PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 44, NO. 3–4, PP. 303–316 (2000)

OBJECT-COOPERATION IN REAL-TIME: A CONTRACT BASED PROTOCOL

Balázs GOLDSCHMIDT, Károly KONDOROSI and Zoltán LÁSZLÓ

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
H–1521, Budapest, Hungary
balage@inf.bme.hu, kondor@iit.bme.hu, laszlo@iit.bme.hu

Received: April 10, 2000.

Abstract

In this paper we shall introduce a contract-based protocol for object oriented real-time systems, and a simulation environment for this protocol. We outline the problems of object orientation in real-time systems, and the possible solutions. Among the solutions we show the contract-based model, which should handle overloaded situations effectively, and the primary elements of the model. Later on we introduce the simulation environment designed to measure the characteristics of the model in simulated situations. We show the architecture and the inner working of it. We analyse the results it produced. Finally we look at the directions of further development as well.

Keywords: object orientation, contracts, overload, real-time systems, simulation.

1. The Contract Based Model

1.1. Real-Time Systems

It is not only the correctness of results, but the time the results are produced that is important in real-time systems. Some timing criteria have to be fulfilled in order to produce correct results.

Csaba PÁLOS [4] had the very strong opinion that the most important question is how the system behaves under overload. It is almost a commonplace to mention *graceful degradation* here. We say a system is overloaded if there is no schedule that will meet the deadlines of each task, performing the best response activity to each event the system recognized. Overload situations may occur because of extraordinarily high external event density or performance degradation in some parts of the system. In the best case it is only the quality that suffers, but in the worst some events won't be handled.

There are two major possibilities to avoid the severe consequences of overload:

- exclude the overload, or
- enable the system to work even under overload and minimize its consequences.

The conventional real-time design strategy chooses the former. This is the (off-line static) worst-case design method. Worst-case maximum event densities and worst-case maximum computing needs are evaluated, and the hardware will be sized to offer enough computing power to work under overload even in the worst possible case. This approach though has several drawbacks [3]:

- As the worst case hardly ever occurs, most of the computing power will never be utilized.
- As the maximum computing needs are supposed to be predictable some basic programming techniques (dynamic data structures, unbounded loops, recursion) are excluded. Also object orientation may cause several problems because of its dynamic nature (late binding, etc).
- Creating off-line static schedules is a very expensive work, thus reorganizing or changing the system on-the-fly is hardly possible.
- Finally, although these systems were designed thoroughly to avoid it, there occurs sometimes a not predicted density of events or some software or hardware failure that cause overloaded situations.

The other way allows the system to get overloaded in exceptional situations, but also to implement some on-line dynamic methods in order to limit the harmful consequences. We have some degrees of freedom that can help to achieve this goal:

- There is a tendency of having more and more luxury tasks in real-time systems. Their completion is not very important. The different cost of hurting this or that deadline allows us to choose a better distribution of processor power under overload by giving priority to crucial tasks and hurting the deadlines of less important ones. A static scheme of task priority seems not to be sophisticated enough for handling the frequently changing importance of functions in complex systems.
- There is an increasing number of incremental algorithms that give imprecise results in short time, and as time goes on the precision grows. That way if we abort the algorithm before completion we can get a less precise but useful result, and at the same time save the deadline.

The difficulty in on-line scheduling is its complexity. To really gain in performance, responsivity and power, a huge amount of task-specific momentary information should be considered.

1.2. Object-Orientation

In the past object-orientation and real-time systems were thought hard to reconcile for the former has such constructions that make the estimation of maximum computation time impossible.

The main problem is late binding for there is no way to estimate the execution time if we do not know which code segment will run at a given place. Any off-line solutions would cancel the benefits of object orientation, either the information hiding or its dynamic nature. However, there is an increasing interest towards using object orientation in real-time systems.

