Simulation of Multi-Tier Webapplications using Anylogic

Thomas Strehl, e7701969, TU-Vienna

2015 12 15

Contents

1 Introduction **3**

2 Methods **4**
2.1 Queuing Networks 4
2.2 Web Application Mechanics 8

3 Model building **11**
3.1 Anylogic Building Blocks 11
3.2 System Under Simulation 15

4 Simulation parameters **17**
4.1 Server component configuration 17
4.2 StoryBoard resource allocation 18
4.3 Runtime variable simulation parameters 19

5 Simulation results **19**
5.1 Standard operation 20
5.2 Application server JVM Garbage Collector 20
5.3 Application server JVM Database Connection Pool 21
5.4 JVM Full GC StopTheWorld event 21
5.5 Database IO performance breakdown 22

6 Model verification **24**

7 Discussion **25**
7.1 ServerComponentModel implementation 26
7.2 Problems with Anylogic PLE 27
7.3 Future enhancements 27

8 Conclusion **28**

Abstract

This work demonstrates the simulation of a multi tier web application based on queuing networks. The server setup includes a web, an application, a database and a legacy system server. Each is comprised of one or more of the building blocks tcp connection, multicore cpu, java virtual machine and different types of delays. The java virtual machine contains a worker thread pool, heap memory with a generational garbage collector and a connection pool. The user interaction with the application is formulated as a story board of use cases that issue requests to the system, each consuming resources on the servers it visits. Threads, memory and pooled connections are treated as limited resources, that raise an exception if not available within a specified timeout interval. Resources and performance indicators are measured, logged and displayed on a dashboard. The simulation is designed to study the behavior of a multi tier web application under equilibrium and transient operating conditions, including long java garbage collector events, slow downstream server response or network outages. Monitoring data taken from a real world system are used for qualitative model verification. Simulation results assist in finding component configuration parameters such that overall system fault tolerance to anomalies of limited duration is increased.

1 Introduction

Queuing theory describes scenarios of consumers requesting some kind of resource and waiting for it to be delivered. Governing parameters are customer arrival rate and service time, and are used to predict performance measures like throughput, utilization, number of customers waiting, response time, or drop rate in case of limited queue size.

Queuing networks have been applied to a wide range of problems, from super market waiting lines to Monte Carlo simulation for High Energy physics experiments [FrÃijhwirth and Regler, 1983]. Modeling of web applications and the IT infrastructure these run on introduces several new concepts. Servers provide cpu, disk and memory resources, and are connected by tcp networks. The token of execution is routed from server to server and eventually back to the user by the application logic. The user interacts with the system by a mixture of requests with individual service demands on the components visited.

Closed solutions exist for the calculation of performance measures for a wide range of scenarios [Robertazzi, 2000], but these become prohibitively complex as the detailed mechanics of real world web servers or java virtual machines (JVM) need to be modeled. Discrete event simulation (DES) of queuing networks is used to handle such non tractable problems, opening up a wide range of scenarios to laboratory examination. Being stochastic by nature, DES shifts the complexity of closed form calculations to sampling measures from the simulation.

Closed form solutions are calculated for the queuing network in equilibrium state, and thus provide parameters that apply to a system in undisturbed operation. This type of condition is typically assumed in web capacity planning on estimation of resources required for e.g. maximum throughput during peak hours [Menasce et al., 2001]. Burstiness in the request structure and heavy-tailed service time distributions may disturb the assumed equilibrium to such a degree that this type of capacity planning estimations are rendered unfeasible.

This is particularly true for transient states induced by intermittent system failure or inevitable interrupts like long running garbage collection (GC) events in a JVM. Such events typically lead to queues being flooded or piling up beyond capacity planning estimated high watermarks. Out of reach for analytic solutions, DES can be used to reproduce such transients, analyze circumstances and develop strategies to increase system fault tolerance (see e.g. van AS [1984] for telecommunication processes).

Several software packages including Enterprise Dynamics, Matlab/Simulink and Anylogic exist for the DES of queuing networks, but only few implement IT infrastructure entities like multicore servers (e.g. Bertoli et al., 2009). Anylogic is used in this work for its graphical design interface, the rich queuing model library, and, in particular, the option to extend the system with custom functionality written in java. Components that can be modeled along the queuing paradigm are constructed with standard library objects including route, queue, wait, service, and resource seize and release. Other components that do not lend themselves to be formulated in terms of queues, like the generational garbage collector that runs inside a JVM, make use of the Anylogic java extension framework.

The concepts implemented in this Anylogic 'Process Modeling Library' project are based on close examination of the "Bank of Austria" "eBanking for CEE countries" production system. The complex setup has been condensed to its core components, the storyboard for the user interaction was derived from actual traffic analysis, and the operation parameters are sized along actual values.

The document is organized as follows: Section Methods introduces basic concepts of discrete queuing networks, their simulation and how these relate to the work flows found in web applications. Section Model building gives details on the components implemented in Anylogic with simulation scenarios and observations thereof presented in Sections Simulation parameters and Simulation results. The document concludes with some comments and an outlook for future enhancements in Sections Discussion and Conclusion.

2 Methods

This section first introduces basic operating parameters for a single queue with Poisson arrival process and exponential service times. It demonstrates that the number of customers in the queuing system is sufficient to fully describe its operation. The results can be used as a first approximation to mean values in many situations.

The section on web application mechanics outlines the entities and workflows used to describe a typical application architecture. The role of the web, application and data tier servers is described, and some details are given about the components these are comprised of and how these interact.

2.1 Queuing Networks

The simplest possible model of a web service consists of just one service station. Requests enter the station with the arrival rate λ and are serviced with the service rate μ. Requests are queued in case they find the service station busy. Once serviced requests leave the system. Properties like mean system response time or mean throughput are the quantities of interest. This black box type of approach to modeling a service is called 'System Level Modeling'. This work will treat each service on the 'Component Level', taking into consideration inherent processes.

2.1.1 State Transition Diagram

The basic parameter to describe a queuing system is the number k of requests in the system, currently being serviced or waiting for service. The number of possible states is therefore enumerable. The transitions between possible states are depicted in the so called

'State Transition Diagram', short STD. The transition of a state with k requests into one of its neighboring states with k+1 or k-1 requests is triggered by the arrival of a new or departure of a serviced request.

2.1.2 Memoryless Property

The assumption that only the number of requests is needed to fully describe the state of the queueing system implies that the history of the system is not relevant for determining its future. This property is called the 'memory less' or 'Markovian' assumption on the workings of a queuing system.

2.1.3 Equilibrium State

The basic properties of a queuing system calculated in this section pertain to the system in 'equilibrium state'. This means that the number of requests arriving at the system equals the number of requests being serviced over the observation period. Equivalently the number of requests in the system are required to be the same at the beginning and the end of the observation.

2.1.4 Flow-In Flow-Out Equations

Because the system is in its equilibrium state, the probability to leave the current state must be the same as the probability to enter the current state. This pertains to all possible states, see Fig. 2.1. This 'flow in' = 'flow out' principle is the very foundation for the exploration of further properties like 'number of requests in the system', response time or throughput.

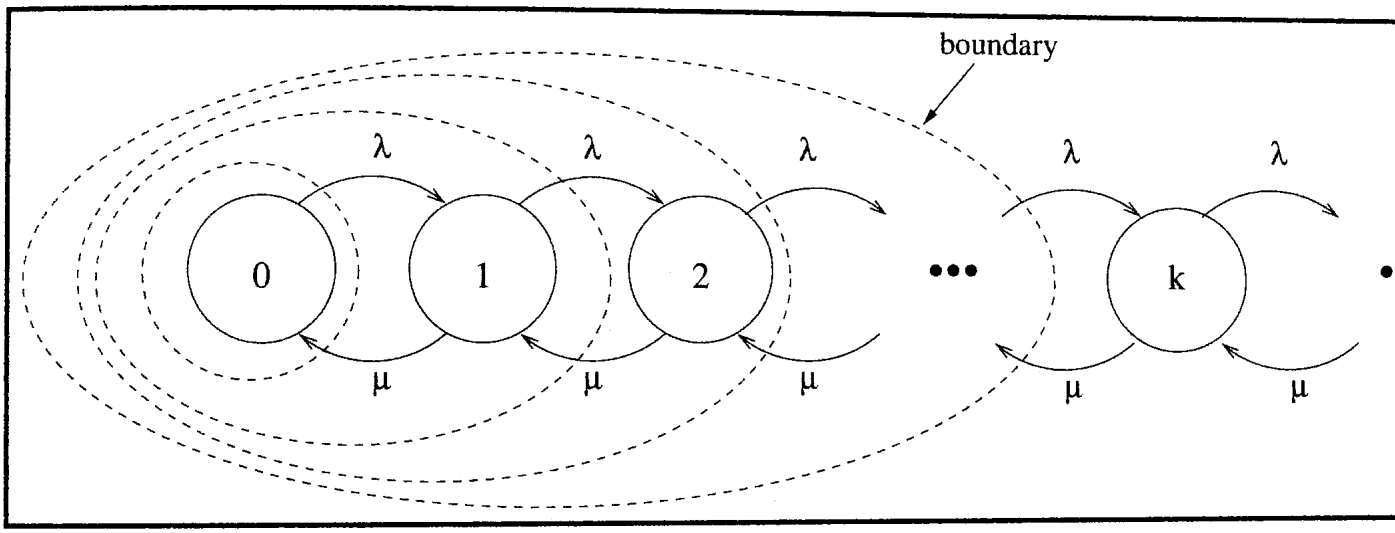

Figure 2.1: StateDiagram [Robertazzi, 2000]

2.1.5 Characteristic Properties

Based on the Flow-In Flow-Out equation the probability p_k that there are k customers in the system can be calculated [Robertazzi, 2000] as

$$p_k = p_0 \left(\frac{\lambda}{\mu}\right)^k = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^k \tag{2.1}$$

The probability that the queue is empty and no customer is being serviced is given as

$$p_0 = 1 - \frac{\lambda}{\mu} \tag{2.2}$$

where λ is the arrival rate, and μ is the service rate. All other measures are derived from the probabilities p_k. Equation (2.2) implies $\lambda < \mu$, that is, the arrival rate for a system in equilibrium state must always be less than the service rate, which makes immediate sense. In other words, the utilization, that is the probability the system is not empty, is given as

$$U = 1 - p_o = \frac{\lambda}{\mu} \tag{2.3}$$

The mean number N of requests in the system is calculated as the mean value of the states the system is in in equilibrium state, and evaluates to

$$N = \frac{U}{1 - U} = \frac{\lambda}{\mu - \lambda} \tag{2.4}$$

The variance of N is given as

$$\sigma_N^2 = \frac{U}{(1 - U)^2} \tag{2.5}$$

which states that system variability in terms of numbers of requests queued increases dramatically as the utilization approaches one (see Fig. 2.2).

