# Overload Protection for CORBA Systems with Time Constraints

Widell, Niklas; Nyberg, Christian; Kihl, Maria

2002

# Overload Protection for CORBA Systems with Time Constraints

Niklas Widell, Christian Nyberg and Maria Kihl
Department of Communication Systems
Lund University
Box 118
SE-221 00 Lund, Sweden
Phone: +46 46 2227195
{niklasw,cn,maria}@telecom.lth.se

**Abstract**

Scalable and reliable distributed object-oriented computing (DOC) middleware systems is an important technology in, for example, telecommunications service logic and distributed web servers. The Common Object Request Broker Architecture (CORBA), developed by the Object Management Group (OMG) is a specification of a common platform for DOC systems. CORBA acts as middleware, by inserting itself between the Operating System (OS) layer and the Application layer on a host. CORBA provides support for transparent interaction of objects situated on different nodes. The original CORBA specifications had no support for timing constraints in applications and very little support in the terms of performance optimizations. Present extension to CORBA include support for real-time applications and a number of performance enhancements such as load balancing. However, no work so far address the issue of overload in a CORBA system. This paper presents a discussion of overload issues in distributed CORBA systems with time-constrained tasks. First a performance model of a CORBA system is introduced. Second, overload in distributed CORBA systems is discussed. Third, a number of classic overload protection mechanisms are applied to the performance model and investigated using simulation. The simulations show that even by using very simple protection mechanism, a good throughput can be achieved.

## 1 Introduction

Scalable and reliable distributed object-oriented computing systems like CORBA are being used to implement, for example, telecommunications service logic and distributed web servers. As the systems become larger, it becomes more difficult to predict the needs of the applications for dimensioning purposes, in particular since the needs may change during the execution of the application.

Depending on the time constraints put on a system, it can be classified as either real-time or best-effort. A real-time system is a system where the correctness of a task depends not only on the logical result of the task, but also on the time at which the results are produced. A best-effort system is one where the time it takes to complete a task has no impact on the correctness of the execution.

In general, a task in a real-time system has a deadline associated with it. The deadline is the time when the task is supposed to be finished. A real-time system can be further classified into being either *hard* or *soft* real-time. In a hard real-time system all deadlines must be met and the failure to meet a deadline may have disastrous results. A hard real-time system need not be fast, but it must be very predictable. In the soft real-time case occasional timing violations may occur, as long there is no system failure. A typical performance criteria for a soft real-time system is to minimize the number of missed deadlines. Jensen [1] provides a discussion of soft real-time systems.

Soft real-time systems include systems where full predictability is not possible, or not even necessary. It may be that the timing of the completed task is important, but the overhead introduced by real-time operations and/or complexity of the system may prohibit the use of a full real-time environment. Further, there might be a dynamic request pattern, as arrival times for new tasks is not known.

An e-commerce site may be regarded as a soft real-time system, even though it is not implemented using a real-time infrastructure. The traffic to the site is not likely to be known with great accuracy. This means that the system may become overloaded, causing long response times. If a response from a server takes to long to reach a client it may mean that the client leaves the site without making any purchases, resulting in a loss of revenue for the site. In a market-place where a customer can chose between many different e-commerce sites, poor performance can be disastrous for a site. Also, for an e-commerce site, it is the *user-experienced* performance that matters, and nothing else.

CORBA is a suitable platform for soft real-time applications such as e-commerce systems. In order to improve performance, support for load balancing, migration of objects and distributed scheduling exist in several different implementations of CORBA. An object distribution mechanism decides how objects are distributed on physical nodes, taking into account the possible migration of objects from one node to another. A load balancing mechanism attempts to distribute the workload so that the system performs as efficiently as possible. These facilities allow given systems to work with better efficiency and to approach soft real-time capabilities.

In many cases load balancing and object distribution mechanisms are not sufficient to guarantee the timing requirements of a system. These mechanisms are only sufficient when the workload caused by arriving tasks is well below system capacity. In, for example, web servers and telecommunications service logic, tasks may arrive at so high rate that system capacity is exceeded and the system becomes overloaded. If no overload protection is used in this case, task throughput suffers and overloaded servers cause response times to grow above acceptable levels. The fundamental observation for overload protection is that in some cases it is better to prevent some tasks from entering a system, than to let all tasks enter and thereby let all tasks experience poor QoS.

