j2eeprof – a tool for testing multitier applications

Paweł Kłaczewski and Jacek Wytrębowicz

Institute of Computer Science of Warsaw University of Technology
P.Klaczewski@elka.pw.edu.pl, J.Wytrebowicz@elka.pw.edu.pl

Abstract. Quality assurance of multitier application is still a challenge. Especially difficult is testing big, distributed applications written by several programmers, with the use of components from different sources. Due to multi threaded and distributed architecture, their ability to be observed and their profiling are extremely difficult. *J2eeprof* is a new tool developed for testing and profiling multitier applications that run in the J2EE environment. The tool is based on the paradigm of aspect insertion. The main goal of *j2eeprof* is to help in fixing of integration errors and efficiency errors. This paper presents the concept of *j2eeprof* and gives some insides of *j2eeprof* development. On the beginning we give some introduction to the methods of software profiling, and a brief characteristic of existing profilers, i.e., *JFluid*, *Iron Track Sql*, *OptimizeIt Server Trace* and *JXInsight*. Next we present the architecture of *j2eeprof*, and we describe how it collects data, what protocols it uses, and what kind of analysis it supports. On the end we demonstrate how *j2eeprof* works in practice. In conclusions we list the strong and weak points of this tool, which is still in a beta version. *J2eeprof* is planned to be offered as an open source for the programmer community.

1 Introduction

Software testing and software profiling are time consuming tasks, especially during development of multitier, distributed applications. Sometimes these tasks take more time than coding. They are crucial when the target application is safety or business critical. We mean by testing the process of defect discovery in a developed code. We mean by profiling the process of performance analysis of an application.

Because Java Platform Enterprise Edition (J2EE) is a widely used programming platform for developing and running distributed multitier architecture applications, we have focused our attention on testing and profiling applications that run in the J2EE environment. The result is *j2eeprof* [7] - a new tool to help in fixing of integration errors and efficiency errors. Integration testing and profiling need very similar methods and tools. We shortly describe them.

To make not frequent or exceptional conditions testable we have to extend the tested application to make controllable its execution flow. During an execution flow a programmer collects selected data for subsequent analysis. Selection of the data depends on programmer aim, it could be: remote function checking, bottleneck discovery, time consumption of selected functions and memory consumption. In general, there are two methods of data gathering: sampling and tracing∞. The advantage of sampling is that this method slightly influences the tested application in contradiction to the tracing

method. The advantage of tracing is the possibility to achieve very high accuracy but when accuracy is higher – the execution time is more and more disturbed.

Extensions that make the execution flow controllable are included in the application code by a programmer. Sampling can be performed without any modification of the application code. Tracing can be achieved by altering the code or by modification of its environment, or both. The Java Platform Debugger Architecture (JPDA), which is a collection of APIs to debug Java code, is a good example of a tool for environment modification. A disadvantage of JPDA is the limited set of low-level events that the programmer can observe. The abstraction level of virtual Java machine is not suitable for J2EE application analysis. The programmer gets too much low level data, which are difficult to analyze. Altering of the application code can be done by hand, can be processed by a compiler (e.g., as for *gprof* Unix tool), or after compilation. There are Java libraries, e.g., BCEL[1], ASM [3], which allow altering a Java bytecode during loading. The programmer has to point where and how the automatic code altering should be performed.

The amount of data collected during an application run is usually huge. Sometime some compression or aggregation methods have to be used for their collection. A programmer needs to have some tools for filtering the collected data and for their visualization in an interactive manner. G. Ammons, T. Ball and J. R. Larus [1] have proposed to build a structure called Calling Context Tree (CCT) – as an aggregation method. Every tree node keeps some measurements of an executed function. Any path in the tree represents a possible execution sequence of modules (the module can be a method, a component, a layer, or a node belonging to a distributed system). Figure 1 depicts an exemplary execution path of a function X that executes 6 modules (AB notation means that A module calls B module). A tree representation is more expressive.

