An Object-Oriented Modeling Scheme for Distributed Applications

D. Anagnostopoulos
Department of Geography
Harokopion University of Athens
El. Venizelou Str, 17671
Athens, Greece
Tel.: (+) 301 – 9549171
Fax: (+) 301 – 7275214
Email: dimosthe@hua.gr

M. Nikolaidou
Department of Informatics
University of Athens
Panepistimiopolis, 15784 70
Athens, Greece
Tel.: (+) 301 – 7275614
Fax: (+) 301 – 7275214
Email: mara@di.uoa.gr

Abstract

A modeling approach is here introduced for distributed applications. During the last years computer networks have dominated the world, forcing the development of applications operating in a network environment. Since new technologies, as WWW, middleware and co-operative software emerged, distributed applications functionality became rather complex and the requirements from the underlying network increased considerably.

Distributed applications usually consist of interacting services provided in a multi-level hierarchy. In order to effectively evaluate their performance through simulation, we introduce a multi-layer object-oriented modeling scheme that facilitates the in-depth and detailed distributed application description and supports most popular architectural models, as the client/server model and its variations. Application functionality is described using predefined operations, which can be further decomposed into simpler ones through a multi-layer hierarchy resulting into elementary actions that indicate primitive network operations, such as transfer or processing. Important features of the modeling scheme are extendibility and wide applicability.

The simulation environment build according to this modeling scheme is also presented, along with an example indicating key features and sample results.

1. Introduction

Simulation modeling is widely adopted in the computer network domain for performance evaluation purposes. During the last decade, numerous simulation tools were constructed, aiming at analyzing the behavior of complex, user-defined network environments ([1], [2]). Application performance exploitation is thus closely depended on the network infrastructure. In most cases, applications running on a network environment are viewed as generators of network traffic, while application operation mechanisms are often overlooked.

The outburst in network technology forced the development of new types of applications, such as distributed information and control systems, e-mail and WWW applications, distant learning environments and workflow management systems. Most are based on the client/server model and its variations, such as the two-tier and three-tier models [3], and are generally called *distributed applications*. Distributed applications extend to multiple sites and operate on multi-platform networks. As distributed applications become more complex and new services are emerging, the detailed description of operation mechanisms is more significant, considering the fact that network delays are often negligible. Thus, even though distributed applications depend on the supporting network, detailed modeling of application operation mechanisms is a prerequisite for their in-depth performance evaluation.

In current research, a number of cases with a different orientation can be referenced. Simulation modeling of customized applications is usually performed analytically using mathematical models (i.e. the corresponding functions/distributions) to represent network load generation ([5], [6]). In other cases, the QoS provided by the network to support specific application requirements is exploited. When performance evaluation is oriented towards issues as the above, it is performed using modeling solutions that are restrained to these specific objectives, without emphasizing the application operation mechanisms.

Application operations are examined in approaches, such as the ones presented in [2], [7] and [8], where object-oriented modeling is widely adopted. Application operation is expressed at the primitive action layer, using a series of discrete requests for processing, network transfer, etc., in terms of predefined, primitive actions. This, however, cannot be effective when application decomposition is not supported through a mechanism that transforms operations into primitive actions through intermediate ones, which conform to the various architectural models (e.g. the client/server model) and standards. Decomposition is thus accomplished in an "empirical" manner. When determining the effect of applications without analyzing the operation mechanisms, an accurate estimation of application load is not feasible. Extendibility and wide applicability, to support variations of the architectural models as well as customized implementations, are also not supported. Establishing a generic modeling scheme is thus required to facilitate the representation of different types of applications, i.e. primitive (e.g. FTP) and complex (e.g. distributed databases), according to common modeling principles, as well as the interaction between applications and the underlying network.

The modeling scheme introduced in this paper facilitates accuracy in distributed application description y using a multi-layer *action* hierarchy. Actions indicate autonomous operations describing a specific service and can be decomposed into simpler ones, resulting in elementary actions similar to those described in [2] and [7]. The modeling framework supports the client/server model and its variations and can be further extended to support other architectural models. Main features of the modeling scheme are modularity, extendibility and wide applicability.

