Discrete-Event Simulation of Network Systems Using Distributed Object Computing

Weilong Hu
Arizona Center for Integrative M&S
Computer Science & Engineering Dept.
Fulton School of Engineering
Arizona State University, Tempe, Arizona, 85281-8809
Email: weilong.hu@asu.edu
http://www.acims.arizona.edu

Hessam S. Sarjoughian*
Arizona Center for Integrative M&S
Computer Science & Engineering Dept.
Fulton School of Engineering
Arizona State University, Tempe, Arizona, 85281-8809
Email: sarjoughian@asu.edu
http://www.acims.arizona.edu

Abstract: A modeling and simulation environment supporting scalable network system analysis and design is described. The environment, called DEVS/DOC, enables simulation modeling of network systems where hardware and software layers of the system are modeled separately and combined based on the concept of quantum Distributed Object Computing. The key benefit of this reengineered simulation environment is support for scalability. Of particular interest is the ability to evaluate alternative network configurations where many software and hardware components of a system can be accounted for and analyzed simultaneously. A series of simulations are developed to show the capabilities of distributed object computing modeling and its realization using the DEVSJAVA modeling and simulation environment.

Keywords**:** computer networks, discrete-event system specification, distributed object computing, HW/SW, performance analysis, scaleable simulation, quantum modeling.

1 INTRODUCTION

Numerous simulation modeling frameworks, methodologies, and techniques have been proposed for designing software or hardware aspects of network systems. Some of these simply provide add-on capabilities to handle simulation modeling since each generally is devised for modeling hardware or software layers of a networked system [1][2][3]. In contrast to using ad-hoc means to support combined software and hardware simulation modeling of networked system, it is important to have a methodology that inherently supports hardware/software co-design. Such a methodology, similar to co-design for embedded systems, can have overcome inherent challenges in accounting for tradeoffs in network system design.

Distributed object computing (DOC) [4] provides an approach to modeling and simulating distributed object computing systems as a set of software components mapped onto a set of networked processing nodes. Distributed object computing was extended and implemented using the discrete event system specification [5] and DEVSJAVA environment [6]. It is intended for co-design of networked systems. Distributed object computing provides three abstractions to model and simulate a system's *software* and *hardware* layers and their interactions [7][8][9]. The software layer is captured as a distributed cooperative object (DCO) model to present interacting software objects, both local, to a hardware node or a set of distributed hardware nodes. The hardware layer represents a loosely coupled network (LCN) model of processing nodes, network gates, and interconnecting communications. The distributed DCO software assigned to LCN hardware forms an object system mapping (OSM). The simulation models of DCO, LCN, and OSM component structures and behavior dynamics were formally characterized using the DEVS formalism [8][9].

As a modeling and simulation framework DEVS/DOC, and in particular DOC 1.0, was built on top of DEVSJAVA 2.63. DOC 1.0 supports modeling of distributed network systems such as information systems and supply chain networks (e.g., see [10][11]). It provides important features which can help users build their own simulation models using the principle of component-based modeling. It also offers a user friendly environment for simulation experiments. The visualization and controlled manipulation features of simulation execution provide a variety of simple to powerful features (e.g., concurrent event handling and processing) for extensive simulation studies.

Recent advances in software design, programming languages, and development environments led to DEVSJAVA 2.7, a redesign of DEVSJAVA 2.63. The new DEVSJAVA environment offers capabilities such as real-time simulation and powerful data structures, which are key for modeling complex structural and behavioral aspects of software and hardware network components [6]. These capabilities in turn offer important features to be incorporated into DOC. For example, one of the main differences between DEVSJAVA 2.63 and DEVSJAVA 2.7 is the use of the Java Collection API. The use of DEVSJAVA 2.7, therefore, necessitated redeveloping DOC 1.0.

* To whom all correspondences should be directed to.

In the remainder of the paper, we will describe DOC and its application to analysis of a client/server network system. In Section 2, we review the basic distributed object computing concepts and some aspects of the DEVSJAVA 2.7 modeling and simulation environment. In Section 3 we highlight the core elements of the DOC 2.0 simulation model components. To demonstrate DEVS/DOC 2.0 capability in terms of scalability, we present an example of a simple network consisting of a few to several hundred software and hardware components in Section 4. In Section 5, we discuss future work and present some future research directions.

2 BACKGROUND

2.1 *Distributed Object Computing Approach*

Distributed Object Computing (DOC) is a conceptual framework for modeling hardware and software components of networked systems. DEVS/DOC is a realization of DOC using the DEVSJAVA modeling and simulation environment. This environment supports simulation of combined software and hardware for structural and behavioral networked systems. A DOC model consists of a Loosely Coupled Network (LCN), Distributed Cooperative Object (DCO), and Object System Mapping (OSM) models. An LCN model represents the hardware structure (topology) and behavior (interaction) of interconnecting hardware entities. A DCO model specifies software components and structures. It defines how the DCO software components interact both internally and externally (i.e., when executed on the distributed LCN hardware components). An OSM model specifies precisely how DCO components are to be mapped onto LCN hardware components. The separation and integration of DCO and LCN are important in enabling modelers (analysts and designers) to study (i) alternative software designs given some hardware architecture, (ii) alternative hardware designs given a collection of interacting software components, (iii) or combinations thereof. Figure 1 presents an abstract view of the distributed object computing framework [8][9].

