A Distributed System Simulation Modelling Approach

M. Nikolaidou

Department of Informatics - University of Athens
Panepistimiopolis 15771, Athens, Greece
tel.: (+) 302107275614
fax: (+) 302107275214
email: mara@di.uoa.gr

D. Anagnostopoulos

Department of Geography –Harokopio University
70 El. Venizelou Str, 17671 Athens, Greece
tel.: (+) 302109549171
fax: (+) 302109514759
email: dimosthe@hua.gr

Keywords

Distributed Systems, Distributed Application Modelling, Distributed System Simulation, Performance Evaluation

Abstract

The employment of network-based technologies, such as the WWW and middleware platforms, significantly increased the complexity of distributed application, as well as the Quality of Service requirements for the underlying network. Distributed application modelling is nowadays far more demanding than network modelling, where numerous solutions are already employed in commercial tools. We introduce a simulation modelling approach for distributed systems, giving emphasis to distributed applications. The proposed scheme enables the in-depth description of application functionality, the accurate estimation of network load and the extension of existing application models to support further customisation. It supports widely employed architectural models, such as the client-server model and its variations, and is based on multi-layer decomposition. Application functionality is described using predefined operations, which can be further decomposed into simpler ones, ultimately resulting into elementary actions corresponding to primitive network operations, such as transfer and processing. Even if realisation of this scheme proves to be time demanding, individual application modelling is performed with consistency and considerably lower overhead. The distributed system simulation environment built to realise the proposed modelling scheme and a case study indicating key features of the overall approach are also presented.

1. Introduction

The outburst in network technology gave rise to different types of applications operating in a network environment. Most of them are based on multi-tiered client-server models [1], and are generally called *distributed applications*. Distributed applications extend to multiple sites and operate on multi-platform networks. Distributed applications and the network infrastructure form a *distributed system* [2]. Most commercial information systems, such as banking and flight control systems, e-mail and WWW applications, distant learning environments and workflow management systems, fall in this category. Development of middleware standards [3], such as CORBA that allows the interaction between heterogeneous, autonomous applications, and of programming languages, such as Java that provides native distributed programming support, have established a well-defined framework for distributed application development.

Simulation modelling has been widely acknowledged as an efficient technique for performance evaluation. Numerous methodological and practical approaches for distributed system simulation have appeared in the literature. In most cases [4, 5, 6], application performance exploitation is closely depended on the network infrastructure. Thus, applications running on a network environment are viewed as network traffic generators and application operation mechanisms are not emphasised. Investigation of the Quality of Service (QoS) provided by the network to determine whether application requirements are efficiently supported has also been the objective of simulation studies [7], where applications are usually represented using analytical models. In these cases, distributed application operation is not emphasised due to the significance of networking issues.

When orientation is towards evaluating an aggregate distributed architecture, system components are analytically described and component-specific models are employed. Distributed system modelling is mostly based on the client-server model. However, both client and server functionality is usually represented at an abstract layer and, due to the number and complexity of potential component combinations, behavioural characteristics of individual models are roughly modelled [8, 9]. In-depth performance evaluation approaches have also been provided, especially for customised applications, where load generation is modelled at a low

layer mostly using mathematical models [10, 11, 12], thus not promoting the reusability of simulation models. In [13] and [14], UML is used to model distributed system functionality, while mathematical modelling, specifically queuing networks, is adopted to estimate application performance. Use of UML sequence diagrams [15] facilitates the description of client-server architectures, process triggering and information exchange. However, the detailed description of process functionality is not facilitated. When examining the operation of distributed applications [6,7,9,10,12,13], object-oriented modelling techniques are usually adopted. Application operation is directly mapped at the elementary action layer as a series of discrete requests for processing, network transfer, etc., using predefined, elementary actions. We consider that such approaches lack efficiency and wide applicability, as:

1. The outcome of application decomposition is rather empirical when not supported by a consistent mechanism transforming operations into elementary actions through intermediate layers.

2. Intermediate layers are required to support application decomposition in terms of the various standards and architectural models, such multi-tiered client-server models.

3. Determining application load through an empirical analysis does not permit an accurate estimation.

4. Extendibility and applicability to support variations of the architectural models and customised implementations are - generally - not supported.

We argue that a generic modelling scheme should be established in order to facilitate the uniform representation of different types of applications (i.e. elementary, such as FTP, and complex, such as distributed databases) and the interaction between applications and the underlying network. We propose a modelling framework for distributed system entities, emphasising the in-depth description of application operation mechanisms. Network modelling is not discussed, as traditional approaches have provided effective solutions [16]. The scheme introduced promotes accuracy in distributed application description using a multi-layer *action* hierarchy. Actions indicate autonomous operations describing a specific service and are further

decomposed into simpler ones, ultimately resulting in elementary actions. The proposed elementary actions are similar to the ones described in [10], [12] and [13]. The modelling scheme supports the client-server model and its variations and can be further extended to support other architectural models. The proposed application modelling approach is independent of simulation implementation and can be easily incorporated into existing simulation environments.

Based on the proposed scheme, a simulation environment, namely Distributed System Simulator (DSS), was also constructed for the evaluation of distributed application performance. DSS enables the exploitation of various types of distributed applications, including user-defined ones, as well as of the network infrastructure. Object-oriented modelling and component preconstruction is employed. Both network and application entity models reside in model libraries. Performance issues in DSS operation were also addressed to ensure that the duration of simulation experiments remain within acceptable boundaries.

The rest of the paper is organised as followed: in Section 2, we address modelling issues, emphasising the description of distributed applications. In Section 3, the components of Distributed System Simulator are presented. Simulation model extension and validation issues are discussed in Section 4. A case study where DSS is used for performance evaluation of a distributed banking system is presented in Section 5, while conclusions reside in Section 6.

2. Distributed System Modelling

Distributed systems are modelled as a combination of two types of entities: distributed application and network infrastructure entities. Both are described in terms of their elementary components [17]. The modelling scheme introduced for the representation of typical distributed architectures is depicted in Figure 1, as a decomposition diagram. Distributed applications are described in terms of processes (clients and servers), files and user profiles. Processes and files are elemental. Files are accessed only through servers of a specific type (File Servers). User behaviour is modelled through User Profiles. The network infrastructure consists of nodes,

either processing (depicting workstations and processor pools) or relay (depicting active communication device, such as routers), storage devices and communication links. Distributed architecture modelling is based on the workstation-server and the processor-pool models, both of which are widely acceptable [2]. Client processes are executed on workstations, while server processes are executed on dedicated servers or processor pools.