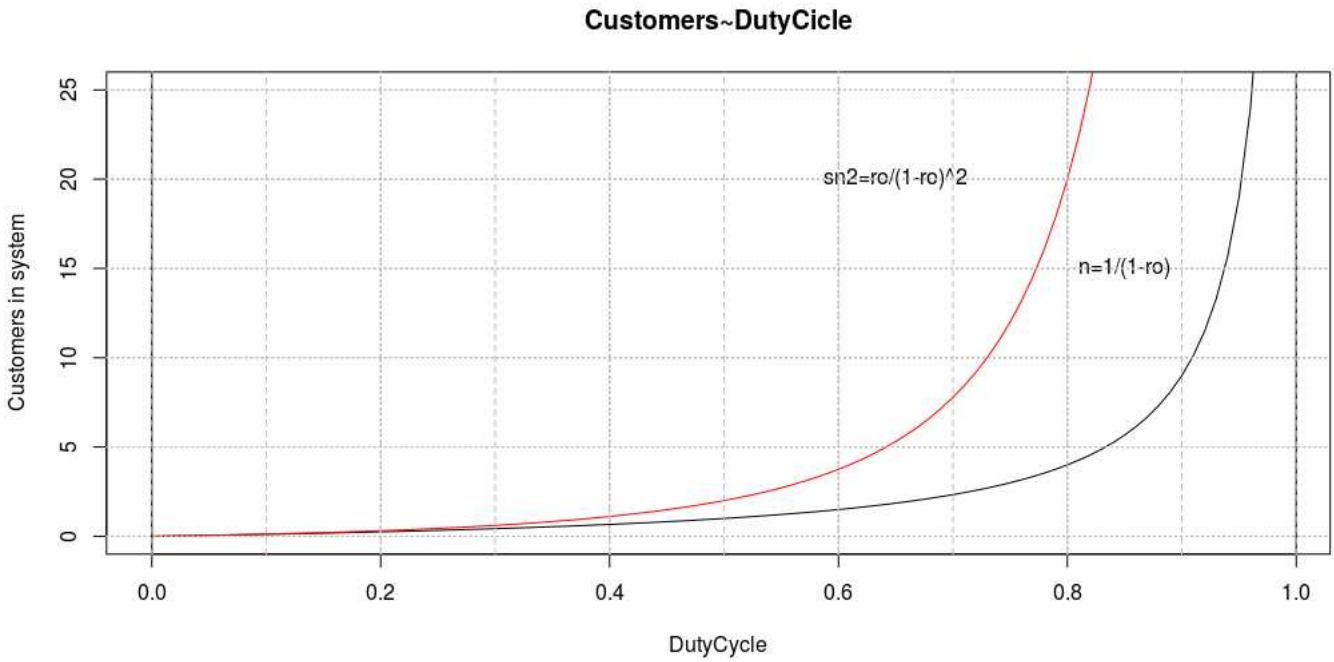

Figure 2.2: Number of customers in system

The throughput X of the system is μ whenever a request is being serviced. Thus

$$X = U * \mu = \lambda \tag{2.6}$$

This is expected for a system in equilibrium state as all requests are being serviced in the observation period. The mean response time is calculated using Little's Law

$$N = R * X \tag{2.7}$$

$$R = \frac{N}{X} = \frac{1}{\lambda} * \frac{\lambda}{\mu - \lambda} = \frac{1}{\mu - \lambda} = \frac{S}{1 - U} \tag{2.8}$$

where S is the service time. Equation (2.8) states that the response time is close to the service time while the utilization is low, and increases beyond bounds as it approaches one.

As indicated above, the formulas in this Section hold for the equilibrium state of a queuing system with unlimited queue length, requests generated by a Poisson process with arrival rate λ for an unlimited population, and served with an exponentially distributed service time with mean service rate μ.

2.1.6 Generalized systems

Real world queuing systems require generalization in several aspects, including non-poisson arrival process, limited queue length, limited population, and non-exponential service time distribution. These generalizations are often written in Kendall's notation defined by

$$GI/G/n/W \tag{2.9}$$

where GI is the type of arrival process (most general form denoted as 'Generally Independent'), G the type of service process (most general form denoted as 'General'), n the number of service stations, and W the maximum size of the queue including the service stations. Several distributions for GI and G are used, with M denoting a negative exponential process (Markoff process).

2.1.7 Limited queue size

The formulas introduced in this Section pertain to the $M/M/1$ model in Kendall notation, with $M/M/1/W$ the same with maximum queue size W.

$$p_k = p_0 \left(\frac{\lambda}{\mu}\right)^k \frac{1}{1 - \left(\frac{\lambda}{\mu}\right)^{W+1}} = \left(1 - \frac{\lambda}{\mu}\right) \left(\frac{\lambda}{\mu}\right)^k \frac{1}{1 - \left(\frac{\lambda}{\mu}\right)^{W+1}} \tag{2.10}$$

2.1.8 Closed user groups

The generalization to a limited population describes a queuing system for a closed user group, that is, for users returning to the service station after a 'ThinkTime' Z. The number of customers in the system is constant and for $Z = 0$ equals the total number of customers at all times. For a $M/M/1$ closed system with $Z = 0$, Little's law states that the response

time is proportional to the number of users. In real world web applications the arrivals consist of requests originating from both new and returning users and thus constitute a mixture of open and closed system properties.

2.1.9 Queuing networks

Generalizing to a network of queues that are connected by random routing, it can be shown that the probabilities for each possible system state $p(\underline{n})$ can be written as a product of the initial state and the throughput and service rates of each of M queues. The equations governing networks of queues are thus called product form solution.

$$p(\underline{n}) = \prod_{i=1}^{M} p_i(n_i) = \prod_{i=1}^{M} \left(1 - \frac{\theta_i}{\mu_i})\right) \left(\frac{\theta_i}{\mu_i}\right)^{n_i} \tag{2.11}$$

Comparing the terms of the product form solution to the decomposition of a joint probability into its marginal probabilities for independent random variables suggests that the queues act independently from each other. That is, the state of a queue at a specific point in time cannot be inferred from the state of the other queues.

2.1.10 Closed form solutions

Several further generalizations with closed form solutions of increasing complexity exist [Robertazzi, 2000, Menasce et al., 2001, van AS, 1984]. This includes queues with multiple service stations, mixtures of service demands, and advanced service scheduling strategies.

2.1.11 Discrete Event Simulation

Simulation is used for systems that cannot be described in terms of queuing networks with closed form solutions. The simulation technique typically used to build models of a communication type network is stochastic discrete event simulation (DES). The events that occur during simulation of the queuing network (arrival, departure, routing, servicing) are mimicked such that the states the system goes through can be observed in detail.

Scheduling of events like inter arrival times, service times, or routing is based on a stream of pseudo random numbers. A discrete event simulation thus is a statistical experiment, with the performance characteristics of the model (throughput, resident time, or queue lengths) themselves random variables. Mean values and confidence intervals are obtained from multiple or stratified simulation runs.

The sensitivity of results to changes of parameter values is estimated by rerunning with sets of different values for the parameters under consideration. Another approach uses likelihood ratios, that utilizes the effect of the natural variation of the random process with respect to the parameters.

2.2 Web Application Mechanics

Real world web applications often follow the classical three tier architecture of web, application and data tier. The servers typically run a preemptive time slicing operating system on multicore cpu hardware. Communication between servers/services is established with application level protocols on top of tcp networks.

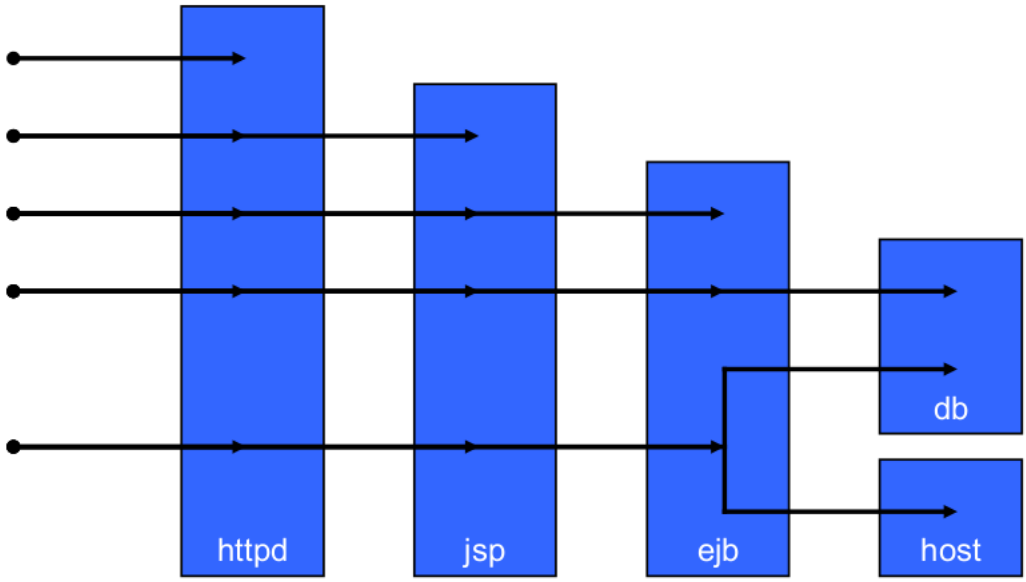

Figure 2.3: Multi tier application

As seen from the client, the 'token of execution' (request) is passed from the client system (browser) over a tcp connection to the web server, which forwards the token via another tcp connection to the application server, that in turn uses another tcp connection to talk to it's downstream data services. Once a response is prepared by a system, control is passed back to the calling system, freeing up the resources that were allocated (memory, tcp connection or worker thread) on the way. In this sense, the 'token of execution' visits most of the runtime resources twice, once for allocation on the way from the client to the data sources, and once for de-allocation on it's way back to the client. For session based applications (via e.g. login or cookie) the 'session' related objects remain allocated until logout or timeout.