1.3. Contracts

The objects in the system have to complete timing constraints. In order to estimate timing correctly they need to know not only their own execution time but that of the other objects they rely upon. Some level of trust is needed between objects.

The main idea of [4] was that we should take an analogy from economics: every company produces some goods, and uses the products of other companies. They have to complete serious timing limitations, and they have to trust their suppliers. The protocol of these relations is the system of bilateral contracts.

Contracts contain the most important parameters: the type of service, the quality, the timing constraints, the remedial actions in case of vis maior, and the price of the service. Some contracts describe not only one delivery but a long term connection. In such cases the actual deliveries will be triggered by some external events, by very simple actual orders or by the calendar.

In [4] the author was certain that a similar protocol based on inter-object contracts, could serve as a distributed coordination mechanism. We expect the following benefits:

- *Overload-resistance:* The contracts may define remedial actions in case of an overload or of a forecasted deadline-miss so that the consequences can be minimized.
- *Load-balancing:* Contracts can change with the situation. Changes in the load distribution will be handled without human intervention. Less loaded objects will offer better service and client objects will change to them. The same mechanism can handle the object migration between nodes [2].
- *Maintainability:* The system can grow and change evolutionarily as new objects are added to the system and old ones are removed. Also the computing power may be resized by simply adding new nodes to the system and let objects migrate there.

However, we should take into account some possible drawbacks as well:

- *Administrative overhead:* In order to handle the advertisements the blackboard may use sophisticated algorithms, thus it has some inner latency when an advertisement can appear to the customers, or when a customer can be notified of interesting advertisements. On the other hand, the customers and suppliers themselves have to implement some protocol to make a contract, and it has timing overhead too.

Table 1. A typical advertisement

Advertisement	
Supplier:	`BA-8-Z23`
Service:	`Computation #493`
Precision:	`1E-2 to 1E-5`
Time:	`about 28ms`

- *Intelligence for evaluating offers:* Both the suppliers and the customers have to be smart enough in a complex system, to make the optimal or suboptimal deal, and to take into account the most important aspects in order to utilize the benefits of the contract model best.

1.4. The Protocol

There may be several variants of the protocol itself. First we introduce the common characteristics and give some examples.

In the protocol we defined two major kinds of objects: *suppliers* and *customers*. Of course a certain object may have the characteristics of both kinds. Thus a directed graph topology may form, where the sources are the end-suppliers, and the sinks are the end-customers.

The only centralized (but in an advanced variant it can be made distributed) server in the system is the *blackboard*.

At first the suppliers have to advertise their services, which they can do using the blackboard. The customers may search on the blackboard. The implementation of the blackboard may vary from the minimal version (it only stores the advertisements, and passes them unsorted to the customers) to the advanced one (it notifies the subscribed customers, if a new advertisement of their interest arrives). An advertisement may look like that in *Table 1*.

In the next phase the customer finds the suppliers using their advertisements, and asks them for an offer (here too the implementation may vary from a pure minimal to a full advanced version), and signs a contract with the one of the best offer. A typical offer may look like that in *Table 2*.

If we really need an overload-tolerant system, the contracts should have amendments about the handling of exceptional events (acceptable deadline slips, decreasing quality, etc).

A customer may even make deals with backup-suppliers so that if the main supplier cannot fulfil its task, the backups may continue and give an acceptable result.

In the system the results are triggered by orders. In the simplest case the contract itself is the order too. In the case of long term contracts there are separate orders, perhaps with unique parameters, that make clear some unspecified aspect

Table 2. A typical offer

Offer	
Supplier:	`BA-8-Z23`
Customer:	`ZZ-9-B11`
Service:	`Computation #493`
Precision:	`1E-4`
Time:	`26ms`
Deadline miss:	`Prewarning, Quality loss, Abort, Slip`

of the original contract.

Both parties may cancel the contract. The customer may do it because it does not need the service anymore, or it has been given a better offer recently. The supplier's cause may be that it has lost its resources it relied on, or it has an order of higher priority.

