JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©**JOT, 2005**

Vol. 4, No. 4, May–June 2005

Remote Objects: The Next Garbage Collection Challenge

Witawas Srisa-an and Mulyadi Oey
Department of Computer Science and Engineering
University of Nebraska-Lincoln, United States

In this paper, we present an analysis of *remote objects* in Microsoft's *Shared Source Common Language Infrastructure*. The contribution of our work is three-fold. First, we analyze the behavior of remote objects. We find that the basic behavior of these objects significantly differ from the ones of local objects, which have been thoroughly studied in the past. We also study the behavior of remote objects with respect to different activation modes (i.e. single-call, singleton, and client-activated). Second, based on those behavioral differences, we study the impacts of managing remote objects with a generational garbage collector that is designed essentially to manage local objects. We find that the garbage collection efficiency degrades significantly when the heap is interspersed with both local and remote objects. Third, we suggest various optimization techniques to improve the garbage collection efficiency in distributed objects environments.

1 INTRODUCTION

Web services computing is an emerging technology for developing and deploying distributed applications. The main goal of Web services is to provide an infrastructure for applications to communicate with each other using the World Wide Web [24, 33]. By using Web services, applications from different system architectures (e.g. mainframe, client/server, etc.) can easily exchange information without having to rely on vendor specific middleware components.

Currently, the two major technologies that support Web services are Sun's *Java 2 Enterprise Edition* (*J2EE*) and Microsoft's *.NET Framework*, which is based on the *Common Language Infrastructure* (*CLI*) standard [16]. According to industry observers, Sun will have 40% of the new enterprise application market and Microsoft will have 30% by the end of this year [22]. They also reported that "Web services will become the dominant distributed computing architecture in the next ten years and will eventually define the fabric of computing" [19]. Thus, it is not surprising that the revenue for Web services is expected to be in excess of twenty one billion dollars by 2007 and twenty seven billion dollars in 2010 [19].

In addition to Web services, distributed applications can be developed and deployed using Microsoft's *.NET Remoting* framework. Semantically, remoting is very similar to Java *Remote Method Invocation* (*RMI*). However, remoting is architecturally different from RMI in that remoting does not require stubs nor interfaces [34]. In .NET, Web services is a special type of remoting that runs under *Internet Information Services* (*IIS*) [2]. Remoting is not only utilized for cross-platform communication and heterogeneous systems, as in Web Services, but also is optimized for communication between .NET-centric applications. More importantly, .NET Remoting provides mechanisms that work with stateful objects. This feature will potentially allow remoting to be the foundation of the next distributed applications generation [20].

To the best of our knowledge, there have been no previous work to characterize the behavior of distributed objects related to either Remoting or Web services. In addition, the information about the effect of these objects on the garbage collector performance is very limited. This paper represents one of the first attempts to study the behavior of remote objects and their impacts on the *garbage collection* (*GC*). The contributions of our work are as follows:

- Characterization of remote objects—we classify remote objects as any objects created directly or indirectly to service requests from remote clients [17]. Our study encompasses different *activation* modes that include both *server-activated* (*single-call* and *singleton*) and *client-activated*. The results of our study show that the majority of remote objects in our benchmark program live significantly longer than local objects when singleton and client-activated modes are used.

- Analysis of generational garbage collection—we analyze it by monitoring the number of remote objects that are long-lived. We hypothesize that the efficiency of generational garbage collector is greatly reduced because the heap is interspersed with both local and remote objects; and those local and remote objects behave differently. The experiments show that the long-lived remote objects greatly degrade the efficiency of the garbage collector.

To perform our study, we have created a calendar program that can be configured to use different activation modes. The workload is varied by changing the number of clients that are simultaneously requesting services. It is worth noting that our benchmark program is created to represent the basic mechanisms of invoking remote methods and interfacing with remote objects. Our application is intended to be a case-study of remote object behavior based on different activation methods, which are fundamental to distributed application development in .NET infrastructure. In spite of a small amount of work performed for each client's request, we have already seen that remote objects have longer lifespan than that of the local ones in our application. Thus, with heavier/real-world workload, the lifespan of remote objects may even be longer and result in less efficiency for generational garbage collection.

The remainder of this paper is organized as follows. Section 2 introduces related work and background information. Section 3 discusses the rationales for our hypothesis. Section 4 presents the experiments. Section 5 presents the results obtained from our experiments. Section 6 discusses the future work and the last section summarizes this paper.

2 RELATED WORK AND BACKGROUND

This section provides an overview of other studies of the object behavior. It also provides background information related to the *Shared Source Common Language Infrastructure* (*SSCLI*) which is the experimental platform of our research. The related work is provided in section 2 and the background information is provided in section 2.

Related Work

As of now, very little research effort has been spent to study the object behavior in distributed Object-Oriented applications. While the extensive study of SPECjvm98 [26] benchmarks by [4] yielded many insights into the allocation and garbage collection behavior of Java programs, SPECjvm98 does not represent distributed application and server workloads. In [32], a study was performed to investigate whether the CLI implementations can be "useful for high-performance computing." A large set of benchmarks was used to compare the performance of four CLI implementations. Again, none of the benchmarks represent server-type and distributed workloads. In [11], a performance study of SPECjAppServer2002 [28] was performed. SPECjAppServer2002 is an application Server benchmark based on the Java 2 Enterprise Edition technology. The study mainly concentrated on the micro-architecture level and not at the object behavior level.

In [1], the the effectiveness of pretenuring is studied. Pretenuring is an extension to the generational garbage collection to more efficiently managed long-live objects. A profile based approach is used to select objects for pretenuring. The idea is to provide profile information to the compiler so that the optimal object's birthplace is available at runtime. They classify objects into three different classes-short-lived, long-lived, and immortal. They report up to 30% improvement in GC time and 7% improvement in the overall system performance. It is worth noting that their study is mainly based on desktop applications with one server that is not distributed in nature. We expect that their approach may generate higher performance gains in distributed environment.