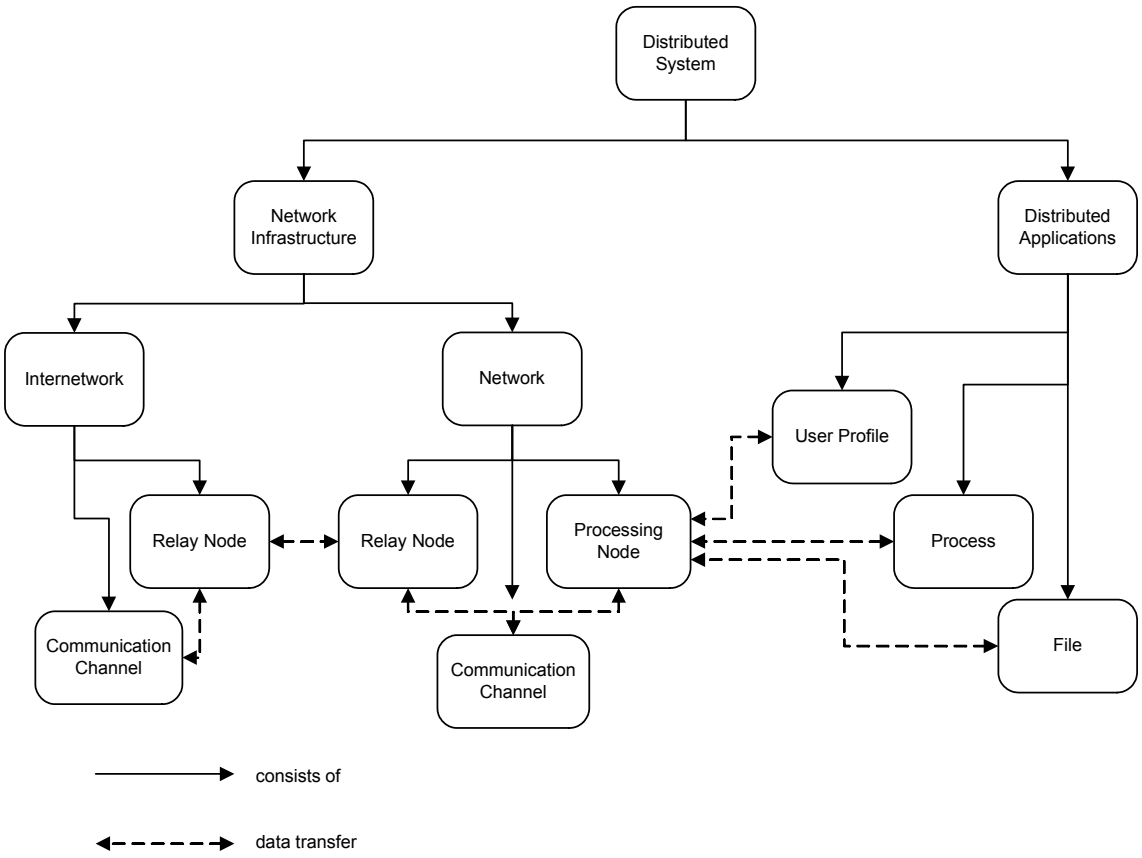

Figure 1. Distributed System Decomposition Scheme

2.1. Distributed Application Modelling

As discussed in [2], numerous architectural solutions may be employed for the design of distributed applications, regarding the functionality provided by clients and servers and the replication scheme. In most contemporary systems, distributed application operation is based on the client-server model. According to *two-*

tiered client-server model, application functionality is merely embedded in the clients, while servers deal with data manipulation and consistency issues [1]. After the explosion of the WWW, this model was no more viable, as a. functionality was embedded in Web Servers to minimise communication delay, and b. the aggregate functionality was dispatched into more than one layer with the presence of intermediate layers between clients and servers (middleware) in order to offer common services to clients. This is widely known as the *three-tiered client-server model*. In the proposed application-modelling scheme, *multi-tiered client-server model* variations are supported. Two types of processes can be defined: clients, which are invoked by users, and servers, which are invoked by other processes. Specific interfaces must be defined for each process, acting as the process activation mechanisms. The operation scenario corresponding to the invocation of each interface must also be defined, comprising the actions occurring upon process activation.

Actions are described by qualitative and quantitative parameters, such as the involved processes and the amount of data sent and received. The actions included within the operation scenario are executed sequentially, that is, each action is executed when the previous one is completed. The operation scenario also supports the requirement for parallel action execution through specifying groups of actions that obtain the same sequence number, which indicates their execution priority.

The proposed modelling scheme supports the following basic actions for application description – note that these are not the elementary actions:

- *Processing*: indicating data processing
- *Request*: indicating invocation of a server process
- *Write*: indicating data storage
- *Read*: indicating data retrieval
- *Transfer*: indicating data transfer between processes
- *Synchronise*: indicating replica synchronisation

Each process is executed on a processing node. *Processing* action indicates invocation of the corresponding node processing unit. Server processes can be invoked by other processes, both clients and servers. *Request* action indicates invocation of a server process and is characterised by the name of the server process, the invoked interface and the amount of data sent and received. It also invokes network services, as request and reply messages must be transferred between the invoking and the invoked process. DSS currently supports RPC, RMI and HTTP protocols.

Storing data is performed through *File Servers*. Two actions are provided for data storing, namely *read* and *write,* which are characterised by the file server invoked and the amount of data stored and retrieved, respectively. Temporary data can also be stored in local disks, resulting in the invocation of the corresponding node storage element. File Server process supports two interfaces, *read* and *write*, corresponding to the aforementioned actions. *Transfer* action is used to indicate data transfer between processes.

Replication of processes and data is a common practice to enhance performance in distributed applications. While process replication is easy to implement, data replication is more complex and is accomplished through allocating data replicas and defining a synchronisation policy. For the latter, we need to determine the process responsible for the synchronisation (the invoking process or a process replica), timing issues (i.e. if synchronisation is performed whenever a change occurs or periodically, at pre-specified time points) and the synchronisation algorithm.

DSS supports the definition of both process replicas, which may operate on different nodes, and data replicas, stored at different file servers. Although it does not provide constructs for the automated embedding of specific synchronisation policies in the simulation model, it enables the "low-level" description of the synchronisation policy through elementary actions. Specifically, it describes the logical connection between replicated processes and data during process definition and provides the *synchronise* action for the specification of the synchronisation policy. This action corresponds to the invocation of the *synchronise* interface, which must be supported by all process replicas. The corresponding operation scenario is user-defined. *Synchronise* action

parameters include the process replicas that must be synchronised and the amount of data transferred. User behaviour is modelled through *User Profiles*. Each profile includes user requests to the client interfaces that may be invoked by the user. Execution parameters, such as the execution probability, are also specified for each profile. User profiles are associated only with processing nodes.