2.2.1 Web Server

The webtier often consists of a webserver that essentially only controls the communication between the client browser and the application server. Functions employed are handling of tcp connections (http 'keep-alive'), encryption, manipulation of http headers, URL rewrite and redirects, as well as load balancing/failover to the downstream application server(s). Apache webserver is a typical choice in the open source domain. The runtime environment consists of a watch dog and a dispatcher process, that relays incoming requests to multithreaded child processes, that are started/shutdown as load requires. Communication with the application server(s) is often run via the ajp13 protocol (a variant of http) that features a connection pool for better performance. Controlling the number of client side request via the keep-alive configuration and the number of server side connections via connection pool parameters is essential for proper operation.

2.2.2 Application Server

The application server usually runs some 'application' or 'portal' server software that provides the framework for the deployment of the actual business logic services. Several of

these framework servers (including liferay, tomcat, jboss, ...) are written in Java. For these the configuration of the Java virtual machine runtime is essential for proper operation. Key parameters include the number and speed of CPUs, memory resources, and java worker thread pool as well as connection pool configuration for communication with other systems.

Java VMs typically feature a generational memory model that is comprised of an area for 'permanent' objects (classes, statics, ...), a 'young' or 'eden' generation that is used for initial allocation of application memory, and an 'old' generation, that is populated with long lived objects. Once allocation in any of these areas hits a high water mark, memory that is no longer referenced by the application is reclaimed during a 'garbage collection' (GC). A GC in the 'eden' generation removes free objects and moves still used ('surviving') objects to the 'old' generation, optionally employing one or more intermittent 'survivor' generations. Current GC algorithms run most of the memory reorganization in the background, and only need exclusive ('synchronized') access to application memory during a final 'StopTheWorld' event. The duration of such 'StopTheWorld' events typically last from a few to many dozen milliseconds in the 'eden' generation, and can take up to a few dozen seconds in the 'old' generation, depending on memory size, memory allocation patterns, cpu speed and number of cores.

2.2.3 Data Servers

Data storage and retrieval is typically handled via a dedicated database and/or preexisting legacy systems. While the operation of the database in a given setup may be accessible via configuration of connection, cpu or disk resources, legacy systems or off-premise services essentially act as a 'black box'.

2.2.4 Concert Operation

As seen from a particular server, proper operation of the down stream systems are vital for it's own operation. In particular, throughput as dictated by incoming client requests needs to be handled by the downstream systems in a timely manner, or queuing in the upstream systems will inevitably impact operation, up to complete outage. Several options exist for a server to protect against problems on downstream data services. These include connection and response timeouts, connection pools, and asynchronous communication. Connection timeouts protect against systems not reachable on the network level, response timeouts against long runners that block resources over extended periods of time. Connection pools and related timeouts help to retain costly connections when needed and release when idle for a specified period of time.

While it may be tempting to configure resources such that queuing of requests could cope with any downstream anomaly over extended periods of time, this approach generally runs into one or more problems:

- local resources may be effectively limited by underlying systems (e.g. physical memory or memory on 32bit systems, number of threads in JVM, number of tcp connections on OS level)
- continued queries to an unusually slow or unresponsive downstream system may further deteriorate its state
- continued connection attempts to an unavailable system leads to excessive queuing while not providing service to the user

In addition, web application response times beyond some ten seconds will inevitably lead to the user cancel the request in most cases. Thus a system configuration that enables excessive queuing of requests consumes system resources while not being able to deliver its service as expected.

Consequently a system configuration that is able to mask intermittent problems from the user while handling substantial problems effectively is desirable. Parameter settings for such a design goal are difficult or impossible to calculate. Systematic studies of system behavior for different sets of parameters are difficult given the fact that many problems occur only on production system.

The Anylogic 'ServerComponentModel' presented in this work constructs a set of web application servers as a network of queuing services that allows analysis of critical parameters under laboratory conditions.

3 Model building

This section describes the servers and components that have been implemented in Anylogic. 'Process Modeling Library' objects have been used where possible, often customized with java code snippets in Anylogic predefined places. Other components like the database connection pool in the JVM have been implemented from scratch using only Anylogic input/output interfaces.

3.1 Anylogic Building Blocks

The web, application, database (DB) and legacy backend (BE) servers are built from a selection of these components:

- a tcp connection that is the entry point into the server
- a time sliced multicore cpu system
- a Java virtual machine featuring a garbage collector, a thread and a connection pool
- delay units that draw delay times either from a distribution or a data file

Except for the cpu and the delay units all components are visited twice, for allocation and de-allocation of resources.

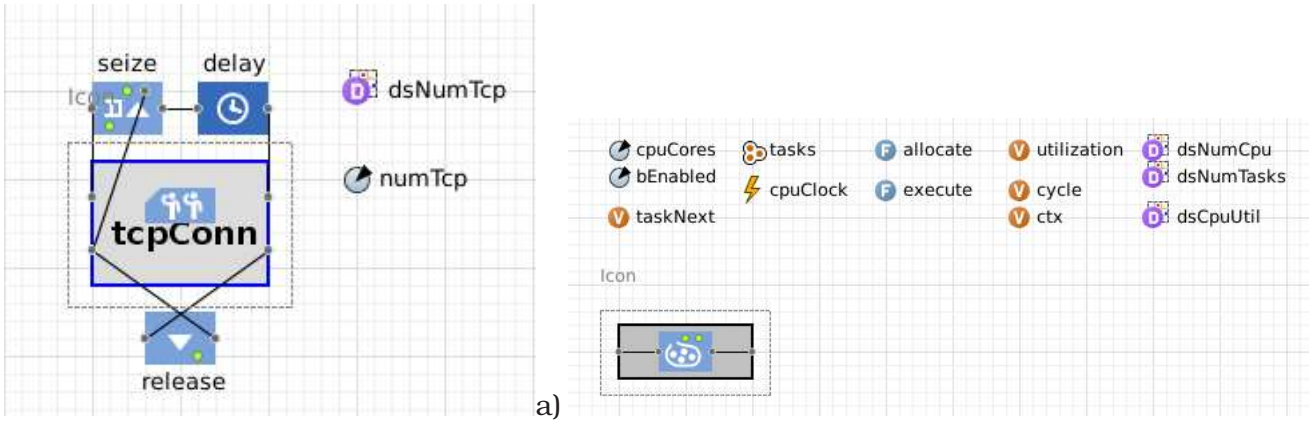

Figure 3.1: ServerComponentModel: a) ResTCP b) ResCPU

3.1.1 ResTCP

Allocation of a tcp connection is implemented with a 'seize' block and a resource pool. A dummy 'delay' ensures time propagates as a resource is picked. The size of the resource pool is limited, and the timeout channel of object 'seize' is enabled. Exception handling follows the standard procedure described in 3.2.3. Release of a tcp connection is handled by a release block that a request visits on leaving the server.

3.1.2 ResCPU

This component emulates a preemptive time slicing operating system on a multicore/cpu hardware. It is implemented with a wait block, a task list and an execution function that is triggered by a periodic event. Visiting agents deposit a request for a certain amount of CPU units on the task list, and are stored in the wait block until all requested CPU units have been executed. Each invocation of the execution function removes as many units from the task list as there are cores in the cpu, serving pending tasks in turn. The trigger event is fired with 100Hz, thus each unit represents 10ms on cpu. Utilization is implemented as the ratio of the maximum possible number of cpu units that can be removed per second (number of cores * 100) and the actual number worked. In support of a 'StopTheWorld' event triggered by a Full GC in the ResMEM resource of a JVM, invocation of the execution function can be enabled/disabled on the fly.

3.1.3 ResTHREAD

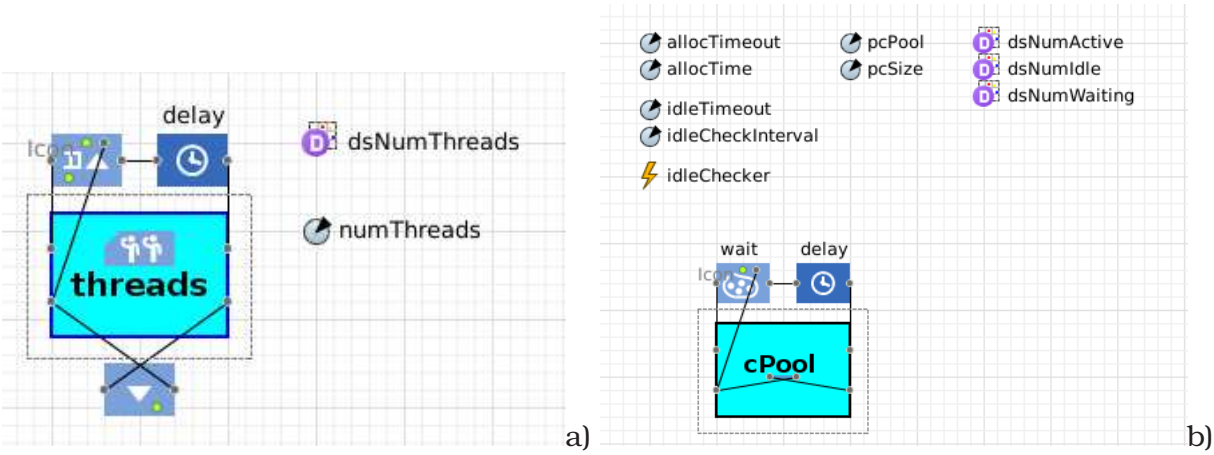

Figure 3.2: ServerComponentModel: a) ResTHREAD b) ResCP

This component resembles the 'worker thread pool' concept implemented by many java application servers. It is a component of the Java Virtual Machine building block. Functionality is similar to the ResTCP component, but is affected by the 'StopTheWorld' GC events in the JVM. During these periods, all JVM internal activity is suspended, including allocation of new threads. The default size of the thread pool is one hundred, and requests waiting longer that a timeout of ten seconds for their thread are flagged with a thread timeout exception.

