A C++ Workbench with Accurate Non-Blocking Garbage Collector for Server Side Internet Applications

Piotr Kołaczkowski and Ilona Bluemke
{P.Kolaczkowski, I.Bluemke}@ii.pw.edu.pl

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland

Abstract. At the Institute of Computer Science Warsaw University of Technology a workbench for building server-side, dependable, Internet applications was designed and implemented. This workbench is a collection of C++ classes. The design and implementation of these classes are briefly described. The most important part of the workbench is the web server, implemented as a C++ class that can be used in a standalone application. To implement the web server a precise, concurrent garbage collector was designed. Our garbage collector is based on the concurrent mark-and-sweep algorithm and smart pointer pattern. It makes the risk of memory access faults or memory leaks much lower than in standard C/C++ applications. The advantages of our workbench are shown in some experiments. We have measured the overhead of our garbage collector and the performance of the workbench. A comparison with other systems is also given.

1 Introduction

Automatic memory management techniques have been successfully employed for years. Existence of a garbage collector increases the dependability of applications and makes programming easier. A garbage collector is a core component of many programming environments e.g. for languages like Java or C#. On the contrary, originally designed without automatic memory management, uncooperative environments are still used. The C++ language belongs to such environments. Although conservative garbage collection techniques are quite popular [1], the design of an accurate, non-blocking garbage collectors is a very complex task and those collectors are rather uncommon. In the paper we show, how the accurate, non-blocking garbage collector designed and implemented at the Institute of Computer Science Warsaw University of Technology [2, 3] was employed in a C++ workbench for dependable server side Internet applications. A very important issue is to provide the high quality and the dependability of such applications. Although there are many techniques for building an Internet application [4], it is not easy to develop it in widely used C++ programming language. A C++ programmer can use only few techniques: CGI [5], FastCGI [6] or .NET environment. One of the problems that has to be solved by the programmer in C++ programs is the memory management. Errors in the memory management are often the source of defects and security holes.

The paper is organized as follows. In the next section some general memory management approaches in C++ applications are briefly described and their usability for building Internet applications is discussed. In Section 3 the implemented workbench is

Please use the following format when citing this chapter:

Kołaczkowski, P., Bluemke, I., 2006, in IFIP International Federation for Information Processing, Volume 227, Software Engineering Techniques: Design for Quality, ed. K. Sacha, (Boston: Springer), pp. 15–24.

presented. The following aspects of the web server are mentioned: memory management, application interface, concurrency support, session handling. In Section 4 some experiments are described. The throughput and the response time of sample C++ applications, prepared with the workbench, were measured. The overheads of the garbage collector are examined. The final section contains conclusions.

2 Related work

Boehm and Weiser have developed a good and widely used conservative garbage collector for C and C++ languages [1, 7]. This garbage collector can work in one of the two modes: blocking or incremental. The blocking mode uses the *mark-and-sweep* algorithm. The incremental mode uses the *train* algorithm [8], which is much slower, but reduces the execution delays. In both modes the collector does not guarantee that all *dead* (inaccessible) data are removed, so some small memory leaks are possible. The probability of such leaks is higher for applications with larger heaps e.g. extensively caching webservers. Memory leaks can be disastrous in long-running applications.

Barlett proposed a generational, mostly-copying, conservative garbage collector for C++ [9], which according to the benchmarks presented in [10] performs better than the Boehm-Weiser's collector. It is also more accurate, because the programmer provides special procedures enabling the garbage collector to find pointers in objects. On the other hand, some parts of memory are still treated conservatively, so the problem of possible memory leaks remains. Additionally, the programmer can give erroneous relative pointer locations and mislead the garbage collector. This can be a cause of severe memory management failures.

Detlefs studied the possibility of using the C++ template metaprogramming techniques to achieve the garbage collector's accuracy [11]. *Smart pointers* can be used to track references between objects. This allows for the accurate garbage collection without the need to manually specify the relative pointer locations. Detlefs used this technique in a reference counting collector. His measurements show that reference counting can impose a time overhead of over 50% which is probably too high for being successfully used in a high performance web application.