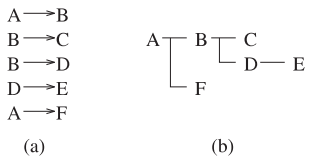

Fig. 1. Execution path visualization a) sequential, b) context tree

It helps to find bottlenecks related to different load of data or user connections. There are more ways of execution path visualization as Fig 2 shows. Nowadays profilers generate a layered representation of full tree of execution calls (Fig 2b). The width of every rectangle may depict execution time of relating module. Complex applications give very big trees. To make them more readable reduced graphs can be generated (Fig. 2c, 2d). Most profilers allow for simple filtering of presented data with predefined set of views. However there are exceptions: a programmer using *ejp*[2] can implement own filters. *XDSE* profiler [2] stores full execution trace in an XML database. Next a programmer can define filtering by XQuery language and select a visualization form.

[1] http://jakarta.apache.org/bcel
[2] http://ejp.sourceforge.net

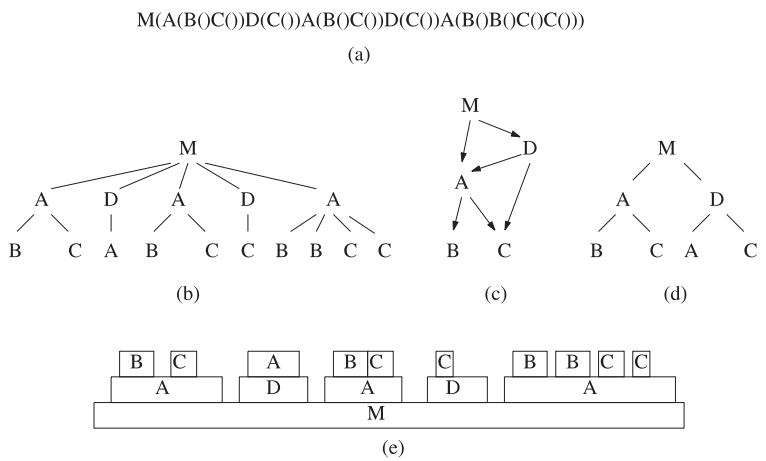

Fig. 2. Execution path visualization a) layered representation, b) full tree of execution calls, c) reduced call graph, d) context call tree. e) trace graph

Profiling of a distributed system is difficult. Every distinct element has to be observed independently. Next, a profiler has to correlate collected data before filtering and presentation. A correlation method based on independent clocks is not accurate and leads to interpretation errors. Much efficient is to include tracing into a communication mechanism used by separate instances. Authors of [8] describe a tool that traces TCP messages. For better efficiency, a profiler could use some marking of messages that concern the analyzed application/purpose. *Pinpoint* project [6] is based on modification of *Jboss*[3] application server – in this way a distributed application, which works on *Jboss* servers, can be easily and efficiently traced. When a programmer uses CORBA, then we can take advantage of built in interceptor mechanism for message marking. The interceptor is a function written by the programmer and called during communication.

There are several commercial profilers addressed to J2EE environment, but we do not know any such a tool from public domain. Profilers created for Java programmers, not only for those who use J2EE, are more numerous. Let take a look on some of them – the most interesting in our opinion. *JFluid* profiler [5], from Sun Microsystems, works only with the *NetBeans* programmer framework. It provides some means for analysis of: memory consumption, execution time and execution flow. Programmer can point some Java methods for analysis. *JFluid* process the code statically to discover all methods, which could be executed by those selected. Next it alters them to make them traceable. It visualizes only the traces that belong to the execution context of selected methods. The altered code has constant time overhead, that allows subtracting it from measured values, and present more accurate data. Because *JFluid* co-works with extended (tuned for it) virtual java machine it is a fast and efficient tool.

Iron Track Sql[4] is a free tool for performance monitoring of java applications that interact with databases. It builds a log of every database query, its time and duration. It allows for some filtering, e.g., to register only these queries whose duration overcomes

[3] http://www.jboss.org
[4] http://www.irongrid.com/catalog/product_info.php?products_id=32

a defined threshold. It is based on a database proxy, which makes all required logs. The programmer has to use the *p6spy* driver (an element of *Iron Track Sql*) in place of standard jdbc driver.

OptimizeIt Server Trace is a Borland profiler addressed to J2EE. It can gather data using probing or tracing. It can monitor memory consumption. With this tool the programmer can visualize execution paths as a context tree or as a full tree of execution calls. OptimizeIt presents j2ee services trace using sets of abstract words. In example word "ejb load" stands for ejb load life cycle method. Tool hides application server internal implementation of ejb load and presents it to user in simplified form. Profiling statistics are then more readable and free of unnecessary information. This feature makes *OptimizeIt* much more effective tool for J2EE application tracing than standard profiler. *OptimizeIt* can point hot spots, can display execution time of every layer, e.g.: JDBC 23,68%, JNDI 15.31%, servlets and jsp 57.84%, EJB 3,17%. It can even register and visualize RMI communication.