To evaluate distributed application performance, a simulation environment, namely Distributed System Simulator (DSS), was constructed. DSS enables the exploitation of various types of distributed applications, including user-defined ones, as well as the exploitation of the network infrastructure through its graphical components. Object-oriented modeling and component preconstruction is employed.

In the following sections we present Distributed System Simulator components and emphasize the modeling scheme introduced for distributed application description and model extension and validation issues, crucial for our approach. An example using DSS to evaluate the performance of a distributed banking system is also presented.

2. Simulation Environment

Distributed System Simulator was initially developed as part of a distributed architecture design environment, called IDIS ([9]). Since requirements for network and application modeling, experimentation and model management increased considerably, DSS evolved into a standalone environment. DSS is based on object-oriented and process-oriented simulation and its current version is implemented using MODSIM III ([10]) for model construction and Java for all other modules.

DSS is modular, as presented in figure 1, and consists of a graphical user interface (GUI), which co-operates with individual modules for simulation program construction and model manipulation and a model base.

Model and experimentation specifications are provided through *GUI*. *Model Generator* constructs the simulation program, using component models that reside in *Model Base*.

Completeness and validity of specifications must be pre-ensured, and this is accomplished through the *Compatibility Rule Base*, which includes a representation of all models residing in *Model Base* and compatibility rules. *Model Manager* is invoked during the model extension process.

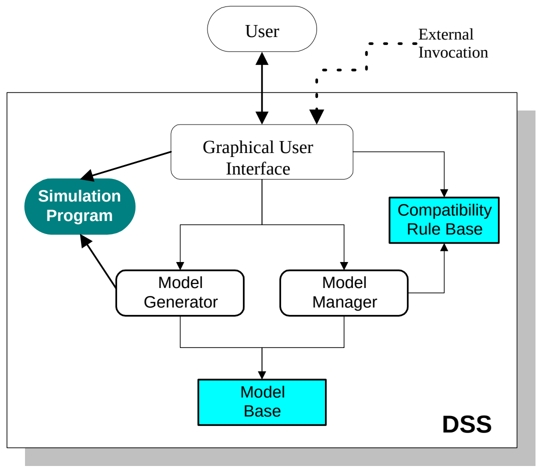

Figure 1. Distributed System Simulator components

Line connections in figure 1 indicate module invocation and data access. When experiments are completed, results are subjected to output analysis in order to a) determine whether distributed applications operate efficiently and b) determine the ability of the network infrastructure to support the requirements imposed by distributed applications.

3. Model Definition

Object-oriented modeling provides an almost natural representation of multi-entity systems, as distributed systems, since modularity enables the in-depth description of all their components. In simulation modeling, modularity often results in a hierarchical structure, according to which components are coupled together to form larger models ([11]). Distributed systems are modeled as a combination of two types of entities: distributed application and network infrastructure entities. Both are described in terms of their elementary components ([12]). Network model composition is a complex task due to the increased number of network technologies and standards. Since modeling solutions for communication network architectures are already employed by commercial simulation environments, as Comnet and OpNet ([2], [1]), this topic is not further discussed in this paper.

In most contemporary systems, distributed application operation is based on the client-server model. When designing distributed applications, as indicated in [3], there are many architectural solutions that may be employed regarding the functionality provided by clients and servers and the replication scheme. There are two variations of the client/server model that are widely adopted: the *two-tier* and the *three-tier* models. According to the two-tier model, application functionality is merely embedded in the clients, while servers deal with data manipulation and consistency issues ([3]). After the explosion of the Internet and the WWW, this model was no more viable, since functionality was embedded in Web Servers to minimize communication delay. Furthermore, the aggregate functionality was dispatched into more than one layer, with the use of intermediate ones (middleware) between clients and servers, thus offering common services to clients. This is the three-tier model.