Several papers have discussed load balancing and load sharing in computer networks. Othman *et al* [2] presents the load balancing scheme used in the TAO real-time implementation of CORBA. The Realize Resource Management system described in Melliar-Smith *et al* [3] describes a full system that takes care of the run time needs of real time distributed applications. Realize supports replication for fault tolerance, migration and dynamic scheduling of objects using an off-the-shelf ORB. Kreimen *et al* [4] discussed load sharing algorithms for distributed systems. Kunz [5] examined how the network nodes should exchange load status information to be used in load balancing schemes.

Overload protection has been studied extensively for telecommunication systems,

see for instance Körner *et al* [6] for a survey of early results. Berger [7] compares the efficiency of two classic protection schemes, call gapping and percent thinning. The ACTS project MARINER has published extensively on using market-based agent technology to protect distributed systems in Intelligent Networks, see [9]. Kihl [10] describes a simple throttle scheme for the Telecommunications Information Networking Architecture (TINA). TINA is an architecture for next generation telecommunication networks, see the TINA Consortias home page [8]. Jordan *et al* [11] investigated call admission policies for communication networks that support several services. In the ACTS project MARINER CORBA-based IN systems were investigated and some results on the use of agents for load balancing in CORBA system can be found in Conor *et al* [12]. Also in [12] is an investigation in how to assign a number of objects to to a number of processors in order to minimize, for example, the mean session completion time.

Other than Kihl [10] and Rumsewicz [13], no research has been performed in the area of overload protection schemes for distributed object oriented systems. The reason for this is probably that overload control has not been deemed necessary for the systems under study, due to the assumption that the offered workload is not above a certain acceptable limit. However, for some systems, typically where the resources are used by "outsiders", such as CORBA based web servers, the overload problem can be very serious and must therefore be studied. While some results from classic telecommunication research can be used, new problems, such as how to identify overload conditions in a distributed system and how to react to overload when it has been identified in a way that still provide good service to the users etc. remains to be investigated.

This paper focuses on the introduction of overload protection in CORBA systems with timing constraints. The main objective is for the system to preserve the user-experienced QoS during short periods of overload. The paper is organized as follows: Section 2 provides background on CORBA. Section 3 introduces a performance model of a distributed CORBA system. Section 4 discuss the philosophy behind overload protection. Section 5 mentions how overload protection may be implemented in CORBA. Section 6 describes a number of simulation cases using the performance model together with some classic protection mechanisms. Section 7 summarizes the work.

# 2 CORBA

CORBA is an open standard for distributed object computing under development by the Object Management Group [16]. By the use of standards, CORBA aims to provide independence from programming language, computing platform and communication medium.

A CORBA platform provides many necessary functions for distributed processing, such as object registration, location and activation, parameter passing among others. CORBA defines a set of mechanisms that allow a client object to invoke a method on a server object on a remote node with full location transparency. Location transparency means that neither the client object nor the server object needs to know anything about on which node they are executing, since the CORBA infrastructure hides the distribution from the application.

An object in CORBA is an encapsulated entity with a distinct identity whose services can only be accessed through well defined interfaces. Thus an object's services and how the services are implemented is separated. The interfaces of an object are specified in the *Interface Description Language, IDL*. The IDL is then used to generate

stubs and skeletons in a some programming language, such as C++ or Java. The stubs are used by a client object to make method invocations to a server object. The server uses the skeleton to implement the service.

The CORBA Object Request Broker (ORB) acts as a message bus to provide seamless interaction between client and server objects. Using the General Inter-ORB Protocol (GIOP) and the TCP/IP specific Internet Inter-ORB Protocol (IIOP) heterogeneous ORBS can interoperate.

In the original CORBA specifications there were no support for real-time applications. Real-time CORBA 1.0 [17] adds support for static scheduling of tasks. The proposed Real-time CORBA 2.0 specification will include support for "pluggable" dynamic schedulers.

### 2.1 CORBA performance improvements

A number of features exists in CORBA that improves the performance of applications. These include, for example, object distribution, object migration and load balancing.