The DOC modeling approach called DEVS/DOC 1.0 was first developed using the DEVS framework and finally realized using the Java technology. The DEVS framework enables modular, parallel model execution and thus it offers a systematic basis for handling multiple, concurrent events as well as capabilities to organize a collection of entities and their manipulations [6]. The capabilities offered by DEVS/DOC 1.0 template simulation models extend the general purpose (core) DEVS atomic and coupled models. As shown in Figure 1, software and hardware objects belonging to the DCO and LCN layers are synthesized via OSM. For example, the swObject model supports handling input/output ports and messages offered by atomic models (partial class hierarchy of DEVS/DOC 1.0 is shown in Figure 2). Similarly, a processor model is defined as a coupled model consisting of transport, router, and cpu atomic models. Atomic and digraph are elementary DEVS models.

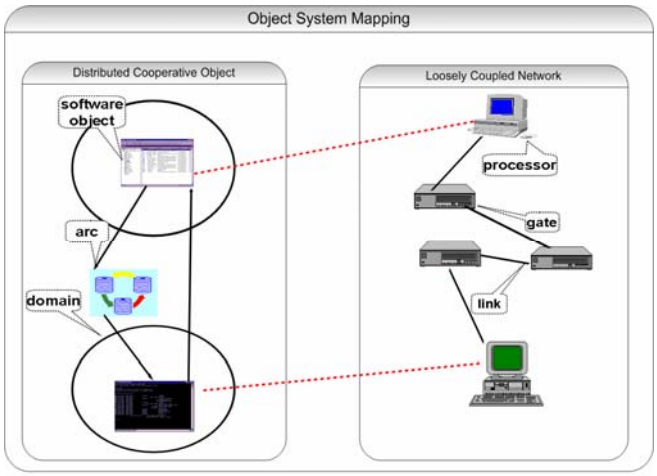


Figure 1. Distributed Object Computing Approach

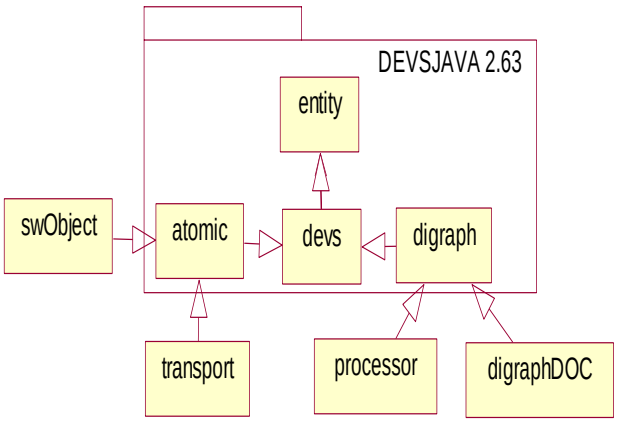


Figure 2. DEVS/DOC 1.0 Partial Class Hierarchy

2.2 *DEVSJAVA 2.7 Simulation Environment*

Many implementations of the DEVS framework have been developed in popular programming languages including Java. The DEVSJAVA 2.7 environment is a new generation of DEVS-based modeling and simulation environments that separates the modeling and simulation engines and the user interface. This environment offers new capabilities not found in DEVSJAVA 2.63 such as model type discovery and run-time execution using logical clock or wallclock (see [12]). DEVSJAVA 2.7 offers strong separation between modeling and simulation engines, which is important for supporting distributed simulation using

alternative technologies such as CORBA [13] and HLA [14].

One of the key DEVSJAVA APIs is GenCol [15]. Based on SDK 1.4 [16], it consists of a set of containers and utility classes necessary for all other modules. This API contains the base class entity, which serves as the root class for the *modeling* (genDevs.modeling), *simulation* (genDevs.simulation), and *visualization* (simView) modules [15]. In DEVSJAVA 2.7, the modeling module provides basic modeling elements including atomic and coupled models such as devs and atomic. The devs class extends class entity with methods to add input and output ports so that a model can send and receive messages—i.e., it supports distributed communication. Extending from devs, the atomic class provides state assignment and functions to process external and internal events. With DEVSJAVA 2.7, all atomic models can run as a separate component with viewableAtomic and all coupled models can be executed as a separate system with viewableDigraph (see Figure 3). The other basic modeling elements in the modeling module are ports and coupled, which allow coupling of atomic and coupled models.

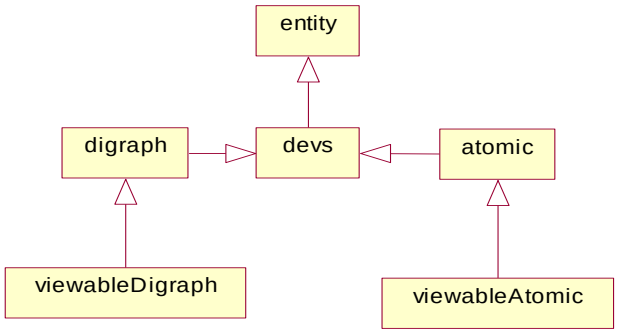


Figure 3. viewableAtomic and viewableDigraph Classes in DEVSJAVA 2.7

Similar to the *modeling* package, the *simulation* package contains classes that execute the atomic and coupled models and thus transmit messages. The classes in the simulation package are realizations of the DEVS parallel abstract simulator [6]. The atomicSimulator and coupledSimulator are two of the key classes for handling the timing of atomic and coupled models and their exchange of output and input events. The other main package is the *SimView,* which provides GUI services for visualizing atomic and coupled models (viewableAtomic and viewableCoupled), configuring simulation models, and executing and viewing the simulation dynamics.

An example of an atomic LCN model is called *hub*. The execution speed of the hub can be controlled via the "real time factor." A model component can receive input messages on one or all of its input ports (e.g., inLink1) and concurrently produce output messages on its output ports (e.g., outLink2). The states of the hub atomic model (e.g., execution phase (passive) and the duration in which the hub stays in phase passive (∞)) can be viewed at run time.