JInspired company offers the *JXInsight* profiler. This tool is very similar to *OptimizeIt Server Trace*. The difference is that *JXInsight* has more functions for monitoring of database queries. It can display correlations between distributed events using CORBA interceptors. Both *OptimizeIt* and *JXInsight* are very complex and powerful tools, which allow multitier visualization of execution paths.

There are many other profilers but most of them suit only development of standard Java programs running on a single machine. They are inefficient for development of J2EE applications, which are distributed and use a server code. Usually the programmer does not know the server code (it is a black box for him). And the server code is a significant part of the application. The only corrections and optimizations, the programmer can make, are inside his code. Hence only tools like *OptimizeIt* and *JXInsight* can really help to profile J2EE applications.

2 j2eeprof insides

J2eeprof is profiler designed for applications running in J2EE environment. J2EE provides variety of services. Programs work in a container i.e., servlet container or ejb container. Container provides services, can manage component life cycle and enhance program behavior. The way program uses services can be specified in code or configuration descriptor. When configuration is used it is impossible to inspect program behavior only by reading its code. This makes testing more difficult to the programmer. Another problem arises, when J2EE application is profiled using standard java profiler. There is huge amount of container implementation code execution registered together with program code. The performance impact is large and results contain plenty of superfluous information.

In order to capture accurate view of execution flow, *j2eeprof* uses tracing. *J2eeprof* comes with ability of selective program tracing. It registers J2EE services and program execution at high level of detail. By inspecting trace programmer can find out all the interactions of J2EE services with program. The tool has significant ability to shape profiling scope. *J2eeprof* addresses also distributed nature of ejb components. It is able

to track communication between remote ejb components and deliver distributed system trace.

J2eeprof is designed for profiling applications that run in a distributed environment. Thus tool itself is distributed as well. There are three major modules of *j2eeprof*: data collection module, transport module and visualization module. The data collection is installed on distributed system nodes and acts as client in the client-server *j2eeprof* architecture. Visualization module is responsible for trace analysis and visualizations. The data is transported from remote data collections modules to visualization module by transport module.

2.1 Data collection

Data collection module uses tracing method to collect profile data. Its implementation is based on the aspect oriented programming (AOP). Aspect is a program module that implements some common functionality and has no dependencies on other program modules. AOP consists of two elements: aspect weaver and composition language. Aspect weaver is responsible for composition of aspects and other modules into final application. Composition language controls the weaver. *J2eeprof* uses *Aspectwerkz*[5], open source AOP library, as a basis for data collection module. *Aspectwerkz* weaver is capable of dynamic aspect insertion. This feature enables profiler to temporarily modify tested code and change profiling scope on every program execution. *Aspectwerkz* uses *AspectJ*[6] composition language. The point of program code, where aspect can be inserted, is called join point. It can be i.e., a method or a construction invocation. Pointcut is *AspectJ* definition that pick out a set of join points. *AspectJ* gives *j2eeprof* capability to define profiling scope with detail. Important feature in J2EE environment is that a join point can define interface and polymorphic execution. J2EE is specified by a set of interfaces. *J2eeprof* can profile application server standard services by tracing them at the interface level. This method provides the right level of abstraction. Tracing implementation details of application server not only has negative performance impact, but also has no value for the application developer, as he cannot modify server code. Still the application code can be traced with much greater detail – up to every method call.

Data collection module implements a set of aspects. Data collection aspect is responsible for registering information on program execution. AOP composition language allows mixing of aspects in order to register traces on different detail level. Data trace representation (see Fig. 3) in *j2eeprof* consists of 4 elements. *PathNode* is a node of trace path. *PathNode* can contain other *PathNode* in the way it make call tree. *PathNode* is a base class for a concrete node, which may represent method execution or distributed call. Nodes belong to an execution thread, which is represented by *ThreadNode*. *SystemNode* is a node of distributed system. *System* abstracts whole observed system. The representation can describe nodes on different level of abstraction.