It is important to note that a cancellation of a contract may cause an avalanche of further cancellations. A good protocol implementation should avoid this situation.

One may even modify a contract on the fly for some needs or possibilities have been changed, but it should not happen when the system is just overloaded.

2. The Testing Environment

Using any of the above protocol variants we could construct a dynamic and flexible real-time system. The question is how useful the protocol really is, and what performance it could provide. We can use two major approaches to observe the characteristics of the protocol. Either we create a pure mathematical model, based on probability and queuing systems theory, or we can implement a computer simulation of the problem. We chose the latter one. Our aim was to create a testing environment we can test and compare several variants with.

2.1. The Core of the Simulation

The simulation has now two major packages, the core package (`consim`), and the prototype implementation `minimal` package (see *Figs. 1 and 2*).

In the core package there is the scheduler (`Clock`, a singleton object [1]). Objects implementing the `ClockClient` interface may register themselves to it

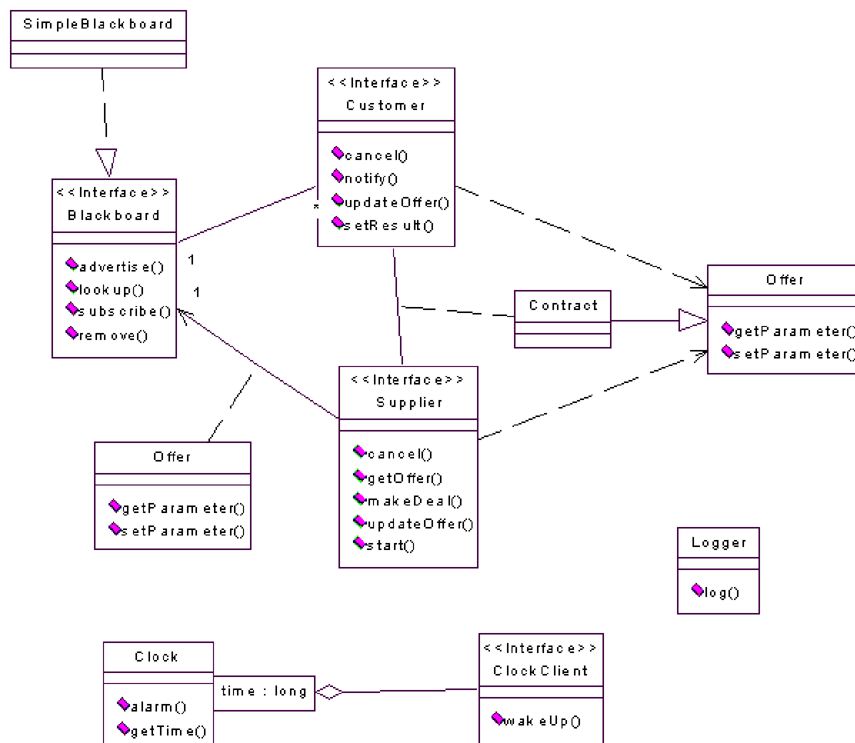

Fig. 1. Class diagram of package consim.

so that they will be notified when their time has come, and ordinary objects may ask the `Clock` for the current (virtual) system time.

The `Logger` (also a singleton) logs the given events into a log-file.

The `Offer` is a general purpose offer type, which can hold any kind of parameters with a name. Its subclass is the `Contract` that has the reference of its supplier and customer obligatory.

There are three more interfaces in the core package: the `Supplier`, the `Customer` (they define the communication interface to the suppliers and the customers), the `Blackboard` defines the interface of the blackboard. There is even a minimal blackboard implementation (`Simple Blackboard`) that can store the advertisements (`Offer` objects) of the suppliers, and when a customer asks for a service, it returns all of the offers.