3 DISTRIBUTED OBJECT COMPUING ENVIRONMENT

In this section, we describe the new DEVS/DOC modeling and simulation environment with particular emphasis on the details of atomic and coupled DEVSJAVA model specifications for software and hardware. We show the new DOC model components and their role in supporting simulation of relatively large-scale network systems (a few thousand hardware and software model components) using DEVSJAVA 2.7.

3.1 DOC 2.0 Software and Hardware Layers and Their Mapping

The Distributed Object Computing part of DOC 2.0 is designated for modeling software objects and interaction arcs. These software objects form a computational domain. A software object contains both attributes (data members) and methods (function). The size of a software object is defined by the collective memory requirements of these attributes and methods. When a software object is invoked, the size parameter loads the supporting LCN processor memory. Message arcs and invocation arcs are defined as software-object interactions. A message arc represents peer-to-peer exchanges between objects, while the invocation arc represents client-server type interactions between two software objects. The LCN components model the hardware aspect of a network.

As mentioned above, Object System Mapping plays a key role in modeling the behavior or performance of distributed object computing systems. It maps the DCO software objects onto the LCN nodes so that the abstract behavior dynamics of the software architecture are constrained by the capacity of resources such as processor speed, memory size, and bandwidth. This distributed architecture glued by OSM provides flexibility and enables users to analyze and design different software on the same hardware or different hardware for the same software.

3.2 New DOC Specification

The packages of DEVSJAVA 2.7 provide the basic capabilities that are used in DOC 2.0. As mentioned earlier, since DEVSJAVA 2.7 was completely redesigned, a new design for DOC 2.0 became necessary. The atomic models in DOC 2.0 are extended from viewableAtomic, which supports greater GUI and execution capabilities compared to DOC 1.0. Similar to atomic models, coupled models are extended from digraphDOC. Coupled models in DOC 2.0

extend viewableDigraph, which supports system level visualization of the simulation models. Also unlike DOC 1.0, when users apply DOC 2.0 to set up their own modeling and simulation system, there is no need to resort to low-level (customized) programming to visualize simulation executions. That is, as long as the application model extends from the digraphDOC class, the simulation engine simView handles the entire execution of the simulation.

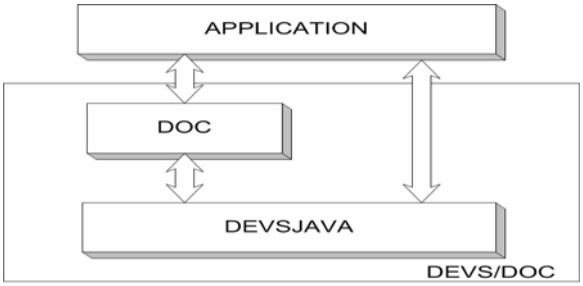


Figure 4. Application Domain Modeling Using DEVS/DOC

The LCN and DCO atomic models in DOC 2.0 extend from viewableAtomic. These models therefore allow visualization of hardware and software models. DOC 2.0 supports coupled models at the hardware and software levels, and their mapping at the OSM level. A simple example of a coupled model in DOC 2.0 is the processor model. It contains cpu, router, and transport atomic model components.

3.3 Application Layer

DOC 2.0 provides users with flexible facilities and methods to set up their own modeling and simulation applications. Figure 4 shows the relationships between DOC 2.0, DEVSJAVA 2.7 and user defined applications. In the application layer, hardware or software components can be defined as descendents of viewableAtomic (vA) and coupled models can be defined as descendents of viewableDigraph (vD). A system which contains several coupled models can be defined as digraphDOC (see Table 1).

As a modeling and simulation system, DOC 2.0 provides a mechanism to control and monitor the simulation process and to collect and analyze output data from the simulation process using the concept of experimental frame [4]. Traditionally, an experimental frame includes three components: generator, acceptor, and transducer. The generator stimulates the system under investigation, the acceptor monitors an experiment to see the desired conditions are met, and the transducer observes and analyzes the system outputs.

With the help of viewableAtomic in DEVSJAVA 2.7, DOC 2.0 sets up the experimental frame with two classes, acceptor and transducer. The acceptor can also stimulate the system by sending fire messages.

For an atomic or a simple coupled model, one can define a transducer to observe simulations. However, for large simulation models, multiple *transducers* are needed to handle different parts of simulations separately. DOC 2.0 provides the transd_tuples class to collect and analyze the data from other transducers (see Table 1).

4 A NETWORK SYSTEM EXAMPLE

As a layered, distributed framework, DOC 2.0 supports study of hardware/software co-design. As alluded to earlier, this environment is particularly useful for analysis and design of concurrent hardware and software behaviors. DOC 2.0 extends the concept of co-design system development from the execution of one or more processes on a single device (embedded system) to the interdependent execution of many processes running on multiple, distributed, and networked heterogeneous devices (system of systems). The LCN, DCO, and OSM building blocks equip the DOC 2.0 environment with important flexibility when used as a toolkit for studying distributed systems. Next, we illustrate the key role of separation of software and hardware layer modeling using two client/server system configurations.

4.1 Simulation Experiment Set-up

As we discussed above, this paper's goal is to show the scalability of DEVS/DOC 2.0. This environment supports development of large-scale simulation experiments for studying client/server network protocols. A client/server application executing over a network requires a server software component receiving requests from one or many client software components for processing executions over a set of hardware components.

A model of a single-client/server network consists of three hardware components (two processors and one link) and two software components. A two-client/server network model has an additional pair of processor and software simulation model components (see Table 1 and Figure 5).

An N-client/server model can be generalized—a model needs to have additional processors, software objects, and links (see Table 2 for the LCN, DCO, and OSM simulation model components). Each client executes its methods, sends requests (messages) to the server, and receives processed requests (messages) after some time period. Partial specifications for the hub ethernet, processor, software, and acceptor simulation models are given in Table 3.