Within the DSS framework, a basic scheme was introduced to facilitate the description of applications, regardless of their complexity and architecture, supporting both the above architectures. Two types of processes can be defined: clients, which are invoked by users, and servers, which are invoked by other processes. The specific interfaces, acting as process activation mechanisms must be defined for each process, along with the operation scenario that corresponds to the invocation of each interface. Each operation scenario comprises the actions that occur upon process activation.

Actions are described by qualitative and quantitative parameters, e.g. the processes being involved and the amount of data sent and received. In most cases, the operation scenario is executed sequentially (each action is performed when the previous one has completed). However, there are cases where actions must be performed concurrently. This is supported through specifying groups of actions that have the same sequence number.

The basic actions used for application description are the following:

• *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
• *Synchronize*:	indicating replica synchronization

Each process is executed on a processing node and, thus, *Processing* action indicates invocation of the processing unit of the corresponding node.

According to the client-server model, communication between processes is performed through exchanging messages. Server processes can be invoked by other processes, clients or servers. *Request* action indicates invocation of a server process and is characterized by the name of the server process, the invoked interface and the amount of data sent and received. It also implies activation of the network, since the request and the reply 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*. There are two actions available for data storing, which are *read* and *write,* which are characterized by the amount of data stored and retrieved, respectively, and the file server invoked. Temporary data can also be stored in the local disk, resulting in the invocation of the corresponding node storage element. File Server process supports two interfaces, namely *read* and *write*, corresponding to the aforementioned actions.

Transfer action is used to indicate data exchange between processes.

Replication of processes and data is a common practice in distributed applications in order to enhance performance. While process replication is easy to implement, replication of data is accomplished through defining process replicas, for handling data, and a synchronization policy. In the latter case, there are many issues to be resolved, such as determining the process responsible for the synchronization (the invoking process or a process replica), when synchronization is performed (i.e. whenever a change is made or periodically at pre-specified time points) and the synchronization algorithm.

Definition of process replicas operating on different nodes and data replicas stored at different file serves is supported. DSS does not support specific synchronization policies. It allows the description of the logical connection between replicated processes and data during process definition and provides the *synchronize* action to facilitate the specification of synchronization policy. This action corresponds to the invocation of the *synchronize* interface, which must be supported by all process replicas. The corresponding operation scenario has to be defined by the human operator. *Synchronize* action parameters include the process replicas that must be synchronized and the amount of data transferred.

User behavior is modeled through *User Profiles*. Each profile includes user requests to the client interfaces that may be invoked by the user. For each profile, execution parameters, such as the execution probability, are also specified. User profiles are associated only with processing nodes.

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*, which indicates the size of the corresponding data. *Deposit* operation scenario includes actions, such as *read* (indicating program download) and *request* (indicating the actual deposit) that activate the corresponding operation scenarios of *Local Database* and *File Server*. The first parameter of each action indicates the execution sequence.

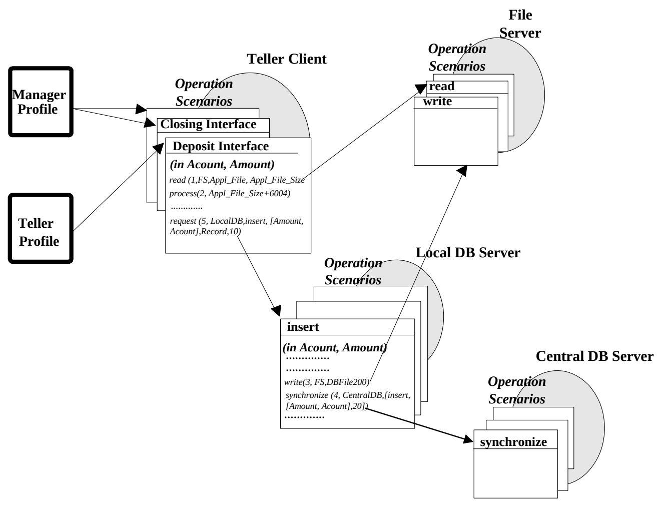

Figure 2. Distributed Application Description example

Local Database is a replica of *Central Database*, thus *synchronize* action is used to indicate the need for data synchronization between the local branch and the main system. After data is stored in *Local Database*, *Central Database* is also updated. Since the synchronization algorithm is application-specific, the corresponding operation scenario is defined by the user. Server process activation is performed through *read, write, request* and *synchronize* actions and is indicated by dotted lines.