To start the simulation we have to call the `start` method of the `Clock` object. It starts its own thread, and wakes up subsequently (in temporal order) all the subscribed `Clock Clients`. An object may subscribe in run-time also.

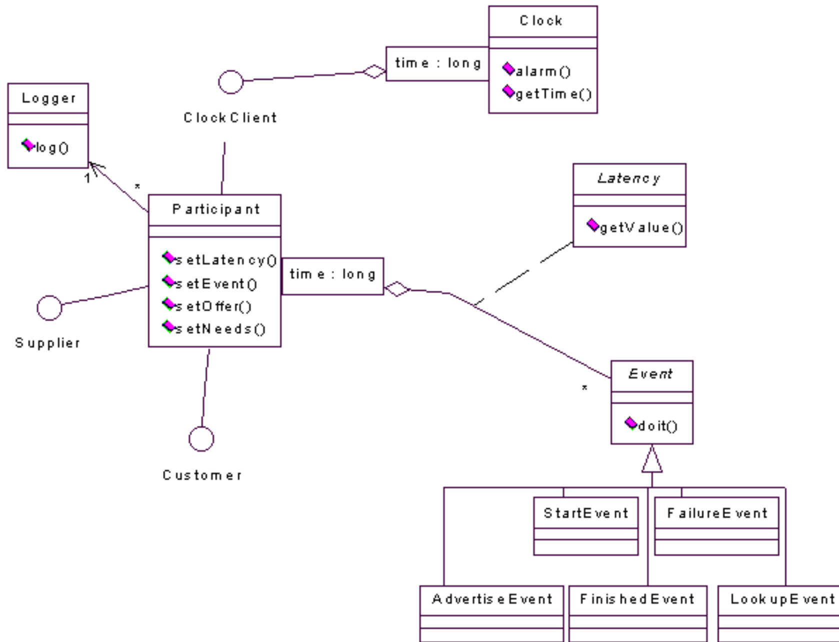

Fig. 2. Diagram of the implementation classes.

2.2. The Protocol Implementation

The most important element of the protocol implementation is the `Participant` class, which implements the `Customer`, the `Supplier`, and the `Clock Client` interfaces. Its behaviour is triggered by `Event` classes. The timing is directed by subclasses of the `Latency` abstract class, thus we can use fix time, uniform distribution, and Gaussian distribution (both with the appropriate mean value and deviation).

The implemented protocol is very simple. In a normal case (with `type-writer` setting we show the class of the event):

1. `Lookup`: the participant searches for the appropriate suppliers asking the blackboard. It does so until it has a supplier for every needed service. Finally, if it has some services to offer, it advertises on the blackboard.
2. `Start`: it starts to run. First starts its suppliers, and when each has sent a `setResult` message, it starts its own work.
3. `Finished`: the work is done. It notifies its customers about the results, and waits a given time, then starts from the beginning.

A pure supplier starts only when its customer starts it. The customer waits a preset time interval between looking up suppliers and starting the work.

If a supplier is contacted, and a contract is made, the supplier removes its advertisements from the blackboard.

There is a problem in two cases: either there was a cancellation, or some inner error occurred. In the former case it can either be a cancellation from the supplier or a cancellation from the customer. In this minimal protocol the cancellation means the breaking up of the contract, there is no way to change the contract on the fly.

If a supplier was sent a cancel message, it just stops working and advertises its service once again. If a customer was sent a cancel message, then it first verifies if it has backup suppliers for the given service. If it has not, then it looks up a new supplier, and cancels each contract it is a supplier in.

A failure event may occur during the run of a supplier (its probability may be set). In that case the supplier cancels every contract it has, and waits for a given time to be "repaired". When the time has come it advertises again.

Here is the summary of the settable parameters:

- The number of suppliers and customers.
- The restart time of the customers. After a successful run they wait that much time before starting again.
- The time the customers wait after an unsuccessful lookup.
- The failure probability of the suppliers.
- The work time of the suppliers and the customers.
- The type of the service the customer looks for and the supplier offers.
- The threshold of deadline for the clients.

