications we experimented with. Experimental results are presented in Section 4. Conclusions and future work are given in Section 5.

2 I-RMI - Motivation, Design, and Implementation

2.1 Motivation

There is a plethora of work addressing runtime performance management in distributed server systems, ranging from system-level solutions like process/load migration or request throttling [26, 27, 32], to application-level tradeoffs in the quality of server responses produced for clients vs. server response time [10, 18], to the creation of new middleware or system abstractions that support the runtime adaptation of applications and systems in response to changes in user requirements or platform resources [7, 13, 15, 20, 24, 25].

The premise of our research is that modern distributed applications created with development platforms like those based on the J2EE standard are sufficiently complex to make it difficult, if not impossible, to design application-wide methods for optimizing their runtime behavior. Instead, we address the simpler problem of curtailing or limiting the spread of performance problems across distributed client/server subsystems. Examples of this problem occur in the enterprise system run by one of our industrial partners: (1) a backup job run by an administrator during system operation can generate a sufficient level of I/O to slow down file system operations for another subsystem running on the same machine, or (2) the logging of operational data contained in files to a backend database slows down other subsystems that use or produce this file data. One result of such slowdowns is that they cause other subsystems' request queues to build up, including those from the front ends used by clients, potentially leading to operational failures (e.g., inappropriately long response times) or revenue loss (e.g., clients going to alternate sites). The problem, of course, is that performance degradation in one part of the system (i.e., the storage subsystem) leads to performance degradation elsewhere. In other words, the system does not adequately deal with or *isolate* the *performance dependencies* inherent to this distributed application.

Our approach to limiting performance dependencies in distributed enterprise applications like those described in [5, 35] is to enhance middleware with functionality that offers improved levels of performance isolation, thereby creating a performance analogue of the firewalls used in computer security: (1) by examining middleware to identify points along the code path that are vulnerable to performance dependencies, termed *isolation points*, and (2) by re-coding these points and enhancing them with a generic and extensible API that permit developers to define runtime reactions to violations of application-specified measures of performance exhibited by applications, represented as *adaptation modules*. The outcome is the creation of performance 'firewalls' that prevent the spread of performance problems across different components of distributed applications. Our implementation approach addresses the broad class of web service-based applications, by associating instrumentation and support for performance firewalls with the RMI/IIOP implementations used in interactions between web, application, and backend servers.

2.2 Architecture and Implementation

Isolation Points

Isolation points (IP) are associated with identified performance vulnerabilities, enhancing them with a monitoring and control architecture [23] [34]. An *isolation point* uses resource monitoring to detect performance issues and reacts through its enforcement mechanism to prevent their further spread. The specific actions taken are determined by user- or developer- defined policies.

I(solation)-RMI

I(solation)-RMI (I-RMI) is a version of RMI-IIOP enhanced with several isolation points. Our current implementation uses the three isolation points listed below to cover intra- and inter- process interaction. The monitoring and adaptation methods used at these points utilize well-established techniques. The goal is to create an implementation of I-RMI suitable for the information-flow architectures prevalent in today's enterprise computing systems rather than developing new techniques. I-RMI currently defines three isolation points as shown in **Fig. 1**, an interesting aspect being that they are backed by occurrences in enterprise software observed by our industrial partners. We note here that as with related abstractions developed in earlier work [14, 20, 28], the changes made by isolation points occur at the middleware level and can be realized and carried out without requiring modifications to application code.

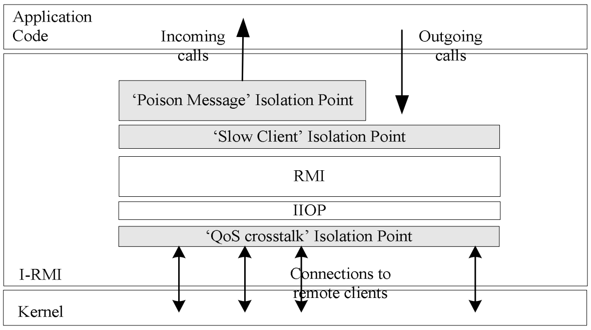
Application Code
Incoming calls
Outgoing calls
'Poison Message' Isolation Point
'Slow Client' Isolation Point
RMI
IIOP
'QoS crosstalk' Isolation Point
I-RMI
Connections to remote clients
Kernel