There are 2 generic aspects that trace method executions: *MainAspect* that registers only method signatures and ParametersAspect that registers also parameter values. An aspect collects information about several attributes: start and end time, information on

[5] http://aspectwerkz.codehaus.org
[6] http://www.eclipse.org/aspectj

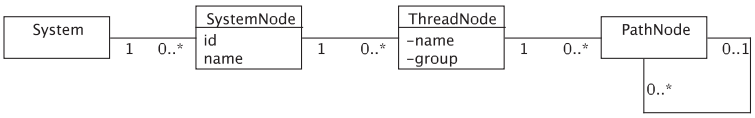

Fig. 3. Trace model

exception, path node name (e.g. method signature) and execution thread. There is also one additional attribute – category that is specified in aspect definition, and it is used later for data analysis.

2.2 Data transport

Gathered data are transmitted by transport module. The module consists of three parts: data sender, transport protocol and data receiver. Data transport module can write data to file or send over TCP/IP. The most important element is the protocol. *J2eeprof* uses binary protocol that is built in a way to keep network traffic low. We have executed several tests to measure *j2eeprof* overhead. The results (Table 1) have shown that the most time consuming is I/O. The more data is sent the more impact on performance is made (see test 3 and 4 in Table 1). During execution of test 3 all gathered data been discarded, during execution of test 4 the same data have been written into a file. I/O slow down factor was about 6. Addition of a simple compression method resulted in better overall performance. *J2eeprof* uses dictionary compression for most frequently sent data – event labels. *MainAspect* sends approximately 30 bytes per start method event and 22 bytes per exit method event. Executions with tracing turned off (test 1) and with *AspectWerkz* (test 2) empty aspect have shown a difference of performance overhead. Encoding overhead (test 3) is 3,232.98 ns but 509.68 ns (test 2) is the effect of using *AspectWerkz* and cannot be avoided. Write to the file (test 4) slows down by 17,421.38 ns. *J2eeprof* performs almost twice better as *Log4J*[7] (test 7). The maximum time was taken from *j2eeprof* statistics. It indicates that writing into a file gives more stable effects compared to sending over TCP/IP, however the second choice is much more convenient for a user.

Tab. 1. Measured performance overhead

no	test	mean time[ns/per call]	max time[ms/per call]
1	no aspects	31.63	
2	NullAspect	509.68	
3	MainAspect (no I/O)	3,232.98	
4	MainAspect (file)	20,654.36	58
5	MainAspect (tcp local)	33,639.00	308
6	MainAspect (tcp)	36,767.00	949
7	Log4J (file)	41,199.31	

2.3 Distributed tracing

J2eeprof can profile distributed J2EE systems. Execution path on each distinct node of analyzed system is recorded. But it is also required to match right local paths and

[7] http://logging.apache.org/log4j

reconstruct distributed path. Tagging messages exchanged between nodes can do this. This method has top accuracy over others, and is not affected by time differences of the nodes. EJB protocol – RMI/IIOP supports sending additional information in protocol tier, without changing interface on an application tier (Fig. 4). Corba Interceptor documentation [4] describes this feature. *J2eeprof* tracing mechanism can be enabled in the configuration file of application server, with no need to modify program or server code. The method is protocol dependent; *j2eeprof* comes with implementation for standard EJB protocol RMI/IIOP and *Jboss* RMI. But this solution is well suited for J2EE environment. J2EE specification requires application servers to provide transaction support and user authentication over remote calls. These services are defined in application configuration descriptors. Thus communication protocols must be able do support rpc-level communicates tagging. *J2eeprof* inject into EJB communication apart of transaction id and user information his own data.

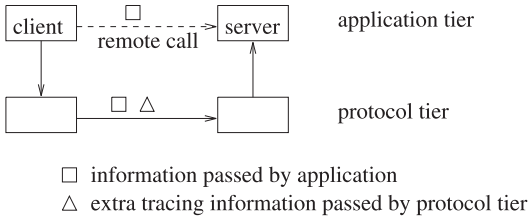

Fig. 4. Protocol tracing

Distributed paths require trace model to be improved. Model is extended by addition of two new nodes (*PathNode* subclasses). *RPCCallNode* (RC) represents an rpc call on the client side. *RPCReceiveNode* (RR) represents an rpc call on the server side. Figure 5 depicts reconstruction of a distributed path. On rpc call event – *j2eeprof* tags outgoing message with *rpcId* – auto generated id, unique in jvm scope, and *node id*(specified in configuration file). On rpc receive event – *rpcId* tag and *node id* are added to RR event. *Node id* attribute is saved in RR.sourceNodeId field. Paths merging is performed by matching RC-RR pairs. Match criteria is:

1. RC.rpcId=RR.rpcId
2. RC is registered on system node defined in RR.sourceNodeId

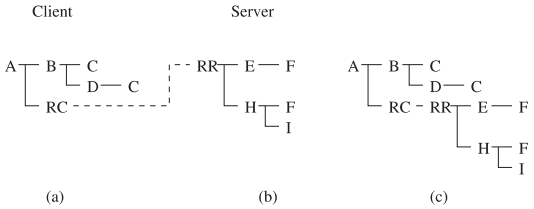

Fig. 5. Reconstruction of a distributed path a) local path on client side b) local path on server side c) completed distributed path

2.4 Visualisations

J2eeprof supports several visualizations. Profiler provides data analysis on summarized trace data as well as on raw trace. Many of these visualizations are found in other tools, but distributed trace view is an original extension of them.

J2eeprof can summarize trace in form of CCT and flat list. Both views display total number of invocations, total, mean, minimum and the maximum execution time. CTT view provides ”drill up” and ”drill down” functions. ”Drill up” displays all contexts in which selected node was called. ”Drill down” selects all possible executions rooted in a selected node.

Raw trace can be visualized as a graph or tree. Figure 6a shows graph of a trace. The Graph is similar to tree view but every node has a rectangle form. The length and position represents execution time. For the purpose of more readable view, there is an option for displaying only top-level trace nodes (Fig. 6b). Raw trace data can be queried. The result is indicated in graph view (Fig. 6) by changing color of nodes. Raw trace views are connected each other. When user selects node in the tree list, focus in other view is set to this node.

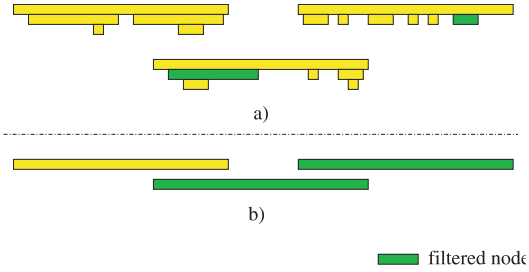

Fig. 6. Trace visualization a) detail, b) summary

Ability to collect distributed trace is quite uncommon in profilers. Thus there are not many ready to use solutions. Distributed trace requires special view. *J2eeprof* comes with original solution to this problem.

Figure 7 depicts ”rpc view”. The view captures distributed path on all nodes it belongs to. Apart of the path itself, the view contains also context of path on each distinct node. The view is horizontally divided in two zones. On the top, there is distributed path. On the bottom, there is context of the fragment of graph view. The view has also several vertical zones, each on every node along the distributed path. Double vertical lines divide system nodes. Doted lines mark time margin zone. In margin zone the top part of view is frozen on the contrary to the context shown in bottom part of the view. Timeline in context view is wider than in distributed path view. Thus in a case when distributed path execution of given system node is very short, still the context view show some information.

The path on Fig. 7 starts on Node 1, paths a and b. Execution of c is an rpc call. That part is shown on left part of the figure. D path is executed on Node 2 - middle part of the graph. Paths a,b,c are marked with grey color as they do not belong to Node 2. Last part of the figure, on the right, displays end of paths back on Node 1.