3.1.4 ResMEM

ResMEM implements a generational memory structure as outlined in 2.2.2. It features a newGen and an oldGen memory area with corresponding Garbage Collectors and related 'StopTheWorld' (STW) events. The length of these STW events is set to 0.1 second in the newGen, and 5 sec in the oldGen, but is prepared for dynamic calculation (size of the generation, number of memory blocks removed/retained). Three types of memory are allocated by requests: Short lived with a fixed lifetime of 0.1 second, medium lived with a lifetime of the request duration, and long lived session memory that is allocated on login and released on logout (see Section 3.2.2 for details of the user session). If the Full GC in the oldGen memory pool cannot free enough memory to accommodate survivers from the newGen, the allocating request is flagged with an 'Out Of Memory' (OOM) exception state.

The core JVM components are complemented with several 'stop' and 'buffer queue' objects to make visible the frozen state of the JVM to the outside components. Also, working only on garbage collection during a StopTheWorld event, the cpu is disabled for request handling during these periods.

Details are implemented in java classes JvmMemBlock.

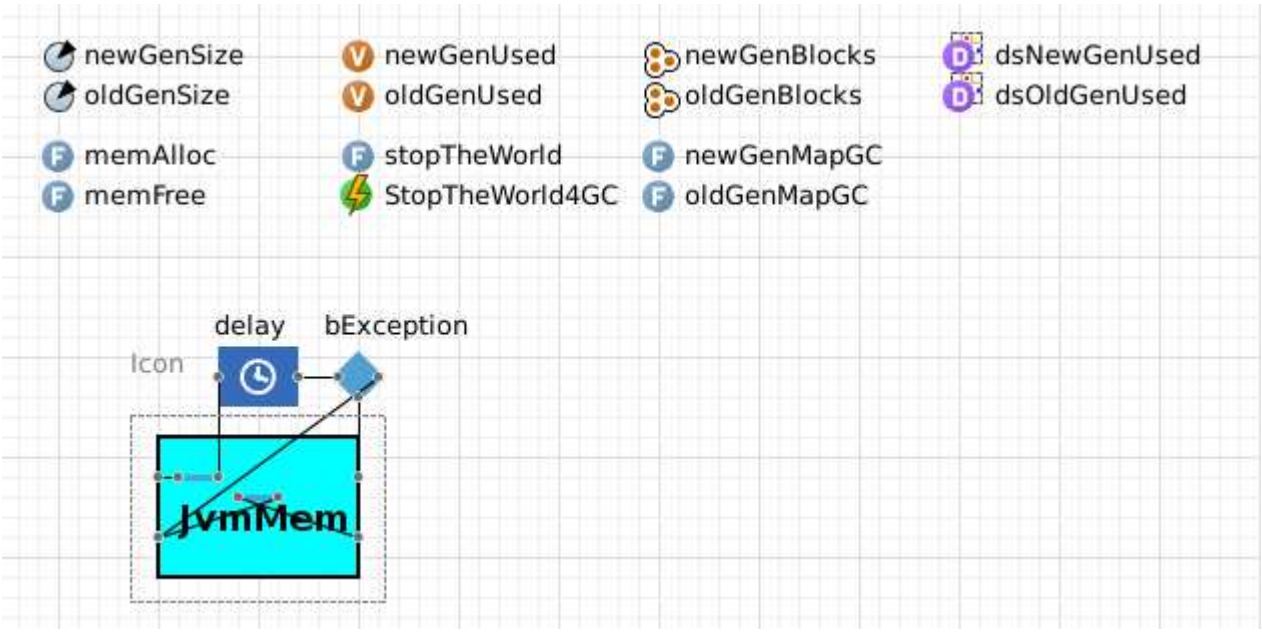

Figure 3.3: ServerComponentModel: ResMEM

3.1.5 ResCP

In the present setup the configuration pool is used by the JVM in the application server to connect to the database server. Establishing a new connection to a database is a costly operation as it often involves starting a new process on the database server. Real world connection pools offer complex functionality, including pre-request or periodic connection probing, and forced reclaim of unused or stale connections. ResCP implements a 'least recently used' (lru) reuse of idle connections, a idle timeout with periodic timeout checker, a fixed delay of 0.5 seconds for creating a new connection, and a allocation timeout of five seconds with corresponding exception flagging of the offended request. The latter is implemented by a 'wait' block with activated timeout. Active connections are returned to

the idle state on return of the allocating request from the downstream database server. The connection is handed over to the oldest request in the waiting queue, if any.

Details are implemented in java classes CPconnectionPool and CPpooledConnection.

3.1.6 ServerWeb

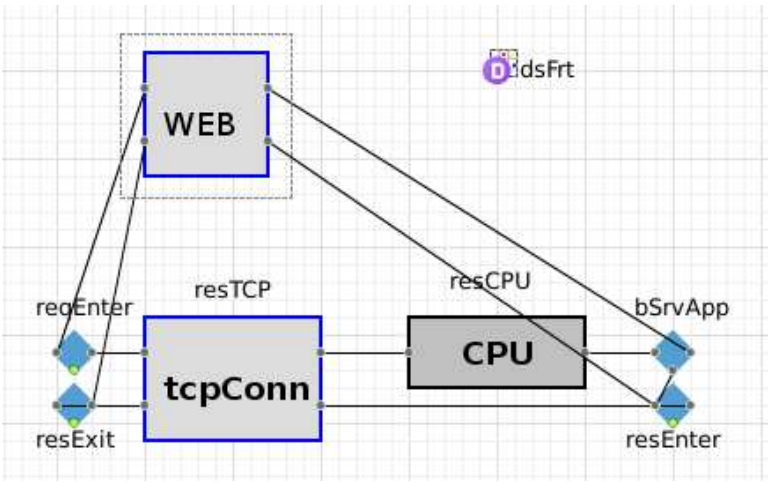

Figure 3.4: ServerComponentModel: ServerWeb

The web server is implemented with a tcp connection and a cpu. It connects the client browser to the application server without advanced features like keep-alive or connection pooling. While the present work concentrates on emulating concepts found on the application server, a complete setup would require more elaborate emulation of the web server features.

3.1.7 ServerApp

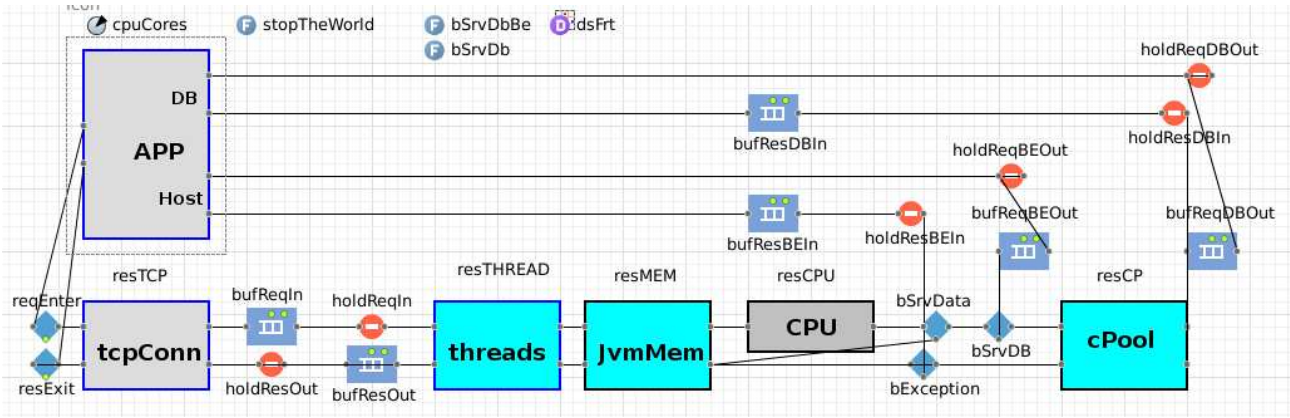

Figure 3.5: ServerComponentModel: ServerApp

The application server is comprised of a tcp connection, a cpu and a Java VM, that in turn consists of a thread pool, a generational garbage collector and a connection pool (all described in detail earlier). The 'StopTheWorld' events triggered by the GC are implemented

with a total of six 'hold' blocks, two in the direction of the web server, and two on side of each of the database and backend servers. These are switched to the 'blocked' state during the STW events, effectively blocking all traffic in and out of the JVM. In addition, the CPU is halted during STW events. Furthermore custom routing directs requests to the database and/or backend server if so scheduled by the StoryBoard for the particular request.

All hold blocks are complemented by infinite buffer queues that accept requests from up/down stream systems during STW events. These are provided for technical reasons and do not resemble real world behavior very closely. They may be dropped in a future version.

3.1.8 ServerData

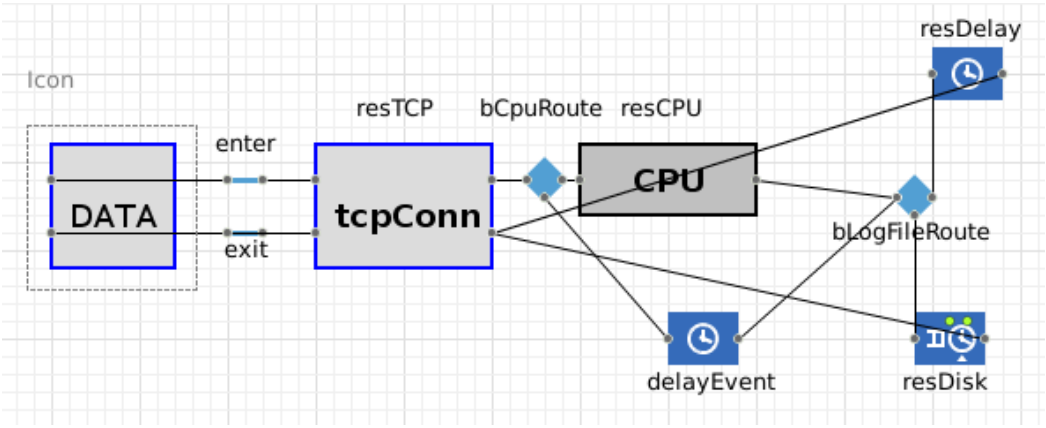

Figure 3.6: ServerComponentModel: ServerData

The database and backend servers are variants of a general 'data' server, that consists of a tcp connection, a cpu and a delay that either reads the delay time from a file or samples from a distribution. The implementation of the database is comprised of a tcp connection, a cpu and a delay emulating disk io. The backend server consists of a tcp connection and the response time is read from a data file via a delay block.

