Simulation efficiency analysis method of Java Enterprise Edition application

T. GÓRSKI
gorski@wat.edu.pl, tomasz.gorski@rightsolution.pl

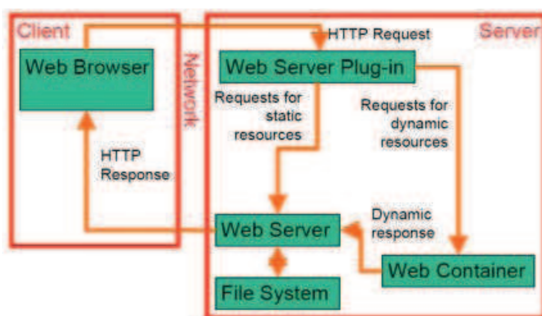
Institute of Information Systems
Cybernetics Faculty, Military University of Technology
Kaliskiego Str. 2, 00-908 Warsaw

In this article efficiency analysis method of Java EE applications was presented. Efficiency's measures of such kind of applications were described. Furthermore, discrete-event simulation modelling method Event Graph and its extension LEGOS were presented as well. Moreover, model of Java EE application was presented. An implementation of proposed model in Java and SimKit package was presented. In the paper, a project of simulation application was also described. The article encompasses description of simulation experiment used in efficiency analysis of Java EE application and example of results from such experiment.

Keywords: software engineering, simulation, enterprise applications, performance

1. Architecture of Java EE application

An application is called package of software with interface by which user gets access to functionality offered by this package [1]. An internet application we can define as a dynamic set of various software components deployed on application server which use different resources (e.g. database) due to deliver services via WWW infrastructure [1]. The main advantage of this kind of application is access to application logic through Internet browser. So, there is no need to install software on client computer (only Internet browser). Moreover, changes in application logic do not require changes on client machine. Practically, all business logic is located on application server. In the picture Pic.1 pattern of client's request realization is presented in model request-response in architecture Java Enterprise Edition (Java EE).

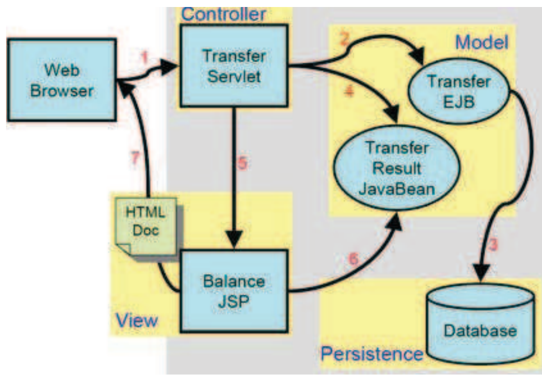


Pic.1. Service of client's request in Java EE architecture [2]

In Java EE architecture fundamental architectural pattern is Model-View-Controller (MVC). Main tasks of Model components are realization of business processes and communication with database. Services offered by this layer are used by two other layers: View and Controller. Model is independent from View and Controller. View components are responsible for model visualisation. They dynamically generate html documents with content provided by Model, which are sent to client. Components of the third layer – Controller – are responsible for service of client actions. They are responsible for receiving client requests and service realization by invoking appropriate services, offered by Model and View components.

In MVC architectural pattern we can distinguish following layers: Client, Controller, View, Model, Data [2, 3]. In the picture Pic.2 client request realization in MVC pattern is depicted: 1. Client request; 2. Change of Model state; 3. Request to database; 4. Objects creation for data transfer; 5. Invoking JSP page; 6. Reading data from data objects; 7. Results presentation to client.

In this paper, it was assumed that the main object of modelling and conducting efficiency analysis is Internet application in architecture Java EE.



Pic.2. Client's request realization in MVC [2]

2. Efficiency measures of Java EE application

The main issue in the paper is efficiency of appropriate execution of Java EE application functionality. Following measures were used to analyse efficiency of Java EE application:

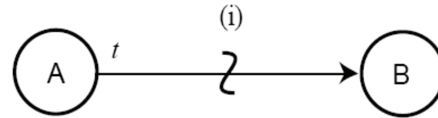
- *Response time* – this is a time between arrival of request to application server and beginning of sending application response to client,
- *Throughput* – this is a number of requests, which application realizes in time unit,
- *Utilization of memory* – this is percentage value of used memory in specified period of time.

3. Event Graph and LEGOS methods

Event Graph methodology is a way of presenting discrete-event simulation models in a simple and elegant language-independent manner. The methodology uses graph which represents events occurring in modelled system and their relationships.