In [10], a thorough characterization of memory system behavior of SPECjbb2000 and SPECjAppServer2001 was performed. SPECjbb2000 [29] is a benchmark to evaluate the performance of servers running typical Java business applications. It represents an order processing application for a wholesale supplier. SPECjAppServer2001 [27] is a client/server benchmark to measure the performance of Java Enterprise Application Servers using a subset of J2EE API's in a complete end-to-end web application. Their work [10] mainly studied the performance of cache memory with a portion of the work for a study of the effect of garbage collection on cache performance. In [13, 21], workload characterization of SPECjbb2000 and VolanoMark [12] was performed. VolanoMark is a multithreaded chat room application. Again, both studied the performance at the computer architecture level and not from the object perspective.

Background Information

Shared Source Common Language Infrastructure (SSCLI)

The main objective of the CLI is to allow programmers to develop component-based applications where the components can be constructed using multiple languages (e.g. C#, C++, Python, etc.). ECMA-335[1] (CLI) standard describes "a language-agnostic runtime engine that is capable of converting lifeless blobs of metadata into self-assembling, robust, and type-safe software systems" [31]. There are several implementations of this standard that include Microsoft's *Common Language Runtime* (*CLR*), Microsoft's Shared Source Common Language Infrastructure (SSCLI), Microsoft's .NET Compact Framework, Ximian's Mono project, and GNU's dotnet project. For this research, we use the SSCLI due to the availability of the source code. Moreover, it seems to be the most mature implementation than the Mono or GNU's dotnet projects.

SSCLI is a public implementation of ECMA-335 standard. It is released under Microsoft's shared source license. The code base is very similar to the commercial CLR with a few exceptions. First, the SSCLI does not support ADO.NET and ASP.NET which are available in the commercial CLR. ADO.NET is a database connectivity API and ASP.NET is a web API that is used to create Web services. However, the SSCLI does support *remoting*, which enables us to construct basic Web services mechanisms via remoting with *Simple Object Access Protocol* (*SOAP*) encoding and *Hypertext Transfer Protocol* (*HTTP*) communication channel. SOAP is an open standard sanctioned by the World Wide Web Consortium (W3C). Second, the SSCLI uses a different *Just-In-Time* (*JIT*) compiler from the CLR. The latter provides a more sophisticated JIT compiler with the ability to pre-compile code. Notice that both implementations of the CLI adopt JIT compilation and not interpretation mode as in some earlier Java Virtual Machine implementations. Third, it is designed to provide maximum portability. Thus, a software layer called Portable Adaptation Layer (PAL) is used to provide Win32 API for the SSCLI. Currently, the SSCLI has been successfully ported to Windows, FreeBSD, and MacOS-X operating systems.

Remotable versus Remote Objects

The .NET Remoting framework classifies objects into two categories: nonremotable and remotable. There are three categories of remotable types: *marshal-by-value*, *marshal-by-reference*, and *context-bound* [15]. *Marshal-by-value* remotable type is declared by using the *Serializable* attribute. *Marshal-by-reference* remotable type is defined by deriving from *System.MarshalByRefObject*. Simply deriving from this class enables the instances (plus the methods) of the type to be remotely-accessible. Context-bound remotable type is a refinement of marshal-by-reference type. Context-bound type is derived from *System.ContextBoundObject* class, which itself inherits from System.MarshalByRefObject. Therefore, remotable objects can be "accessed outside their application domain or context

[1]European Computer Manufacturers Association

using a proxy, or they can be copied and these copies can be passed outside their application domain or context; that is, some remotable objects are passed by reference and some are passed by value" [18].

In addition, there are two ways to activate remotable objects—client and server activations. The *server-activated* objects can be further categorized into *single-call* and *singleton*. A *client-activated* object is a server-side object whose lifetime is mainly controlled by the client application [2]. It is commonly used in multi-tier client/server applications that allow clients to make a series of method invocations on the same server to complete one long and complex business transaction [17]. On the other hand, the lifetime of a *server-activated* object is managed by the server. A single-call object is created by the server to serve only one client request. When the client invokes a method on a single call object, the object constructs itself, performs appropriate actions, and then is subject to garbage collection [2]. Thus, single-call objects are stateless. Single-call is often used for load balancing [14]. On the contrary, a singleton object is created by the server to serve multiple clients that share information. As a result, singleton objects are stateful and tend to live for the duration of a program [2]. Singleton is often used as listener threads in web server and database server [3].

In this paper, we define *remote objects* as objects created to service remote requests. Specifically, remote objects are objects created within a remote method; moreover, if the remote method calls other methods, objects created within those methods are also considered as remote objects. Thus, our definition of remote objects encompasses marshal-by-value and marshal-by-reference remotable types as well as all connected objects that they reference. Figure 1 shows the difference between remotable and remote objects. The squares denote remotable objects and the three circles within a semicircle denote remote objects. Those remote objects are created by a remote method within the remotable object as part of method calls by the clients.

Garbage Collection in the SSCLI

The SSCLI adopts generational collection technique to manage objects. It is worth noting that any objects larger than 85,000 bytes are considered large objects and managed differently. The generational scheme employs two generations: ephemeral and mature generations. The default sizes as specified in the SSCLI source code are 800 K-bytes and 1 M-bytes, respectively. Intergenerational pointers are maintained using card marking [9, 25] where each card is 4 K-bytes. Objects from unmanaged code are also managed by the garbage collector; handle list is used to store references coming from unmanaged environment. In this approach, all objects that are not large objects are created in the ephemeral space. Any objects that survive ephemeral collection would then be moved to the mature region.