The process of distributing objects on multiple nodes is fundamental to CORBA. However, due to the extra processing time necessary to make method invocations between nodes, the distribution of objects can have large impact on performance. A good distribution will improve performance, while a poor distribution will decrease performance.

A CORBA application with a given object distribution can be re-configured to another object distribution during run-time using object migration. However, object migration is a complex and computationally expensive operation and its use under conditions of short term overload remains to be investigated. Object migration makes it possible to change the system if the conditions for the system change. Such changes include new task types being added, new nodes being added or if the traffic pattern of arriving tasks changes.

To make distribution efficient, support for load balancing mechanism is required. Load balancing allows the system to spread the load from arriving tasks to objects on different nodes so that no node has excess spare capacity while another node in the system is overloaded. In CORBA, this means that given a set of distributed objects to chose from, the mechanism should chose an object residing on the node with the least load.

The current load balancing mechanism in CORBA is simple (see Othman *et al* [2]. The mechanism works by sending all requests to one node until that node is overloaded, then the mechanism decides which other applicable node has the least load and sends all further requests there, until that node is overloaded.

## 3 Performance Model

This section presents a performance model of a distributed CORBA system to aid in the discussion of overload protection.

Consider a system containing a network with $m$ nodes. Nodes are labeled $N_1$ to $N_m$. As an example, figure 1 shows a simple five node ($N_1$ to $N_5$) network. To allow for an inhomogeneous network, node $i$ has a relative processing speed $s_i$. If $s_i = 1, 1 \leq i \leq m$, then all nodes have equal processing speed. For the present model, it is assumed that it is the processors that are the performance bottlenecks. In addition, assume that the network is very fast compared to the nodes, and switching

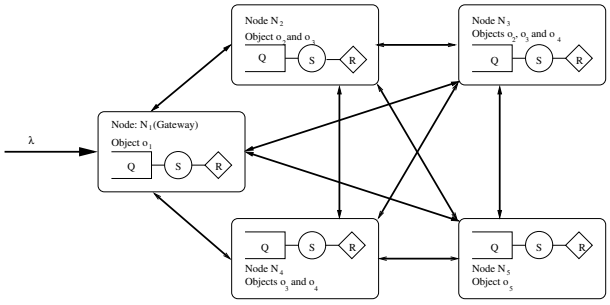


Figure 1: A five node network with five objects

and transmission times are negligible. Therefore the nodes can be considered as fully connected. Each node is modeled as a single server queue. The queuing discipline depends on the scheduling mechanism used. However, for simple analysis a FIFO queue is used.

Distributed in the network are software components called object of $n$ different classes. In the performance model, the set of objects all instantiated from a single class $i$ is labeled $o_i$. Objects in $o_i$ each has a number of services called methods. By saying that $o_i$ is located on node $N_j$, it is meant that the services of class $i$ can be accessed at run-time at node $N_J$. Referring to figure 3, there are five objects sets ($o_1$ to $o_5$) distributed in the network.

For every method $m$ of object $o_i$ invocation time $t_{o_i,m}$ is used to model the time a method invocation executes before returning or making a new invocation to another object. This is a simplification, since the actual method invocation times may vary depending on the application. It is assumed that the behavior of each method invocation is independent of context, meaning that processing time and other resource usage does not depend on why or when the method was invoked.

The process of translating the parameters transfered during a method invocation between two objects is called marshalling for the client object and unmarshalling for the server object. If a method invocation must be transfered over the network, the marshalling/unmarshalling process is modeled as an extra service time $t_m$ for the node with the client and on server object node side an unmarshalling time $t_u$. If both client and server objects are located on the same node, then $t_m = t_u = 0$.

The node at which a new task arrives is called a gateway. There can be as many gateways as there are nodes. The gateway represents the entry point for a task. All communication between system and user passes through the gateway.

Tasks of different types, $T_i$, arrive at the system according to some stochastic process, with a a mean rate $\lambda_i$ . A task $T_i$ is described by: a sequence of method invocations ($SMI_i$), a deadline $d_i$ and a importance $v_i$. $SMI_i$ describes which objects are used and in what order they are used by the task. $d_i$ is the time the task must be finished for it to be considered as useful. $v_i$ is the value of the task to the system, as defined by the implementer. A mission-critical task will have $v_i = \infty$. The value of a task may be used during overload in order to reject tasks with lesser value to give more valued tasks more resources.