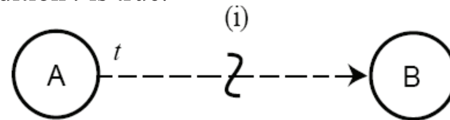
An Event Graph consists of nodes and directed edges. Each node corresponds to an event, or state transition, and each edge corresponds to the scheduling of other events. Each edge can optionally have an associated Boolean condition and/or a time delay.

Picture Pic.3 shows the fundamental notation for Event Graphs and is interpreted as follows: the occurrence of Event *A* causes Event *B* to be scheduled after a time delay of *t*, providing condition (*i*) is true (after the state transitions for Event *A* have been performed). By convention, the time delay *t* is indicated toward the tail of the scheduling edge and the edge condition is shown just above the wavy line through the middle of the edge. If there is no time delay, then *t* is omitted. Similarly, if Event *B* is always scheduled following the occurrence of Event *B*, then the edge condition is omitted, and the edge is called an unconditional edge. Thus, the basic Event Graph paradigm contains only two elements: the event node and the scheduling edge with two options on the edges (time delay and edge condition).



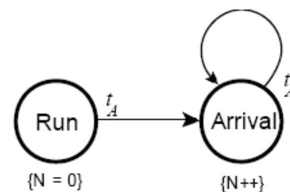
Pic.3. Event addition to list of events in Event Graph [5]

Second type of edge (Pic. 4) is represented by dashed line and means: occurrence of event *A* will cause deletion from event list the first occurrence of event *B* (if exists on the list), if condition *i* is true.



Pic.4 . Event deletion from event list in Event Graph [5]

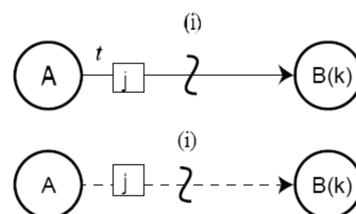
The special type of event in Event Graph is *Run*. The *Run* event is placed on the Event List at time 0.0 but is otherwise an ordinary event with associated state transitions and scheduling edges. The aim of this event is starting simulation. Picture Pic.5 shows arrival process model with *Run* event.



Pic.5. Arrival process model in Event Graph [6]

Run event simply initializes cumulative number of arrivals (N) to 0 and schedules the first arrival. The state transition for the *Arrival* event is that the cumulative number of arrivals (N) is incremented by 1.

The important extension of Basic Event Graph is the ability to pass parameters on edges to event nodes. This is presented in picture Pic.6 for both scheduling and cancelling edges.



Pic. 6. Edges with parameters in Event Graph [6]

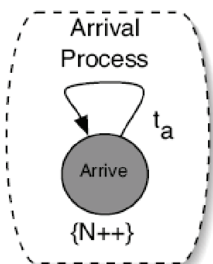
The interpretation of the constructs in the picture Pic.6 is as follows. For the scheduling edge with parameter: when Event *A* occurs then, if condition (*i*) is true, event *B* is scheduled to occur after a delay of *t* time units; when *B* occurs, its parameter *k* will be set to the value given by the expression *j*. For the cancelling edge with parameter: when event *A* occurs then, if condition (*i*) is true, the first scheduled event of type *B* whose parameter *k* exactly matches *j* is removed; if no such event is found, then nothing happens; when event *B* occurs, the value of expression *j* is that which it had when the scheduling event *A* occurred.

Listener Event Graph Objects (LEGOS) [7] are fundamentally Event Graphs, but have two simple, but important, additions. The first of these is encapsulation, and the second is the listener pattern.

An Event Graph which represents self-contained functionality can be encapsulated to create an Event Graph object, which can subsequently be treated as an atomic component in other models. Those components can be combined to create more complicated models. This hierarchical approach assures high scalability of created models.

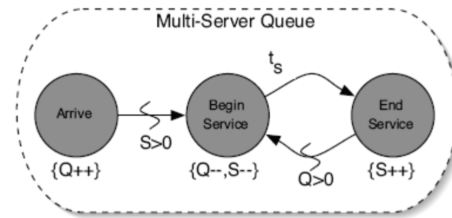
Listener pattern provides capability of connecting components without need of changing inner structure of those components [3, 9].

In practice we need at least two modules to show capabilities offered by LEGOS. We will use considered earlier arrival process model as the first of them (Pic.7).