Fig. 1. Overview of I-RMI

Slow Client

The idea of isolation points applies both to client-server- and event-based distributed applications. Consider the structure of typical enterprise information systems described in [21]. Events generated at the edge of a system trigger chains of message passing and processing inside the system, where each processing step augments, personalizes, or otherwise transforms the original event. An example of such a system is deployed at one of our industrial partners, a major airline company, which feeds ticket reservation events into a revenue estimation system. Each event results in 20-30 subsequent calls to other modules inside this system. The application uses asynchronous messaging to decouple senders from receivers.

Message-based distributed applications have to be constructed and administered so that the rates of delivering messages into queues do not exceed the rates of extracting messages from queues and processing them. Jitter in rates [9] can both lead to queue buildup and put pressure on servers' available memory resources. This in turn can deteriorate server performance and its ability to meet target performance levels. A concrete set of examples studied in this paper addresses data-intensive applications, our intent being to explore the uses of J2EE infrastructures for manipulating the large data items implied by future applications in tele-medicine or -presence, remote collaboration, remote access to rich data sources [3], and data mining. For example, for the multimedia or document management applications described in Liferay Portal [1], we expect message sizes to be quite large, and any additional delays in processing queued messages by remote clients can result in substantial server-level performance degradation.

The 'slow client' isolation point added to RMI-IIOP is intended as a generic mechanism for handling the case described above. This point is inserted in the call path before call argument marshalling. The logic we inject into the path monitors queue behavior (system or application-level queues) indirectly, by monitoring the respective incoming and outgoing request rates[1]. By combining estimates of queue lengths with resource utilization on the local and remote nodes, the injected code can detect situations where a slow node is causing serious queue buildup that might lead to performance degradation on the server. A sliding window is used to measure these rates, one window per causally connected incoming and outgoing APIs, one window for local resource information, and a third window for resource information on each remote machine. The specific action taken to reduce queue buildup is decided at runtime by user-supplied logic. Possible actions include: decreasing the sizes of call parameters to reduce the processing required on the target server, rerouting the call to another host, rejecting the call and having the sender deal with this exception, etc.

Poison Messages

Our next scenario is derived from an airline enterprise information system (EIS). System administrators strive to provide consistent performance levels for the operation of their system. An occasional surge in resource usage, traced back to a particular uncommon request type, can cause other subsystems' requests to build up, including those from the front ends used by clients, ultimately threatening operational failure (e.g., inappropriately long response times) or revenue loss (e.g., clients going to alternate sites). Such uncommon request/message types are termed *Poison Messages*.

The poison message isolation point addresses this class of isolation problems. We monitor a server's steady state throughput (see **Fig. 2**) using request counting methods similar to [13]. When a sudden drop in throughput (L_1) and at the same time, a sharp increase in resource utilization is detected, we identify a potential 'poison' state. A snapshot (S_1) of every request currently executing in the J2EE is taken and stored.

[1] Understanding the causal relations between incoming and outgoing requests is necessary in order to translate request rates into meaningful queue behaviors. The detection of such causal relations is beyond the scope of this work. Aguilera et al [2] articulate possible ways to automatically detect such relations.

When the server later recovers and throughput rises (L_2) again, we take another snapshot (S_2) of all requests currently executing in the server. The set difference ($S_1 - S_2$) represents a list of requests that are potential suspects. This procedure is repeated every time we encounter such abnormal behavior and eventually, the suspect list narrows down to a few request types. The specific action against the potential suspects is left to the user to define, possible actions are reject or re-route to another server.