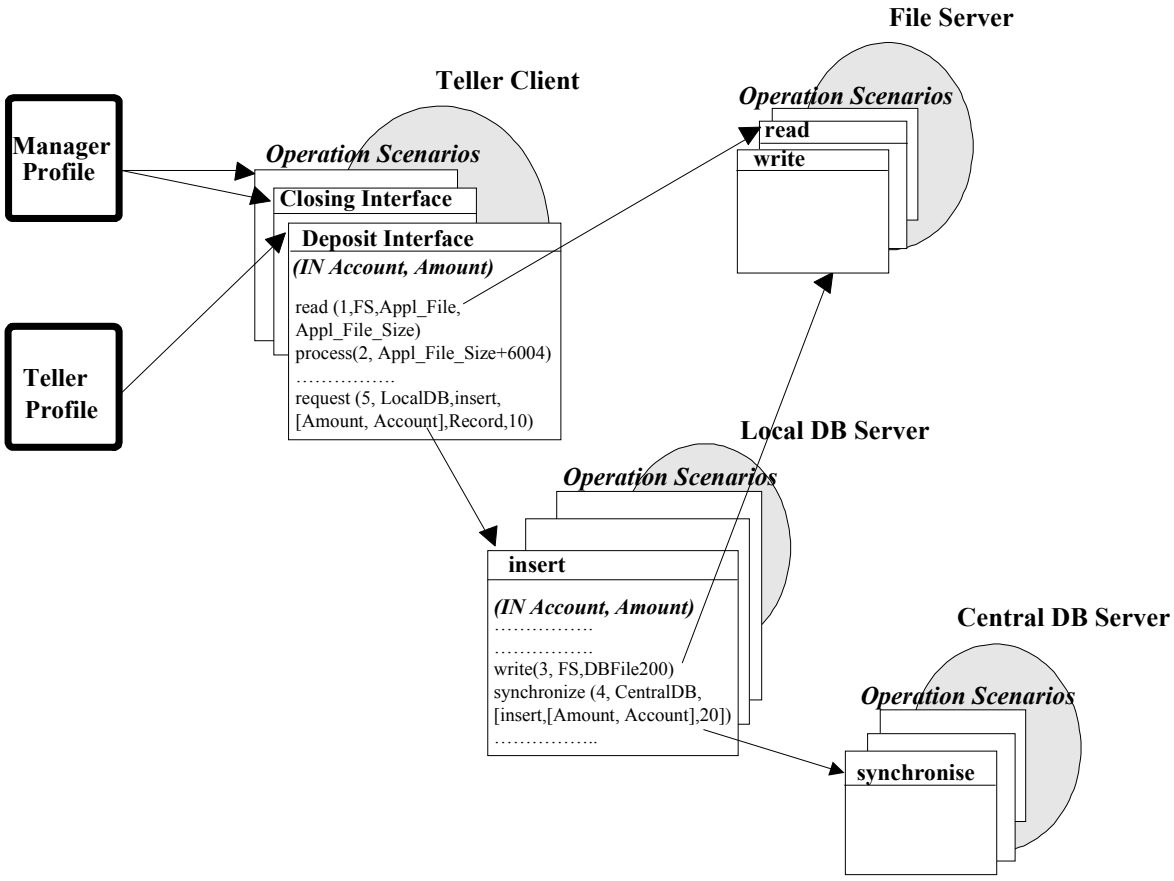

Figure 2. Distributed Application Description Example

In Figure 2, an example of the processes involved in a distributed banking system is presented. Tellers are represented through *Teller Profile,* which activates *Teller Client* by invoking the *Deposit Interface*. The teller manager, represented by *Manager Profile,* can also activate *Teller Client* by invoking *Closing Interface*. *Deposit interface* corresponds to a deposit in a client account and is invoked with two parameters, *account* and *amount*. *Deposit* operation scenario includes actions, such as *read* (indicating program download) and *request*

(indicating program activation) activating the corresponding operation scenarios of *Local Database* and *File Server*. The first parameter of each action indicates its execution sequence.

As *Local Database* is a replica of *Central Database, synchronise* action is used for data synchronisation between the local branch and the main system. When data is stored in *Local Database*, *Central Database* is also updated. As the synchronisation algorithm is application-specific, the corresponding operation scenario is user-defined. Server process activation is performed through *read, write, request* and *synchronise* actions.

Processes are modelled as composite objects with static properties, such as the process type, and dynamic properties, such as lists of interfaces and operation scenarios. Each operation scenario is a composite object, which includes a list of actions. DSS user can store specific instances of processes, such as the *DB Server*, for potential reuse. Such issues are discussed in Section 3.

The actions used to define operation scenarios are either elementary or higher-layer actions. In the latter case, they can be decomposed into elementary ones. For instance, *write* is expressed through *process* and *request* actions activating a *File Server*. All actions are ultimately expressed through the three elementary ones, namely *processing*, *network* and *diskIO*, each indicating invocation of the corresponding infrastructure component [17]. Action decomposition is accomplished through intermediate steps. The action decomposition scheme is presented in Figure 3.

It acquires the following features:

1. Maintaining consistency, as all high-layer actions are decomposed in terms of specific intermediate ones. Avoiding an empirical approach also contributes to the accurate estimation of application requirements from the network and processing resources.
2. Uniform representation of applications based on the widely used architectural models, as intermediate actions conforming to these models are preconstructed.
3. Extendibility and automation of the decomposition process are supported through a rule-based mechanism.

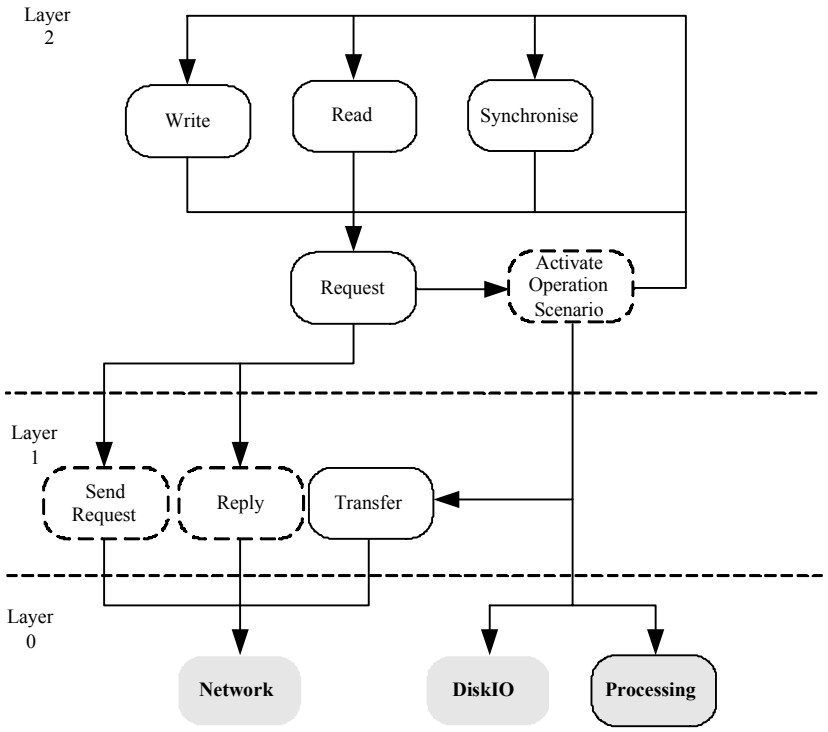

Figure 3. Action Decomposition Scheme

Dotted-border rectangles represent *intermediate* actions and black-border rectangles represent *application* actions used when defining operation scenarios. Grey rectangles represent *elementary* actions. Note that even though *processing* is an elementary action, it may be used in the definition of operation scenarios. This diagram can be further extended to include user-defined, domain-oriented actions, conforming to specific architectural models, as discussed in Section 3. However, alteration or creation of elementary actions is not allowed.