3.2 System Under Simulation

The user interaction with the 'System under Simulation' (SUS) is modeled by a 'Request' object that extends Anylogic class agent and is orchestrated by a StoryBoard and a TaskList. The experimental setup is complemented by simulation level exception handling, a response time measurement facility, and a trace and debug log.

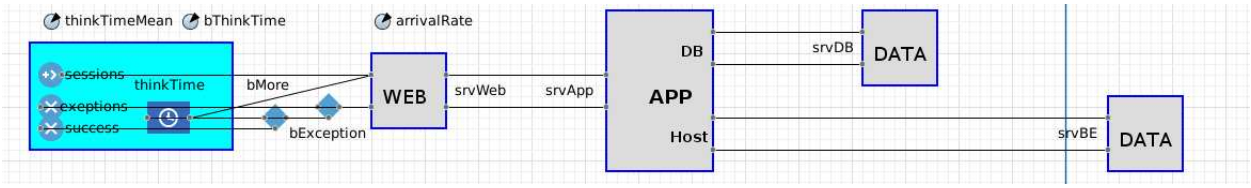

Figure 3.7: ServerComponentModel: main

3.2.1 Main and Request

The Anylogic 'main' agent brings together the SUS server landscape, the user interaction with the SUS and the visualization of the data acquired during simulation. The user issues requests to the system along the StoryBoard, optionally taking a pause ('think time') between requests. The duration of the 'think time' and the frequency of 'request' agents originating from the 'session' source block can be influenced by sliders coupled to corresponding parameters during runtime. Custom routing based on 'SelectOutput' blocks are used to forward requests that have more use cases on their TaskList to the web server, dispose agents that have the exception flag set to the 'exceptions' sink block, and release successfully finished sessions to the 'success' sink.

3.2.2 StoryBoard and TaskList

User behavior is simulated along a StoryBoard that starts the session with a login, followed by four use cases, each consisting of a menu navigation and a transaction kind of request, plus a final logout. Each of these ten requests consumes a specified amount of different resources on one ore more servers. Menu navigation requests are answered by the web and application server alone, while transactional requests include a visit to the database and backend servers. The StoryBoard is implemented in java class StoryBoard as a blueprint for the actual task list for a specific session. The task list contains a pointer to the StoryBoard blueprint plus state information on the session executing, including timers for response time measurement and servers visited.

Details are implemented in classes ResSrv, ResUcSrv, StoryBoard and TaskList.

3.2.3 Exception handling

Exceptions are implemented by the exState flag of the 'Request' agent. It is set on specific events, mostly on timeout waiting for a resource or e.g. the JVM running out of memory. A request that has the exState flag set is returned to the user on the shortest path possible (freeing up any held resources on the way) and is deposited by custom routing into the 'exception' sink block.

3.2.4 Data acquisition and visualization

Resource usage and response time is measured on the block, server, request and session level.

Each server exposes request response time by a 'Data set' object named dsFrt. On main, these data are fed into corresponding 'Histogram Data' objects called hstFrt<server>. Both are visualized in the data acquisition panel with a point plot and a histogram, respectively. In addition, the number of tcp connections and the cpu utilization is visualized for each server. Furthermore detailed information on the internal state of selected components are plotted, including of the usage of the JVM connection pool and generational memory areas. The panel is completed by plots showing overall system throughput in terms of requests handled per second and the number of exceptions that occurred so far.

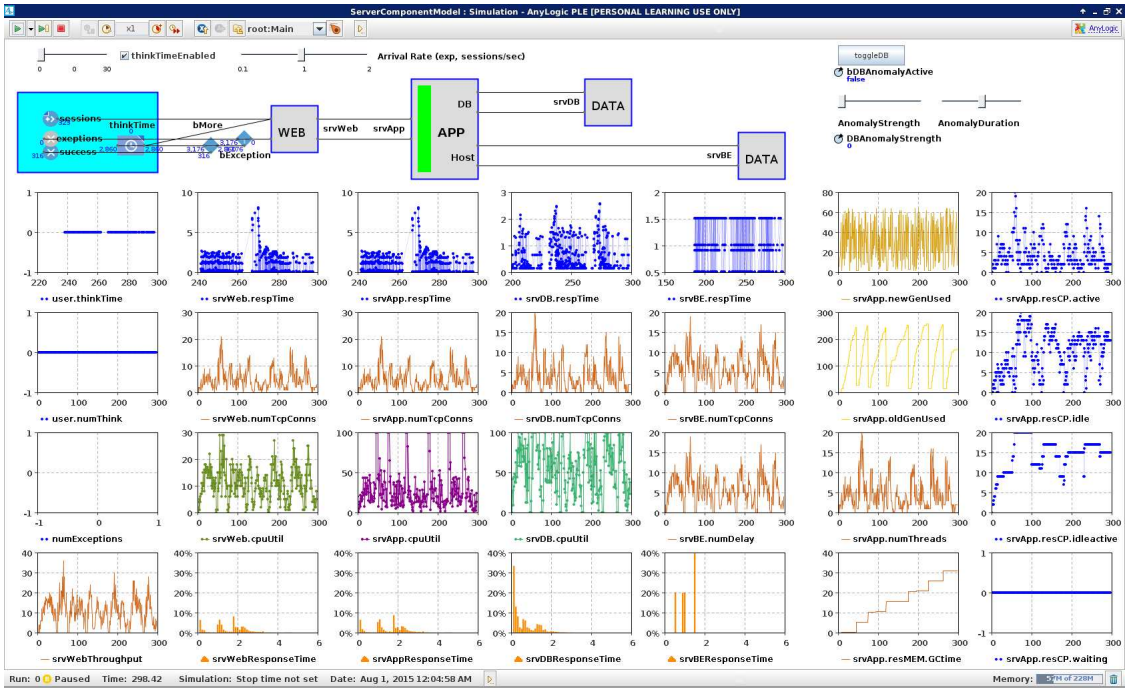

Figure 3.8: ServerComponentModel: DashBoard

4 Simulation parameters

Simulation parameters are implemented in three stages:

- configuration of server components (design time Anylogic parameters)
- specification of service demands in a session storyboard
- parameters adjustable during simulation (runtime Anylogic parameters)

Actual values are inspired by data recorded in a real world online banking system. The runtime parameter ranges are designed such that critical component states can be provoked. Like this system under stress behavior can be observed and analyzed.

4.1 Server component configuration

Components are configured to resemble real world system behavior while at the same time exhibiting typical phenomena like the JVM Full GC events in minutes rather than hours. Thus some parameters are substantially scaled (e.g. size of JVM heap memory), while others (including database connection pool timeout and GC StopTheWorld duration) are close to actual values.

4.1.1 Web server

- CPU consumption is sized such that the web server contribution to the total response time is about one percent.

4.1.2 Application server

- The JVM worker thread pool (the pool of threads that handle requests coming in from the web server) is set to one hundred, with an exception immediately being thrown in case of pool exhaustion. This ensures no more than one hundred requests are being served at any one time. The actual value is subject to tuning of system configuration and needs to be aligned to other parameters, including traffic volume and database connection pool timeout.
- The JVM young gen is configured for 64mb, the old gen for 256mb. This is around a factor of ten lower than settings common to loaded 32bit production systems.
- GC StopTheWorld times are ten ms for young gen, and five seconds for old gen. GC times in real world systems heavily depend on the type of GC configured, the sizing of the generational memory pools and CPU speed. Values of several seconds for Full GC in old gen are observed frequently for the CMS (Concurrent Mark and Sweep) collector (available from JDK 1.5) with pool sizes beyond one GB.
- The memory allocation is 1mb for session memory, 1mb for the duration of each request, and 1mb short lived memory for request setup/destroy.
- Session timeout is sixty seconds. Typical values in real world online systems range from ten minutes to maybe an hour. This value comes into play for sessions that are not logged out by the user or run into an unrecoverable exception. It heavily influences the amount of memory that needs to be reserved for objects with a lifetime of session duration. For the purpose of the simulation this timeout is effective for sessions that run into an exception.

4.1.3 Data servers

- The DB connection pool is configured with a max size of twenty, one second setup time for a new DB connection, thirty seconds idle timeout and five seconds allocation timeout. Among these parameters the connection setup time and idle timeout are exaggerated to better show effects. Pool size and connection timeout typically have to be aligned with other parameters like system load in terms of number of DB requests per unit of time and JVM thread pool size.
- DB IO response time is modeled as a device with discrete, constant service time and unlimited queue capacity. Upstream queuing effects manifest themselves as the initially discrete, constant response time being increasingly smeared out towards longer duration. DB cpu consumption is implemented using the same multicore, preemptive HW and OS model as in the application and web servers.
- BE response time are discrete, constant delay values between 500ms and 1000ms, with granularity of 100ms. Thus the contribution of the legacy system to the total response time will be visible as discrete, constant values, independent of system load.

4.2 StoryBoard resource allocation

The storyboard specifies the number and type of requests issued in the course of a user session, and the amount of resources that are consumed at each server component.

- the total duration of all requests in one session is around ten seconds
- the number of requests per session is ten
- the proportions of total time spent in each system is around one percent in the web server, some fifteen percent in the application server, and the rest distributed among database and legacy backend servers
- about half of the requests pertain to menu navigation that consume minimal resources
- a lot of data are processed during login, reflected by high resource allocation

server	resource	trxLin	navFis	trxFis	navAcc	trxAcc	navIzv	trxIzv	navArc	trxArc	trxOut	sum	prop
web	**cpu**	10	10	10	10	10	10	10	10	10	10	**100**	**1%**
app	**cpu**	800	20	200	20	100	20	100	20	100	120	**1500**	**15%**
db	**cpu**	250	0	500	0	50	0	100	0	50	50	**1000**	**10%**
	service	500	0	1000	0	100	0	200	0	100	100	**2000**	**20%**
be	**delay**	1500	0	500	0	1000	0	1500	0	900	0	**5400**	**54%**
sum		**3060**	**30**	**2210**	**30**	**1260**	**30**	**1910**	**30**	**1160**	**280**	**10000**	**100%**
prop		**31%**	**0%**	**22%**	**0%**	**13%**	**0%**	**19%**	**0%**	**12%**	**3%**	**100%**	

Figure 4.1: StoryBoard resource allocation

4.3 Runtime variable simulation parameters

Three parameters critical to overall system behavior can be modified during simulation runtime

- the request arrival rate (in terms of new sessions per second)
- the user think time (the time interval between a response delivered by the system and the next request issued by the user)
- a disturbance factor that increases the demand of the disk and cpu resources of the database server.