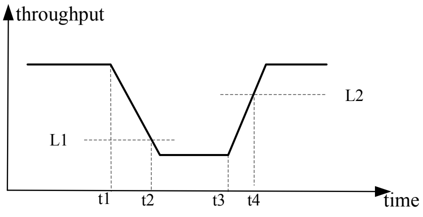

Fig. 2. Dynamically detecting poison messages at runtime

Our current implementation identifies requests only by their API names. To deal with server overload caused by changes in request parameters, the implementation has to be extended to also scan and analyze request parameters [37]. Additional detection logic is necessary if poison state is caused by a sequence of messages.

‘QoS Crosstalk’ due to Parallel Concurrent Streams

Tennenhouse describes `QoS crosstalk` as the effect of multiple concurrent streams on server performance [31]. We include an isolation point in I-RMI to manage and minimize such crosstalk effects. This section describes its implementation and demonstrates the potential of poor performance isolation in the presence of multiple concurrent request streams with varying request sizes. Such request streams are common in information flow applications between front-end Web/UI servers and backend business process servers.

The RMI-IIOP implementation we use dedicates a separate reader thread per client connection. When a server is subjected to invocations from multiple clients, all of the corresponding reader threads are activated, as they all receive notifications of data being available on their underlying sockets. It is up to the underlying kernel thread scheduler to decide which thread to run next. Assuming a round-robin scheduler and equal buffer sizes on all connections, it is common for streams with very small request sizes to receive better treatment compared to streams with large request sizes. Note that this analysis also applies to writer threads.

Behaviors like those explained in the previous paragraph can be unacceptable for certain application deployments or client connections. Known control methods addressing them include changing the socket buffer sizes for certain connections, altering threads priorities, or both. Setting the right buffer size for each connection requires that such a value be calculated uniformly for all connections. Toward these ends, we insert an isolation point at the IIOP reader thread level, and we re-implement parts of RMI-IIOP to use a single reader thread and non-blocking I/O. The single reader thread provides a single point of control where the ‘right’ buffer sizes can be calculated and applied. The resource monitor is responsible for tracking how many parallel streams are active. The enforcement logic dynamically adjusts the buffer sizes for each connection to achieve the desired relative weights. This modified implementation is backwards compatible and also scales better than the original implementation.

3 Representative Applications and Experimental Results

To demonstrate the importance of performance isolation in the J2EE environment, we select WebSphere as a representative software platform. We use the *Trade3* application [16] developed by IBM as our test bed. The Trade3 benchmark models an online stock brokerage application and is built to cover most of J2EE’s programming model, including JSPs, EJBs, transactional aspects and database access. We deployed the Trade3 benchmark with the UI web component on a separate machine from the backend EJB components, WebSphere dynamic caching was not enabled in our experiments.

Experimental Setup
Experiments are run in Georgia Tech’s enterprise computing laboratory, using Version 5.1 of IBM WebSphere J2EE server running on an x345 IBM server (hostname: dagobah), a dual 2.8GHz Xeon machine with 4GB memory and 1GB/s NIC, running RedHat Linux 9.0. The server runs against Version 8.1 of DB2 which runs on a separate machine with an identical configuration. Clients, secondary servers and load generators run on an IBM BladeCenter with 14 HS20 blade servers installed (hostnames awing1-awing14). Each blade has dual 2.8GHz Xeon processors with 1GB RAM and 1 Gb/s NIC card running RH Linux 9.0. We use the Tomcat 5.0.25 servlet container for hosting the front end of the Trade3 benchmark. Httperf [22] is used to generate the workload for the trade benchmark.

3.1 ‘Slow Client’ Isolation Point

Consider the distributed application shown in Fig. 3. The external source injects events into the system, by sending messages to a primary server where they are queued for processing. A worker thread selects messages from the queue and sends them to the secondary server. The primary server also provides auxiliary services to an external client. The external event source generates a 512KB message every 10ms. A client makes repeated requests to the server; each request carries a return parameter of 1MB, the server caches the 1MB object and uses it to serve all client requests.