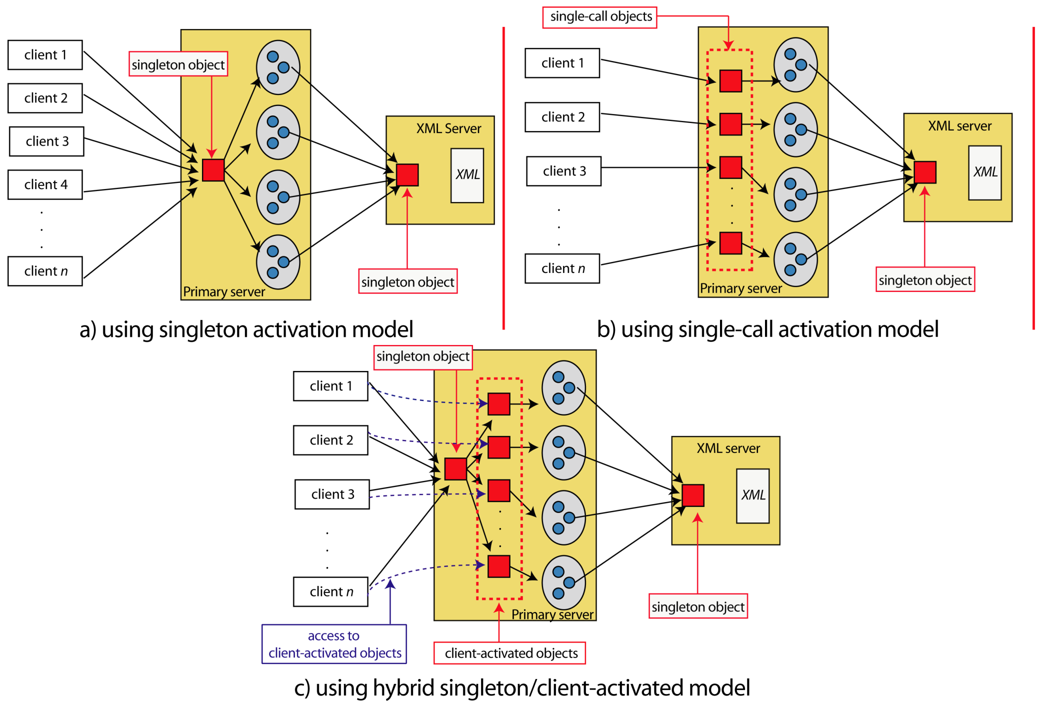

Figure 1: Overview of different activation models in the benchmark program.

3 RATIONALE

As stated earlier, the main goals of this research are *(i)* to characterize the behavior of remote objects and *(ii)* to study their effects on garbage collection. We hypothesize that the behavior of remote objects differ from those of local objects and, thus, may degrade the efficiency of generational garbage collectors (such as the one in the SSCLI). The rationales for our hypothesis are as follows:

1. In distributed computing, the lifespan of remote objects would be much longer than local objects. We base our hypothesis on the work by [6] where the experimental results show that over 80% of objects that are connected together tend to die together. Based on [2], singleton object tends to live for the duration of the server program. For a client-activated object, the default lifespan used in the SSCLI is five minutes. If the client still needs to access the object after the initial lease time, re-leasing mechanism can be used to further extend the object's life. It is also possible that programmers can override this initial lease time with other values (e.g. 30 seconds). Since these objects are the roots of objects clusters, other objects connected directly or indirectly to either singleton or client-activated objects should be long-live as well.

2. Since all objects (except those larger than 85,000 bytes) are initially created in the ephemeral space and more remote requests result in more remote objects being created, remote objects will occupy the majority of the ephemeral space. As mentioned

> in 1, these objects may be long-live. Therefore, the initial premise of generational schemes that “the majority of objects die young” may no longer be true in remoting applications. In fact, the generational garbage collection efficiency may degrade significantly.

Our goal of this paper is to prove the correctness of our hypothesis. We perform experiments to show that there are a lot of long-lived objects in distributed applications. Once we establish the fact that a large number of objects are long-lived, we perform more analysis to show that the majority of these long-lived objects are remote. Lastly, to make sure that we have adequate ephemeral generation size for our experiment, we compare the survival rate of remote and local objects based on the allocation request for each type in between two GC invocations. In doing so, we demonstrate that remote objects are indeed long-lived.

4 EXPERIMENT

This section describes our experimental environment. The description includes the detailed overview of our benchmark programs, the detailed modifications made to the SSCLI, and the analysis tools to study the behavior of remote objects. We conduct our experiments on Windows XP Professional workstation running on 2.6 GHz Pentium IV with Hyper-Threading [8]. The system has 512 MB of memory.

Benchmark Program

Currently, there are no standardized benchmark programs for distributed application in .NET environment. In addition, the SSCLI does not have a complete set of libraries as in the commercial CLR. Therefore, the benchmark for our experiment must utilize basic mechanisms to invoke and interface with distributed objects and execute correctly in the SSCLI. As a result, we create a benchmark program that can be easily customized to use different activation modes and to support a large number of concurrent clients. As stated in the introduction, it is **not** our intention to create a benchmark program that emulates real-world workloads as different applications would exhibit different workload behavior. On the contrary, our application is intended to be a case-study of remote object behavior based on different activation modes (singleton, single-call, and hybrid). The three activation modes are used. We then compare the differences in object behavior among different modes. The basic characteristic of the benchmark program is summarized in Table 1.