In spite of some small programming inconveniences introduced by smart pointers (their usage differs a little from the C++ built-in pointers) [12], we proposed how to use them with a concurrent *mark-and-sweep* algorithm to get an accurate, non-blocking garbage collector [3]. Our research did not show how the garbage collector performs in a real-world application. Only benchmarks for single memory operations were done. Some interesting recent benchmarks can be found in [13], but these don't cover real-world server side applications, too.

Henderson designed a different technique for building an accurate garbage collector based on the code preprocessing approach [14]. The preprocessor inserts additional instructions into the original C code. These instructions enable the garbage collector to find exact pointer locations. Although Henderson didn't implement a multithreaded garbage collector, he proposed how to do it using his approach. He also performed some simple benchmarks and obtained promising results.

While garbage collection techniques were being improved, engineers and researchers were independently creating new ways of building internet server side applications. The latters can be divided into two main categories: scripts and applications servers. Scripts may be used for small applications. Severs are dedicated for more complex ones, even distributed. An overview of techniques for Internet applications can be found in [4].

The script is a file containing some instructions. By processing these instructions a WWW server is able to generate the Internet page. There are many kinds of scripts e.g.: CGI [5, 6], PHP [15], Cold Fusion [16], iHTML [17], ASP [18]. The script is invoked for each request by the WWW server. The script technique is simple but it can be used to build applications like portals or Internet shops. Some script languages e.g. PHP contain special constructs useful in such applications like: data exchange, access to data bases, interfaces based on MVC (Model View Controller) patterns. Due to the short time of living of the script process, scripts don't take much advantage of garbage collectors.

Application servers can be used for building complex, multilayered, distributed applications. Such applications may communicate with users by Internet browsers. The application is active all the time and receives HTTP requests from the browser. Java servlets and JSP [19, 20] operate this way. In application servers some optimization techniques can be included e.g. caching data or keeping a pool of open database connections. Servers often provide advanced services e.g. load balancing, distributed transactions, message queuing. Components for MVC model are also available. Application servers usually run on virtual machines. These environments need a lot of memory but the execution of the application is more efficient than in script interpreters. Due to longer run times, this approach usually requires employing a good garbage collector. Virtual machines like JVM or CLR often contain more than one built-in garbage collector implementation.

3 The C++ workbench

The C++ workbench designed and implemented at the Institute of Computer Science is dedicated to small and medium size Internet applications. A very important issue is to provide the dependability and the high quality of applications prepared with this workbench. Efficiency and good memory management had also high priorities in the design process.

Main functions required were sending/receiving text and binary data to/from Internet browsers, sending and receiving HTTP cookies, session handling. A set of C++ classes was implemented. These classes constitute two components: the WWW server and the garbage collector. The garbage collector enables the programmer to create software of higher quality than using standard, manual memory management schemes. Automatic memory manager would never deallocate the same memory region twice or deallocate memory being used, causing a memory access error or a potential leak. Although it is possible to avoid these errors without a garbage collector, using it can significantly reduce the total software production time. The garbage collector finds inaccessible objects by analysing references among objects. This collector is accurate and is able to find all inaccessible objects. Objects created with the functions provided by our collector are destroyed automatically. Our collector does not influence any code that manages memory