The supported actions are categorised into 3 layers. The lowest layer includes only elementary actions and the highest only actions built upon existing ones. User-defined actions are also placed at this layer. Each action can be decomposed into others of the same or lower layers. During action decomposition, all parameters of the invoked action must be determined. As an example, *request* action decomposition is presented in Figure 4. As indicated in the figure, the *request* action, e.g. the invocation of another process, is performed using a simple request-reply protocol (step1). Instead a request-reply-acknowledge protocol could be modelled by customising the *reply* action functionality to also include an acknowledgement from the calling process. In this case, the *reply* action would be decomposed to an additional *network* action (step 2). This, however, would not be visible

to the entities using the *request* action contributing to the flexibility and modularity of the action decomposition mechanism.

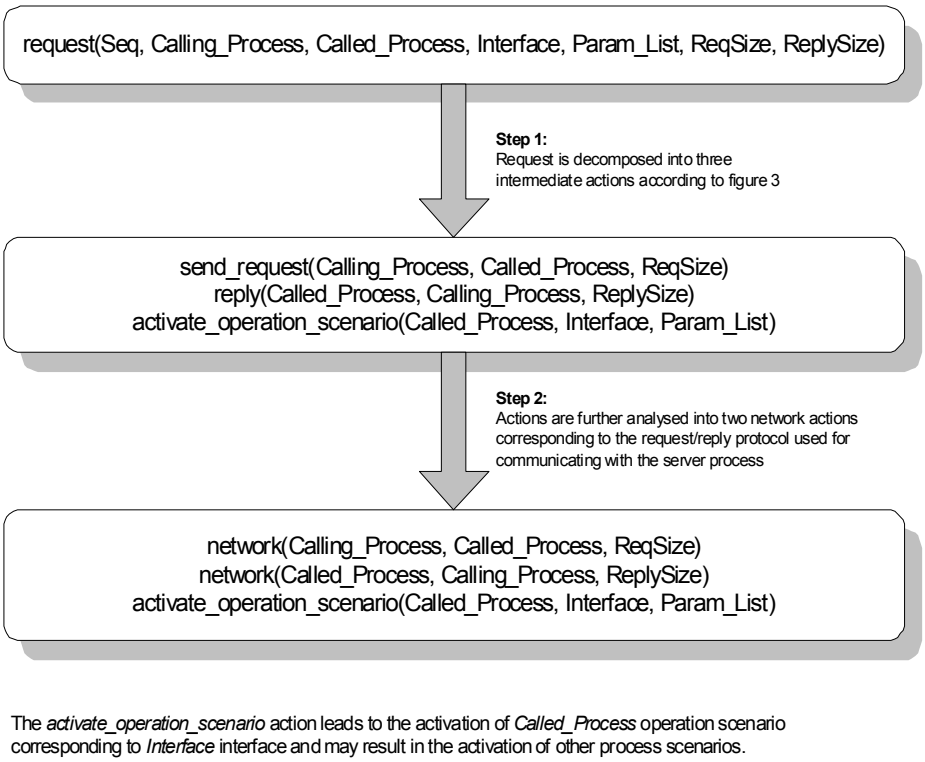

Figure 4. Request Action Decomposition

2.2. Network Modelling

Despite the fact that network infrastructure is obviously a task of significant complexity, network modelling is only briefly discussed as relevant solutions are widely employed [16]. DSS modelling scheme complies with the following requirements: uniformity in the representation of entities of the same type, such as communication protocols, compatibility with the application modelling scheme and automated construction of the network aggregate model.

Network infrastructure is considered as a collection of individual networks and internetworks that exchange messages through relay nodes (note that *networks* are here distinguished from *internetworks* and refer only to local networks). Both network types consist of nodes and communication links. Networks include processing

and relay nodes, while internetworks only include relay nodes. Protocol suites are represented through the communication element entity, which consists of two parts, the *peer communication element* and the *routing communication element*. The first corresponds to peer-to-peer protocols (OSI layers 4-7) and the latter to routing protocols (OSI layers 2 and 3). The protocols of the peer communication element have to be common for all processes of the same distributed application.

The communication element is modelled on the basis of a layered scheme, close to the OSI/RM. The layering scheme enables embedding of protocols and inter-protocol communication in the communication element model through assigning them to either one or multiple layers. It also provides the capability to either support specific layers or not and to model protocols corresponding to more than one layer (protocol suites) or more than one protocol corresponding to a single layer, as in the case of TCP and UDP. In this way, uniform modelling of protocols and protocol suites providing the same functionality (e.g. TCP/IP and ATM) is supported.

Processing node entity represents devices with processing capabilities (workstations, servers, etc). It consists of the *processing* and the *communication elements*, and an optional component, the *storage element*. When a processing node includes a storage element, a *File Server* process may operate on this node. *Relay node* entity represents switching and routing devices. Routing devices are modelled as a pair of relay nodes, each being a member of one of the interconnected networks. The addition of a new routing device to the network infrastructure thus corresponds to the addition of two relay node models in the aggregate model, while the addition of a protocol to a router or a switch device is handled through the use of the extended relay node model.

3. Simulation Environment

Distributed System Simulator (DSS) development started as a common project with the development of distributed system architecture design tool, called IDIS [18]. IDIS is a knowledge-based system that reaches an

optimum solution by composing alternative distributed architectures using exhaustive search algorithms. DSS environment was incorporated within IDIS architecture for the experimental performance evaluation of each proposed solution, resulting either the acceptance of the proposed architecture or the re-activation of IDIS search engine. As specific distributed system components may appear to be problematic, appropriate design decisions (e.g. relocation of applications) are employed prior to the initiation of the next experimental evaluation. Since the requirements for network and application modelling as well as model management increased considerably, DSS evolved into a standalone environment. DSS employs object-oriented and process-oriented simulation and its current version is implemented using MODSIM III [19] for modelling purposes and Java for all other modules. As presented in Figure 5, DSS consists of a graphical user interface (GUI), model construction and manipulation modules and a model base.