Our benchmark is a three-tier client/server calendar program, as depicted in Figure 1. The primary server waits for client connections, forwards each client’s requests to the secondary XML server, which queries corresponding information from an XML file, and then returns the results back to the clients. Since the primary and secondary servers operate in two different application domains, they also need remoting mechanism to communicate with one another. The connection between the primary server and the XML server is

Application	Configuration	Number of Clients	Allocated Objects by bytes	Allocated Objects by objects	Methods Compiled	Thread Created
Calendar	single-call	1	972,115	13,502	3,433	6
		15	6,061,786	98,067	3,448	17
		30	11,438,351	188,033	3,445	24
		60	22,697,904	374,633	3,731	29
	singleton	1	971,989	13,600	3,433	6
		15	5,974,442	96,123	3,455	19
		30	11,251,682	184,679	3,455	23
		60	22,210,707	369,422	3,632	29
	client-activated/hybrid	1	1,029,305	14,560	3,459	6
		15	6,650,288	100,979	3,473	20
		30	12,506,027	192,807	3,467	23
		60	24,633,000	384,595	3,695	25

Table 1: Benchmark Characteristics

established using the singleton mode. It is worth noting that we use HTTP channel and SOAP formatter to emulate the behavior of Web-service applications.

The server and the client programs can be configured to use single-call, singleton, or client-activated mode for the remotable object activation. The basic overview of the three activation models is given in Figure 1. In our implementations of the single-call and the singleton models, every client makes two similar requests—each request is to retrieve all appointment details for a particular day. For the client-activated model, each client makes two different requests—the first one is to retrieve only the appointment subjects for a particular day and the second is to get the detailed information of a particular appointment.

It is worth noting that a hybrid implementation is used to exercise the client-activated model. The hybrid implementation is preferred to its pure client-activated model counterpart because no compiled server objects needs to be shipped to the client. Shipping compiled server objects violates the principle of distributed objects and is undesirable due to deployment and versioning issues [17]. In the hybrid model, a client makes a connection through a main singleton object in the server, which then instantiates a client-activated object for the corresponding client. The singleton object serves as a factory object to create query objects for the calendar program. One additional proxy object is then set up on the client side so that it can directly access the newly-instantiated client-activated object in the server. The terms client-activated and hybrid are used interchangeably throughout this paper.

We vary the testing workload by changing the number of simultaneous clients (i.e. 15, 30, and 60 clients) for each of the three—single-call, singleton, and hybrid—models. The benchmark allocates over 384,000 objects and 24 M-bytes in the hybrid model with 60 clients. Notice that the number of objects created in our benchmark is similar to the number of objects created in well accepted client/server benchmarks such as VolanoMark [12]. For instance, in our benchmark with 15 clients, the number of objects is about 200,000. For VolanoMark with 20 clients, there are about 257,000 objects [7]. The program also compiles over 3,400 methods during its execution. The comparison of object behavior among these three models is given in section 5.

Trace Generation and Analysis

We perform our experiments by running the benchmark with different configurations on the modified SSCLI. The modified SSCLI is an SSCLI in which the source code is modified to generate information related to thread creation, JIT compilation, object allocation, garbage collection, thread termination, etc. The information is then piped to a trace file. The basic overview of our experimental platform is provided in Figure 2.

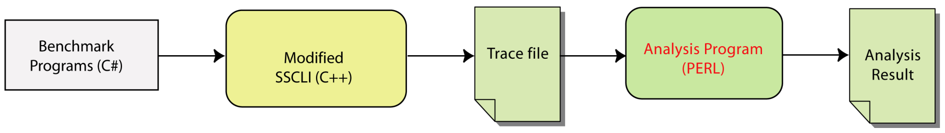

Figure 2: Overview of the experimental platform.

Each time a remote method is entered, the modified SSCLI will generate a line in the trace file. The line includes the method name and a unique identification number of a particular thread executing the method. Any objects created by the same thread while the remote method is still active are considered to be remote objects. Since our benchmark is multithreaded, we segregate concurrent allocation requests from different threads by generating a thread identification number as part of the object allocation trace line. For example, since the workstation used for the experiment has a Hyper-Threading capability, it is possible for two threads from the same application domain to execute simultaneously. However, by recording the thread identification number as part of the object allocation trace line, we can precisely identify whether the object is local or remote.

It is also possible for the thread that executes a remote method to delegate some works to other threads in the same application domain. The most common approach is to create a pool of threads for the delegated tasks. Since these threads also perform works to satisfy remote requests, the objects created by these worker threads are also considered as remote objects.

The complete trace file is then used as the input of the analysis tool, which is used to identify each object's type (i.e. local or remote), calculate the garbage collection efficiency, compute the object size distribution, and so on. The result is then outputted to an analysis result file. The result is discussed in the next section (section 5).

5 EXPERIMENTAL RESULT

Basic Behavior of Remote Objects

We initially hypothesize that the majority of objects in the heap will be remote objects (i.e. remotable objects and any objects connected to them). However, as shown in Figure 3, this is not the case. The ratios of remote and local objects in both single-call and singleton models are close to half (52% for local and 48% for remote). For the hybrid model, each client request results in more local objects created. Such objects include leasing managers and other structurally complex objects for keeping information across states, to support additional runtime requirements. Therefore, the amount of space for local objects also increases as reflected in Figure 3.