Possible values of the arrival rate range from ten to two hundred percent of the default of one new session per second. The session duration and thus amount of memory needed for objects of session lifetime is directly proportional to the user think time. The disturbance factor is used to provoke different critical states in the database server. This is either a slow response time over an extended period of time or brown out of limited duration.

5 Simulation results

The following results are based on two simulation runs of 300 sec each. The service demand at each component is defined as described in Section 3.2.2. The first run demonstrates the application in standard operation at a nominal arrival rate of one new session per second, with all components parameterized such that the utilization remains below fifty percent. E.g., the web server is equipped with one cpu, the application server with four, and the database with two. Several parameters are hard coded to make explicit their effects.

The second simulation run starts out with 120 seconds standard operation, then the IO of the database is slowed down by a factor of ten for 60 seconds, and returns to normal operation for the remaining 120 seconds.

The plots in this section are generated by gnuplot from data written by the logging framework to the 'duration' and 'debug' logs. The duration log records the 'time in service' for each request, with a hierarchy that includes session, usecase, server and component. The debug log contains information about the status of all components, including number of tcp connections, number of idle connections in the database connection pool, and number of tasks on cpu.

5.1 Standard operation

Fig. 5.1a) shows the response time of each of the 316 sessions executed, and the response time of each of the ten use cases a session is comprised of. The four menu navigation only use cases (all plotted in black) involve the web and application server only and thus are very fast. The other use cases involve the database and/or back end server, with the login and izv (domestic payment) requiring the most resources in terms of IO and cpu.

Fig. 5.1b) dissects the total session response time into the time spent on each tier and server (web, application, database, legacy backend). As expected, the use case, web and application server times are almost identical, as establishing a tcp session is fast, and the cpu consumption on the web server is low. The application server response time is comprised of the time for the internal cpu consumption, the wait time for a database connection, possibly the JVM garbage collector StopTheWorld times, as well as the database and legacy backend response times. The JVM GC StopTheWorld times (five seconds hard coded) manifest as peaks in the use case response time and as gaps in the creation of database and backend requests. The database (plotted in blue) consumes cpu and disk io, thus the response times depends on the service demand and the number of requests in the system. The legacy backend (plotted in red) is setup such that the response time exactly matches the service demand and thus manifests as horizontal lines.

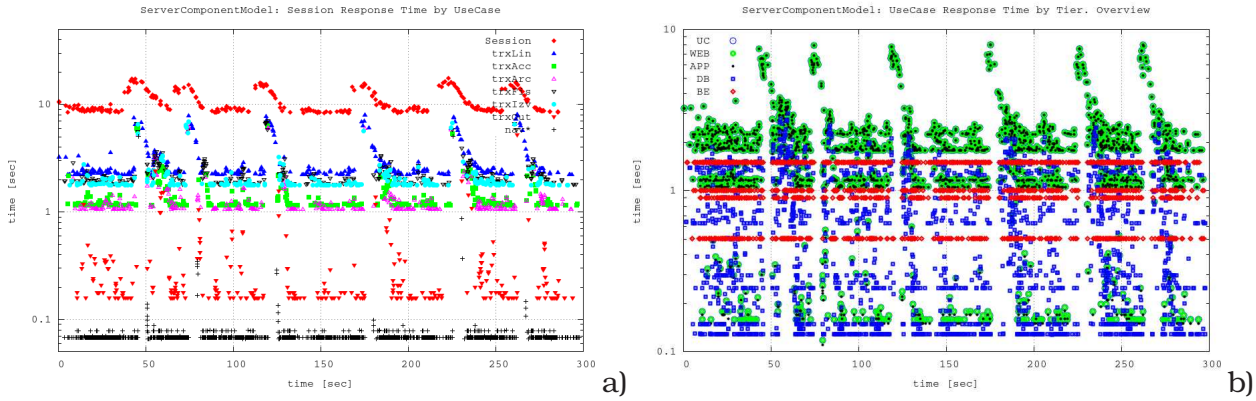

Figure 5.1: Response Time. a) by UseCase b) by Tier for UC=trxLin

5.2 Application server JVM Garbage Collector

Fig. 5.2a) demonstrates the work of the generational garbage collector. Objects are allocated and stay in the young generation (plotted in red) until this memory pool runs full. A GC in the young generation frees up the space taken up by objects no longer used, and

promotes the still referenced objects to the old generation. The StopTheWorld time for a GC in the young generation is fast (hard coded 0.01 seconds), and the young gen pool empty after GC. Objects surviving a young generation GC are accumulated and aged in the old generation (plotted in blue). Once the old generation hits a high watermark, a Full GC with a StopTheWorld event that lasts several seconds (five seconds hard coded) takes place and reclaims memory taken by objects no longer referenced.

The total time in GC StopTheWorld events (plotted in green) is just above 30 seconds after a 300 second simulation run. The caused ten percent throughput penalty is not unusual for badly tuned JVM's. Optimum values are in the range of one percent and below. The behavior in this simulation is due to the generation sizing and the amount of memory allocated by the application, both parameterized to demonstrate GC characteristics in short simulation time. About one Full GC occurs every minute here, while a very few per hour are not unusual for loaded production systems.

5.3 Application server JVM Database Connection Pool

Fig. 5.2b) gives details on the operation of the database connection pool. The maximum pool size is twenty, the connection idle timeout is set to ten seconds, the idle timeout checker runs every ten seconds, and the connection timeout is five seconds before an exception is thrown. The number of active connections is shown by blue points, the idle connections as green lines, and the sum of these in red. Whenever all connections established so far are active, new connections (plotted in black) are allocated and the total number of connections rises. This usually happens at the end of a Full GC StopTheWorld event when a burst of new db connections are required for the requests that have piled up at the JVM. Connections idle longer than the idle connection timeout are closed by the idle connection timeout checker, thereby reducing the total number of connections in the pool (red line). Fig. 5.2b) demonstrates that all parameters behave as expected, albeit actual values of e.g. idle timeout will be longer in productive systems.

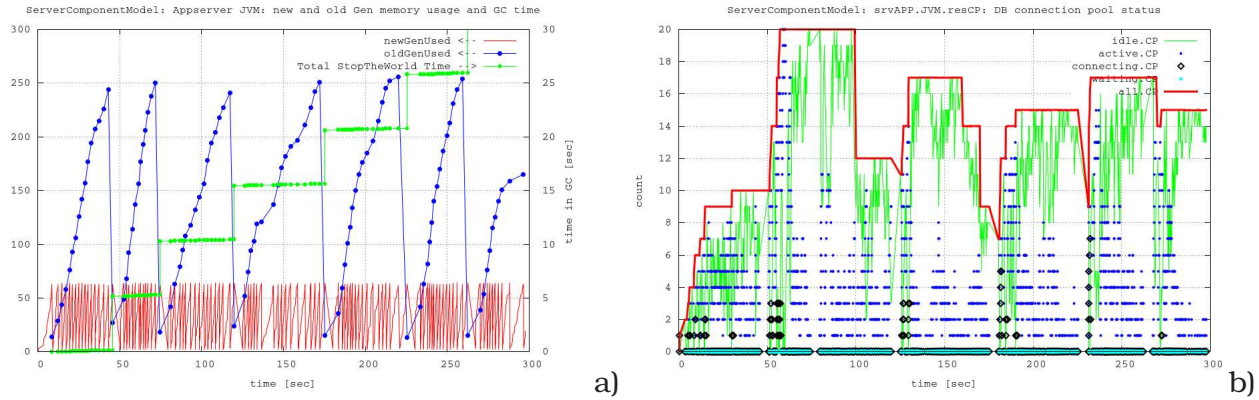

Figure 5.2: Component Detail. JVM: a) GC b) ConnectionPool

5.4 JVM Full GC StopTheWorld event

During a GC StopTheWorld event all processing in the JVM is suspended. No new requests are given a thread as unit of execution, and no response from downstream systems is handled. As seen from the web server, the application server is stalled, and requests pile up in the operating system handled tcp layer. As seen by the application server JVM, the

response time of requests pending in down stream servers increases by the duration of the GC event.

Fig. 5.3a) examines the flow of operations of one session that is interrupted by a Full GC event. The x-axis value marks the start of a event, the y-axis the duration (log scale). As expected, a use case is started (big blue circles), immediately afterwards arrives at the web server (green medium sized circles) that relays the request to the application server (big black dots). There first the cpu is consumed, then the database request is executed (blue squares), followed by the (optional) call to the legacy backend server (red diamonds).

Two events in Fig. 5.3a) are noteworthy: First, the backend call of the fifth use case is only started after the GC event (at 125 sec, and lasting for 1 sec). That is, the application server was interrupted by the GC event while executing the database call, and thus the call to the backend server was delayed until the end of the GC. In the current implementation, the database response consists of only one packet (the request agent being passed back to the application server by the Anylogic event engine). Thus, as seen from the database, the request is executed without being impacted by the GC event, because the answer has been successfully put into the queue buffering the tcp connection between application and database server.

Second, the database call of the seventh use case is started only about one second after the use case (at 127 sec), correlating to the high cpu load triggered by the burst of requests after the end of the GC event.