The average round trip time for the client is listed in Table 1. In the first case, “Unloaded”, the secondary client runs with a very light load. Under these conditions, the average queue length is under 3 units, and the client average RTT is 35 ms/call. The second scenario, “Stress Load”, imposes a heavy workload on the secondary server. We use the stress utility to run 8 CPU intensive threads. This results in a significant drop in the ability of the secondary server to process its requests and subsequently, creates queue buildup on the primary server. As a result of this buildup,

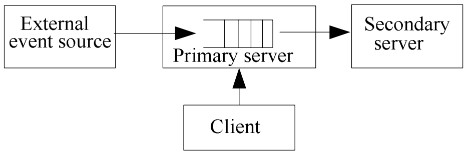

Fig. 3. Abstract view of nodes in an operational information system (OIS)

Table 1. Average round trip time for client calls

Scenario	**Average RTT [from client side]**
Unloaded secondary server	35 ms/call
Secondary server stress loaded	80 ms/call
Secondary server stress loaded + Primary server uses I-RMI	35 ms/call

available free memory drops on the primary server and garbage collection is triggered more often (JVM memory was set to max. to 120MB). This results in an increase in client average RTT to 80ms/call and a 56% drop in throughput. These can be attributed to increased garbage collection on the primary server (see Table 2) due to memory pressure resulting from queue buildup.

To demonstrate I-RMI effectiveness in controlling such effects, we repeat the above experiment using I-RMI on the primary server. Fig. 5 shows the rate of calls coming in and going out of the primary server. Rates are measured by dividing the number of calls that occurred during the last N seconds, where N is the width of the sliding window we use. At T=10000 the secondary server is subjected to CPU stress load, and queue buildup is evident from the difference between the rates of incoming and outgoing calls. The increased garbage collection (GC) activity (as shown in Fig. 4) leads to a drop in the server's ability to service events from the external event source. As the CPU utilization crosses a predefined threshold (1.0 in this experiment), the isolation logic decides to cancel calls outgoing to the secondary server (occurring at about T=14,000). This results in the apparent increase in the rate of outgoing calls. The application is unaware of the short circuit applied by the isolation logic, still thinking that its calls are being completed. Note that this example uses the simplistic approach of call elimination, to focus on the performance isolation properties of our approach. Realistic systems will use any number of techniques, including request rerouting, queuing for later submission, application-specific reductions in request volume [12] and others.