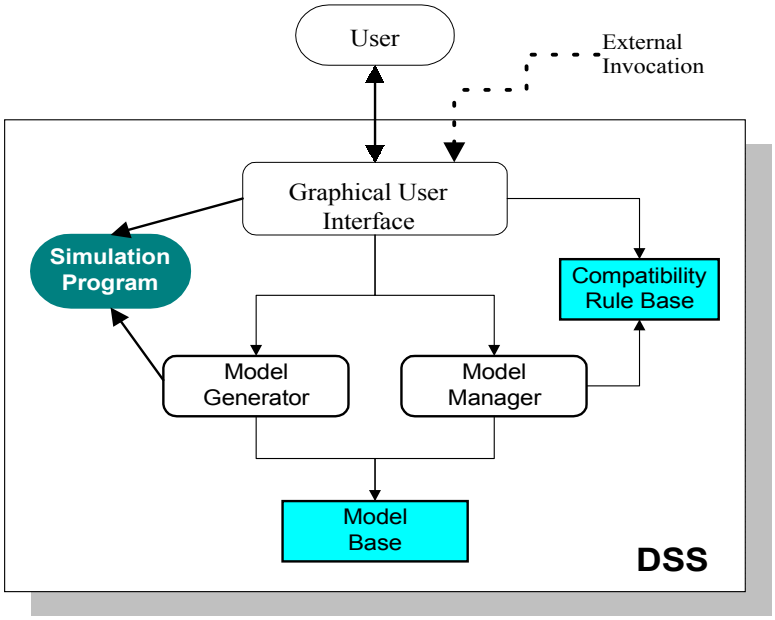

Figure 5. Distributed System Simulator Components

Model specifications, defining the system under study, and experimentation parameters are provided through *GUI*. *Model Generator* constructs the simulation program, using component models residing in *Model Base*. Models, either atomic or composite, are implemented as objects and are organised in object hierarchies. As completeness and validity of specifications must be pre-ensured, this is accomplished by the *Compatibility Rule*

Base, which includes a representation of all models residing in *Model Base* and a set of compatibility rules. *Model Manager* is invoked during the model extension process.

Line connections indicate module invocation and data access. When experiments are completed, results are subjected to output analysis with the following objectives: a. determining whether distributed applications operate efficiently, and b. determining whether network infrastructure supports the requirements imposed by distributed applications. When co-operating with an architecture design tool, simulation output analysis either approves the proposed solution or suggests that alternative architectures should be examined.

4. Model Extension and Validation

Object-oriented simulation modelling was employed for implementation purposes, enabling an almost natural representation of multi-entity systems and an in-depth description of individual entities. In simulation modelling, modularity often results in a hierarchical structure, according to which components are coupled together to form larger models [20]. When referring both to elementary (e.g. process) and composite entities (e.g. network node), hierarchical layering enables the construction of more complex models through extending the behaviour of existing objects and ensures that models of a single entity, organised in a single class hierarchy, are accessed through a common interface, using polymorphism [20]. We use preconstructed models corresponding to the potential distributed system components, including composite entities. Implementation of this scheme proves to be notably time- consuming when not supported by automated generation and manipulation capabilities, such as the ones provided by DSS. However, it promotes model availability and reduces the time required when composing customised models.

Extending distributed application constructs is a strong requirement for the modelling scheme, noted as a pitfall of current simulation tools. The proposed scheme facilitates extending application component (e.g. action) functionality, which is required for the description of custom applications, as well as for storing specific component instances (e.g. *DB server*) for possible reuse. Similar capabilities are provided for network

components, such as network protocols. Model extension focuses on application modelling to deal with the varied functionality of emerging applications. Model extension is performed through the invocation of *Model Manager* and *Compatibility Rule Base*. Processes, actions and communication protocols are the most common entities for which new model components need to be provided. *Model Base* is extended using hierarchical layering, ensuring that models of a single entity are accessed through a common interface [21]. The object-oriented framework described in [21] for computer network modelling facilitates the definition of composite network entity models (e.g. Ethernet Network Node) as ancestors of abstract ones (e.g. Network Node). A similar approach was adopted for extending DSS object structures. Using the example presented in paragraph 2.1, a new *insert* action model is constructed as a descendant of the abstract *application action* model and a *DB process* model as a descendant of *BE process* model (Figure 6).

To ensure the validity of the modelling scheme, specific restrictions apply when extending object hierarchies. User-defined action models are either of *intermediate* or *application* type. Existing actions cannot be altered. When creating a process model, interfaces and operation scenarios must be fully defined. Although a new operation scenario (e.g. *insert*) can be stored within a new process model (e.g. *DB process*), the *operation scenario* model cannot be extended, as the addition of new scenarios not belonging to a specific process is not supported. While describing an application, the user may temporarily store an existing operation scenario, as well as other entity instances, within *Compatibility Rule Base*.