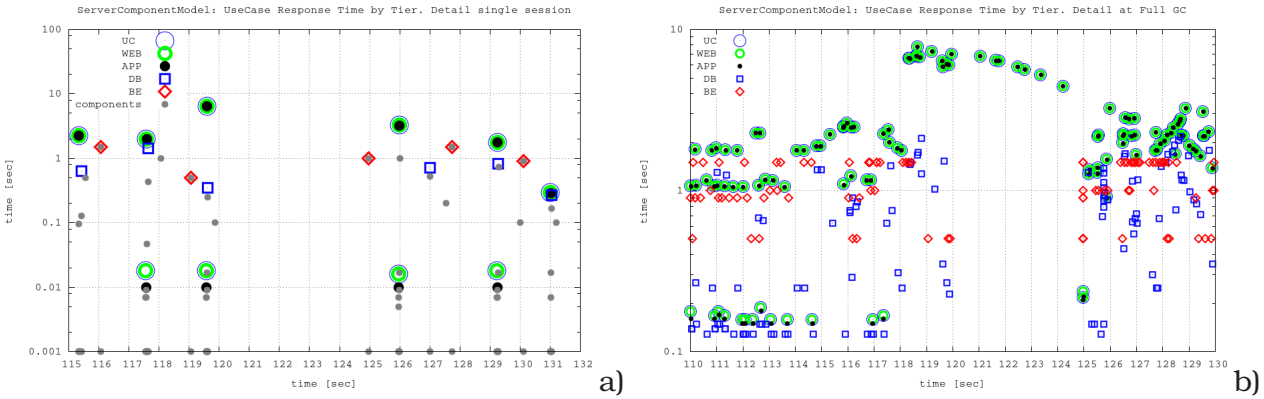

Figure 5.3: Response Time detail during FullGC: a) one complete session b) all requests

Fig. 5.3b) captures the situation during a Full GC event for all requests on all servers. The GC event sets in at t=119.962 sec, and lasts till t=124.962 sec. The response time of requests that are started just before the Full GC is increased by the full five seconds, those that are started during the StopTheWorld event are delayed by the corresponding fraction.

5.5 Database IO performance breakdown

The second simulation run examines system response to an IO problem in the database server lasting for 60 seconds starting at t=120sec. Fig. 5.4a) exemplifies response time behavior using the trxLin (login) use case. Legacy backend response times remain constant, while database response times increase from around 0.5 seconds to above five seconds. Subsequently login response time initially rises from around two seconds to about seven seconds. In the next 30-40 seconds login response time increases to a maximum of about 15 seconds, then the number of successfully completed logins drops significantly.

The root cause of this increase of login response time from 7 to 15 seconds remains to be understood.

During this period of slow IO in the database server the cpu consumption of all servers does not seem to be significantly impacted, see Fig. 5.4b). Web server cpu utilization (red line) is between 10 and 20%, application server (blue line) 10 to 40%, with peaks up to 80% after Full GC StopTheWorld events (oldGen usage see gray line). Volatility of cpu utilization on the DB server (green line) is very high and shows several periods 100% full load, often, but not always, related to a Full GC event. Fig. 5.4b) shows the number of use cases completed by second (black line with points). As expected this number drops significantly during StopTheWorld events, otherwise exhibits high volatility around its nominal mean of ten finished use cases per second (one session per second arrival rate, and ten use cases per session).

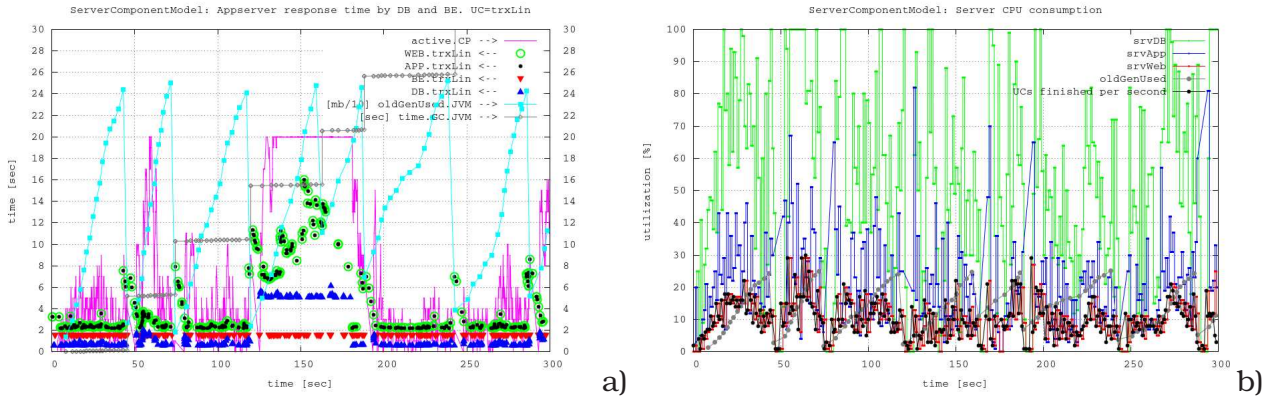

Figure 5.4: Database slow response: a) UC=trxLin Response Time by Tier b) Server cpu consumption

Fig. 5.5 gives details on the state of the JVM during the database slow IO period. Coincidentally, a Full GC event takes place at the same time the database IO performance breakdown sets in. After the StopTheWorld event is completed processing of queued requests sets in, and the number of threads (red line with dots) in the JVM rises together with the number of active connections (blue line with dots) in the database connection pool. As usual, new connections (black diamonds) are established. Because the database response time is slow, db connections are returned at a lower rate, and the number of active connections hits the high watermark after around 10-15 seconds into the IO problem event.

From this point in time, requests for a connection to the database are being queued (gray line with dots), and as expected, the number of threads (red line with dots) in the JVM increases accordingly. In the next 20 or so seconds, up to around 30 requests pile up waiting for a DB connection. Then, some 35 seconds into the DB IO anomaly, queued requests start to run into the connection pool timeout and throw a timeout exception (line with black circles). Within the next ten seconds 36 exceptions are thrown. The corresponding requests are returned to the user, and the number of threads in the JVM as well as the number of queued requests drops by roughly the same number. Towards the end of this period accidentally another Full GC is executed, adding to the complexity of events by causing a burst of some 20 new requests for a DB connection (see peaks at around t=170 sec). As the database returns to nominal performance at around t=180 sec, standard system operating conditions are restored within some 20 seconds, although another Full GC

events takes place at around t=190.

It should be noted that part of the timeout exceptions occur during a Full GC StopTheWorld event. This is an inconsistency in the current implementation of the work flow of the application server, its JVM and the DB connection pool.

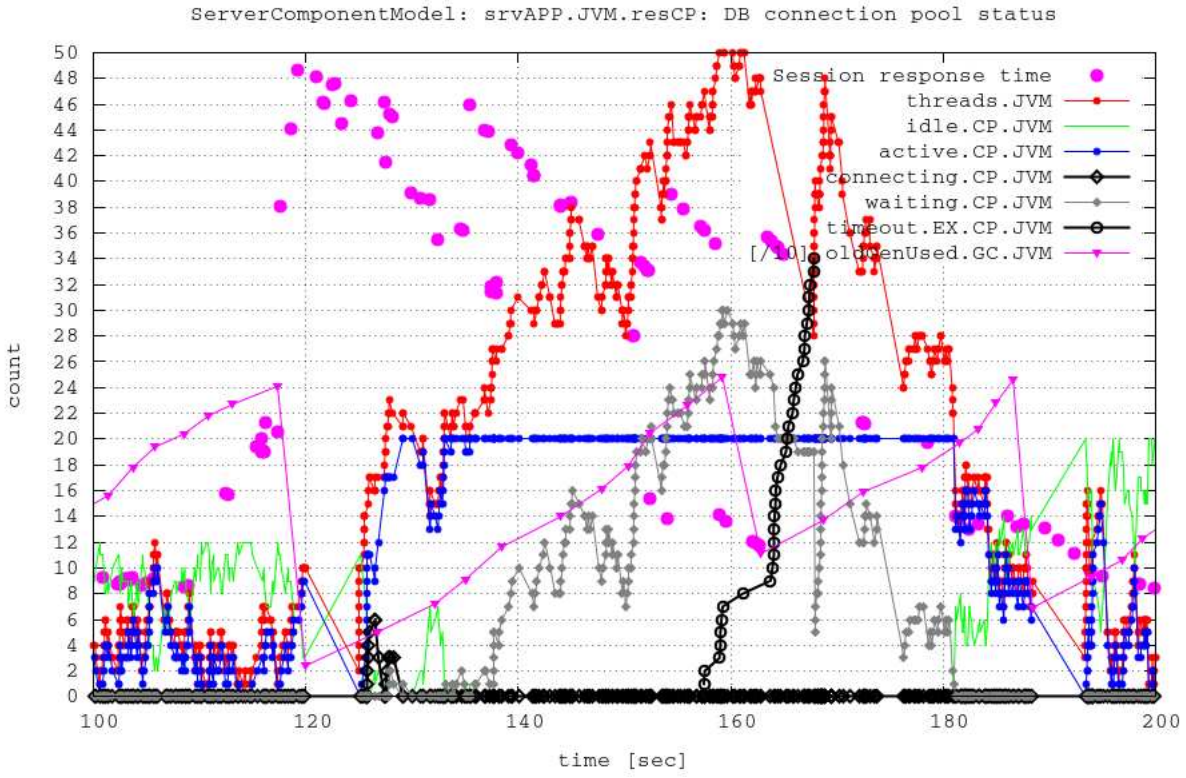

Figure 5.5: Database slow response: Appserver JVM: threads, connection pool and timeout exceptions

6 Model verification

The entities of the ServerComponentModel were designed along insight gained from application and system monitoring data acquired from the production servers of the BankAustria eBanking for CEE countries. Comparison of the simulation results reported in the previous section to system states observed in the real world system demonstrate qualitative, if not quantitative, conformance in all aspects modeled. The plots in Figs. 6.1 and 6.2 demonstrate examples of typical behavior of the application server running on a multicore cpu, with java virtual machine and database connections.

Fig. 6.1a) shows that some 500 sessions are active, with 0.2 to 0.3 new sessions per second. The number of concurrent transactions is around ten, with volatility ranging from five to twenty. Fig. 6.1b) documents the connection status between the clients and the web server, including number the of request per second, the number of bytes transferred, and the number of connections actively handling requests.

Fig. 6.2a) demonstrates activity of the GC. Memory used before a GC is plotted in blue, the memory used after a GC in red. Big jumps in memory consumption indicate Full GC events. Fig. 6.2b) shows the number of active connections to three database servers.

An intermittent event around 12h is visible in the in Figs. 6.1a) and b) as well as 6.2b). The number of client connections increases sharply, as does the number of concurrent active transactions. The root cause of the problem is an intermittent database problem, as documented by the sharp increase of database connections from around five to above thirty five.