Table 1. Two-Client/Server Network Components

Application objects		*1 server*	*2 clients*	*1 link*	*3 network interfaces*	*3 processors*	*9 transducers*	*1 acceptor*
DOC	*class*	swObject		link_ethernet	hub_link	processor	transducer	acceptor
	package	**DCO**		**LCN**			**Experiment Components**	
DEVS	*class*	vA				vD	vA	
	package	**simview**						

Table 2. DOC Simulation Model Components in the N-Client/Server Network System

LCN Layer		*DCO Layer*		*OSM*
N processors (Proc-1, …, Proc-n)	1 link (Link)	N-client (Client-1,…, Client-n)	1 server (server)	each software object is assigned to a processor

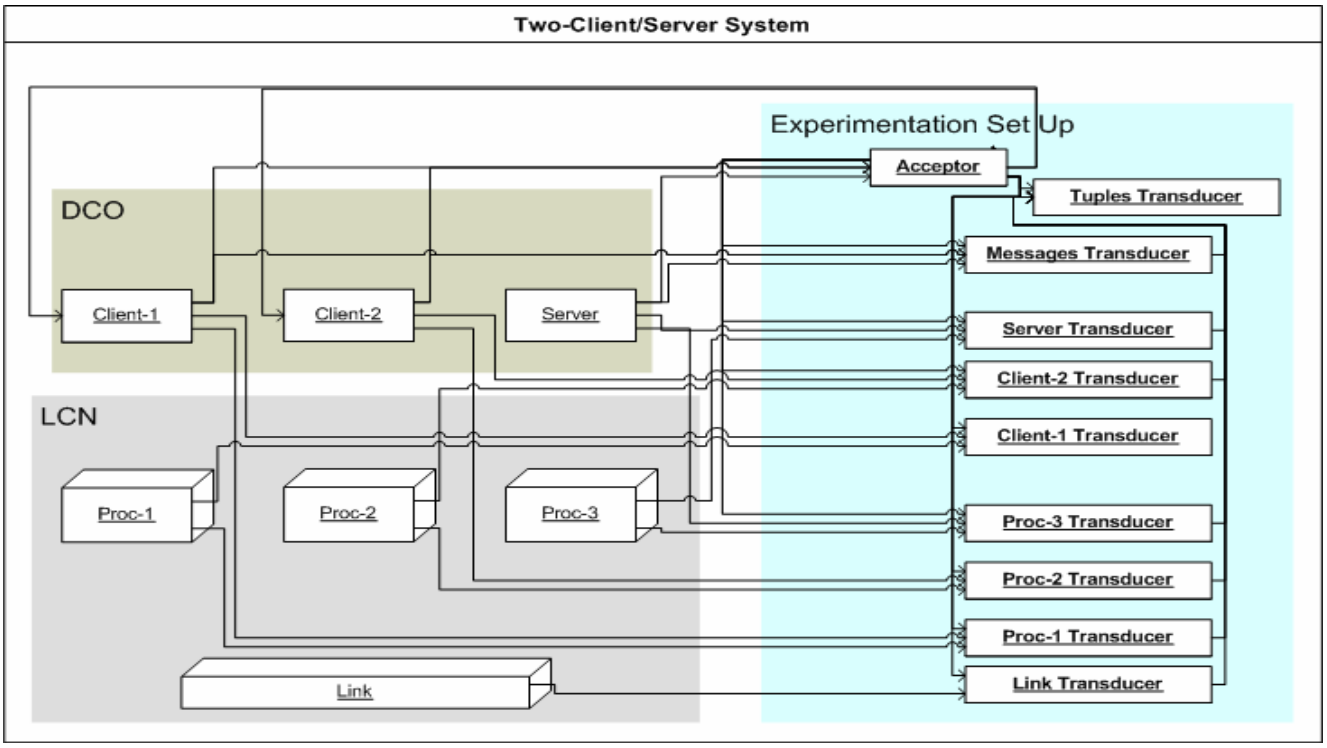


Figure 5. System Decomposition of a Two-Client/Server Network

It should be noted that for a given network topology, multiple links between the clients and server may be used. However, given our choice of the MAC (media access control) protocol and the purpose of these experiments, we have used one link as an abstraction of multiple links.

To study behavior of a network system via simulation, it is important to design experiments. For example, as shown in Figure 5, there are two transducers where one monitors a link and another monitors a processor. Similarly, there are transducers to monitor the software objects and their message exchanges (see Figure 5). To control the execution of the model, an acceptor is defined which in part defines the start and termination of a simulation scenario.

The simulation scenario considers each client to have three tasks to perform, two of which can be processed by the client itself (i.e., using its own designated processor) and one sent to the server (i.e., processor assigned to the server) via the link. Once the server receives a task from a client, the server processor begins to process the task and sends the completed task back to the requesting client through the link.

4.2 Performance Analysis

The model and interaction described above are straightforward for a simple network (i.e., one that has a small number of hardware and software components and interactions). However, as the number of components increases, the dynamics of the model quickly become difficult to predict via direct extrapolation from small-scale simulation models.

The client/server models briefly described above can be studied in terms of their components. For example, we can collect data (e.g., *collision number, successful transmissions,* and *bandwidth utilization*) for the link. Each experiment has an observation time during which data can be collected. Using the DEVS/DOC interface, users can change the execution speed via the simulator's real-time factor—the simulation 'speed up' or 'slow down' rate can be adjusted in terms of the simulator's real-time clock. The observation time needs to be carefully selected. The simulation observation time needs to be greater than the time required for the simulation models to satisfy some criteria. For example, a suitable criterion is to have the observation time be greater than the time it takes for all the software objects (clients and the server) to complete their activities.