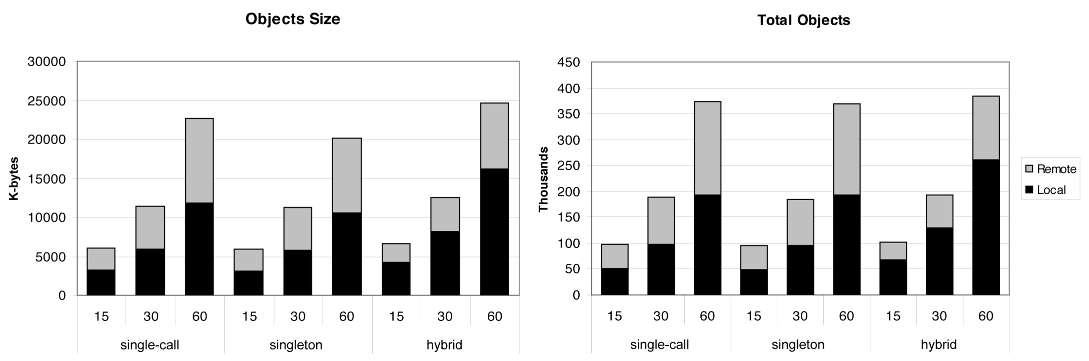

Figure 3: Ratio of remote versus local objects in the benchmark applications.

We also find very few differences between the local and remote object size distributions. Studies by [4, 35] find that the average object size of Java programs is less than 50 bytes. Another study by [23] finds that a high percentage of objects is smaller than 512 bytes. Figure 4 depicts the local and remote size distributions for all the three models with 60 clients. It shows that most of the objects (local and remote) are smaller than 128 bytes.

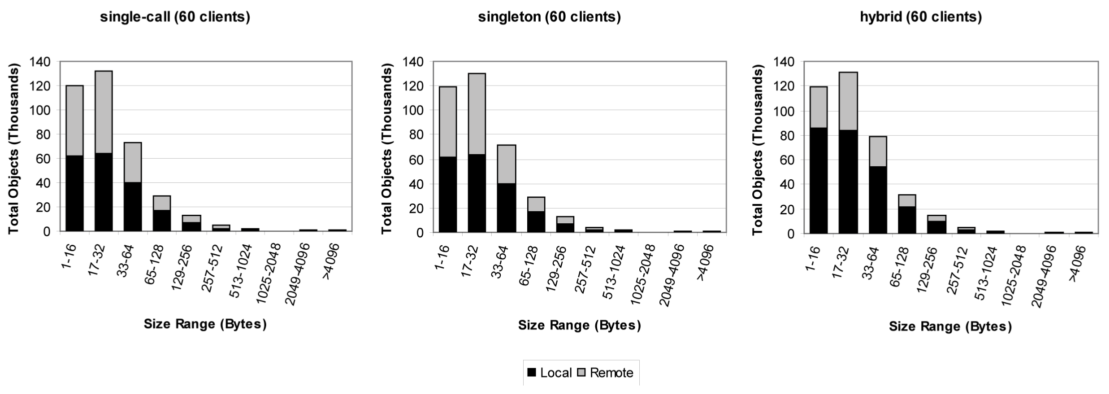

Figure 4: Size distribution of remote and local objects in the benchmark applications.

In terms of lifespan, we find that a large number of objects survive ephemeral collections. A detailed illustration of such objects is given in Figure 5. For example, in the single-call model with 30 clients, we find that the survival ratios are about 40%. Notice that when the number of clients increases to 60, the SSCLI enlarges the ephemeral generations from 800 K-bytes to 2,600 K-bytes in case of single-call, 3,500 K-bytes in case of singleton and 5,800 K-bytes in case of hybrid toward the end of the execution.

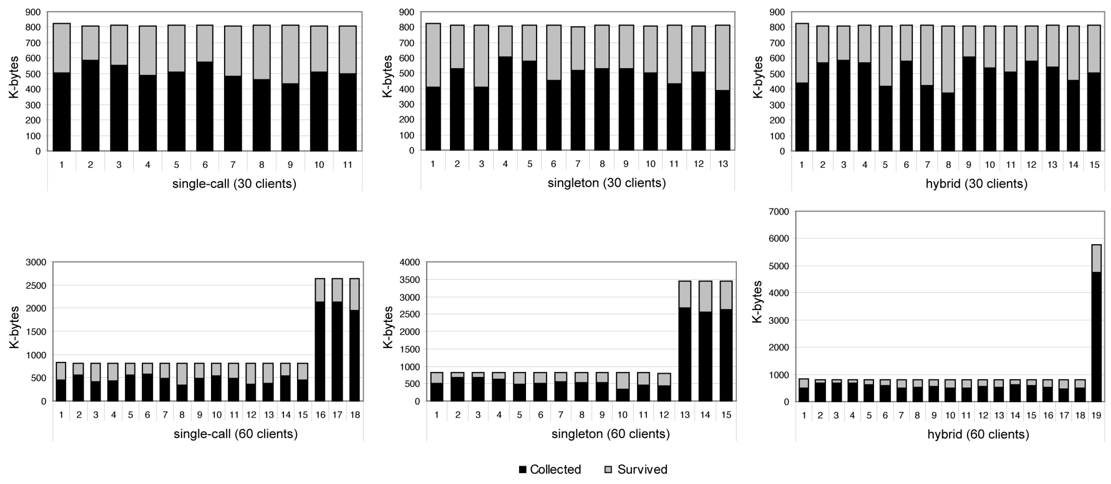

Figure 5: Illustration of surviving objects for each GC invocation.

Application	Configuration	Number of Clients	Survival Ratio			Ephemeral GC Invocation
			Min	Max	Average	
Calendar	single-call	30	27.79	46.85	37.38	11
		60	19.01	57.95	38.73	18
	singleton	30	25.29	52.35	39.44	13
		60	15.53	59.70	33.15	15
	client-activated/hybrid	30	24.74	53.71	36.71	15
		60	16.62	44.59	30.36	19
Word count (desktop)		-	10.5	25.30	15.93	11

Table 2: Survival rates

Table 2 provides the aggregated results of our finding. For example, the hybrid model with 30 clients has the survival rates ranging from 25 to 53%, with an average of 36.71%. We find that the averages of the survival rates fall between 30 to 39% in all configurations. For a comparison, we use a *word count* utility program, which is included as part of the SSCLI test suite, to represent a common desktop application. The utility is used to count the number of characters, words, and lines in one of our trace files. The application allocates 8 M-bytes of memory and invokes 11 ephemeral garbage collections. We find that the objects' average survival ratio in the application is only 16% and that the maximum survival ratio is only 25%, which is smaller than the average survival ratio in our remoting benchmark application.