Table 2. Number of times primary server garbage collects per 100 client calls

Scenario	**GC**
Unloaded secondary server	6
Secondary server stress loaded	102

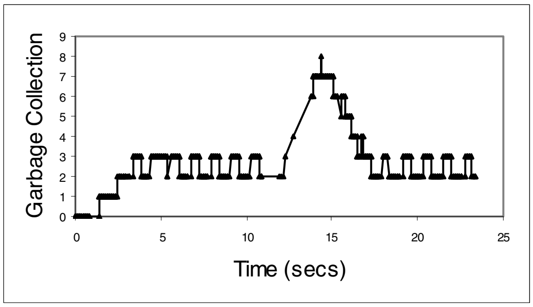

Fig. 4. Garbage collection at Primary Server

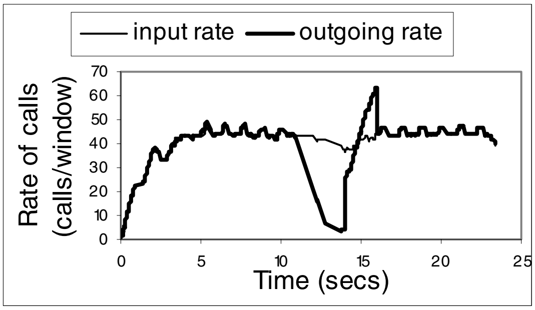

Fig. 5. Call rates measured at the primary server

3.2 'Poison Message' Isolation Point

To demonstrate the effect of a poison message, we run a steady state workload against the Trade3 benchmark. The workload generator (httperf) simulates 4 concurrent sessions with 0.1 seconds think time. The resulting server average request execution times are shown in Fig. 6, respectively (from T=0 to T=50). The small spikes at T=(3, 10, 24, …) are due to garbage collection on the server. At T=59, we manually call a special API added to the benchmark application. This API allocates 10KB byte arrays in a tight loop for 4 seconds. The effects of this API are evident in the graphs as a sudden sharp increase in the average execution times for requests, a drop in server throughput, and a dramatic increase in garbage collection activity on the server (Fig. 7). The rising edge at T=59 triggers our detection algorithm, and it takes a snapshot of all requests currently in the server. As the poison API finishes and server load levels return to normal, the falling edge at T=63 triggers the detection algorithm and it takes another snapshot of all API currently executing on the server. The difference between these two snapshots correctly reveals the poison API in this simple example. Subsequent calls to this API are filtered by the isolation point.

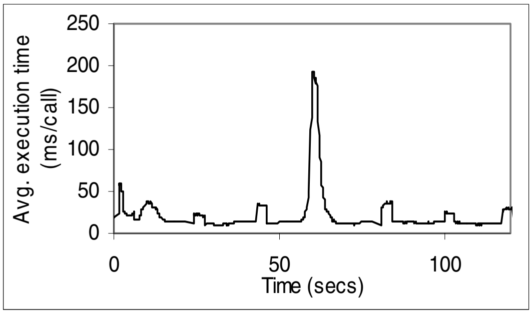

Fig. 6. Average call execution time (ms/call)

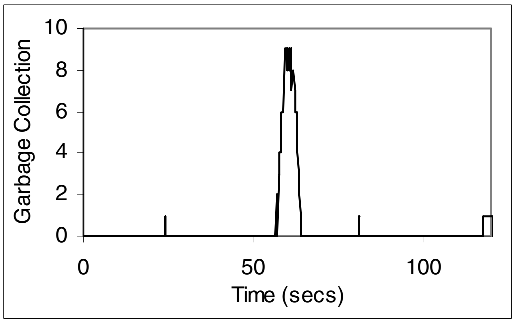

Fig. 7. Garbage collection frequency at the Primary Server

3.3 'QoS Crosstalk' Isolation Point

In this experiment, we demonstrate QoS effects in WebSphere and how I-RMI can provide some control over such behavior. All times reported here represent the time needed to read data for a request at the IIOP level. Request assembly time increases proportionally with message size in the case of one client communicating with the server. In the presence of a second client sending messages of constant 2K size, the time needed to assemble the large request message more than doubles. In Fig. 8, the lower curve shows message assembly times for a client, and the top curve shows message assembly times for the same client in the presence of a secondary request stream of size 2KB/request. This is attributed to the fact that the server processes both streams with equal priority. We ameliorate the above behavior by controlling the socket buffer size for each connection. A larger buffer enables us to read more data per system call. We note here that this approach works only if there is data available at the server socket for reading. This observation indicates the need to associate additional system-level knowledge with isolation points.

In Fig. 9, we plot assembly time against message size for a request stream running against a 2K/request secondary stream. The different lines represent different socket buffer size settings. The top curve labeled '1x' represents equal buffer sizes for both

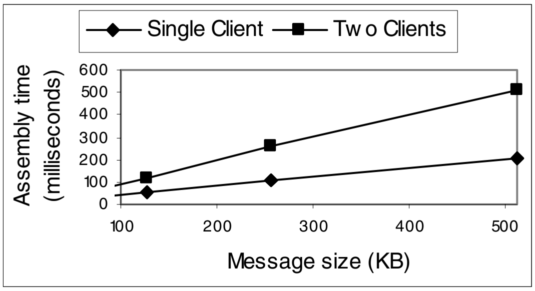

Fig. 8. Time to assemble a request for one vs. two clients

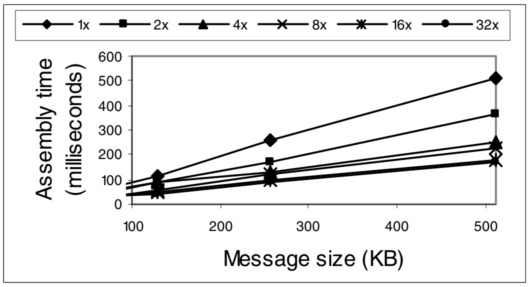

Fig. 9. Time to assemble a request at different buffer sizes

streams. For the next curve, labeled '2x', we set the socket buffer size for the main stream to be twice that of the secondary stream and so on. A larger buffer size clearly reduces the time needed to assemble a large request and therefore, reduces the effects of parallel request streams.