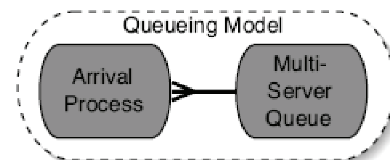
Pic.7. Module LEGOS of arrival process model [7]

As the second module is multi-server queue model with one queue and many servers. In this model we can distinguish following events: request arrival to queue, begin of service and end of service and two state variables: number of elements in queue *Q* and number of idle servers *S*. Module which presents mentioned above model is shown in the picture Pic.8.



Pic. 8. Module LEGOS of Multi-Server Queue [7]

Our aim is building queuing model which uses both Arrival Process model and Multi-Server Queue model. In order to accomplish this, we need a mechanism by which events occurring in one object trigger events in another object, without breaking the encapsulation. This is accomplished using the listener pattern. The loosely-coupled nature of the listener connection is an important distinguishing feature of the LEGOS framework. In the queuing model, the listener connection means that each occurrence of the Arrive event in the Arrival Process LEGOS stimulates the occurrence of the Arrive event in the Multi-Server Queue (Pic.9).



Pic. 9. Queuing Model [7]

4. SimKit library

SimKit library is LEGOS implementation. Using SimKit we can create independent components and combine them in more complicated models. So, it is well suited to model Java EE application which is itself, set of collaborating components.

One of the advantages of SimKit is separation of code which is responsible for modelling of system's actions from code used to collecting statistics during an experiment. The same design pattern was used here – Listener. During experiment, classes responsible for statistics collection are listening changes of state of model classes. This is simple and flexible solution because connection between classes can be configured in code without need of changing those classes.

5. Java EE application model

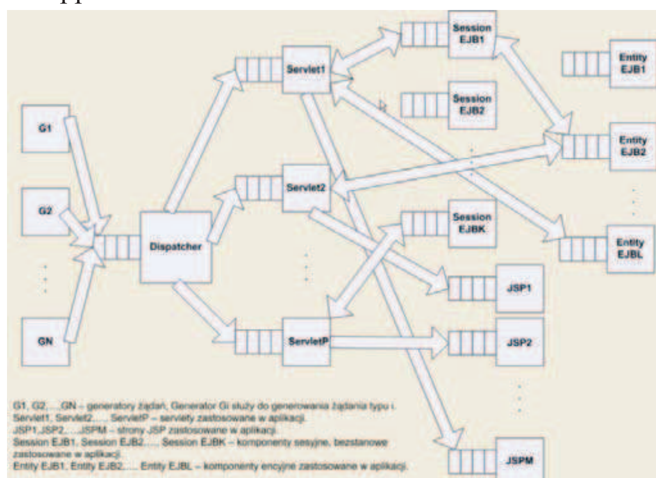
Model assumptions

In the model of Java EE application have been made following assumptions:

- Types of requests are distinguished by path of service realization. Type of request is marked as k_i, where:
  $k_i \in K, i = \overline{1,n}, n \geq 1$
- For each type of request was set distribution time between two consecutive requests of this type. This distribution for type of request k_i is denoted as g_i, where:
  $g_i \in G, i = \overline{1,n}, n \geq 1$
- Each path of service realization starts from servlet and ends on JSP page.
- A certain amount of memory is required to realize each request. This amount depends on type of request. For each of type request amount of memory is specified by distribution. Distribution of memory amount for type request k_i is denoted by m_i, where:
  $m_i \in M, i = \overline{1,n}, n \geq 1$
- Request transmission time from client to server and response transition time from server to client were omitted.
- Communication with database is realized by entity Enterprise Java Beans (EJB). Service time of component entity EJB encompasses service time of database.
- Lengths of queues in the model are unlimited.

Model elements

Picture Pic.10 presents example of Java EE application model.



Pic. 10. Example of model of Java EE application

Presented model is queue model. In the model we can distinguish following types of elements:

- Requests generator,
- Dispatcher,
- Servlet,
- JSP (Java Server Pages) page,
- Entity Enterprise Java Bean,
- Stateless Session Enterprise Java Bean.

In the following part of the paper models of types of elements were presented: Requests generator, Pool and Servlet. Furthermore, path types of request realization were also described.

Requests generator

Request generator generates requests of specified type.