Distribution of Long-lived Objects

In this section, we perform a detailed comparison between the distributions of long-lived local and long-lived remote objects. By monitoring the objects that are promoted during each ephemeral garbage collection, we can compare the amount of remote and local objects that are long-lived. For example, we calculate the amounts of local (L_o) and remote (R_o) objects in the young generation before and after each GC. We also calculate the amounts of local (L_m) and remote (R_m) objects that survive the ephemeral collections (i.e. promoted to the mature generation). Thus, we can express the percentage of survival as $PL = \frac{L_m}{L_o}$ and $PR = \frac{R_m}{R_o}$,where PL and PR are the percentages of survived local objects and survived remote objects, respectively.

Figure 6 provides detailed distributions between long-lived remote and local objects per GC invocation. The main objective is to illustrate that the majority of long-lived objects are remote. Due to space limitation, we only include the distributions in the three models with 60 clients as representatives of other workloads. As illustrated in the figure, most objects that survive the majority of GC invocations are remote objects.

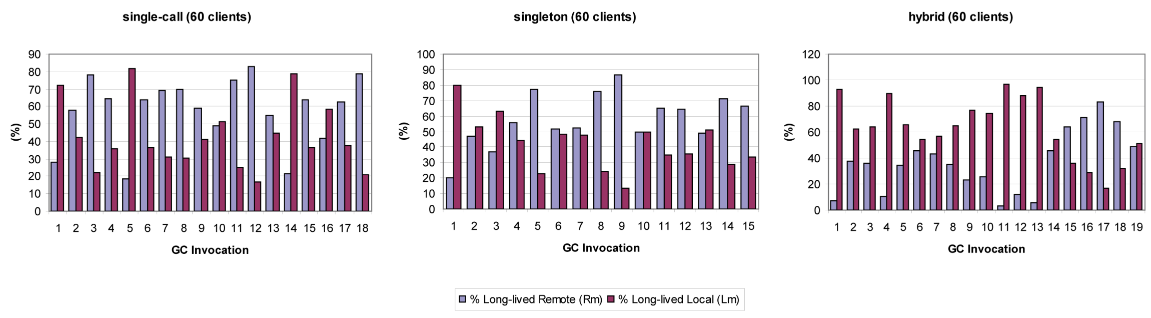

Figure 6: Distributions of long-lived objects per GC invocation.

Potential Effect to the Local Collector

As stated earlier, we hypothesize that the introduction of remote objects to the same heap would degrade the efficiency of generational GC. As a reminder, the efficiency of generational GC is mostly determined by the amount of objects that has to be migrated to the mature region. The basic rationale behind generational scheme is that “the majority of objects die young” [9] and thus, less object will be promoted during each ephemeral collection. The efficiency of a garbage collector is defined as the amount of garbage collected in a given time [9]. As illustrated thus far, remote objects in distributed applications tend to live longer. Therefore, the amount of garbage collected in each invocation would be less and the collection time would be longer. We intend to prove our hypothesis by showing that the remote objects are indeed long-lived.

To perform our preliminary analysis, we compare the behavior of local and remote objects based on the amount of allocation during an ephemeral garbage collection. To illustrate our technique, let us consider the following. Figure 7 depicts the distribution of allocated objects that are remote and local. We record the information just prior to each GC. For example, we investigate the fifth and sixth GC invocations. We find that the amount of remote allocation in the fifth invocation is nearly identical to the amount of local allocation in the sixth. Again, the size of ephemeral generation is maintained at 800 K-bytes. What we want to monitor here is the amount of survived objects after an ephemeral GC is invoked. In this example, we find that the percentage of long-lived local objects in the 5th invocation is 30% whereas the percentage of long-lived remote objects in the 6th invocation is 52%. In this approach, we eliminate one of the common pitfalls in lifespan study of the generational scheme which is the adequacy of the generation size. It is possible that the default ephemeral generation size is too small and therefore a large number of objects are migrated regardless whether they are local or remote. When the young generation size is too small, both remote and local objects would have a very similar chance of getting promotion. However, we can see that generation size is not a big factor in our analysis. As clearly shown in Figure 8, given the same allocation space, remote objects are likely to live longer than local objects.

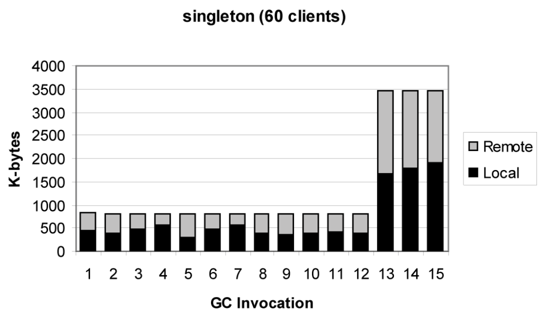

Figure 7: Allocation distribution of remote and local objects in singleton model with 60 clients.