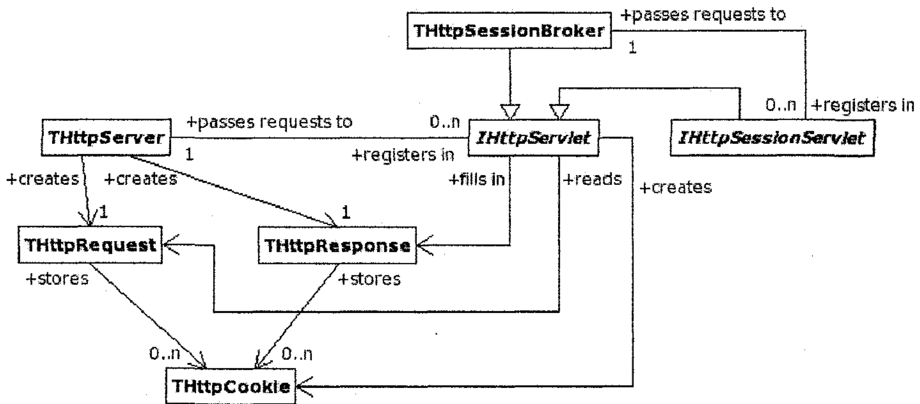

Fig. 1. Class diagram of the WWW server

manually. Objects created by built-in *new* operator should be destroyed manually. Automatic garbage collection introduces some overhead compared to manual memory management. Our garbage collector works concurrently with the application. Execution of application's threads is interrupted for a predictable, limited amount of time. This time does not depend on the number of objects and is so short, that will not be noticed by a user waiting for an Internet page. The implementation details of this garbage collector are given in [2]. Automatic memory management by this collector was used in other components of the C++ workbench i.e. the WWW server. The WWW server is a very important part of the C++ workbench. It is implemented by the class THttpServer presented in Fig. 1. This class interacts with the browser by the HTTP protocol. A programmer has to create an object of this class, set some attributes and call the Run method. As the server is one class only, several servers listening on different ports can be created. In Fig. 1 some other classes are also shown. These classes are used to improve functionality of the embedded WWW server and are described in sections 3.1 – 3.4.

3.1 Servlets

Servlets are objects registered in the server handling HTTP requests. Servlets are created by the programmer. Each servlet implements the IHttpServlet interface. The application programming interface of the workbench never uses raw C++ pointers to pass data to or from the class methods, instead it uses smart pointers provided by the garbage collector component. Hence, to register the servlet, a smart pointer to it must be passed to the server. The server with registered servlets, handles a request in the following manner:

1. The WWW server receives a request and checks its correctness. If it is incorrect, an appropriate error message is returned to the browser.
2. During the registration process the servlet receives an identifier. Basing on the URI identifier included in the request, the server identifies the servlet responsible for handling it. If the servlet can not be found, an error message with a HTTP 404 code is sent to the browser.

3. The server creates an object of the THttpRequest class and fills it with the data send by the browser. The object contains identifier of a resource requested by the client, HTTP headers and a pointer to the opened input stream.
4. The server creates an object of THttpResponse class.
5. The objects created in steps 3. and 4. are given to the servlet. The server waits for the next request.
6. The servlet reads data from the THttpRequest object and generates a response by writing appropriate data into the THttpResponse object.
7. The server closes the connection.

3.2 Receiving and sending data

When a HTTP request comes, the data sent by the Internet browser are written into the THttpRequest object, which is then passed to the servlet. This object has methods returning e.g.: the requested HTTP method (GET, POST or others), the identifier of the requested resource (URI), the HTTP cookies, the text and binary data sent in the 'body' of the request. The servlet receives also a THttpResponse object. This object is used to send a response to the web browser. The following methods in this object are available:

- setting a status code of the HTTP message,
- setting a header,
- setting, modifying or deleting HTTP cookies,
- setting content-type MIME of the sent document,
- opening a buffered output stream and sending the body of the response.

All arguments are passed either by value, by reference or by smart pointer. The raw C++ pointers are not used. This allowed to achieve an easy programming interface. Objects allocated on the heap inside the methods of the THttpRequest and THttpResponse classes are always created by the garbage collector.

3.3 Threads

At its start the server initialises a pool of threads. These threads are waiting on a semaphore for HTTP requests. The main thread listens on a local port (initially 80) and passes requests to one of the waiting threads. If the pool of waiting threads is empty, the main thread stops receiving new requests. The user can set the total number of threads in the pool. Handling concurrent request may cause some problems with common data accessed or modified by several threads at once. To alleviate this problem some simple mutex based synchronization is provided by the servlet container. There is no distributed transaction monitor.