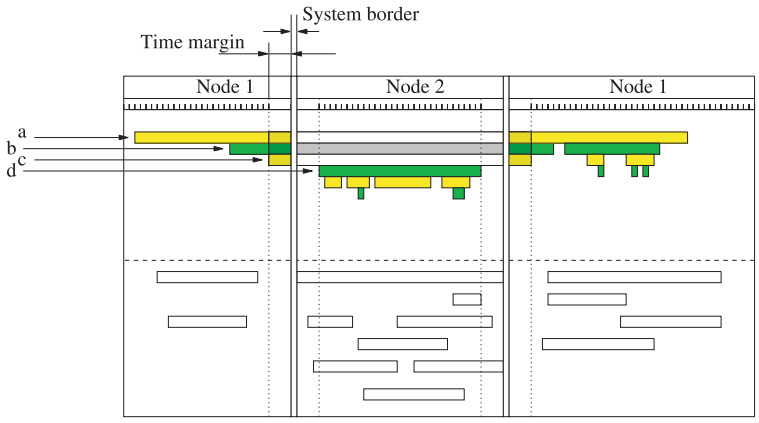

Fig. 7. Distributed trace view

3 j2eeprof in practice

J2eeprof was tested with *Rubis*[8] [9] – J2EE auction site benchmark. *Rubis* was created to compare performance of several distinct implementations of the same program. Each implementation uses different framework or technology. *J2eeprof* was tested with two of them. First is BMP_EntityBean_ID_BMP that is based on Entity Beans and bean managed persistence (denoted as bmp). The second EJB_CMP2 (denoted as cmp) uses Entity Beans, Session Facade design pattern and container managed persistence. Two of *Rubis* functions where choosen for the test. *SearchItemsByCategory* shows list of auction items. The second *RegisterUser* registers new auction site user. These functions are very different. First one is data intensive read only function, while the second is transactional read and write function. Profiling scope included all *Rubis* code method calls and tracing of JDBC and JTA services on interface level. Table 2 presents performance overhead of j2eeprof in J2EE enviroment.

Test were performed on the open source application servers: *Jboss* 3.2 and *JOnAS* 4.5.3. *JOnAS* was configured to use iiop protocol, profiling scope included protocol tracing (using CORBA interceptors). Rubis comes with dedicated load test tool. Load tests were set to run for 5 minutes with 10 virtual users. Test where performed on AMD Athlon XP 1600+, 756RAM, Linux Slackware 10 operating system.

Rubis tests contain random factor, thus test count varies between tests. It also depends on test performance. Jonas bmp test with *j2eeprof* has very large overhead and test count is much lower than test without profiling. *J2eeprof* performance overhead factor varies from 1.1 in jboss cmp test to 56.77 in jonas bmp test. On *Jboss* server overhead is related only to profiling scope. Since number of registered events is reasonably small, overhead is up to 77%. IIOP protocol tracing adds overhead to *JOnAS* test results. *JOnAS* tests performed slower than *Jboss* tests with exception of SearchItemsByCategory test (jonas cmp). The reason is that *Jboss* optimizes local ejb calls, *JOnAS* not.

Table 3 presents some insights of Rubis implementation derived from trace data. On *JOnAS*, bmp performed slower than cmp version but jboss cmp is slower than jboss

[8] Rice University Bidding System (Rubis), http://rubis.objectweb.org

Tab. 2. Measured Rubis performance overhead

	SearchItemsByCategory					RegisterUser				
	test count	time[ms] avg	time[ms] min	time[ms] max	avg overhead	test count	time[ms] avg	time[ms] min	time[ms] max	avg overhead
no profiling (Rubis performance statistics)										
jboss cmp	89	233	146	519		17	67	29	279	
jboss bmp	109	196	72	769		8	50	17	144	
jonas cmp	86	198	115	406		10	585	121	2333	
jonas bmp	78	1,037	196	3,484		5	342	174	546	
j2eeprof (Rubis performance statistics)										
jboss cmp	89	395	235	1,329	1.77	7	74	52	198	1.10
jboss bmp	52	226	100	520	1.15	5	76	36	201	1.52
jonas cmp	74	599	211	5,554	3.03	15	5,148	197	19,813	8.80
jonas bmp	12	33,051	16,426	67,706	31.87	1	19,415	19,415	19,415	56.77
j2eeprof (j2eeprof performance statistics)										
jboss cmp	87	290	175	684		6	70	45	134	
jboss bmp	50	146	24	399		5	42	30	55	
jonas cmp	72	718	165	4,710		15	5,104	193	19,252	
jonas bmp	9	42,197	15,220	66,204		1	15,864	15,864	15,864	
j2eeprof (no CORBA tracing, Rubis performance statistics)										
jonas cmp	75	557	177	3,167	1.14	5	3,521	176	16,795	
jonas bmp	46	2052	283	5,799	9.08	6	2,295	314	6,847	