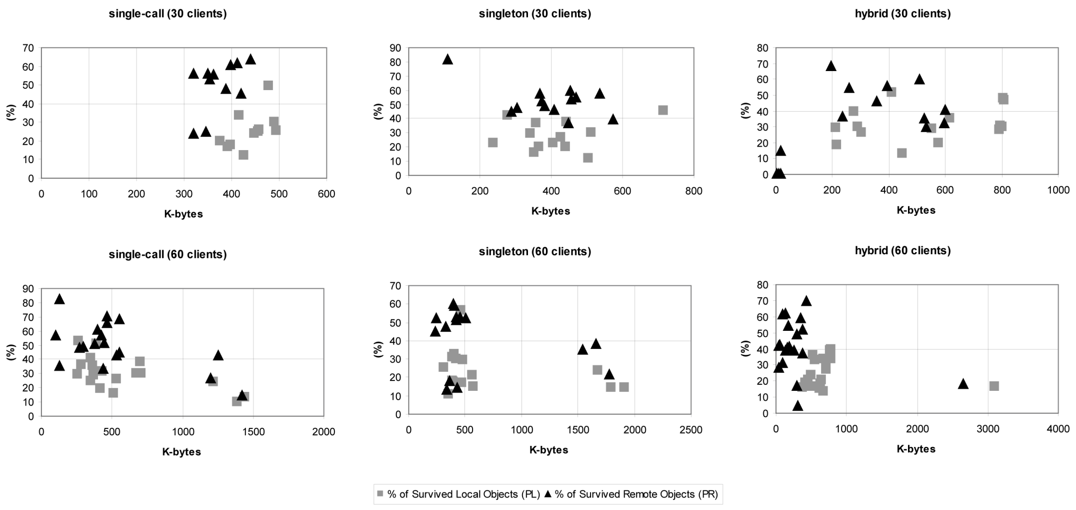

Figure 8: Lifespan comparisons between local and remote objects in three activation models.

To compare the lifespan of remote and local objects, we focus on the amount of survived objects during ephemeral collections. Figure 8 compares the amount of survivors based on the young generation occupancy (L_o and R_o) in single-call, singleton, and hybrid activation models. The X-axis represents the amount of space allocated, in K-bytes, prior to each ephemeral garbage collection. The Y-axis represents the percentage of survived objects. Since we record L_o, R_o, L_m, and R_m before and after each ephemeral GC, we can compare the survival ratios based on the amount of object allocated. When the number of simultaneous clients are 1 and 15, the experimental results indicate that the lifespan of remote objects is longer than that of local objects. When we increase the number of simultaneous clients to 30 and 60, it is conclusive that the lifespan of remote objects is much longer than that of local objects. In the majority of ephemeral garbage collection invocations, over 40% of the remote objects survive. On the other hand, very few GC invocations indicate the survival rates to be above 40% for local objects. It is worth noting that isolated data points in cases of 60 clients (e.g. between 1,500 and 2,000 K-bytes in singleton) are the result of ephemeral space enlargement performed by the SSCLI to accommodate heavier workload. The actual enlarged heap size was previously presented in Figure 5.

Figure 9 shows the percentage of total ephemeral garbage collections that results in more than 40% survival rate for local and remote objects. In case of singleton model with 60 clients, there are 15 ephemeral collections. Out of these 15, there is one collection in which 40% of local objects still alive and there are nine collections in which 40% of remote objects still alive. In other words, only in 6.67% of the total ephemeral collections that local objects have 40% survival rate. However, 40% survival rate for remote objects happen in 60% of the total ephemeral collections.

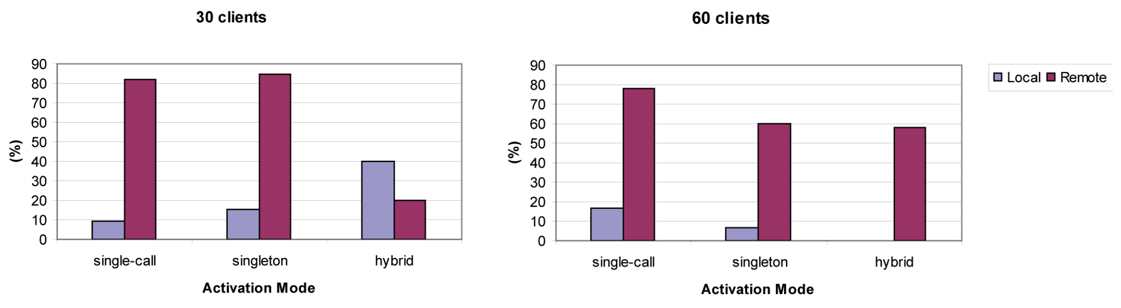

Figure 9: Percentage of ephemeral collections that results in over 40% survival rate.

From our experimental results, it is clear that remote objects are long-lived. Therefore, the garbage collector will be likely to spend additional time and resources to promote these long-lived remote objects.

6 FUTURE WORK

This paper represents one of the first attempts to study object behavior in distributed Object-Oriented applications. Based on the experimental results, we can see that remote objects have different lifespan compared to local objects. For this experiment, we use our micro-benchmark that can be configured to use different object activation modes and support various number of concurrent clients. For future work, we will create a set of benchmarks that performs more complex functions and utilizes more advanced threading techniques (e.g. thread pool for work delegation). As stated earlier, there is presently no standardized benchmarks for remoting.

In this paper, we demonstrate that remote objects tend to be long-lived. In the future, we will experiment with various optimization techniques to improve the efficiency of generational garbage collection. For example, adaptive and dynamic pretenuring [1] can be used to directly create remote objects in the mature generation. We will also experiment to manage remote objects separately. In addition, our results clearly indicate that workload can greatly affect the object creation and garbage collection. We plan to investigate different algorithms that would pre-enlarge the heap during the thread creation or *thread join*; moreover, we plan to investigate the amount of objects shared by multiple threads in distributed applications. If very few objects are shared, it is possible to improve the performance by adopting algorithms such as thread specific and thread local heaps [5, 30].