Our implementation replaces the one-reader-thread-per-connection IIOP model in WebSphere with a single reader thread using non-blocking socket I/O. A single reader thread constitutes a single point at which an enforcement mechanism can be realized. Its presence also removes dependencies on the underlying thread scheduler. Care is taken to prevent blocking of this single reader thread. Processing the socket data and handling it to the ORB for assembly is done in a non-blocking manner through utilization of intermediate hand-off queues. The experiments shown in Figs. 8 and 9 are based on a partial implementation, not employing a dynamic resource monitor, using pre-defined buffer sizes for each connection, and without a decoupling queue between the reader thread and the ORB.

3.4 Discussion of Experiments

Beyond the performance results attained with the isolation points used and evaluated in this section, note that they are representative of the three different kinds of isolation points needed for building performance isolation firewalls for distributed service implementations. (1) The poison message IP monitors and controls behaviors that involve the APIs exposed by the application components on a single node. (2) The slow client IP monitors cross-node communications. (3) The QoS crosstalk IP concerns interactions with the underlying OS/hardware platform. More generally, implementations will use multiple IPs of these kinds, and there will be interactions between the policies implemented by multiple IPs, within each node and across nodes. The experiments shown in this section, therefore, constitute only a first step toward creating performance-robust distributed applications and application components.

4 Related Work

Performance isolation is not a new idea [4] and in addition, prior work has developed many methods for dealing with performance problems in server applications. The latter include request deletion in web servers [27], request prioritization or frame dropping in multi-media or real-time applications [30], and the creation of system-level constructs supporting these application-level actions [25, 33]. Essentially, such methods are specific examples of the more general methods for dynamic system adaptation developed during the last decade [29, 38]. They share with adaptive techniques the use of runtime system monitoring and of dynamically reacting to certain monitoring events, but they differ in that the policy-level decisions made in response to certain events are focused on limiting performance dependencies rather than on exploiting them to optimize the behavior of the distributed system exhibiting these dependencies.

This paper advocates an isolation-based approach to performance management, but differs from prior work in that it also considers performance dependencies that exist across different layers of abstraction existing in current systems, such as dependencies across system-level communication protocols and the middleware-level messaging systems that use them. The specific results attained in this paper for Java RMI-IIOP and J2EE-level method calls are related to earlier work done by our group on the IQ-RUDP [15] data transport protocol, which coordinates middleware-level and transport-level adaptations to better meet application needs. What is new here, however, is that we consider explicit characteristics of the more complex Java middleware environments, including Java's garbage collection techniques.

Hardware, kernel, and application-level protection and isolation have been studied extensively for single Java virtual machines [8]. [17] applies the concept of a Java resource accounting interface to isolate applications inside a JVM at the granularity of isolates to J2EE platforms. In comparison, our work focuses on performance isolation at single request granularity (even within the same application), and we identify three kinds of performance dependences embedded in the middleware implementation of J2EE and WebSphere. Since detection logic is placed into middleware prior to application execution, resource reservation approaches like those described in [17] can be used as an enforcement mechanism, where thresholds are set dynamically by a resource monitor. Note that some of the scenarios present in this paper are not addressed by the isolate mechanism, such as when the vulnerability point is in the lower levels of the middleware before the message is parsed and dispatched to its target application (isolate).

Finally, we point to recent work in performance management for cluster-based web services [19]. A central router classifies and schedules incoming requests to maximize a user-defined utility function based on performance measurements collected from the cluster. While traffic classes represent high-level business value, requests in each class can still have very different operational footprints and can therefore, still experience the performance vulnerabilities presented in this work.