Processes are composite objects acquiring static (e.g. process type) and dynamic properties as lists of objects (e.g. interfaces and operation scenarios). Each operation scenario is also a composite object, including a list of actions. DSS operator can store specific instances of processes, as the *DB Server* in the previous example, for future reuse in other experiments. This is accomplished by properly extending object hierarchies, as discussed in section 4.

The actions used to define operation scenarios are either elementary or of higher layer. In the latter case, they can be decomposed into elementary ones. While *processing* is an elementary action, *write* is expressed through simpler ones, i.e. a *process* and a *request* sent to a *File Server*. All actions can be ultimately expressed through the three elementary ones, *processing*, *network* and *diskIO*, each indicating invocation of the corresponding infrastructure component ([13]). Action decomposition is not performed in a single step. Intermediate stages are introduced to simplify the overall process and maintain relative data. The action decomposition scheme is presented in figure 3.

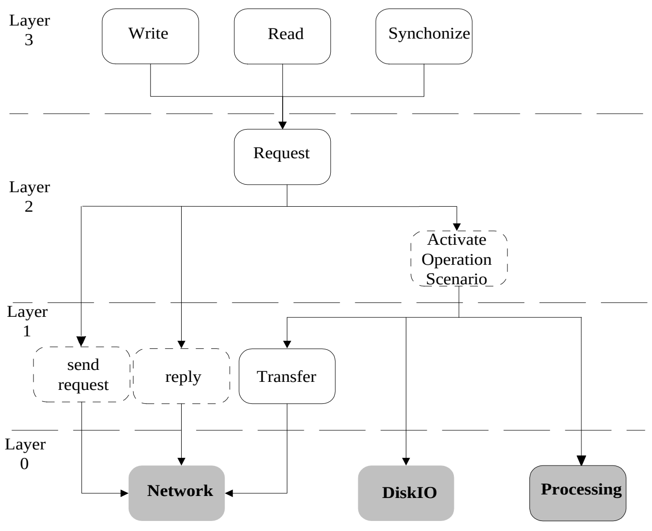

Figure 3. Action composition scheme

Dotted rectangles represent *intermediate* actions, while gray rectangles represent *elementary* ones. Finally, rectangles with black border represent *application* actions used when defining operation scenarios. Note that even though *processing* action is an elementary one, it is used in the definition of operation scenarios. This diagram can be further extended to include user-defined, domain-oriented actions, as discussed in section 4, which conform to specific architectural models. However, although alteration or creation of elementary actions is not allowed.

The supported actions are categorized into 4 layers. The lowest layer includes only elementary actions, while the highest one includes 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 the lower layer. Actions support specific parameters and are derived as ancestors of the *action class*. During action decomposition, all parameters of the invoked action must be defined.

4. Model Extension and Validation

The proposed object-oriented modeling scheme facilitates the extension of application component (e.g. actions) functionality, in order to describe custom applications, and also storing of specific application component (e.g. *DB server*) instances for future reuse. The same capabilities are provided for network components, as network protocols. Model extension is performed by the human operator through the invocation of *Model Manager* and *Compatibility Rule Base*.

Extending distributed application modeling constructs is a strong requirement for the modeling scheme and is noted as a pitfall of current simulation tools. Processes, actions and communication protocols are the most common entities, new models of which (i.e. components) need to be provided. Models are created as ancestors of existing, abstract entity models. A concise modeling framework for extending object structures has been described in ([13]). Considering the example of section 3.1, a new *insert* action model would be constructed as a direct descendant of the abstract *application action* model, while a *DB process* model would be constructed as a descendant of *BE process* model (figure 4).