3.4 Session

The class THttpSessionBroker is a session manager. It is responsible for recognizing clients, assigning session identifiers and directing requests to an appropriate session servlet. The session servlet is an object implementing the IHttpSessionServlet interface.

This interface provides methods for passing information to the session manager about opening or closing a session. There is a separate session servlet created for each session. The manager opens a session for each new client. The session can also be opened explicitly by calling an appropriate method from the manager. The session manager also closes inactive sessions. The session identifier is 128 bits long and is randomly generated. The association between the session identifier and the servlet is made in an associative table. The session identifier is stored implicitly in a HTTP cookie. The session manager is able to find the session identifier as an argument of the GET request or inside the WWW page.

4 Experiments

Below some experiments with the C++ workbench are presented. The goal of these experiments was to measure how the garbage collector influences the performance and response times of the system.

4.1 Performance experiments

In the experiments described below a gratis program http_load [21] prepared in ACME was used. ACME produced also a very powerful Internet server *thttpd* [22]. In our experiments two simple applications were used:

application A Displays a page containing simple text of parameterized length.
application B At the start allocates some permanently reachable objects and produces a constant memory leak per each request.

All the tests were run on a Celeron 2.4 GHz / 256 MB RAM computer. Results of throughput measurements are shown in Fig. 2.

Each server running application A was sequentially sent 10000 requests by the test client residing on the same machine. The length of the response was set to 10 bytes, so that the time of transmitting the data was negligible. The experiment was conducted 10 times. The mean values were calculated and are presented in Fig. 2. Our workbench performed not worse than well known enterprise-level webservers. Response times were also typical (Fig. 3). Under heavy load (Fig. 4), when more users tried to access the servlet at the same time, the performance dropped slightly after exceeding 25 concurrent requests, but was still better than that of Tomcat, running on a JIT enabled Java Virtual Machine 1.4.2. The experiment described above shows, that the implemented garbage collector can be effectively used in Internet interactive applications.

4.2 Overhead of the garbage collector

The overhead was measured using the GPROF profiler from the GCC 3.3.4 package and is presented in Fig. 6. The application B was queried 100,000 times at an average rate of 400 requests/second. It allocated 6 MB at the start and produced 2,5 kB memory leak per request. The measurements show, that it spent most of the time serving requests or waiting for them to come. The total garbage collector overhead was less