Parameters:

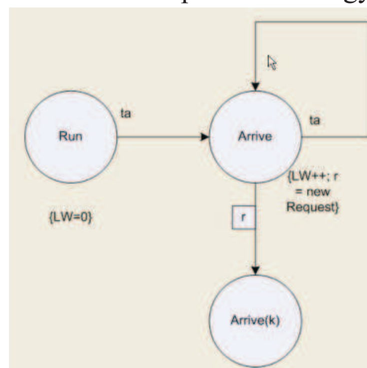
- i – type of requests, which are generated by this generator,
- g_i – distribution function of time between two consecutive requests of type *i*,
- m_i – distribution function of memory required to service request of type *i*.

State variable: *LW* – number of generated requests,

Events:

- Run – event required by methodology Event Graph which adds to event list first event of type *Arrive* for generator g_i,
- Arrive – event of request generation which increases number of generated requests and new request is created – r,
- Arrive(k) – request generation event with parameter *k* – object of request. This event is required to connect Generator with Dispatcher.

Picture Pic.11 presents state graph for requests generator in Event Graph methodology.



Pic. 11. Graph for requests generator

Pool

Elements types like Servlet, JSP page, entity EJB and session EJB use Pool. In case of servlets and JSP pages this is pool of threads. But for entity EJBs and session EJBs this is

pool of objects. Way of pool functioning is the same for all element types. Number of elements in pool n_p is limited by minimal number of elements in pool n_{min} and maximal number of elements in pool n_{max}, $n_{min} \le n_p \le n_{max}$. After starting an application, number of elements in pool is equal to minimal number of elements in pool, $n_p = n_{min}$. In case when this number is not enough to realize requests arriving to pool then new elements are added to the pool. By n_c was denoted number of created elements in specified time. For pool, condition $n_p + n_c \le n_{max}$ should be true. If for certain element in pool there is no request to realize by specified time – timeout – this element is deleted. By n_d was denoted number of excessive elements which are waiting for deletion and condition $0 \le n_d \le n_p - n_{min}$ should be true. By n_f was denoted number of free elements in pool, $n_d \le n_f$.
In time of request arrival to pool there can be one of the following situations:

- In pool there is idle element, $n_f > 0$ and is get to request realization.
- In pool there is no idle element, $n_f = 0$ and $n_p + n_c < n_{max}$. Request is added at the end of pool's queue. Pool manager creates new element which will get first request waiting in queue.
- In pool there is no idle element, $n_f = 0$ and $n_p + n_c = n_{max}$. Request is added at the end of pool's queue.

After request realization in pool there can occur one of the following situations:

- In queue is at least one request which waits for realization – released element gets first request from queue.
- In queue there is not any request which waits for realization and $n_p = n_{min}$ – released element becomes idle element.
- In queue there is not any request which waits for realization and $n_p > n_{min}$ – released element becomes idle element.

We can show pool's parameters:

- n_{min} – minimal number of elements in pool,
- n_{max} – maximal number of elements in pool,
- mem – memory required for one element in pool,
- t_t – timeout,
- t_c – creation time of element.

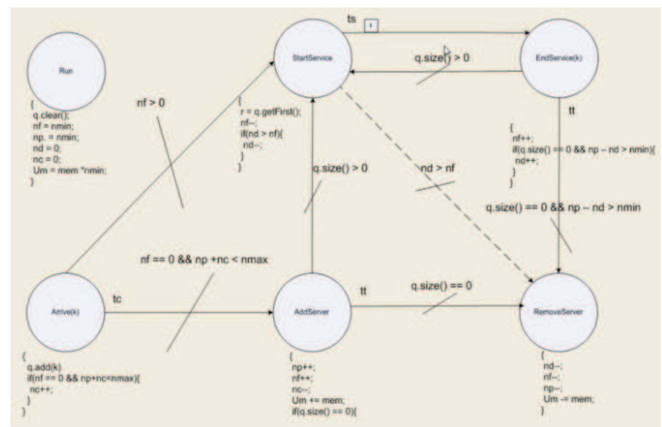
Pool's state variables are as follows:

- n_p – number of elements in pool,
- n_f – number of idle elements in pool,
- n_c – number of created elements,
- n_d – number of elements which wait for deletion,
- q – pool's queue,
- Um – memory required by pool.

Pool's events are following:

- Run – event which initiate state of pool,
- Arrive(k) – arrival of request *k*,
- StartService – start of realization of the first request from pool's queue (r = q.getFirst()),
- EndService(k) – end of realization of request *k*,
- AddServer – element's addition to pool,
- RemoveServer – deletion of an excessive element from pool.

Picture Pic.12 presents state graph for pool in Event Graph methodology.