5 Conclusions and Future Work

This paper builds on previous work in the autonomic and adaptive system domains to address end-to-end performance issues in service-oriented software architectures. The specific issue addressed is *performance isolation*, which refers to the ability to isolate service components from each other with respect to the performance dependencies pervading distributed applications and the systems on which they run. Performance isolation is a necessary element of any solution that seeks to attain end user-desired Service Level Objectives or Agreements (SLAs), preventing the violation of SLAs through circumstances beyond the explicit control of individual services.

To attain performance isolation, our research offers novel middleware abstractions, termed *isolation points*, which both capture performance dependencies and provide functionality that deals with them. The paper first demonstrates the prevalence of performance dependencies in enterprise applications created with J2EE RMI-IIOP-based software platforms and that these dependencies can lead to the *spread* of performance problems through entire enterprise applications. For example, if a ‘poison message’ causes one server to slow down, this server will act as a slow client to its callers, causing their performance to degrade and propagating undesirable performance effects across the entire distributed application. Second, isolation points (IPs) are created to dynamically capture and react to performance dependencies, thereby providing middleware mechanisms for managing and preventing them. Third, a concrete product of this work is I-RMI, which is RMI-IIOP enhanced with isolation points representative of the three different types of IPs required for performance isolation in distributed enterprise applications: (1) IPs guarding service APIs, (2) IPs for inter-node interactions, and (3) IPs for interactions with underlying operating systems and hardware. I-RMI has been integrated and used with IBM’s WebSphere J2EE infrastructure. When using standard J2EE benchmarks, we are able to eliminate performance degradations of up to 56% observed in traditional RMI-IIOP in one case, and up to 85% in another case.

Future work will consider solutions in which multiple IPs cooperate to address potentially complex performance dependencies, across sets of distributed services and service nodes. In addition, we will address the fact that performance dependencies and the need for performance firewalls implemented with IPs are not specific to Java. They appear both in the synchronous call-reply model of RMI and in the message-oriented asynchronous middleware of operational information systems like the one used by our industrial partners [11].

References

1. Liferay: Open source enterprise portal, 2005.
2. Aweya, J., Ouellette, M., Montuno, D.Y., et al. An adaptive load balancing scheme for web servers. *International Journal Network Management*, *12* (1). 3--39.

3. Barclay, T., Slutz, D.R. and Gray, J. TerraServer: A Spatial Data Warehouse *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000.
4. Barham, P., Dragovic, B., Fraser, K., et al. Xen and the art of virtualization *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, 2003.
5. Bernadat, P., Lambright, D. and Travostino, F. Towards a Resource-safe Java for service guarantees in uncooperative environments *IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, 1998.
6. Cardellini, V., Casalicchio, E., Colajanni, M., et al. The state of the art in locally distributed Web-server systems. *ACM Computing Surveys*, *34* (2). 263--311.
7. Cowan, C., Cen, S., Walpole, J., et al. Adaptive methods for distributed video presentation. *ACM Computing Surveys*, *27* (4). 580--583.
8. Czajkowski, G. Application isolation in the Java Virtual Machine *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00)*, 2000.
9. Diot, C. Adaptive Applications and QoS Guaranties (Invited Paper) *Proceedings of the International Conference on Multimedia Networking (MmNet '95)*, 1995.
10. Fox, A., Gribble, S.D., Chawathe, Y., et al. Cluster-Based Scalable Network Services *Symposium on Operating Systems Principles (SOSP 97)*, 1997.
11. Gavrilovska, A., Oleson, V. and Schwan, K. Adaptable Mirroring in Cluster Servers *10th International Conference on High-Performance Distributed Computing (HPDC-10)*, 2001.
12. Gavrilovska, A., Schwan, K. and Oleson, V. A Practical Approach for 'Zero' Downtime in an Operational Information System *The 22nd International Conference on Distributed Computing Systems (ICDCS-2002)*, 2002.
13. Gheith, A. and Schwan, K. CHAOSarc: kernel support for multiweight objects, invocations, and atomicity in real-time multiprocessor applications. *ACM Transactions Computer Systems*, *11* (1). 33--72.
14. Hamilton, G., Powell, M.L. and Mitchell, J.G. Subcontract: A Flexible Base for Distributed Programming *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, 1993.
15. He, Q. and Schwan, K. IQ-RUDP: Coordinating Application Adaptation with Network Transport *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing (HPDC'11)*, 2002.
16. IBM. WebSphere Application Server, Trade3 benchmark.
17. Jordan, M.J., Czajkowski, G., Kouklinski, K., et al. Extending a J2EE^{TM} Server with Dynamic and Flexible Resource Management *International Middleware Conference (Middleware 2004)*, 2004.
18. Krishnamurthy, B. and Wills, C.E. Improving web performance by client characterization driven server adaptation *Proceedings of the eleventh international conference on World Wide Web (WWW '02)*, 2002.
19. Levy, R.M., Nagarajarao, J., Pacifici, G., et al. Performance Management for Cluster Based Web Services *IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003)*, 2003.
20. Loyall, J.P., Schantz, R.E., Zinky, J.A., et al. Specifying and measuring quality of service in distributed object systems *1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 1998.
21. Mansour, M., Wolf, M. and Schwan, K. StreamGen: A Workload Generation Tool for Distributed Information Flow Applications *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, 2004.
22. Mosberger, D. and Jin, T. httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, *26* (3). 31-37.
23. Oreizy, P., Gorlick, M., Taylor, R., et al. An Architecture-Based Approach to Self-Adaptive Software *IEEE Intelligent Systems*, 1999.