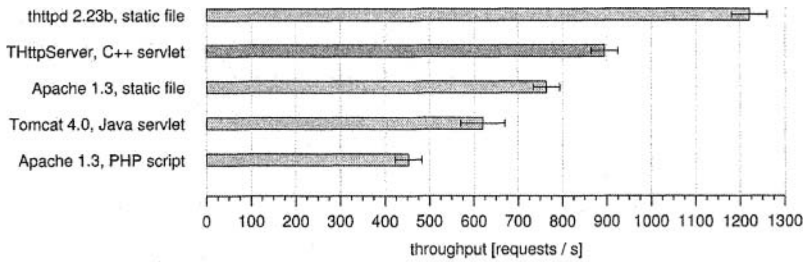

Fig. 2. Comparison of performance of various WWW servers serving application A

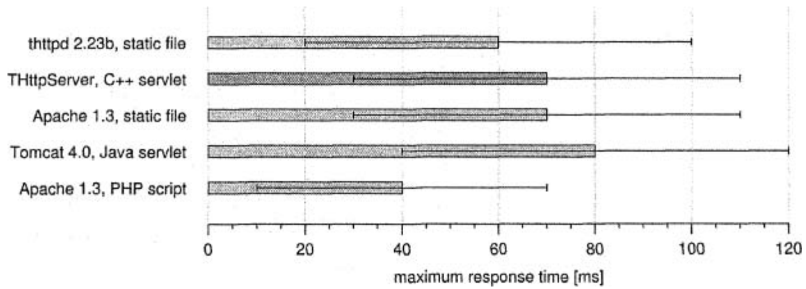

Fig. 3. Comparison of maximum response times of various WWW servers

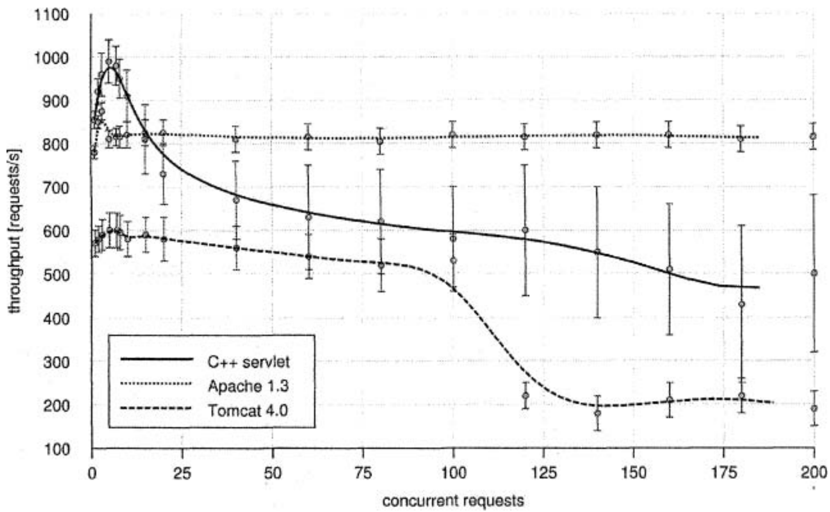

Fig. 4. Performance of the server under heavy load

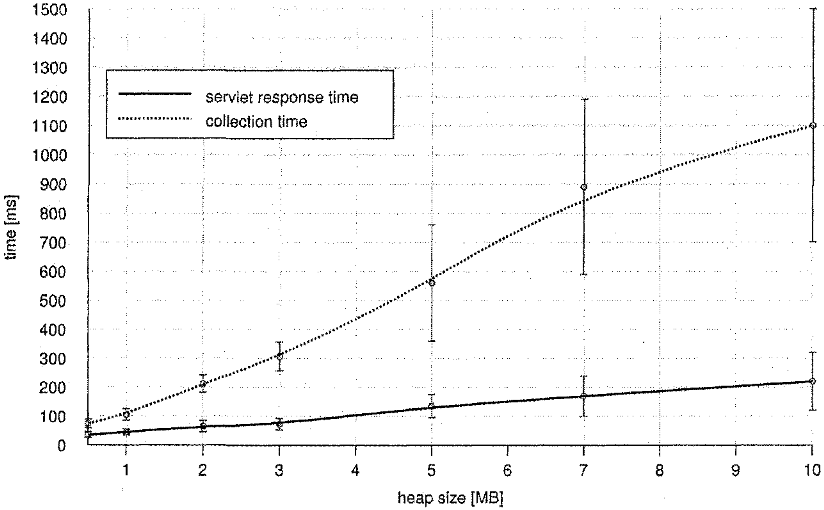

Fig. 5. The total servlet response times and garbage collection times.

than 5%. The part of the collector that is responsible for creating and destroying smart pointers and registering new objects takes much more time than the mark-and-sweep process, so this part should be optimized in the near future. The result of measurements of the garbage collection time (given in Fig. 5.) proves that the garbage collector works concurrently. The requests were handled successfully while the garbage collector was running. The requests were sent at a rate of 100 per second to the Application B. Maximum times from 1500 requests are shown.

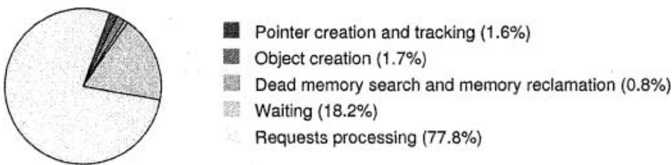

Fig. 6. CPU overhead in the garbage collector

5 Conclusions and future work

In the paper we presented how a non-blocking accurate garbage collector was used as a component in a real-world application – the workbench for C++ server side Internet applications. The workbench is simple, but provides enough functionality to build small and middle size Internet applications. The performance of our www server, as shown in section 4.1 is also pretty good.