Pic. 12. State graph for pool

Servlet
Servlet models way of functioning of servlet's threads pool. Servlet realizes requests which arrive from generator and forwards them to JSP page. Servlet may realize one or many types of requests. Set of request types realized by specified servlet was denoted by K_s and $K_s \subset K$. For each $k_i \in K_s$ distribution function of request service time was specified s_i and $s_i \in S_s$.

Parameters (besides those which possess pool):

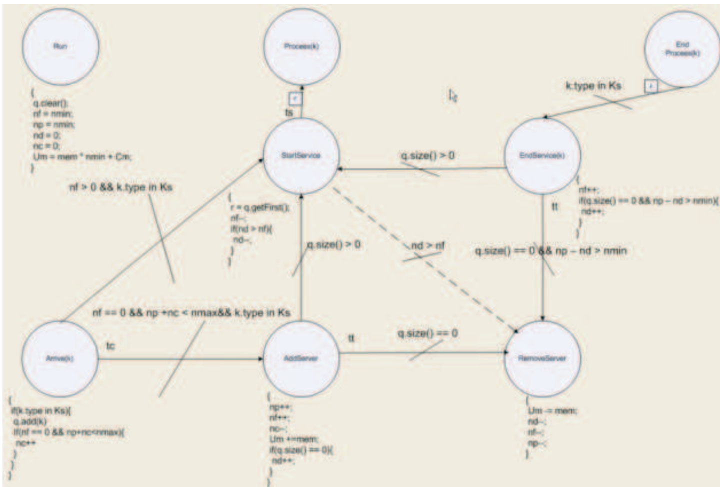
- Cm – memory required by servlet's code,
- K_s – set of request types realized by servlet,
- S_s – set of distribution functions of request service time for each request type realized by servlet.

Events (besides those which possess pool):

- Process(k) – end of request's service by servlet and invoking entity or session EJB,
- EndProcess(k) – end of request's service by external to servlet elements,

- EndService(k) – sending request to JSP page and releasing of servlet.

Picture Pic.13 depicts state graph for servlet.



Pic. 13. State graph for servlet

Path types of request service

It was assumed that service of each request starts from servlet and ends on JSP page. Furthermore, servlet may invoke entity or session EJB and forward result of its service to JSP page. Session EJB may invoke entity EJB. JSP pages and entity EJBs have not possibility to invoke other components.

In the model following path types of request service were identified:

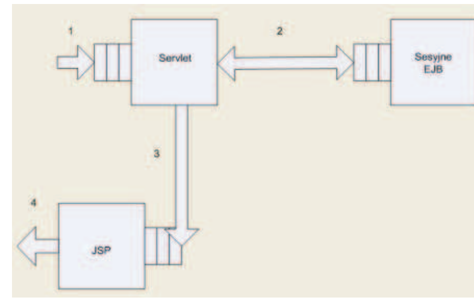
- Request service by servlet > JSP,
- Request service by servlet > session EJB > servlet > JSP,
- Request service by servlet > entity EJB > servlet > JSP,
- Request service by servlet > session EJB > entity EJB > session EJB > servlet > JSP.

For each of paths queue model and Event Graph diagram were created. In the following part of the paper path type of request service with session EJB was presented.

In that case path of service consists of three elements: servlet, session EJB and JSP page. This path is realized in consecutive steps:

- Step 1 – request arrival to servlet (asynchronously),
- Step 2 – invoking session EJB (synchronously),
- Step 3 – request is forwarded to JSP page (asynchronously),
- Step 4 – response is sent to client (asynchronously).

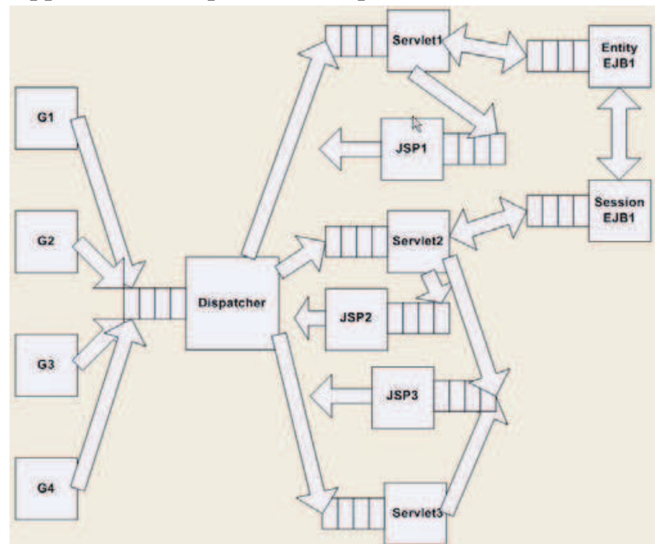
Picture Pic.14 presents path type of request service with session EJB.