Extending object hierarchies is performed according to specific restrictions that ensure the validity of the modeling scheme. User-defined action models are either of *intermediate* or *application* type. Existing actions can not be altered, while new actions must always be described in terms of existing ones. When creating a new process model, the interfaces and operation scenarios are usually 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 can not be extended, since the addition of new operation scenarios not belonging to a specific process is not supported. While describing an application, the user can *copy* a specific operation scenario, since the description of specific instances is temporarily stored within *Compatibility Rule Base*.

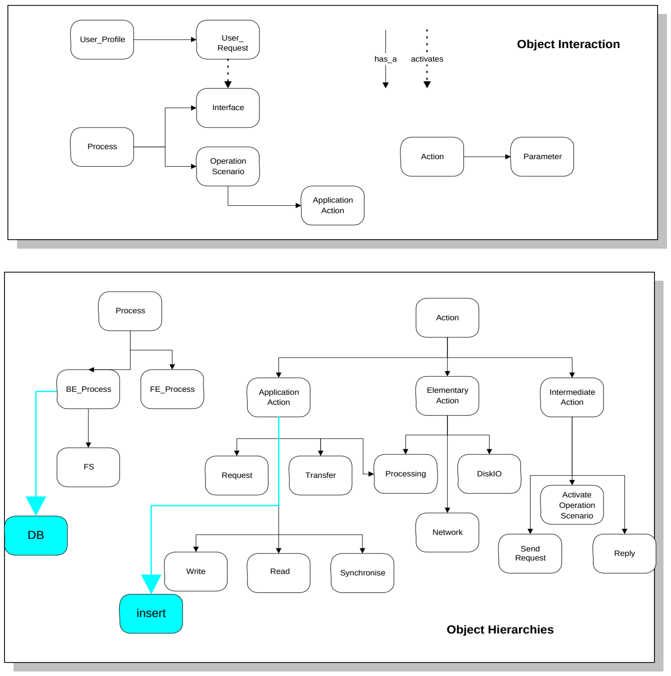

Figure 4. Distributed Application modeling scheme

At the implementation level, the *Model Base* is extended using object inheritance. When extending composite entities (e.g. process), hierarchical layering enables the construction of complex models through extending the behavior of existing objects and ensures that models of a single entity, organized in a single class hierarchy, are accessed through a common interface, using polymorphism ([14]).

When defining a new action, the user declares its parameters and the actions used to describe it, while GUI ensures that all actions are properly invoked (their parameters are properly filled). The code fragment generated when constructing the *write* action is presented in figure 5. As indicated in this figure, *write* is constructed as a descendant of *application action* and results in the activation of a *request* action. Its parameters are stored as object properties, and only the *init* method needs to be modified.