Table 3. Selected Hardware, Software, and Experimentation Components

	Attribute	*Value*	*Unit*
Hub Ethernet	ethernet speed	10^6	bit/sec
	processor internal bandwidth	infinity	bit/sec
	buffer size	infinity	bit
	Attribute	*Value*	*Unit*
Processor	cpu speed	10^8	operations/sec
	cpu memory size	$64*10^6$	bytes/sec
	maximum packet size	$31*10^3$	bit
	processor internal bandwidth	infinity	bits/sec
	processor network interface speed	$2*10^9$	bits/sec
	buffer size	infinity	bit
	packet header size	20	byte
	Attribute	*Value*	*Unit*
Software	size	$2*10^6$	byte
	self-starting duty cycle	infinity	sec
	thread mode	none/method	
	Attribute	*Value*	*Unit*
Acceptor	time for discovering LCN topology	1	sec
	number of times to invoke task	1	NA

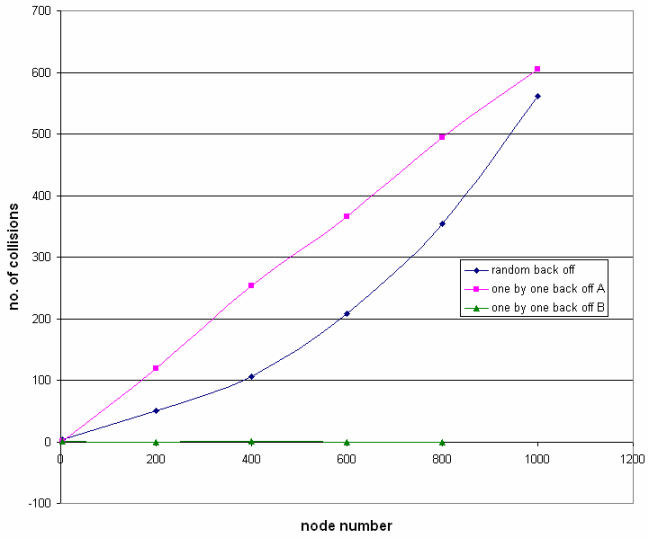
The performance of the network system is affected by many factors. In this paper, the transmission "back off time" is selected to demonstrate the role of DOC and performance analysis of network systems. As discussed above, both clients and server need to send packets (messages) into *link* through the *hub ethernet* model. These clients and server can be viewed as nodes in the network system. The number of the (client and server) nodes is always greater than one in order to study different "back off" schemes in the presence of packet collisions in the link (e.g., one client and the server simultaneously send messages to the link). Usually, after the first collision, each node waits either 0 or 1 time units before trying again. If two nodes collide and each node picks the same random wait time, they will collide again. After the second collision, each node picks 0, 1, 2, or 3 at random and waits that number of time units before sending its packet to the link. If a third collision occurs with 0.25 probability, then the number of time units for a node to wait is chosen at random from the interval 0 to 2^3 - 1. In general, after *m* collisions, a random number between 0 and 2^m - 1 is selected. Figure 6(a) shows the growth in number of collisions given the "random back off" scheme [17] for 2 to 1000 nodes. This is expected since with more and more nodes, the chance of collision increases exponentially. The results shown in Figure 6 are average values from 10 simulations runs.

Given the flexibility to manipulate software and hardware layers of a network system, we use a different *hub ethernet* model, which can be easily composed with the software components from the previous experiment. One of the important attributes of the *hub ethernet* model is the "back off time," which can be calculated using the traditional scheme discussed above, or can be fixed to a series of constants. For example, if there are four nodes in the network, after the first collision, their back off time can be 1, 2, 3, or 4 time units where each of the client and server nodes selects one of them to reschedule its packet retransmission. We call this "one by one back off" scheme.

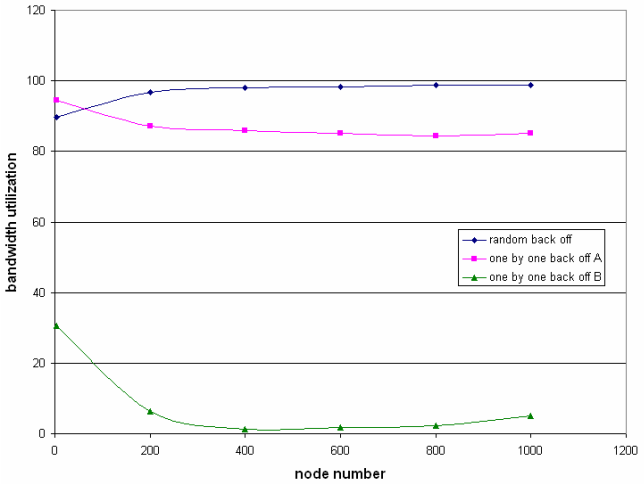
It is natural to suppose this one by one back off scheme will reduce the number of collisions or eliminate them altogether. However, as shown in Figure 6(a), the one by one back off scheme leads to more collisions. The important point here is the relationship between the Server Processing Time (SPT) and one back off time unit (BTU). BTU is the difference between retransmission times of two clients. The reason the one by one back off time seems counterintuitive is that the SPT is bigger than BTU so the server's input to the link interrupts the clients' scheduled back off sequences which results in more collisions.

Before we discuss the details of analysis, we define two types of one by one back off schemes, one called *A* and the other *B*. Type *A* assumes SPT is less than BTU and Type *B* assumes the inverse.