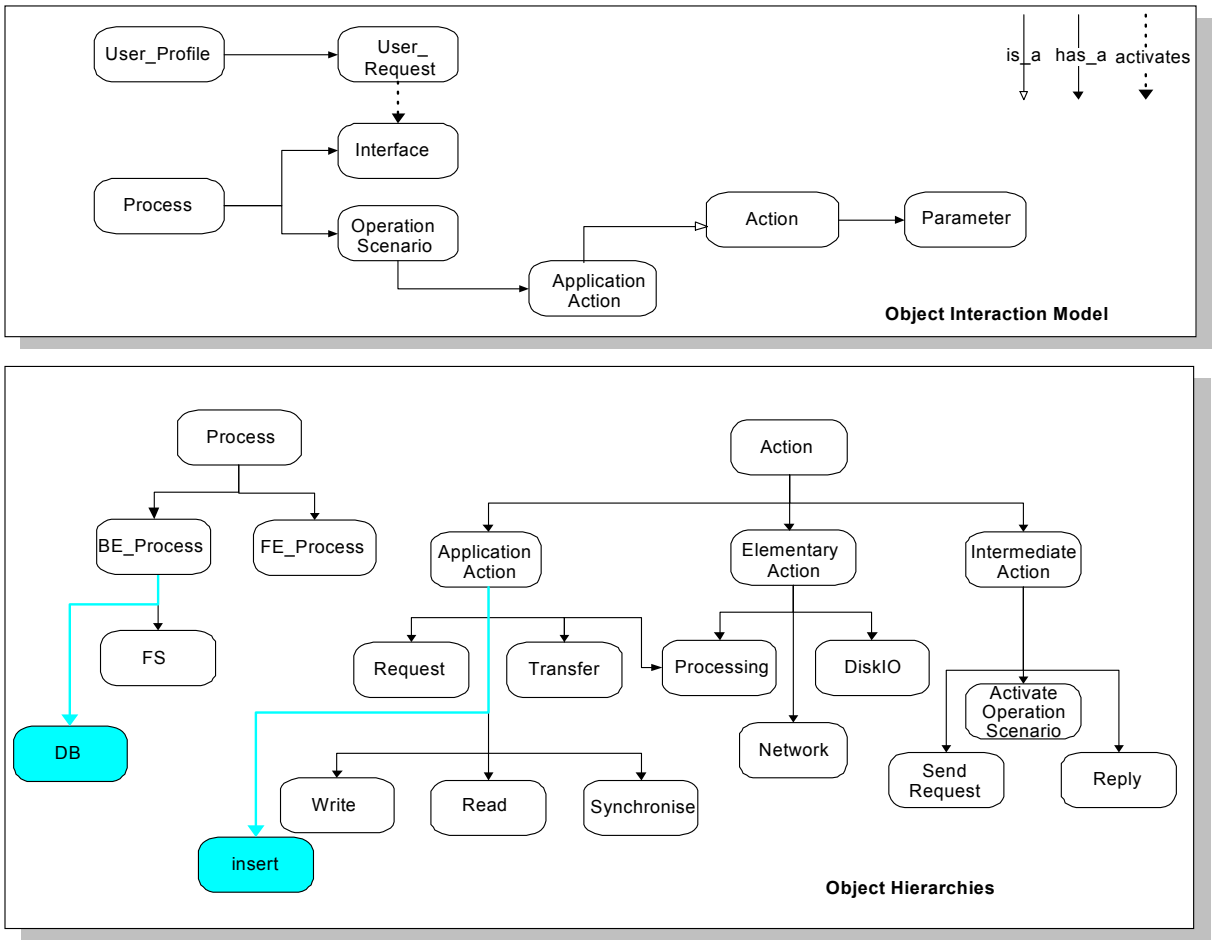

Figure 6. Distributed Application Modelling Scheme

When defining a new action, the user declares its parameters and actions, while GUI ensures that all actions are properly invoked. To give an example, the code fragment generated when constructing *write* action is presented in Figure 7, where *write* is constructed as a descendant of *application action,* resulting in the activation of a *request* action. Its additional parameters are stored as object properties, while the refinement of specific properties as the *CalledProcess* parameters is also feasible. Only the *init* method needs to be modified. User-defined actions are added in *Model Base* in a similar way. The *init* method is explicitly constructed for all user-defined actions, as they support different input parameters included in the *Param* input list, corresponding to the different descriptions stored in the *consist_of* property. As only the method implementation part is modified in the action hierarchy, polymorphism is ensured. As *activate* method is identical throughout the overall action hierarchy, there is no need to override it when defining new actions.

Code generation is performed by *Model Manager*, which establishes the coupling relation between these components. The extension process comprises the following steps:

1. Ensuring validity and compatibility with existing models.
2. Inserting component models in the *Model Base.*
3. Updating *Compatibility Rule Base*.

```
{Object Definition}
ApplicationActionObj=
OBJECT(ActionObj)
     CalledProcess:ProcessObj;
     CallingProcess:ProcessObj;
     Number_Consist_Of:INTEGER;
     Consist_Of:ARRAY_OF_ActionObj_TYPE;
OVERRIDE
     ASK METHOD Init
         (IN Param:ARRAY_OF_STRING_TYPE);
     WAITFOR METHOD Activate;
END OBJECT;

RequestObj=
OBJECT(ApplicationActionObj)
     Seq:INTEGER;
     Interface:InterfaceObj;
     Int_Par_List:ARRAY_OF_STRING_TYPE;
     ReqSize:INTEGER;
     ReplySize:INTEGER;
OVERRIDE
     ASK METHOD Init
         (IN Param:ARRAY_OF_STRING_TYPE);
END OBJECT;

WriteObj=
OBJECT(ApplicationActionObj)
     File:FileObj;
     DataSize:INTEGER;
OVERRIDE
     ASK METHOD Init
        (IN Param:ARRAY_OF_STRING_TYPE);
END OBJECT;
```

```
{Object Implementation}
OBJECT ApplicationActionObj;
…
WAITFOR METHOD Activate;
BEGIN
     FOR i:=1 TO Number_Consist_Of
          WAIT FOR Consist_OF[i] TO Activate;
          END WAIT;
     END FOR;
END METHOD;
END OBJECT;

OBJECT(RequestObj);
…
END OBJECT;

OBJECT WriteObj;
ASK METHOD Init(IN Param:ARRAY_OF_STRING_TYPE);
VAR a_RequestObj:RequestObj;
    Interf_Param:ARRAY_OF_STRING_TYPE;
BEGIN
 Seq:=STRTOINT(Param[1]);
 CalledProcess:=STRTO_ProcessObj_PTR(Param[2]);
 CallingProcess:= STRTO_ProcessObj_PTR(Param[3]);
 File:=STRTO_FileObj_PTR(Param[4]);
 DataSize:=STRTOINT(Param[5]);

 NEW(Interface);
 Int_Par_List[1]:=File_PTR_TO_STR(File);
 Int_Par_List[2]:=INTTOSTR(DataSize);
 ASK Interface TO Init("write",Int_Par_List);
 ReqSize:=DataSize+100;
 ReplySize:=100;
/* fill consist_of list */
 Number_Consist_Of:=1;
 NEW(Consist_Of,1..Number_Consist_Of);
 NEW(a_RequestObj);
 Consist_Of[1]:=a_RequestObj;
 NEW(Interf_Param,1..6);
 Interf_Param[1]:=STRTOINT(Seq);
 Interf_Param[2]:=Process_PTR_TO_STR(CalledProcess);
 Interf_Param[3]:=Process_PTR_TO_STR(CallingProcess);
 Interf_Param[4]:=Interf_PTR_TO_STR(Interface);
 Interf_Param[5]:=STRTOINT(ReqSize);
 Interf_Param[6]:=STRTOINT(ReplySize);
 ASK Consist_Of[1] To Init(Interf_Param);
END METHOD;
END OBJECT;
```

Figure 7. *Write* Action Code Generation

The overall process is depicted in Figure 8. Model validity can only be supported when input specifications are thoroughly examined. Validation is carried out through rule-based mechanisms. When *Model Library* is extended, the *Compatibility Rule Base* is updated with the additional model structures and their relations with the existing models.