A sample task is:

$T_{example} = ([o_1, o_2, o_1, o_3, o_4, o_3, o_1, o_3, o_5, o_3, o_1, o_2, o_1], t_{now} + 0.5sec, 1)$.

To clarify the interaction between objects, figure 2 presents a Message Sequence

Chart (MSC) of $T_{example}$:

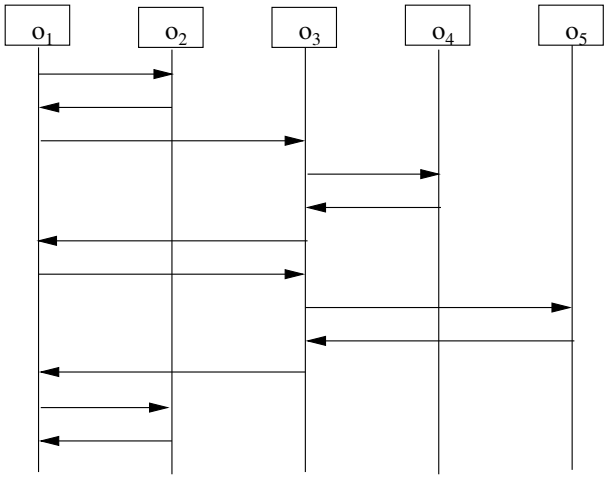


Figure 2: $\tau_{example} = [o_1, o_2, o_1, o_3, o_4, o_3, o_1, o_3, o_5, o_3, o_1, o_2, o_1]$

# 4 Overload Protection Philosophy

This section outlines why overload is dangerous to a real-time system and how overload protection can be used to improve the performance of such a system.

Fundamental queuing theory states that the load, $\rho$, of a system is related to arrival rate $\lambda$ and the task mean service time of $\overline{x}$ as $\rho = \lambda\overline{x}$ [14]. For a single server queue the delay[1] experienced by a task is proportional to $\frac{1}{1-\rho}$. From this relationship it is obvious that keeping load levels down is crucial to prevent large delays. Since a real-time system depends on short and predictable response times to meet deadlines, loads close to one may be catastrophic for the system.

One main observation for overload protection is that in some cases it is better to prevent some tasks from entering a system, than to let all tasks enter and thereby let all tasks experience poor QoS. The objective of an overload protection mechanism is to maximize the amount of useful work which is performed by the system. Useful work is such that is performed to tasks that finish correctly before their deadline. Work done and resources used by tasks that miss their deadlines is wasted. Note that in a soft real-time system deadline misses are acceptable if they are rare, but not if they are common.

The traditional way in which performance has been viewed is from the system operator viewpoint, for instance a telecommunications company. The operator of a system wants maximum capital gain from a given investment, in order to fully utilize the system's available resources. This means that the operator should invest in only enough equipment to support the services offered, while at the same time have enough resources so that no business opportunities are lost due to insufficient resources.

[1]The given relationship is for the Poissonian arrival process, however, for non-deterministic arrival processes the delay typically has a similar behavior

However, with a telecom market moving towards increasing deregulation, and an Internet market where "all sites are equal", the system user viewpoint is becoming more and more important. A user wants quick error-free responses from the system. Good performance is when a requested service is delivered without unacceptable delays.

## 4.1 Types of Overload

The causes for overload can generally be divided into two classes: *resource failure* and *unbounded traffic source*. By resource failure overload it is meant overload that is caused by the system itself, such as by hardware or software malfunction. The failure of a processor, for example, may decrease the capacity so that the offered workload can no longer be finished in time. The failure of a piece of software may cause system instabilities by generating large amounts of erroneous traffic. Unbounded traffic source overload is caused by external sources, such as the users of a system. This type of overload is common in systems where the system itself has no control of the source of the tasks.

While the overload protection schemes discussed in this paper may be applied to both resource failure and unbounded traffic source overload, it is generally aimed at overload caused by the later.