7 CONCLUSION

Based on the experimental results, remote objects in distributed applications have much longer lifespan than that of local objects. We determine the lifespan of each object type by comparing the amount of survived objects given the same amount of allocation space. We find that in most instances, local objects rarely have survival rates of more than 40%. On the contrary, remote objects constantly maintain the survival rates of 40% or more. We conclude that a typical generational collector, such as the one in the SSCLI, would lose its efficiency in a distributed environment because more time will be needed to promote the long-lived remote objects.

8 ACKNOWLEDGEMENT

We would like to thank Dr. Sebastian Elbaum for offering many valuable suggestions. This project was partially supported by the National Science Foundation under grant CNS-0411043 and Univ. of Nebraska Layman's Award.

REFERENCES

[1] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for java. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, Tampa Bay, FL, 2001.

[2] D. Browning. Integrate .net remoting into the enterprise. In *.NET Magazine*, November 2002.

[3] Cunningham. Singleton pattern. Cunningham and Inc. http://c2.com/cgi/wiki?SingletonPattern.

[4] S. Dieckmann and U. Holzle. A study of the allocation behavior of the specjvm98 java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99), Lecture Notes on Computer Science, Springer Verlag*, Lisbon, Portugal, June 1999.

[5] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for java. *SIGPLAN Not.*, 38(2 supplement):76–87, 2003.

[6] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *ISMM*, Berlin, Germany, 2002.

[7] W. Huang, Y. Qian, W. Srisa-an, and J.M. Chang. Object allocation and memory contention study of java multithreaded applications. In *Proceedings of 23th IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 375–382, Phoenix, Arizona, April 15-17, 2004.

[8] Intel. Hyper-threading technology. http://www.intel.com/technology/hyperthread/.

[9] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, New York, NY, 1998.

[10] M. Karlsson, K. E. Moore, E. Hagersten, and D. A. Wood. Memory system behavior of java-based middleware. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 217. IEEE Computer Society, 2003.

[11] S. Kumar. A performance study of specjappserver2002. In *First Workshop on Managed Run Time Environment Workloads*, San Francisco, CA, March 23, 2003.

[12] Volano LLC. Volanomark benchmark. http://www.volano.com/benchmarks.html.

[13] Y. Luo and L. John. Workload characterization of multithreaded java servers. In *Proceedings of 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 128–136, Austin, Texas, 2001.

[14] N. Maharaj. .net remoting-part 3 remoting examples. In *C# Help*, http://www.csharphelp.com/archives2/archive423.html.

[15] S. McLean, J. Naftel, and K. Williams. *Microsoft .NET Remoting*. Microsoft Press, 2003.

[16] Microsoft. ECMA and ISO/IEC C# and common language infrastructure standards. http://msdn.microsoft.com/net/ecma/.

[17] Microsoft MSDN. Broker. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesBroker.asp.

[18] Microsoft MSDN. Remotable and non-remotable objects. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconremotablenon-remotableobjects.asp.

[19] Web Hosting Industry News. http://thewhir.com/marketwatch/idc020503.cfm.

[20] I. Rammer. *Advanced .NET Remoting*. Apress, 2002.

[21] P. Seshadri and A. Mericas. Workload characterization of multithreaded java servers on two powerpc processors. In *In Proceedings of Fourth Annual Workshop on Workload Characterization*, Austin, Texas, December 2001.

[22] D. Sholler. .net seen gaining steam in dev projects. In *ZD Net*, http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2860227,00.html, April 2002.

[23] T. Skotiniotis and J. M. Chang. Estimating internal memory fragmentation for java programs under the binary buddy policy. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2001)*, pages 85–92, Tucson, Arizona, Nov. 4-6, 2001.

[24] B. Sleeper and B. Robins. The laws of evolution: A pragmatic analysis of the emerging web services market. http://www.stencilgroup.com/ideas_scope_200204evolution.pdf. The Stencil Group.

[25] P. Sobalvarro. A lifetime-based garbage collector for lisp systems on general-purpose computers. Technical Report AITR-1417, MIT, February 1988.

[26] SPEC. Standard performance evaluation corporation jvm98. http://www.spec.org/osg/jvm98.

[27] SPEC. Standard performance evaluation corporation jappserver2001. http://www.spec.org/jAppServer2001.

[28] SPEC. Standard performance evaluation corporation jvm98. http://www.spec.org/jAppServer2002.

[29] SPEC. Standard performance evaluation corporation jbb00. (http://www.spec.org/jbb2000).

[30] B. Steensgaard. Thread-specific heaps for multithreaded applications. In *Proceedings of International Symposium on Memory Management*, pages 18–24, Minneapolis, MN, October 15-16, 2002.

[31] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly and Associates, 2003.

[32] W. Vogels. Benchmarking the cli for high performance computing. *IEE Proceedings Software*, 150:266–274, October 2003.

[33] W3C. World Wide Web Consortium. http://www.w3.org/2002/ws/.

[34] R. Wiener. Remoting in c# and .net. *Journal of Object Technology*, 3:83–100, January 2004.

[35] Q. Yang, W. Srisa-an, T. Skotiniotis, and J. M. Chang. Java virtual machine timing probes: A study of object lifespan and garbage collection. In *Proceedings of 21st IEEE International Performance Computing and Communication Conference (IPCCC-2002)*, pages 73–80, Phoenix Arizona, April 3-5, 2001.

ABOUT THE AUTHORS

Witawas Srisa-an is an Assistant Professor in the Department of Computer Science and Engineering (http://www.cse.unl.edu) at University of Nebraska-Lincoln, USA. His research interests include computer architecture, Object-Oriented systems, programming languages, and distributed systems. He can be reached at witty@cse.unl.edu.

Mulyadi Oey is a second year MS student and research assistant in the Department of Computer Science and Engineering at University of Nebraska-Lincoln, USA. He can be reached at moey@cse.unl.edu.