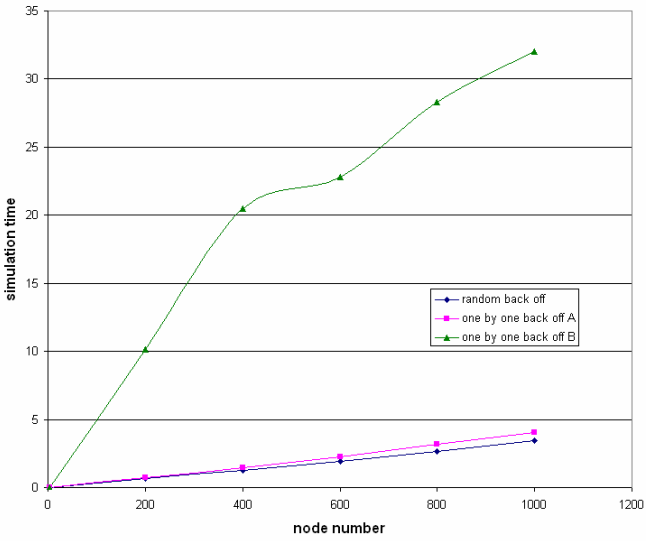
Referring to Figure 7, at time t_i, 5 clients are transmitting their packets to the link—that is, five events are generated concurrently at time t_i (see Figure 7(d)). Because all of these packets go to the link, this results in collisions at the link. After the collision at time t_i, the retransmission of clients number 1, 2, 3, 4, and 5 are scheduled at t_{i+1}, t_{i+2}, t_{i+3}, t_{i+4}, t_{i+5} times as shown in Figure 7(a). The server needs Δt time (one SPT) to process and send the processed packet back. Here, we define $\Delta t = 1.5 * (t_{i+1} - t_i)$ such that SPT is bigger than BTU (Figure 7 (b))—this is Type *B* given above. Figure 7(c) shows that after t_{i+2}, two packets are transmitted via the link, which results in invoking two tasks in the server.



(a). scale and no. of collisions



(b). scale and bandwidth utilization



(c). scale and simulation execution time

Figure 6. Performance Analysis for Back Off Schemes

At time $t_{i+1}+ \Delta t$, the first task in the server is completed and is sent to the link. Then, client 3 sends its packet to the server and there are two tasks in the server again, (see Figure 7(c)). Since Δt is defined as 1.5 BTU, the task for client 2 in the server is completed and sent back at time t_{i+4}. At the same time, client 4 is also retransmitting its packet followed by the second collision at t_{i+4}. This forces both the server and client 4 to back off. The server will back off one-third of Δt, and client 4 still needs to back off 4 BTU. This results in client 4 scheduling retransmission at time t_{i+8}. We define the server's back off time to be fixed—i.e., one-third of the server processing time, $\Delta t/3$. So, at time of $t_{i+4}+ \Delta t/3$, the server makes its retransmission and the packet for client 2 is sent back. At t_{i+5}, client 5 makes its retransmission and again there are two tasks in the server. These two tasks will be completed and sent back at times t_{i+6}, and t_{i+7}.

As stated above, client 4 will try retransmission at time t_{i+8}. At this time, there are no tasks in the server so the task for client 4 will be completed and sent back at $t_{i+8} + \Delta t$. In the above scenario, there are two collisions including one caused by the interruption from the server to the clients that occurred at t_{i+4}. However, this does not mean that constant back off time will always lead to more collisions. The relationship between SPT and BTU determines whether or not the constant back off sequence will be interrupted or not. As shown in Figure 8, if the Δt (SPT) is smaller than BTU, then before the next retransmission happens, the server has already sent the packet back to the client—i.e., there will be no collision due to interruption from server to the client.

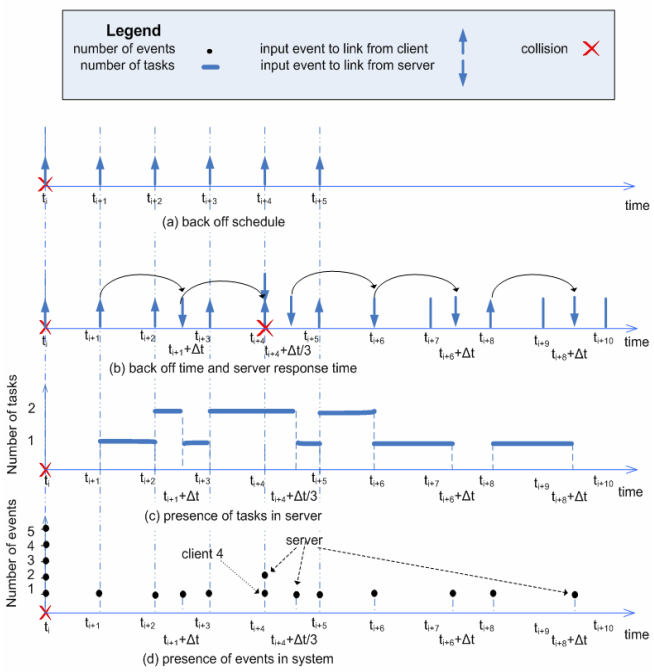


Figure 7. Type *A* One by One Back Off Scheme

In our experiments, the SPT is set to $4*10^{-4}$ seconds. In the one by one back off scheme A, the BTU is set to $1*10^{-4}$ seconds, and in the one by one back off scheme B, BTU is $1*10^{-2}$ seconds. In our implementation of the random back off, the continuous collision number (the number of collisions that happen between two successful transmissions) is less than 10 and the BTU is chosen by the algorithm described above, usually with some constant adjustment (i.e., between $1*10^{-6}$ seconds and $1*10^{-5}$ seconds). That is, before the server is able to complete and send the first task it has received, more tasks are sent to the server. Also, after the first collision happens, the interval for the retransmission is smaller. Therefore, the chance for the collision becomes smaller and also raises the bandwidth utilization (the bandwidth usage in the unit time) as shown in Figure 6(b).