It important to note that overload protection is meant to decrease the problems caused by short periods of overload. If overload continues, then the system is under-dimensioned and other actions are necessary to improve performance.

## 4.2 Overload detection

The detection of overload is a process full of potential pitfalls. The system load may change very quickly and the system itself may change due to the introduction of new task types with new objects, new traffic patterns and so on. The overload protection mechanism must both be ready for instant action, while at the same time being conservative in order to avoid oscillations in order to prevent mechanism instability. In a distributed environment it becomes even harder, since overload on one node not necessarily has any impact on the processing of tasks that do not use the overloaded node. In addition, there is a trade-off between transferring load information often, with resulting communication overhead, or relying on old and possibly inaccurate load information.

There are several system parameters that can be monitored during execution. The following are commonly monitored: resource load, deadline miss rate and system response times. In addition, a profiling system that can make fine-grained measurements on execution times can be used.

The resource load can be the load level of a processor, network element, memory or disk. The resource load is used as a basis for load balancing and object migration mechanisms. However, its applicability as a basis for overload detection in a CORBA environment is hampered by a number of factors. First, overload on node $N_i$ may not have any impact on the processing of tasks that do not use $N_i$. Second, there is a question of *which* measured load that should be used to make the decision, since in a heterogeneous system the same measured load on two nodes with different capabilities may result in very large differences in delay. Third, since the tasks arriving at the system may have widely different service requirements, the variance of the service time may have large impact on the mean service time of a particular node. Finally, if the objective is to satisfy timing constraints, the process of relating a set of measured

loads on a number of nodes to a particular task with its given processing requirements may be non-trivial.

The objective of an overload protection mechanism is to maintain a good throughput of tasks that meet their timing constraints. Thus, seeing how well this objective is met by the system is useful for two reasons. First, as a way to discover overload. Second, as a measurement of how successful the overload protection mechanism is. An increase in the number of missed deadlines would be a good measure that the system is experiencing overload. The ratio of missed deadlines is commonly used in real-time systems as a measurement of success. Lu *et al* has a discussion of Deadline Miss Ratio as overload indicator, see [18]. The main problem with looking at dead-line miss ratio is that it only shows how many deadline misses there are, but gives little warning in the non-overloaded case as to how close the system is to being overloaded. It may therefore fail to adjust to overload until it is to late. To lessen this issue, the mechanism may look at the actual response times of the system and compare these with their respective deadlines.

The response time of the system is a result of load, but also of the variance of the execution times in the system. The main advantage of using system response time instead of load is that the former is what the user of the system sees. For example, a user of an e-commerce site has no knowledge of what the site looks like, how load is distributed, how many nodes are used etc. All that matters for the user is that the *received* QoS is acceptable to the user.

The Realize system [3] includes a Object Profiling System for real-time CORBA, that allows the ORB to monitor execution times of methods, and then feed back this information to the scheduler. The profiling service builds detailed models of how tasks use objects and how long time methods take to invoke.

## 4.3 Overload protection mechanisms

The protective actions to be taken in the event of overload depends on what type of resource that is to be protected and what type of traffic that arrives at the resource. In traditional telecommunications, the most common way to prevent overload from happening is to reject arriving calls according to some algorithm. However, this may not be sufficient in more complex systems.

In the case of resource failure overload, just rejecting tasks is usually counterproductive. Instead, the nodes should try to act to conserve resources by limiting the number of tasks sent.

In the case of unbounded traffic source, depending on the application and how overloaded the system is, the actions may take many forms. Users can be notified that the system is under heavy load and asked to return later (voluntary back-off), offered a simpler service (service adaption) or in the worst case, be rejected. If possible, the mechanism should be such that it discourages reattempts in a short time-scale. If, for example, users make immediate reattempts after failed tasks, the traffic to the system goes up and thus further increasing the load on the system.

Overload protection mechanisms are often designed to operate in three states. First, when there is no overload and the system can finish all arriving tasks without problems, all tasks receive full service. Second, when there is some overload, the system should try to focus on getting as many of the most valuable tasks finished in time, while potentially rejecting tasks with lower priorities. In this state the objective of the overload protection mechanism is to make sure that the throughput of useful tasks remain high. Third, when there is catastrophic overload, the system should concentrate on its own

survival, executing only the most valuable tasks and being very restrictive on what tasks that may enter the system.