User-defined actions are added in *Model Base* in a similar manner. The *init* method is explicitly created for all user-defined actions, since they support different input parameters and correspond to different descriptions stored in the *consist_of* property. Other methods, such as *activate*, maintain the same syntax to facilitate polymorphism and remain the same for all actions.

```
{Object Definitions}

ApplicationActionObj=
Object(ActionObj)
    CalledProcess:ProcessObj;
    CallingProcess:ProcessObj;
    Number_consist_of:Integer;
    Consist_of:
        ARRAY[1..Number_consist_of] of
ActionObj;
Override
    ASK METHOD Init( );
Override
    WAIT FOR METHOD Activate;
END OBJECT;

RequestObj=
Object(ApplicationActionObj)
    Seq:Integer;
    Interface:InterfaceObj;
    Int_Par_List:List of STRING;
    ReqSize: INTEGER;
    ReplySize: INTEGER;
Override
    ASK METHOD Init(IN Seq:INTEGER;
        IN Calling_Procedure: ProcessObj;
        IN Called_Process: BE_ProcessObj;
        IN Interface: InterfaceObj;
        IN Int_Par_List: List of STRING;
        IN ReqSize: INTEGER;
        IN ReplySize: INTEGER);

END OBJECT;

WriteObj=
Object(ApplicationActionObj)
    File:FileObj;
    Data_size: INTEGER;
Override
    Called_Process: FSObj;
Override
    ASK METHOD Init(IN Seq:INTEGER;
        IN Calling_Procedure:ProcessObj;
        IN Called_Process:FSObj;
        IN File: FileObj;
        IN Data_size: INTEGER;);

END OBJECT;
```

```
{Objects Implementation}

OBJECT ApplicationActionObj;

…
WAIT FOR METHOD Activate;
BEGIN
    FOR ALL a IN consist_of
        WAIT FOR a TO Activate;
END METHOD;

END OBJECT;


OBJECT WriteObj;

ASK METHOD Init(IN seq:INTEGER;
        IN calling_procedure:ProcessObj;
        IN called_Process:FSObj;
        IN file: FileObj;
        IN data_size: INTEGER;);


BEGIN

Seq:=seq;
Calling_Procedure:=calling_procedure;
Called_Process:=called_process;
File:= file;
Data_size:=data_size

/* initiate inherited properties */

Interface:=new(InterfaceObj);
Int_Par_List:=[File,Data_size];
ASK Interface TO Init("write", Int_Par_List);
ReqSize:= Data_size+100;
ReplySize:= 100;

/* fill consist_of list */

Number_Consist_of=1;
Consist_of[1]:=new(requestObj);
ASK Consist_of[1] To Init
    (Seq,Calling_Procedure, Called_Process,
    Interface, Int_Par_List,
ReqSize,ReplySize);
END METHOD;

END OBJECT;
```

Figure 5. Code generation when constructing *Write* action

Extension of process models is accomplished based on the same guidelines.

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

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

The overall process is depicted in figure 6.

Model extension and simulation program generation capabilities can only be supported when input specifications are thoroughly examined to ensure model validity. Validation is not trivial, even though models are preconstructed, since models are coupled to form larger ones and are extended to conform to customized implementations. Validation is carried out through rule-based mechanisms during system specification.

Graphical visualization of the existing model hierarchies supports the addition of customized models. *Compatibility Rule Base* is invoked to ensure that consistency is maintained. When *Model Library* is extended, the *Compatibility Rule Base* is updated with the additional model structure, its relations with the existing models and rules concerning its initialization.

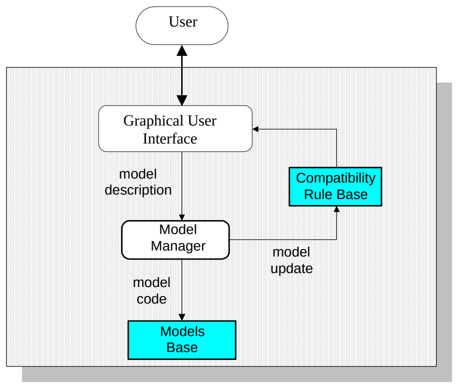

Figure 6. Model Extension process

5. Practical Example of DSS Usage

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, grouped in four categories, which are mainly initiated by tellers. The average transaction number per day in a branch is 500, while the maximum transaction number in central branches is over 1000. The required response time is 15-20 sec for all transactions.

Network infrastructure could be modeled and evaluated using various commercial simulation tools. However, application description was not feasible on the basis of primitive modeling constructs and required an intermediate-layer analysis that gradually extends to the primitive action layer, so that a credible application model would emerge.

The system architecture is based on the three-tier 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 coordinated by a transaction monitoring system, also installed in headquarters. Digital’s RDB database management system and ACMS transaction monitoring system are used. The overall network is a TCP/IP one.

Light client applications are running on user workstations. Client data are stored locally in the branch file server. When a transaction is executed, the corresponding forms are invoked, each having an average size of 3K. ACMS is invoked up to four times for the execution of the corresponding stored procedure. Before finishing each transaction, a log is stored in the local database.