However, in order to control exponential growth of the random back off scheme waiting time, the adjustment period will be kept constant (i.e., the waiting time remains between $1*10^{-4}$ and $9*10^{-4}$ seconds) once the number of continuous collisions is greater than 10 [7]. This technique makes the random back off BTU close to the SPT. The result is increased number of collisions which is also similar to the one by one back off scheme A (see Figure 6(a)).

One additional observation to be made is on the relationship between the number of collisions and bandwidth utilization. With the number of collisions in the random back off greater than 10, the total waiting time between two successful transmissions is still shorter than in the one by one back off scheme A. This explains why the random back off bandwidth utilization is always the best among the three back off schemes (see Figure 6(b)). Similarly, the simulation time for random back off is always the shortest (see Figure 6(c)).

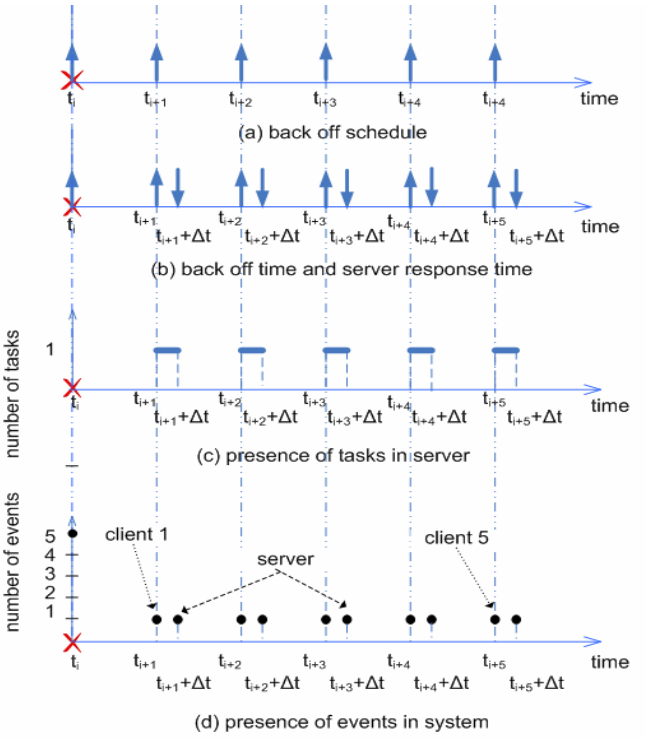


Figure 8. Type *B* One by One Back Off Scheme (See Legend in Figure 7)

Waiting times between two successful transmissions can be described in terms of RTWT (random back off total waiting time), ATWT (one by one back off scheme A waiting time), *rwt* (random back off single collision back off waiting time), and *awt* (one by one back off scheme A single collision back off waiting time). Given n as the number of collisions, we can derive the following relationships which state that the waiting time for random back off is less than the waiting time for the one by one back off scheme A. Given:

$$RTWT = \sum_{i=0}^{n} rwt_i = \sum_{i=0}^{9} rwt_i + \sum_{i=10}^{n} rwt_i$$

$$ATWT = \sum_{i=0}^{n} awt_i = \sum_{i=0}^{9} awt_i + \sum_{i=10}^{n} awt_i$$

$$\sum_{i=10}^{n} rwt_i \approx \sum_{i=10}^{n} awt_i$$

$\sum_{i=0}^{9} rwt_i < \sum_{i=0}^{9} awt_i$, we have $RTWT < ATWT$.

We chose six model configurations based on the number of nodes (i.e., 3, 200, 400, 600, 800, and 1000 node network models). These configurations allow us to study how software and hardware aspects of a network system affect one another. For example, for a configuration with 3 nodes, the one by one back off scheme A has only one collision. More generally, the key properties (collisions, bandwidth utilization, total simulation time) of one by one back off schemes A and B and the random back off can be studies systematically under different hardware and software settings as described above. In particular, we can determine that the one by one back off scheme B has a smaller number of collisions because SPT is less than BTU. Similarly, we can examine why the Ethernet busy time and simulation time are always longer for random back off time scheme B as compared with random back off time scheme A and random back off.

5 RELATED WORK

There are a variety of simulation environments that support modeling of computer networks. Well-known environments for simulation—specialized for computer networks—include NS-2 (Network Simulator [2][18]), GloMoSim (Global Mobile Information Systems Simulation [19]), and TeD (Telecommunications Description Language [20]). An early modeling environment that preceded these is called REAL [2]. This was a popular computer network simulator providing around 30 modules (written in C) capturing details of several well-known flow control protocols (e.g., TCP) and other scheduling disciplines (e.g., Fair Queuing and Hierarchical Round Robin). Since REAL was developed solely for traditional wired networks, the Defense Advanced Research Projects Agency sponsored development of NS-2 to handle more complex network systems [2].

With support for customizing existing models (e.g., unicast routing, multicast routing, mobile networking, and satellite networking), NS-2 has become popular and to some extent is being used for defining customized network models. However, since there is strong dependency among some of the NS modules, it can be challenging to model new protocols or to make changes to existing protocol models. That is, given its complexity and low-level detailed models, it is necessary to have in-depth knowledge of NS-2 for serious simulation [2]. Furthermore, NS-2 is not intended for disciplined integration of software and hardware simulation modeling. It primarily supports simulating software communication protocols for network devices instead of simulating a set of software applications distributed across processors and network devices.