The smart pointer pattern used in the garbage collector resulted in simplicity of the interface. The lack of manual memory management routines in the servlet container and user's servlets enables to achieve high dependability of applications. As was experimentally proved these features neither severely diminish the system performance, nor make response times too long to annoy the user. There was also no significant difference in the performance of the presented system and systems not using the garbage collector at all, or systems running on virtual machines with advanced, generational, copying garbage collectors. This shows that usage of a non-conservative, non-blocking garbage collector in an uncooperative environment like C++ is reasonable and practical.

The garbage collector used in our workbench can be further refined. The object architecture of it makes such modifications easy. So far our workbench was used in rather simple applications. A generational version of the garbage collector is possible and can be a subject of the future research. There is evidence that generational garbage collectors perform better than the non-generational ones [13].

References

1. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Softw. Pract. Exper. **18**(9) (1988) 807–820
2. Kołaczkowski, P., Bluemke, I.: A soft real time precise tracing garbage collector for c++. Pro Dialog (20) (2005) 1–11
3. Kołaczkowski, P., Bluemke, I.: A soft-real time precise garbage collector for multimedia applications. In: V International Conference Multimedia in Business and Education, Multimedia w Biznesie i Edukacji. Volume 2., Częstochowa, Poland, Fundacja Współczesne Zarządzanie Białystok (2005) 172–178
4. Kołaczkowski, P.: Techniques for building server side internet applications. Pro Dialog (18) (2005) 31–59
5. Colburn, R.: Teach Yourself CGI Programming in a Week. Sams Publishing, Indianapolis, Indiana, USA (1998)
6. Open Market, Inc.: FastCGI homepage (2006) http://www.fastcgi.com/.
7. Boehm, H.J.: Space efficient conservative garbage collection. In: PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1993) 197–206
8. Seligmann, J., Grarup, S.: Incremental mature garbage collection using the train algorithm. In: ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1995) 235–252
9. Bartlett, J.F.: Mostly copying garbage collection picks up generations and c++. Technical Report TN-12, Digital Equipment Corporation Western Research Laboratory (1989)
10. Smith, F., Morrisett, G.: Comparing mostly-copying and mark-sweep conservative collection. In: ISMM '98: Proceedings of the 1st international symposium on Memory management, New York, NY, USA, ACM Press (1998) 68–78
11. Detlefs, D.: Garbage collection and run-time typing as a C++ library. In: C++ Conference. (1992) 37–56
12. Edelson, D.R.: Smart pointers: They're smart, but they're not pointers. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA (1992)
13. Blackburn, S.M., Cheng, P., McKinley, K.S.: Myths and realities: the performance impact of garbage collection. SIGMETRICS Perform. Eval. Rev. **32**(1) (2004) 25–36

14. Henderson, F.: Accurate garbage collection in an uncooperative environment. In: ISMM '02: Proceedings of the 3rd international symposium on Memory management, New York, NY, USA, ACM Press (2002) 150–156
15. The PHP Group: PHP documentation (2006) http://www.php.net/docs.php.
16. Adobe Systems, Inc.: ColdFusion documentation (2006) http://www.macromedia.com/support/documentation/en/coldfusion/.
17. Inline Internet Systems, Inc.: User's guide to iHTML extensions version 2.20 (2001)
18. Mitchell, S.: Teach Yourself Active Server Pages 3.0 in 21 Days. Helion, Gliwice, Poland (2003)
19. Goodwill, J.: Pure JSP: Java Server Pages. Helion, Warszawa, Poland (2001)
20. Damon Houghland, A.T.: Essential JSP for Web Professionals. RM, Warszawa, Poland (2002)
21. ACME Labs: Multiprocessing HTTP test client (2005) http://www.acme.com/software/http_load/.
22. ACME Labs: Tiny/turbo/throttling HTTP server (2005) http://www.acme.com/software/thttpd/.