Server processes that were modeled are the following: *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. Since system performance was mainly determined by the interaction of the different system modules and 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. Action parameters are *preprocessing*, *data_accessed* and *postprocessing*. *Data_accessed* parameter indicates the amount of data accessed at each step, while *preprocessing* and *postprocessing* parameters indicate the amount of data to be processed before and after access, as a fraction of the accessed data. 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 modeled as a server process providing the interface *call_ACMS (procedure, inputdata, outputdata, processing)*, which initiates the activation of 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 depicted 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 these three actions, it was possible to describe all applications in a simplified, common way.

Applications were categorized in four groups, each controlled by a different type of user. Applications of the same group are not executed simultaneously by the same user. This led us to depict each group as a client process supporting one interface for each specific application. Users are depicted as profiles initiating the corresponding client application.

Except from building a credible distributed application model, DSS also enabled the estimation of the exact amount of data processed and transferred within and between branches. Modeling advantages that were offered are also simplification in client application description, extendibility and flexibility during process description.

The capability to extend the action hierarchy was important to ensure detailed application description. If only predefined actions could be used, the same description would have to be repeatedly given for all transactions, e.g. form activation. Furthermore, this scheme 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 ensuring the response time of client transactions. Since 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. This proved to be accurate, forcing the bank to upgrade the hardware platform. Second, for the interconnection of branches with headquarters, load estimation indicated that the throughput of specific leased lines should be increased. On the other hand, Ethernet (10BaseT) proved to be efficient for branches, since the average throughput was very low (less than 0.05 Mbps).

6. Conclusions

Exploring the behavior of distributed systems while emphasizing the description of distributed applications was the objective of the modeling scheme introduced. Application modeling extends to the operation and interaction mechanisms and conforms to the various forms of the client/server model. Since distributed system architectures are configurable, considerable effort was put in constructing and organizing the preconstructed component models to ensure their efficient manipulation.

The modeling scheme provides guidelines for modeling the essential, both primitive and composite, distributed system components. The capability to reuse models when implementing customized component models was crucial for the description of different architectures, despite the complicated nature of this process. An important feature of this research is that the modeling guidelines can also be used in other modeling and simulation studies.

References

[1] Mil3 Inc, *Opnet Modeler Modeling Manual*, Washington, 1997

[2] CACI Products, *COMNET III Reference Manual*, San Diego, 1997

[3] J. Shedletsky, and J. Rofrano, “Application Reference Designs for Distributed Systems”, *IBM System Journal*, Vol. 32, No 4, 1993.

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

[5] R. L. Bagrodia, and C. Shen, “MIDAS: Integrated Design and Simulation of Distributed Systems”, *IEEE Transactions on Software Engineering*, Vol. 17, No. 10, October 1991.

[6] V. D. Khoroshevsky, “Modelling of Large-scale Distributed Computer Systems”, *Proceedings of IMACS World Congress 1999*, Conf. 15, Vol. 6, 1999.

[7] S. R. Ramesh, “An Object-Oriented Modeling Framework for an Enterprise-Wide Distributed Computer System”, *Proceeding of the Americas Conference on Information Systems*, Association for Information Systems, August 1998.

[8] 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.

[9] M. Nikolaidou, D. Lelis, et. al., “A Discipline Approach towards the Design of Distributed Systems”, *IEE Distributed System Engineering Journal*, Vol. 2, No 2, 1995.

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

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

[12] J. Kramer, “Configuration Programming – A Framework for the Development of Distributed Systems”, *Proceedings of IEEE International Conference on Computer Systems and Software Engineering*, Israel, 1990.

[13] D. Anagnostopoulos, and M. Nikolaidou, “A Conceptual Methodology for Conducting Faster-than-Real-Time Experiments”, *SCS Transactions on Computer Simulation*, Vol. 16, No 2, 1999.

[14] D. Anagnostopoulos, and M. Nikolaidou, “An Object-Oriented Modelling Methodology for Dynamic Computer Network Simulation”, to appear in the *International Journal of Modelling and Simulation.*