DEVS/DOC is similar to other environments such as NS-2 and provides modelers with ready to use hardware modules such as *cpu*, *router*, and *link*. However, unlike NS-2 and others such as OpNet, it provides greater flexibility and simplicity for creating customized hardware modules as well as software modules and composing them within a well-defined simulation modeling framework. That is, unlike DEVS/DOC, the NS-2 environment does not support distinct software and hardware modeling as layers or their composition. Instead, NS-2 supports very detailed, fine grain modeling of protocols rather than quantum level modeling.

From a usability aspect, NS-2, which is a Unix-based environment, does not offer a user friendly simulation environment. Consequently, the use of NS-2 can require additional time and effort for developing models that are not already available. For example, direct run-time control of simulation offers important aid to modelers in conducting detailed analysis of simulation execution rather than relying solely on data (simulation logs). The comparison of NS-2 and DEVS/DOC indicates the latter to be more suitable for system-level network simulations and experimentations with user friendly interfaces and popular integrated object-oriented development environments.

6 FUTURE WORK

The distributed object computing approach is important for characterizing network systems separately in terms of software and hardware layers. The new DEVS/DOC is an environment that supports experimenting with system specifications where alternative software and hardware designs can be varied and integrated in a well-defined fashion. Given the generic distributed object computing framework and discrete-event system specification, DEVS/DOC offers a basis for modeling network-based systems where it is important to analyze and design a system's conceptual architecture not separately from software or hardware points of view, but instead by accounting for total software and hardware simultaneously. Therefore, DEVS/DOC offers a sound framework for important domains including sensor networks, mission training systems, and supply-chain networks. Applications of interest might include enterprise systems where it is necessary to develop architectural designs across many hundreds of (hardware and software) computational nodes.

Another area of interest is to use DEVS/DOC simulation models with physical systems. This can lead to important capabilities where a portion of a network system executing on a physical computer network can be embedded inside a large-scale simulation model. That is, the

environment can support physical hardware with simulated software or vice versa where software tools are executed on simulated hardware. This capability enables mixed logical and real-time execution of simulated and physical software and hardware components. For example, the conceptual and system-level design of a network of many satellites orbiting the earth can be simulated with a few actual satellites, with the remaining ones being simulated. Finally, large-scale, complex application of DEVS/DOC requires execution of DEVS/DOC in a distributed setting such as a service-oriented architecture.

Acknowledgment

This research was partially supported by NSF DMI-0075557 grant.

REFERENCES

[1] Keshav, S., REAL 5.0 Overview, Cornell University, http://www.cs.cornell.edu/skeshav/real/overview.html. 1997.

[2] Information Sciences Institute (ISI), The Network Simulator - ns-2, University of Southern California, http://www.isi.edu/nsnam/ns/, 2004.

[3] OpNet Modeler, http://opnet.com, 2004.

[4] Butler, J. M. 1995, "*Quantum Modeling of Distributed Object Computing*", Simulation Digest, Vol. 24, No. 2, pp. 20-39.

[5] Zeigler, B. P., T. G. Kim, and H. Praehofer, 2000, *Theory of Modeling and Simulation*, 2nd Ed., New York: Academic.

[6] DEVSJAVA, ACIMS, http://www.acims.arizona.edu /SOFTWARE/software.shtml, 2004.

[7] Hild, D. R., "Discrete Event System Specification (DEVS)/Distributed Object Computing (DOC) Modeling and Simulation", Ph.D Dissertation, March 2000, Electrical and Computer Engineering Dept., University of Arizona, Tucson, Arizona.

[8] Sarjoughian, H. S., D. R. Hild, and B. P. Zeigler, 2000 "*Engineering Distributed Systems: Simulation-Based Co-Design*", IEEE Computer, Vol. 33, No. 3, pp. 110-113.

[9] Hild, D. R., H. S. Sarjoughian, B. P. Zeigler, January 2002, "*DEVS-DOC: A Modeling and Simulation Environment Enabling Distributed Codesign*", IEEE SMC Transactions, Vol. 32, No. 1, pp. 78-92.

[10] Godding, G., H. S. Sarjoughian, K. E. Kempf, Dec., 2003, "Semiconductor Supply Network Simulation, Winter Simulation Conference", pp. 1593-1601, New Orleans.

[11] Sarjoughian, H. S., X. Hu, B. Strini, D. Hild, 2001, "*Simulation-based HW/SW Architectural Design Configurations for Distributed Mission Training Systems*", Simulation, Vol. 77, No. 1-2, pp. 23-38.

[12] Cho, Y., B. P. Zeigler, H. Cho, H. S. Sarjoughian, and S. Sen, March 2000, "Design Considerations for Distributed Real-Time DEVS", AIS 2000, pp. 290-294, Tucson, AZ.

[13] CORBA Basics, Object Management Group, http://www.omg.org/gettingstarted /corbafaq.htm, 2005.

[14] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and rules, IEEE Std 1516-2000, 2000.

[15] Park, S., B. P. Zeigler, H. S. Sarjoughian, Oct. 2001, "Interface for Scalable DEVS and Distributed Container Object Specifications", IEEE Sys. Man. Cyber. Conf., Tucson, pp. 93-98.

[16] Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification, http://java.sun.com/j2se/1.4.2/docs/api/ index.html, 2004.

[17] Tanenbaum, A. S., 1988, *Computer Networks*, 2nd Edition, Prentice Hall, pp. 145-146.

[18] Fall, K., V. Kannan, The ns Manual, March, 13, http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf, 2005.

[19] Gerla, M., R. Bagrodia, L. Zhang, K. Tang, and L. Wang, 1999, TCP over Wireless Multihop Protocols: Simulation and Experiments, Proceedings of IEEE ICC.

[20] Perumalla, K. S., R. M. Fujimoto, 1998, "Efficient Large-scale Process-oriented Parallel Simulations", Winter Simulation Conference, pp. 459-466, Washington D.C..