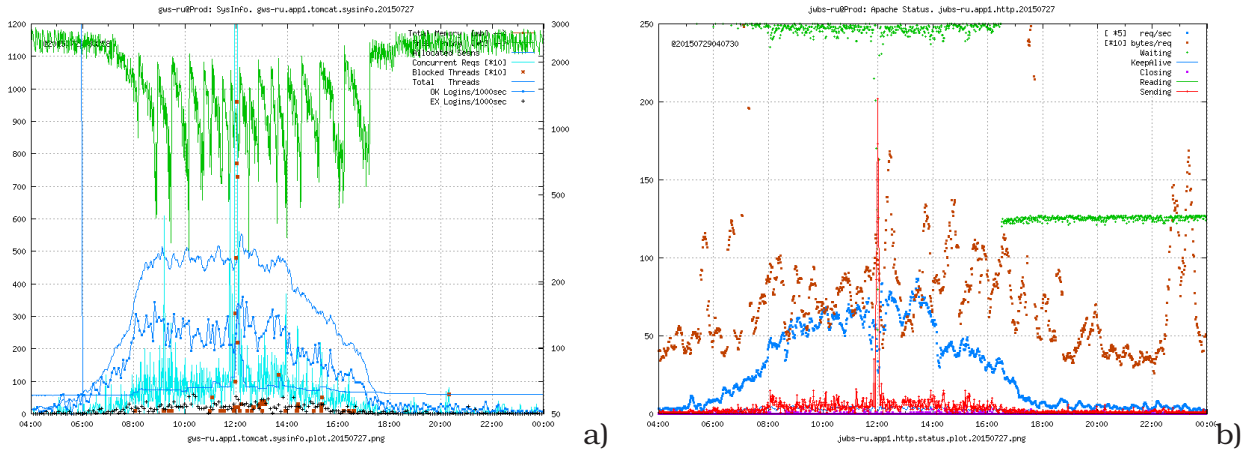

Figure 6.1: BA-CEE-eBanking. a) Number of sessions b) Apache status

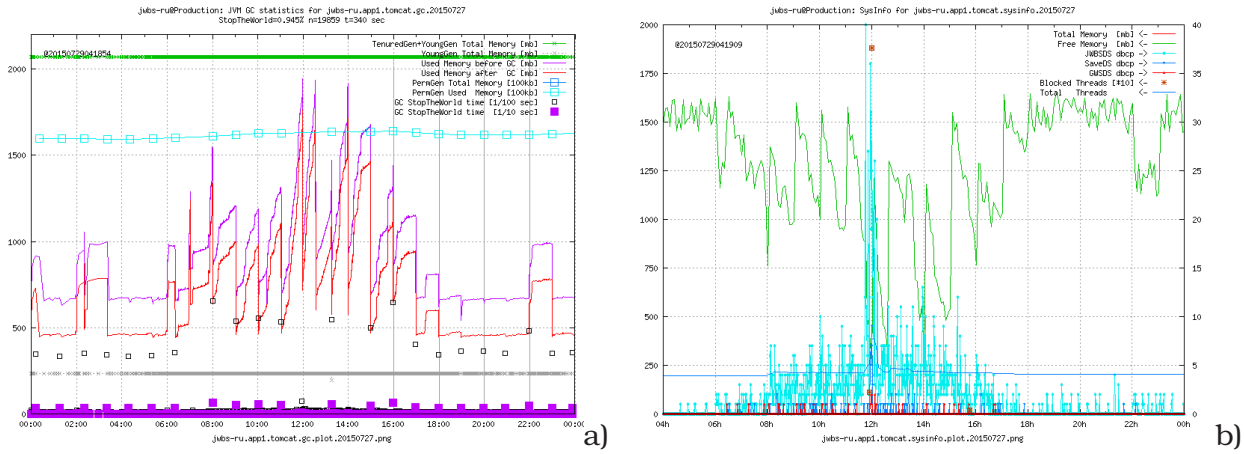

Figure 6.2: BA-CEE-eBanking. a) GC activity b) db connection pool

7 Discussion

The components implemented so far in the 'ServerComponentModel' Anylogic model allow to study critical phenomena observed in real world systems. Among these the

- request queuing during, and request burst after, a JVM GC StopTheWorld event that lasts for several seconds
- upstream queuing caused by slow response or brownout of a downstream subsystem

Comparison of system and application monitoring data of the simulation with data obtained from the BA-CEE-eBanking production system shows similar system behavior in

several respects. In particular, the JVM thread pool usage, the generational GC with its StopTheWorld events, and the database connection pool resemble major characteristics of their real world implementation. For the present simulation runs, several parameters where hard coded to bring to the fore those of particular interest. Some components need a more refined implementation to be able to study corresponding parameters of real world systems. This in particular includes the web server (e.g. apache) and the connection from the web server to the application server (e.g. mod_jk for apache and tomcat).

7.1 ServerComponentModel implementation

The basic design principle of independent request and response channels in each component is found appropriate to reproduce application servers that pass the 'token of execution' along network connections. The simulation of the allocation of a tcp connection by the OS and the allocation of a thread inside the JVM relies on the Anylogic seize, release and resource pool objects. No problems were observed with these components.

The implementation of the multicore time-slicing CPU as a task list worked on by a Anylogic clocked custom event handler is found simple and effective. Compared to queuing models based on multiple service units the algorithm much more closely resembles the actual implementation of a real time slicing OS.

The generational memory manager and garbage collectors of the JVM agent object were found suitable to demonstrate the effects of 'StopTheWorld' events. The duration of the GC events is hard coded in the current implementation, but is prepared to take the number of objects and size of the memory pool into account. Further experiments with GC durations actually observed are key to realistic estimation of effects on request burstiness with a given arrival rate.

7.1.1 DB connection pool

The pool implemented to connect the JVM to the database is a greatly simplified version of the 'dbcp' pool delivered with e.g. Tomcat 6.0. The inner workings of this pool are not easily understood and thus difficult to reproduce. Moreover the dbcp exhibits unexpected behavior when e.g. confronted with a sufficiently big burst of requests, running into an 'out of connections' state while only one connection is used to actually execute pending DB requests.

Thus only basic properties are modeled. For example, the costly setup of a new connection is implemented as a fixed delay for each specimen, while the same action is synchronized in actual code, leading to additional queuing. On the other hand, handling of situations when the pool runs out of connections is cleanly implemented, allowing for queuing of new and execution of pending requests, as opposed to anomalies seen in real dbcp systems.

As Anylogic allows integration of external libraries, one advanced option would be to directly call dbcp's own methods for allocation/deallocation of connection pool entities. With the dbcp library configured to connect to a real database, the Anylogic simulation engine could be used to generate advanced load patterns.

7.1.2 Service demand definition by StoryBoard

The definition of system load by a use case based storyboard closely resembles actual workloads on real world production systems on both user session and component ser-

vice demand level. In particular, key performance indicators like throughput in terms of number of sessions per seconds, number of requests per session, use case response time or number of concurrent active transactions can be tuned to match a wide range of load patterns.

7.2 Problems with Anylogic PLE

The most (non-productive) time consuming tasks during work with Anylogic were found to be

- verifying that the implementation of the server components functions as intended
- gaining programmatic access to Anylogic and custom objects from the many places custom methods are executed

The ServerComponentModel Anylogic DES contains custom code snippets that are distributed over many event handlers, callbacks and other extension points. With the 'PLE' edition of Anylogic it is found difficult to monitor variables other than by printf type of logging, as it is not possible to set break points or use other debugging aids.

Events that are scheduled for execution at the same point in time are executed in no guaranteed order. This can lead to unexpected results and was solved by 'dummy' delays in critical places.

Programmatic access from upper to nested object properties (and the other direction), as well as access from custom java classes to simulation entities (e.g. the current simulation time) was found badly documented, if possible at all. The current implementation calls custom code from within the simulation, passing required objects as parameters.

Several other inconveniences were found, including

- line plots have difficulty with out of x-axis order data
- command completion does not work
- property editor notoriously slow
- no integration with version control in PLE
- only ten agents possible in any one model in PLE
- no obvious path to development of custom DES agents that work akin Anylogic library objects

The last topic is found particularly annoying, as it makes building of hierarchical models difficult.

7.3 Future enhancements

The custom components presented in the 'ServerComponentModel' are just a first step on the way to a library of components for the simulation of distributed web applications. Future refinement will require several enhancements, including, but not limited to

- Construction of a hierarchical model of components as AnyLogic agents like those in Process Modeling library

- Realistic implementation of Web server
- Allow for multiple instances of application and data servers
- Advanced implementation of connection pool
- Dynamic server visits and routing based on StoryBoard
- JVM GC StopTheWorld times calculated from number of objects and size of generation
- Service and delay times drawn from suitable distribution (e.g. log normal)
- Call to actual implementation of real world components by using the Anylogic library import feature

8 Conclusion

The 'ServerComponentModel' discrete event simulation project demonstrates that multi tier web service applications can be simulated in detail by a suitably designed queuing network with custom extensions for non-queuing model akin processing. The current model consists of a web, an application, a database and a legacy backend server, but can be adapted for other server architectures.Each server is built from basic components, including tcp networking, multicore cpu and java virtual machine. The load pattern is defined in terms of a use case based story board.

Given these entities, a real world system can be modeled in terms of architecture, server components and load pattern. This allows to study key parameters on a quantitative level. While helpful for web capacity planning purposes, the strength of this simulation is its capability to scrutinize system and component behavior under transient events. The ServerComponentModel allows to reproduce such intermittent failures under laboratory conditions, study server and component response in detail, and size critical parameters for fault tolerant system behavior.

References

Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1530873.1530877.

R FrÃijhwirth and M Regler. *Monte-Carlo-Methoden: eine EinfÃijhrung*. Bibliographisches Institut, Mannheim, 1983. ISBN 978-3-411-01657-0.

Daniel A. Menasce, Virgilio A. F. Almeida, and Virgilio A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Pearson Prentice Hall, Upper Saddle River, NJ, rev. edition, September 2001. ISBN 978-0-13-065903-3.

Thomas G Robertazzi. *Computer Networks and Systems Queueing Theory and Performance Evaluation*. Springer New York, New York, NY, 2000. ISBN 978-1-4612-1164-8. URL http://dx.doi.org/10.1007/978-1-4612-1164-8.

Harmen van AS. *Modellierung and Analyse von ÃIJberlast-Abwehrmechanismen in Paketvermittlingsnetzen*. PhD thesis, Siegen, 1984.