Pic. 14. Path type of request service with session EJB

Example of Java EE application model

Example of Java EE application model serves four types of requests, K = $\{k_1, k_2, k_3, k_4\}$. In the model there are four generators: *G1*, *G2*, *G3*, *G4,* respectively for request types: k_1, k_2, k_3, k_4. Moreover, there are three elements of type servlet (Servlet1, Servlet2 and Servlet3), three elements of type JSP page (JSP1, JSP2, JSP3) and one element of type stateless session EJB (sessionEJB1) and entity EJB (entityEJB1). Queuing model of this application was presented in picture Pic.15.

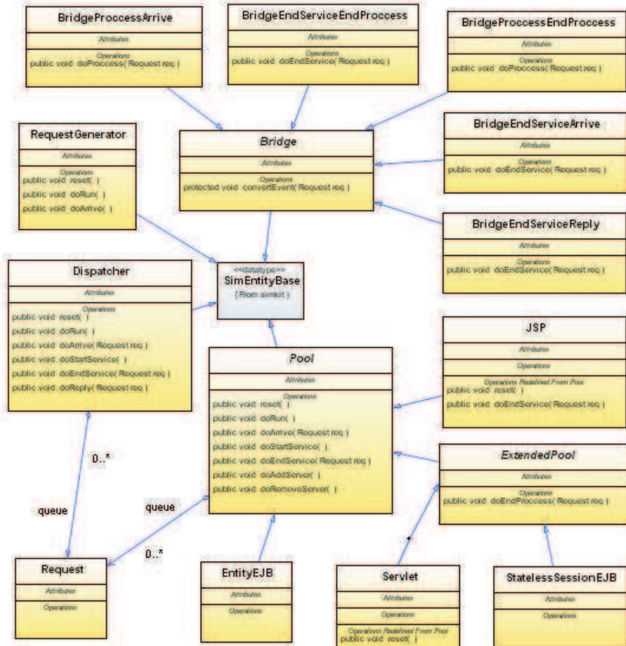


Pic. 15. Queuing model of example application

Path types of request service specify types of elements used to request realization. For example, if two types of requests *A* and *B* have the same path type – path type with session EJB – then *A* type requests are realized by *Servlet1*, *sessionEJB1*, *JSP1* and *B* type requests are realized by *Servlet2*, *sessionEJB2* and *JSP2*. So this is important to specify concrete model elements for paths.

6. Library of Java EE application model elements

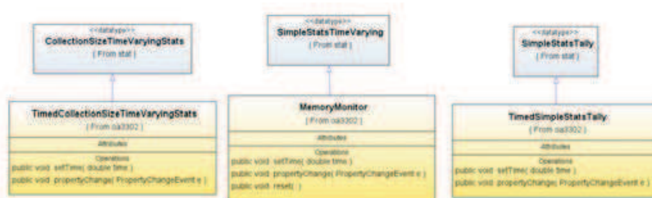
This library was based on SimKit library. Basic SimKit library was extended by classes which implements Java EE application model elements and classes used to collecting statistics. Class diagram which presents classes of Java EE application model was depicted in picture Pic.16.



Pic. 16. Class diagram of Java EE application model

This library was created due to have set of components to configure different models of Java EE applications. Elements of types Servlet, JSP, entity EJB and session stateless EJB are pools so behaviour connected with pool was placed in class *Pool*. Furthermore, servlet and session stateless EJB have ability to invoke other components' operations, so code responsible for such actions was placed in class *ExtendedPool*. Code, common for all bridges, was placed in class *Bridge*. Namesof descendant classes of class *Bridge* determine way of event conversion, e.g. BridgeEndServiceArrive class converts event *EndService* to *Arrive*.