3. The Results

In the simulation we built a two level architecture. In it the suppliers have no sub-suppliers, and the customers provide no service to others. Every supplier offers, and every customer seeks the same service. There are 10 customers and 5 suppliers in the system.

The servers' recovery time was set to Gaussian distribution of mean 500 and deviation 10, the clients' restart time was set to Gaussian distribution of mean 100 and deviation 10.

The clients' re-lookup time (the time elapsed between an unsuccessful lookup and the next) is 1, thus the clients keep looking up till they found a supplier.

The servers' failure probability was set to 0.01, a value typical of real life systems.

The clients' runtime was set to a fix 100[1], that of the servers to a Gaussian distribution of mean 100, the deviation was different in different configurations. That way we wanted to look at the problem of servers missing their deadline in a statistical way.

We made four protocol variants – or models – to investigate:

- **A**: This variant is the plain one. The servers do not know about any deadline. They offer best effort quality of services. It is our reference model for it uses almost no protocol.
- **B1**: In this model the servers offer services with a deadline of 100, and if they miss it, they cancel their contract, and the client has to find someone else to finish. The probability of when the servers realize they miss the deadline has uniform distribution.
- **B2**: This variant is the same as the previous one, except that the distribution of the servers' realization is growing linear (approaching the deadline they may realize with greater probability they will miss it).
- **C**: This variant is the most sophisticated of the four. It is basically the same as **B1**, but here the servers know what the real deadline threshold is, and do cancel according to it. Thus, if the threshold is higher, they know they have more time to complete.

Each simulation took 100 000 units of time. We simulated each variant with deadline threshold values set to 200, 300, 400, 500, 600, and 700; and with deviation of servers' runtime set to 50, 100, 150 and 200. Thus we had $4 \times 6 \times 4 = 96$ configurations.

In *Figs. 3, 4, 5*, and *6* are shown the results of these simulations. In each figure one can see the failure rate percentage (number of deadline misses per all runs) as the function of the deadline threshold, and for each model. The four figures show the different results of the deviations of the servers' runtime set to 50, 100, 150 and 200.

First we have to mention that the clients restart time and the average runtime of both the server and the client side have the same value (100). Thus the average repetition-time for a client is 300 in a normal situation, and for 200 of it the client does not need any server, in other words, the probability of some client to need a server is $p = \frac{1}{3}$. The probability of the situation that every server is busy is

$$P\{\text{Every server is busy}\} = 1 - \sum_{k=0}^{4} \binom{10}{k} \left(\frac{1}{3}\right)^k \left(1 - \frac{1}{3}\right)^{10-k} = 0.2172.$$

That is, in more than one fifth of the situations the system is overloaded, and this is what we want to investigate.

[1] We could make it a Gaussian distribution too, but it would be an equivalent of setting the deviation of the servers' runtime with other values, for the sum of Gaussian distributions is also a Gaussian distribution.

In the figures one can see that each model gives almost the same results when the deadline threshold is very low (equals the expected value of the cumulated runtime of server and client). As the threshold grows, the failure rate gets lower and lower, as we expect. What we should look at is the rate of decrease in each model.

The simplest one (**A**) has very good results. The **B**s are worse, but there is an explanation for it. In the case of **A** if a server is to miss the deadline, it does not care, just finishes, and gives the results to the client. In the **B**s, however, when it is to miss, it cancels the contract, and the client has to find a new server (it costs additional time), and the new has to complete the task from the very beginning, in the average case in 100. And the cumulated time can be much greater than in the case the original server did not cancel. And if the second one is to miss too, the avalanche starts.

The difference between **B1** and **B2** is simple: if the distribution of the probability of cancelling is uniform, then it is more likely to happen sooner than in the case of the growing linear distribution, thus the cumulated runtime is lower.