24. Plale, B. and Schwan, K. dQUOB: Managing Large Data Flows Using Dynamic Embedded Queries *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, 2000.
25. Poellabauer, C., Schwan, K., West, R., et al. Flexible User/Kernel Communication For Real-Time Applications In Elinux *Proceedings of the Workshop on Real Time Operating Systems and Applications*, 2000.
26. Powell, M.L. and Miller, B.P. Process migration in DEMOS/MP *Proceedings of the 9th ACM symposium on Operating Systems Principles (SOSP '83)*, 1983.
27. Provos, N. and Lever, C. Scalable Network I/O in Linux *Proceedings of the USENIX Technical Conference, FREENIX track*, 2000.
28. Pyarali, I., Schmidt, D.C. and Cytron, R. Techniques for enhancing real-time CORBA quality of service. *Proceedings of the IEEE*, *91* (7). 1070-1085.
29. Rosu, D., Schwan, K. and Yalamanchili, S. FARA: A Framework for Adaptive Resource Allocation in Complex Real-Time Systems *he 4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, 1998.
30. Sundaram, V., Chandra, A., Goyal, P., et al. Application performance in the QLinux multimedia operating system *Proceedings of the 8th ACM International Conference on Multimedia 2000*, 2000.
31. Tennenhouse, D.L. Layered Multiplexing Considered Harmful. Rudin, H. and Williamson, R. ed *Protocols for High-Speed Networks*, 1989.
32. Welsh, M., Culler, D. and Brewer, E. SEDA: an architecture for well-conditioned, scalable internet services *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01)*, 2001.
33. West, R. and Schwan, K. Dynamic Window-Constrained Scheduling for Multimedia Applications *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, 1999.
34. White, S.R., Hanson, J.E., Whalley, I., et al. An Architectural Approach to Autonomic Computing *1st International Conference on Autonomic Computing (ICAC 2004)*, 2004.
35. Wiseman, Y., Schwan, K. and Widener, P. Efficient End to End Data Exchange Using Configurable Compression *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, 2004.
36. Wolf, M., Cai, Z., Huang, W., et al. SmartPointers: personalized scientific data portals in your hand *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (Supercomputing '02)*, 2002.
37. Xie, T. and Notkin, D. Checking Inside the Black Box: Regression Testing Based on Value Spectra Differences *IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004.
38. Yuan, W. and Nahrstedt, K. Process group management in cross-layer adaptation *Multimedia Computing and Networking 2004*, 2004.