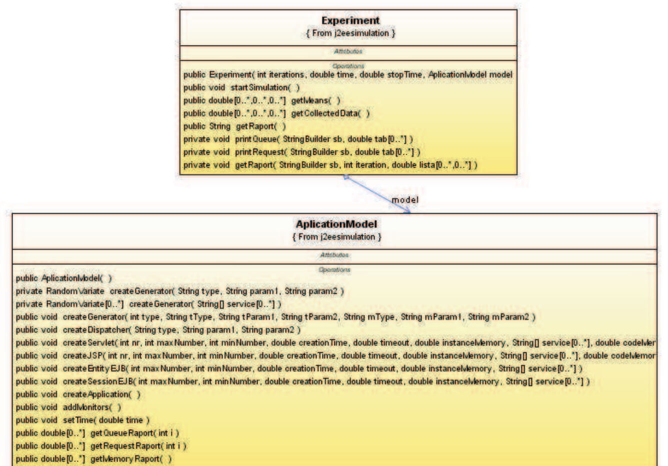
Classes which are used to statistics collection were presented in picture Pic.17. Those classes modify behaviour of standard SimKit classes.



Pic. 17. Class diagram with statistics classes

7. Design of simulation application

Simulation application implements model of Java EE application presented in picture Pic.10. This application uses elements of extended SimKit library. Simulation application allows input of model and experiment parameters, experiment execution and visualisation of its results. For each of four types of requests are collected statistics of response time and throughput. For whole application memory utilization is also collected. Application assures separation of code responsible for user interface from code responsible for application logic. Code responsible for simulation was placed in two classes: *ApplicationModel* and *Experiment* (Pic.18).

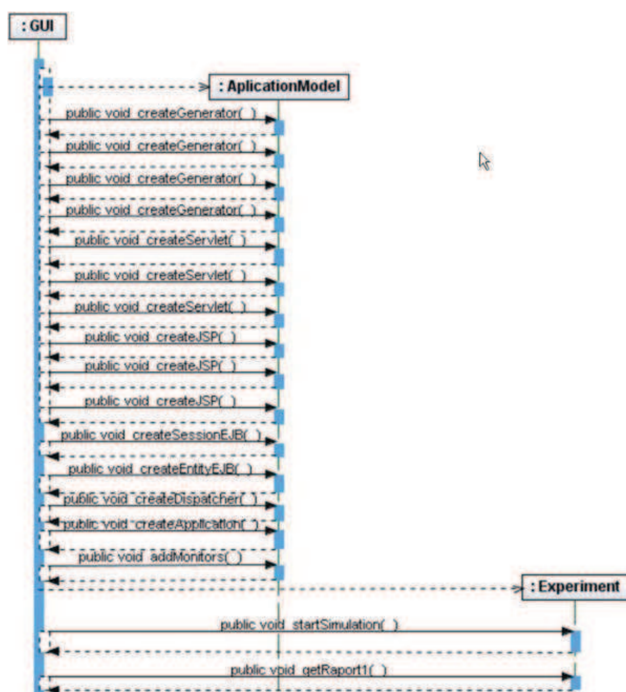


Pic. 18. Class diagram of application logic

Class *ApplicationModel* is responsible for model creation. Class *Experiment* realizes experiment and report generation. Such organization of code allows on preparation of one model and conducting many experiments on the same model. Conversely, we can prepare one experiment and conduct it for different models. Way of using classes *ApplicationModel* and *Experiment* was presented in sequence diagram (Pic.19).

First step is object creation of class *ApplicationModel*. Next, all elements of model are created by using methods: *createGenerator*, *createServlet*, *createJSP*, *createSessionEJB*, *createEntityEJB* and *createDispatcher*. In order to connect all elements in whole model method *createApplication* is invoked. Due to add monitors to the model, method *addMonitors* is invoked. After model creation object of *Experiment* class is created. Reference to object of *ApplicationModel* is available in

object of *Experiment* class. Invoking *startSimulation* method causes conducting of experiment. After experiment *getReport* method is invoked to get experiment's results.