The most important result is that the most sophisticated and smart protocol, **C**, where the servers know the deadline threshold, has better failure rates than the **A** variant. This shows us that the contract based protocol model, when used properly, can give better results than the brute force method, and promises that the way we chose has certain advantages even in this preliminary form. This latter variant could be more efficient, if the server took into account how much the deadline miss will be, and instead of cancelling it would first warn the client.

We have to point out that there is no real quality of service represented in the model yet, and that the incomplete results of the cancelling server are not used by the consequent servers, which could also lower the cumulated runtime, and thus the failure rate.

4. The Possibilities of Development

There are two ways for us now: either to improve the simulation, or to use queuing theory models and analyse the results, according to the conclusion of the previous chapter. We shall look at the former now.

For future development we see that the system is easily extendible now with objects implementing more advanced protocols. First we thought of the extension of the `Supplier` and `Customer` implementations. The extension of the `Blackboard` can wait, for it is useful enough in our simulation (of course we need to improve it in the far future). It is the offers now that need serious improvements above all. We have to include the quality of service in the protocol, and measure its degradation under overloaded situations.

We can make more than two level hierarchy as well, to see what the consequences of avalanches are.

At the time, however, we can state that the simulation environment is stable,

offers good results, and shows even in this early stage the benefits of the contract based model.

Finally we have to admit that we realized that our problem is a special queuing problem. It seems to be worth to study the matured models and tools used in telecommunication for serving competing users (subscribers), and try to adapt them for our case.

5. Summary

In this article we introduced a contract-based model for object oriented design in real-time systems. We showed the basics of the protocol.

We introduced a simulation environment as well, which is intended to show the benefits and drawbacks of the above model. We tested a minimal implementation and analysed the results of the tests. It turned out, that even this minimal version has a cute degradation under overload and well chosen protocol implementation, and it promises us that there are more benefits that can be used in real life projects and systems as well. We even realised the convergence of our work with that of the matured models of telecommunications industry.

Finally we pointed out the possible steps of development and improvement, first of all the inclusion of quality into the protocol implementation to see its degradation under overload, and the effects of avalanches in a multilevel architecture.

References

[1] GAMMA, E.–HELM, R.–JOHNSON, V. R., Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1994.

[2] LANGE, D. B.–OSHIMA, M. Programming and Deploying Java Mobile Agents with Aglets. Addison-Wesley, 1998.

[3] NATARAJAN, S.–ZHAO, W., Issues in Building Dynamic Real-Time Systems. *IEEE Software*, pp. 216–227, Sept. 1992.

[4] PÁLOS, CS.–KONDOROSI, K., A Contract-based Architecture for Active Objects in Real-TimeSystems. In *Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'96)*, 1996.

Appendix

The results of the four protocol variants. The variants were as follows:

- **A**: This variant is the plain one. The servers do not know about any deadline. They offer best effort quality of services. It is our reference model for it uses almost no protocol.