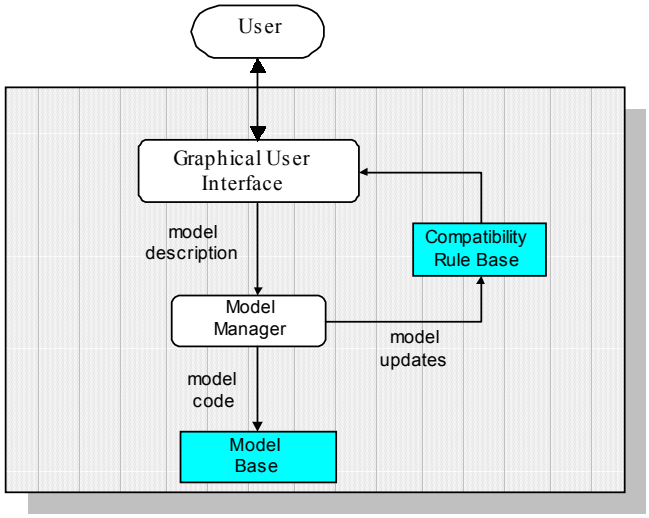

Figure 8. Model Extension Process

5. Case Study

Distributed System Simulator was used for evaluating the performance of a distributed banking system. Except from headquarters, the bank maintains 64 branches. The banking system supports 24 discrete transactions, which are mostly initiated by tellers, and are grouped in four categories. The average amount of transactions/day/branch is 500, while the maximum amount of transactions in specific branches is over 1000. The required response time is 15-20 sec for all transactions. Network infrastructure could be modelled and evaluated using various commercial simulation tools. However, due to the increased number of customised applications, each one involving a relatively high number of actions, and the need to accurately estimate the network load generated, direct mapping of application description into low-level primitives was not feasible.

The banking system architecture is based on a three-tiered client-server model. A central database is installed in headquarters, where all transactions are executed, while transaction logs are maintained in local databases at the branches. The central database supports 33 stored procedures corresponding to the different execution steps of the 24 transactions. Transactions are co-ordinated by a transaction monitoring system, which is also installed in headquarters. Digital RDB database management system and ACMS transaction monitoring system are used. The overall network is TCP/IP based. Light client applications are running on user workstations. Client data are stored locally, at the branch file server. When a transaction is executed, the corresponding forms are invoked, which have an average size of 3K. ACMS is invoked up to four times for the execution of the corresponding stored procedure. Before completing each transaction, a log is stored in the local database.

The following server processes were modelled: *File Server* at headquarters and local branches, *CentralDB, LocalDB* and *ACMS*. Since *LocalDB* represents logging, only a simple *insert* interface had to be implemented for recording the log. *CentralDB* is accessed through the 33 stored procedures, which are implemented and stored in the database. For each stored procedure, a single interface had to be implemented. As system performance was mainly determined by the interaction of the different system modules, not by the internal database mechanisms, we decided to establish a common representation for all stored procedures. A new action called *call_stored_procedure_step* was created and inserted in the action hierarchy with the following parameters: *preprocessing*, *data_accessed* and *postprocessing*. *Data_accessed* parameter indicates the amount of data accessed at each step, while *preprocessing* and *postprocessing* indicate the amount of data that need to be processed before and after accessing the data base, expressed as a fraction of the accessed data size. Using this action, the description of stored procedures was significantly simplified. Each stored procedure consists of one to five steps. The *call_stored_procedure_step* action is implemented as an interface of the *CentralDB* process in a way similar to read/write and includes the activation of *processing*, *read* and *write* actions. ACMS is modelled as a server process providing the interface *call_ACMS (procedure, inputdata, outputdata, processing)*, which activates the corresponding stored procedure.

Client applications involve the invocation and processing of forms, the activation of stored procedures through ACMS and log recording. Log recording is represented through properly invoking the *insert* interface of LocalDB, while stored procedure activation is accomplished through the invocation of the *call_ACMS* interface of ACMS. *Form_access (FS, form_name, processing)* was added in the action hierarchy to depict accessing, activating and processing of a form. Using combinations of the three aforementioned actions, it was possible to describe applications in a simplified and uniform manner. Applications were categorised in four groups, each controlled by a different type of user, as indicated in Figure 9. As a single user does not execute applications of the same group simultaneously, we modelled each group as a client process supporting one interface for each specific application. Users are modelled as profiles initiating the corresponding client application.

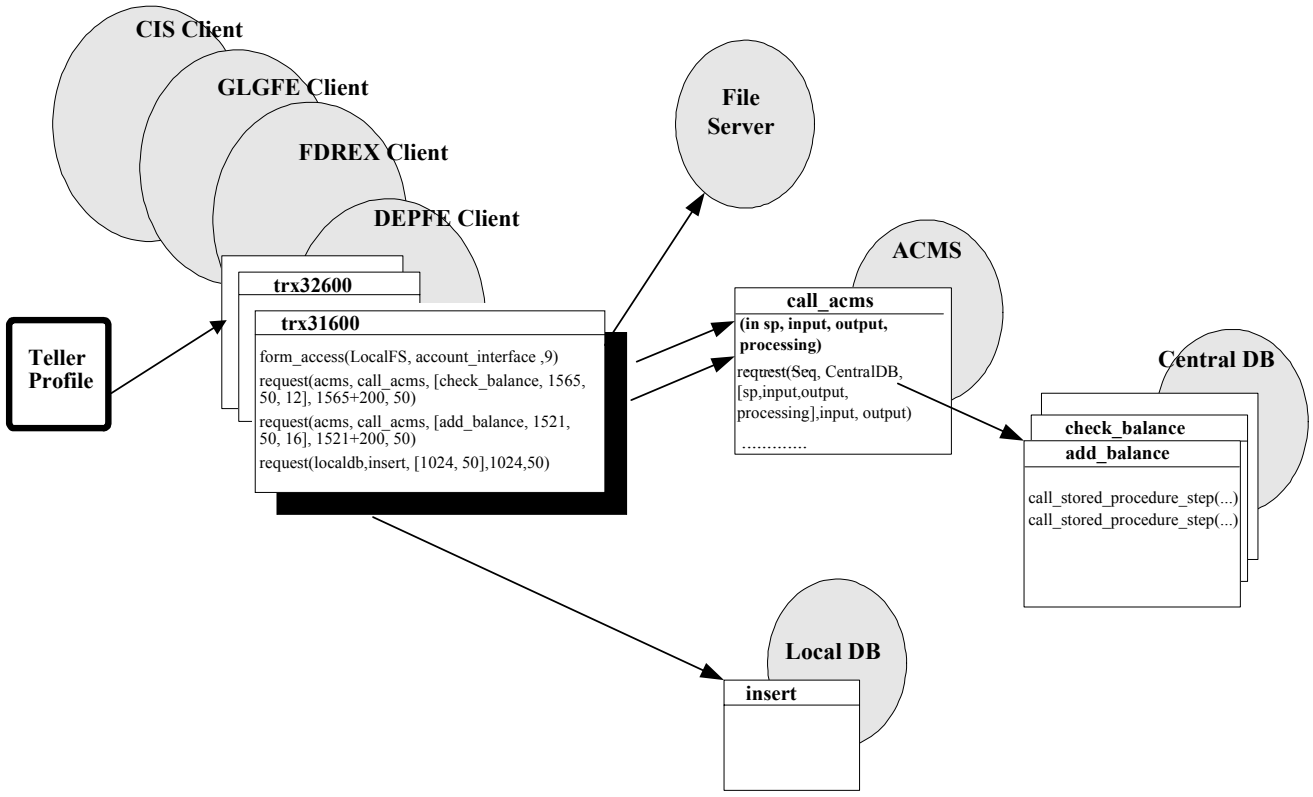

Figure 9. Banking System Transaction Modelling

The representation of transaction *trx31600* corresponding to *Cash Deposit* using the aforementioned user-defined actions is depicted in Figure 9. The steps included in this transaction correspond to the access and activation of the appropriate form, the activation of the Database Server through the ACMS interface and the