Tab. 3. Rubis tests results

	jboss cmp	jonas cmp	jboss bmp	jonas bmp
concurrent threads	7	14	4	18
SearchItemsByCategory				
jdbc (time percent)	51.72	67.84	27.43	3.93
jdbc/ejb.load	1	0.77	1	1
jdbc/ejb.find	1	0.00	1	1
rmi (time percent)	0	2.41	0	18.60
rmi/per client call	0	2	0	42
RegisterUser				
jdbc (time percent)	23.64	42.58	9.05	34.25
jdbc calls/per client request	11	6.33	6	35
rmi (time percent)	0	0.39	0	2.88
rmi/per client call	0	2	0	8

bmp. Bmp *Rubis* implementation calls ejb entity components within web tier that results in large number of remote calls. Such design is described as J2EE anti-pattern. *Jboss* optimizes such calls but on *JOnAS* there is a remarkable average overhead of rmi call – 108.975 ms. Cmp version uses better design – Session Facade that minimizes remote calls, there are only 2 rmi calls in SeachItemsByCategory compared to 42 in bmp test.

The most efficient jdbc use is done by jonas cmp version. Each pair of ejb.find and ejb.load methods result at most one jdbc call. *JOnAS* probably makes use of cache since jdbc calls are performed only in 77% of ejb.find calls in SearchItemsByCategory test. Other *Rubis* versions does not perform jdbc optimization, every ejb.load and ejb.find call results in jdbc.call. Despite of jonas cmp efficiency, the best performer is jboss bmp. *JOnAS* and *Jboss* differ also in number of observed threads. *Jboss* delegates one thread to a server client request so number of concurrent threads is equal to number of concurrent requests. *JOnAS* passes control to different thread in every rmi call. The protocol tracing mechanism is necessary to obtain complete paths in such case, although significantly increases performance overhead.

4 Conclusions

The purpose of *j2eeprof* is to help in testing and profiling of J2EE distributed applications. Using it a programmer can easily analyze interactions between his code and other components or environment. Programmer does not have to modify his code to gather data. *J2eeprof* uses RMI/IIOP to mark and trace communication messages – giving accurate data about interactions between distributed components. The programmer decides on which abstraction level he wish to analyze his code, then he controls the trace information using *aspectwerkz* library. The advantage of the aspect approach is, that the programmer can easily monitor the interactions between his code and a J2EE server code. The strong features of *j2eeprof* are: flexibility in use, ability to fit gathered data to programmer needs, and high accuracy of registered traces from distributed components.

A week feature of *j2eeprof* is remarkable and varied execution time overhead. All profilers that work on tracing basis, in place of sampling basis, have this disadvantage. Because *j2eeprof* gathers full execution trace with programmer-defined data, not just

execution statistic, the overhead is higher than other profilers put in. To obtain accurate time characteristics, the programmer has to take other profiler that works on sampling basis. *J2eeprof* is small and simple tool comparing with commercial *OptimizeIt* and *JXInsight* profilers. Although it is free, easy to use and we find it very useful.

References

1. G. Ammons, T. Ball and J. R. Larus: Exploiting hardware counters with flow and context sensitive profiling. In Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation,pages 85-96, Las Vegas, 1997.
2. C. Anslow, S. Marshall, R. Biddle, J. Noble and K. Jackson: Xml database support for program trace visualization. In Australian Symposium on Information Visualization, volume 35, 2004.
3. E. Bruneton, R. Lenglet and T. Coupaye: Asm: a code manipulation tool to implement adaptable systems. In Adaptable end extensible component systems, Grenoble, France, 2002.
4. Interceptors Published Draft with CORBA 2.4+ Core Chapters, Document Number ptc/2001-03-04. http://www.omg.org/cgi-bin/doc?ptc/2001-03-04
5. M. Dmitriev: Design of jfluid: A profiling technology and tool based on dynamic bytecode instrumentation. Technical report, Sun Microsystems, Nov. 2003.
6. E. Kiciman: Pinpoint: Status and future directions. 2003 www.stanford.edu/˜emrek/pubs/roc-retreat-2003-pinpoint.pdf
7. Paweł Kłaczewski: Testability Issues of Multitier Applications (in polish). Master thesis, Institute of Computer Science of Warsaw University of Technology, 2005.
8. Marcos K. Aguilerai, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds and Athicha Muthitacharoen: Performance debugging for distributed systems of black boxes. In Proceedings of SOSP, Bolton Landing, NY, Oct. 2003.
9. E. Cecchet and A. Chanda and S. Elnikety and J. Marguerite and W. Zwaenepoel: Performance Comparison of Middleware Architectures for Generating Dynamic Web Content, 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June, 2003.