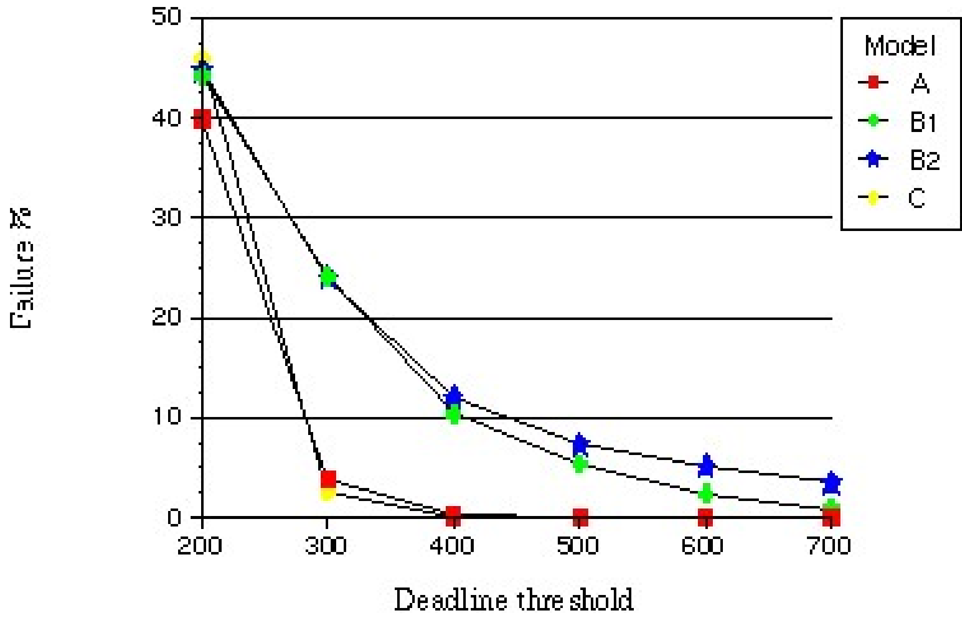

Fig. 3. Failure percentage when server runtime deviation is 50.

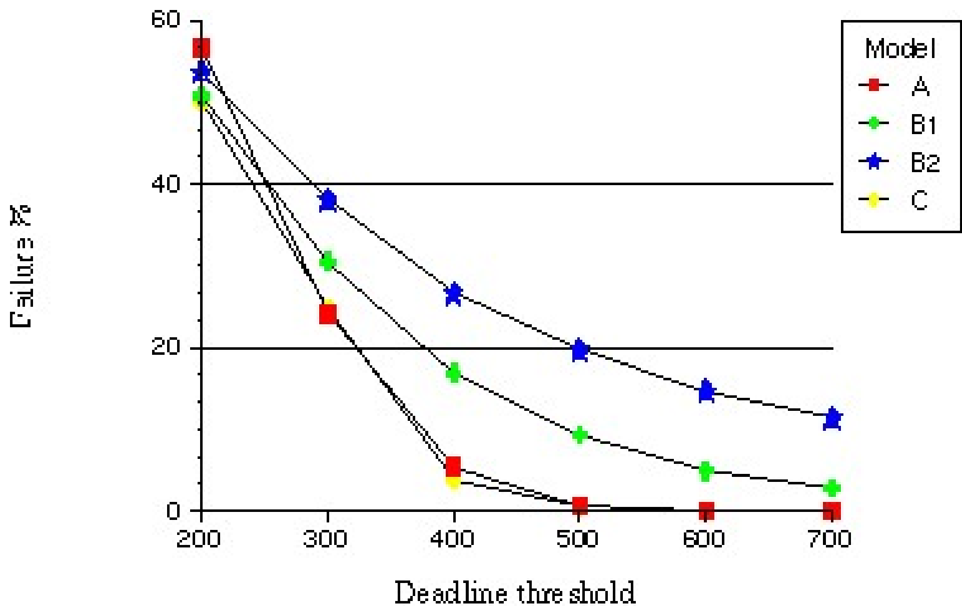

Fig. 4. Failure percentage when server runtime deviation is 100.