update of the local database. The representation of the Cash Deposit transaction results in the invocation of 256 elementary actions used for the description of all involved processes. Thus, it would be difficult to represent it directly using elementary actions. Although the description of all supported transaction results to the invocation of a large amount of elementary actions, varying from 196 to 512, no more than 10 application actions are used for the representation of client and server processes.

The capability to extend action hierarchy was important to ensure the detailed application description. If only predefined actions could be used, the same description would have to be repeatedly given for all transactions. The modelling scheme also facilitated application description at the level of abstraction required by different groups of users. While the system was under deployment, DSS contributed to determining potential weak points and estimating the response time of client transactions. As the main activity of all transactions relates to the invocation of the central database through ACMS, special attention was given to the system performance at headquarters. DSS indicated two drawbacks: first, the processing power of the hardware supporting the Central Database was not adequate to execute client transactions within the predefined response time. After thorough data analysis, the bank was forced to upgrade the hardware platform. Second, load estimation indicated that the throughput of specific leased lines, interconnecting branches with headquarters, should be increased.

6. Conclusions

Exploring the behaviour of distributed systems is not a trivial task due to the complexity encountered in both network and application modelling. Emphasising the description of distributed applications was the objective of the modelling scheme introduced. Application modelling extends to the operation and interaction mechanisms and conforms to the various forms of the client-server model. As distributed system architectures are configurable, the modelling scheme introduced is generic enough to allow the description of diverse applications, while different levels of detail are also supported. As distributed system description results in

complex models, considerable effort was put in constructing and organising the preconstructed DS component models to ensure their efficient manipulation.

The proposed application modelling approach can be easily incorporated into existing simulation environments to facilitate the more accurate application description. However, in order to accommodate the customisation and extension of process and action models, automated code generation capabilities should be supported and this feature is rarely provided by existing network simulation tools. The proposed distributed system modelling scheme is independent of simulation implementation and can be ported to any simulation implementation environment accommodating object-oriented models.

References

[1] J. Shedletsky, J. Rofrano, "Application Reference Designs for Distributed Systems", IBM System Journal, vol. 32, no 4, IBM Corp., 1993.

[2] G.F. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems - Concepts and Design, Third Edition, Addison Wesley Publishing Company, 2000.

[3] D. Serain, Middleware, Springer-Verlag London, Great Britain, 1999.

[4] B. Cahoon, K.S. McKinley, Z. Lu, "Evaluating the Performance of Distributed Architectures for Information Retrieval Using a Variety of Workloads", ACM Transactions on Information Systems, vol. 18, no 1, ACM Computer Press, 2000.

[5] Liu Zhen, Nicolas Niclausse, César Jalpa-Villanueva, "Traffic model and performance evaluation of Web servers", Performance Evaluation, vol. 46, no 2-3, Elsevier Press, 2001.

[6] S. Dumas, G. Gardarin, "A Workbench for predicting the performances of distributed object architectures", in Proceedings of WSC98, SCS, 1998.

[7] V. D. Khoroshevsky, "Modelling of Large-scale Distributed Computer Systems", in Proceedings of IMACS World Congress, IMACS, 1999,

[8] Das Olivia, C. Murray Woodside, "Evaluating layered distributed software systems with fault-tolerant features", Performance Evaluation, vol. 45, no 1, Elsevier Press, 2001.

[9] M. Matsushita, M. Ashita, et. al., "Distributed Process Management System based on Object-Centred Process Modeling", Lecture Notes on Computer Science 0302-9743, No 1368, Springer Verlag, 1998.

[10] E. Ginters, Y. Merkuryev, A. Spungis, "Simulation of Client-Server Distributed Data Processing Systems", in Proceedings of ESM'96, SCS, 1996.

[11] J. Cruz, K. Park, "Towards performance-driven system support for distributed computing in clustered environments", Journal of Parallel and Distributed Computing, vol. 59, no 2, Springer Verlag, 1999.

[12] S. Ramesh, H.G. Perros, "A multi-layer client-server queuing network model with non-hierarchical synchronous and asynchronous messages", Performance Evaluation, vol. 45, no 4, Elsevier Press, 2001.

[13] R. Mirandola, V. Cortellessa, "UML Based Performance Modeling in Distributed Systems", in Proceedings of UML2000, Lecture Notes in Computer Science 1939, Springer-Verlag, 2000.

[14] P. Kaehkipuro, "UML-Based Performance Modeling Framework for Component-Based Distributed Systems", in Proceedings of Performance Engineering 2001, Lecture Notes in Computer Science 2047, Springer-Verlag, 2001.

[15] Object Management Group (OMG), The UML Reference Manual, http:\www.omg.org\uml

[16] A.M.Law and M. G. McComas, "Simulation Software of Communications Networks: The State of the Art", IEEE Communications Magazine, vol. 4, no 3, IEEE Computer Press, 1994.

[17] J. Kramer, "Configuration Programming – A Framework for the Development of Distributed Systems", in Proceedings of IEEE International Conference on Computer Systems and Software Engineering, IEEE Computer Press, 1990.

[18] M. Nikolaidou, D. Lelis, et. al., "A Discipline Approach towards the Design of Distributed Systems", IEE Distributed System Engineering Journal, vol. 2, no 2, IOP Press, 1995.

[19] CACI Products Company, MODSIM III The Language of Object-Oriented Programming - Reference Manual, San Diego, 1999.

[20] B.P. Zeigler, “Object-Oriented Simulation With Hierarchical, Modular Models”, copyright by Author, 1995 (originally published by Academic Press, 1990).

[21] D. Anagnostopoulos, M. Nikolaidou, “An Object-Oriented Modelling Approach for Dynamic Computer Network Simulation”, International Journal of Modelling and Simulation, vol. 21, no 4, ACTA Press, 2001.