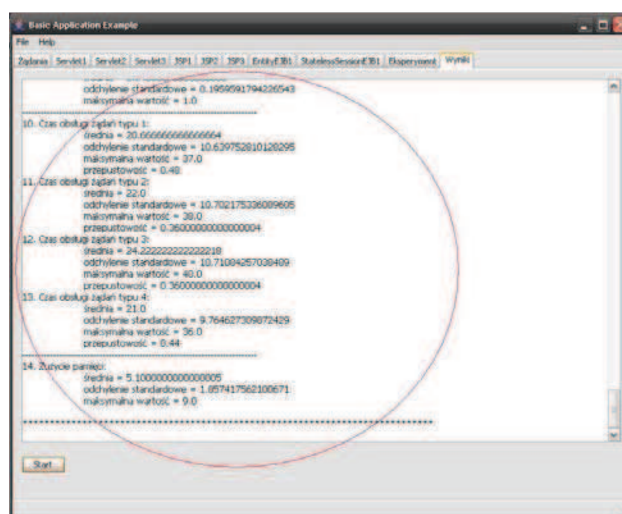


Pic. 19. Sequence diagram of model and experiment configuration

8. Simulation experiment

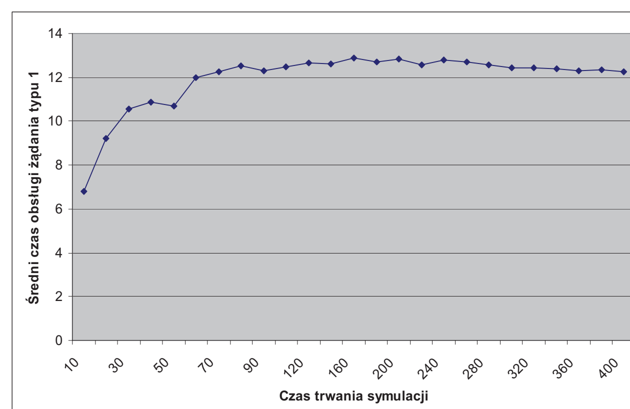
As previously stated, designed application realizes simulation model of Java EE application presented in the picture Pic.10. For each of four types of requests we should determine distribution time between two consecutive requests and distribution of memory amount required to their service. Implemented model consists of three servlets (Servlet1, Servlet2 and Servlet3) and three JSP pages (JSP1, JSP2, JSP3). For each of these elements should be set maximal and minimal number of threads in pool. Furthermore, following parameters should be specified: memory amount for one thread, thread's creation time, timeout and memory amount occupied by servlet's code. Moreover, distribution of request type service time should be specified. For stateless session EJB and entity EJB should be set maximal and minimal number of components in pool, memory required by component, creation time of component and timeout. In the application you can use tabs to enter those values.

After completing an experiment, report with its results is presented on tab „Wyniki" (eng. Results) (Pic.20).



Pic. 20. Report with experiment results

The application allows determination of duration time of unstable state of model during simulation. For example, for mean time request realization of type 1 this measure is stable for simulation which lasts minimum 80 units of time (Pic.21).



Pic. 21. Mean time of type 1 request realization depending on simulation time

9. Summary

The main result is set of model elements of Java EE application. LEGOS methodology was used to describe model of considered application. Extended library, which allows configuration of Java EE application, was designed and developed. In order to present capabilities of extended library, a simulation application was designed and developed. It is worth to emphasize that the library is easy to extend.

It is planned to add to the library remaining elements of Java EE application: stateful session EJB and message-driven EJB. Furthermore, it is considered to build graphical

application which allows model configuration by technique drag-and-drop. The following step will be construction of models which encompass architectural and design patterns. It will lead to creation of method and tool which will allow estimating efficiency (performance) of Java EE systems at the stage of their design.

10. References

[1] Noel J.: *The Evolving Definition of an Application*, www.softwaremag.com,

[2] *Servlet and JSP Development with IBM Rational Software Developer*, IBM WF311, 2006,

[3] Freeman E., Sierra K., Bates B.: *Head First Design Patterns. Edycja polska*, Helion, Warszawa, 2005,

[4] Jain R.: *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley-Interscience, New York, NY, April 1991,

[5] Buss A.: *Basic Event Graph Modeling*, Operations Research Department, Naval Postgraduate School, Simulation News Europe, Monterey, USA, 2001,

[6] Buss A.: *Discrete Event Programming with SimKit*, Operations Research Department, Naval Postgraduate School, Simulation News Europe, Monterey, USA, 2001,

[7] Buss A., Sanchez P.: *Building Complex Models with LEGOS*, Proceedings of the 2002 Winter Simulation Conference, San Diego, California, USA, 2002,

[8] Górski T.: *Symulacyjna metoda badania wpływu niejednorodności obciążenia i decentralizacji rozpraszania zadań na efektywność funkcjonowania rozproszonych sieci komputerowych*, Wojskowa Akademia Techniczna, Warszawa, 2000,

[9] Eeles P., Houston K., Kozaczynski W.: *Building J2EE applications with the Rational Unified Process*, Addison-Wesley, 2003,

[10] Alur D., Crupi J., Malks D.: *Core J2EE. Wzorce projektowe. Wydanie drugie*, Helion, 2004.