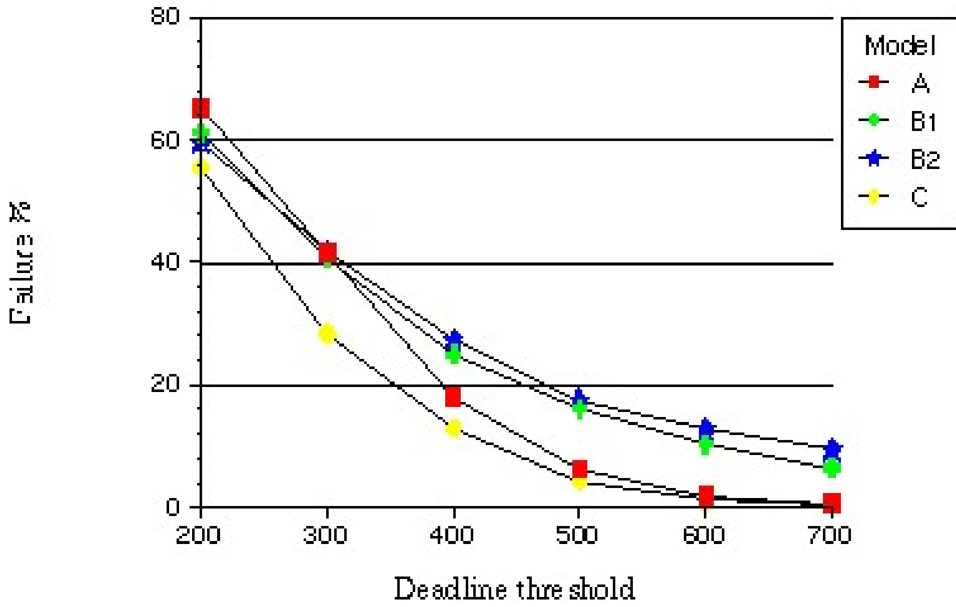

Fig. 5. Failure percentage when server runtime deviation is 150.

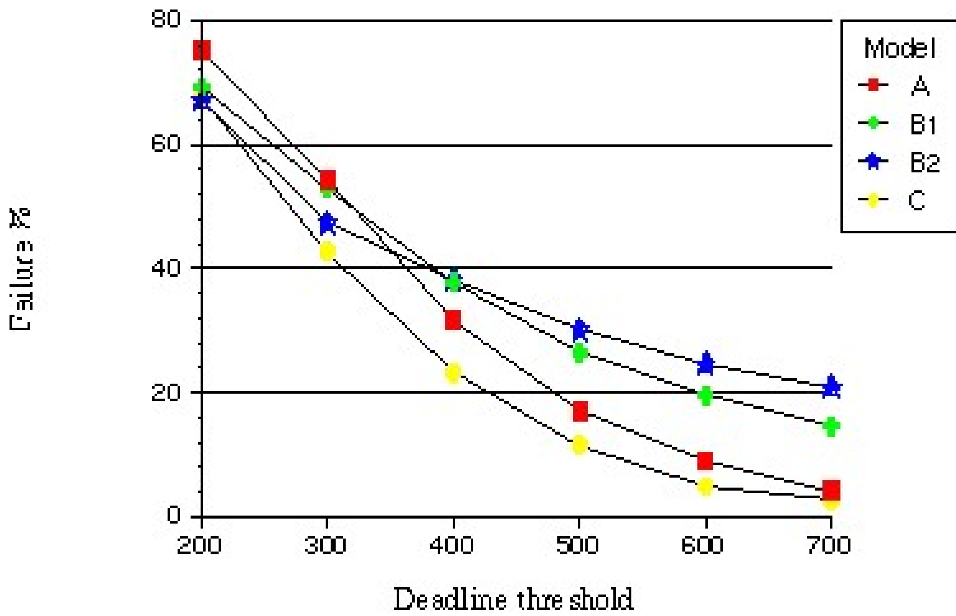

Fig. 6. Failure percentage when server runtime deviation is 200.

- **B1**: In this model the servers offer services with a deadline of 100, and if they miss it, they will cancel their contract, and the client will have to find someone else to finish. The probability of when the servers realize they miss the deadline has uniform distribution.
- **B2**: This variant is the same as the previous one, except that the distribution of the servers' realization is growing linear (approaching the deadline they may realize with greater probability they will miss it).
- **C**: This variant is the most sophisticated of the four. It is basically the same as **B1**, but here the servers know what the real deadline threshold is, and do cancel according to it. Thus, if the threshold is higher, they know they have more time to complete.