The point at which a task is rejected during execution may have great impact on performance. Since the objective of overload protection is to maximize useful work, it is important to reject a task before it wastes to much resources. In a distributed system, this means that, if possible, a task should be rejected at the gateway to the system.

Also it is important that the overload protection mechanism requires low overhead. It must not require too extensive computational resources, making itself a reason for overload.

To measure the success of an overload protection mechanism, tasks can be divided into four classes.

1. Tasks that are rejected at the gateway, that is before they enter the system. This is often called external or network level overload protection.
2. Tasks that are accepted at the gateway but are rejected by a node inside the distributed system. This is often called internal or node level overload protection.
3. Tasks that are not rejected at the gateway or at any other node but are not finished in time, i.e. tasks that fail to meet their deadline.
4. Tasks that are successfully served and finished in time.

The purpose of overload protection is to maximize the number of tasks belonging to class 4. Naturally, the number of tasks of class 2 and 3 should be as small as possible since they are not successful and the processing time spent on them is wasted.

Tasks of class 1 and 2 can be considered as failed operations that must be taken care of by the application or user generating the tasks. Note that there is a risk that if a fault-tolerant system is being used, the system itself may do reattempts, and this interaction must be addressed.

# 5 Implementation in CORBA

This section gives some proposals for the introduction of overload protection in CORBA. The overload protection mechanism can then complement load balancing and object distribution mechanisms as ways to improve performance.

An overload protection function may be divided into two main mechanisms: a set of rules that decides which tasks to reject and a mechanism to clean up the system after a task has been rejected.

The set of rules for task rejection are application specific. Tasks may be admitted independently. For example, simple IN/CORBA services may generate only a single task. Some systems require entire groups of tasks to be successful in order to consider a user's interaction with the system as successful. For example, user interaction session with an e-commerce site requires all tasks to be successful for the session to be considered successful.

The general mechanisms to support the release of resources after a task has been rejected can be implemented by using existing CORBA services. The information of a task rejection can be distributed using exceptions. The Real-time CORBA specification specifies a TASK_CANCELLED exception that can be used. It is obvious that the amount of work necessary to release resources is proportional to the amount of resources used, and also the types of resources used. Stateless objects can be destroyed right away, while objects with states must be handled with more care.

# 6 Simulations

This section describes some classic overload protection mechanisms, a simple feedback loop to make the mechanisms dynamic, and a number of simulation cases that show the efficiency of the mechanisms.

## 6.1 Common overload protection mechanisms

There are two basic forms of overload protection mechanisms that have been used over the years in telecommunications: window based and rate based. These are further described below, with their respective strengths and weaknesses mentioned briefly.

### 6.1.1 Window based overload protection

A window based overload protection mechanism operates by limiting the number of active tasks to the window size $W$. If a new task arrives and there are presently fewer tasks executing than $W$, it immediately continues executing, otherwise it is rejected. The window size is related to the capacity of the system, with one window size giving one system response time.

The window mechanism is robust, since it handles transients well and uses resources efficiently. For the mechanism to work well, the counter of the number of active tasks must be kept updated at all times, for instance by using time-outs to discover failed operations.

The window mechanism used in this paper is similar to the Isarithmic mechanism described by Gerla and Kleinrock [15].

### 6.1.2 Rate based overload protection

A rate based overload protection scheme works by limiting the traffic to the system. The rate-based mechanisms are more susceptible to transients in arrival rates than the window based ones, and control loops updating the algorithm parameters must run on a tighter time scale.

Some variations include (both ar further described in Berger [7]):

**Percent blocking** The percent blocking algorithm admits tasks into the system with a probability $P(admit)$.

**Call gapping** A call gapping algorithm closes for a set amount of time $t_{gap}$ (the gap size). After this interval the next task to arrive is allowed into the system and the throttle is closed again. By varying the gap size the mechanism can set an upper limit to the number of arriving tasks per time unit.

## 6.2 Algorithm control loop

A protection mechanism must be able to react to the changes in arrival rate or task mix in a quick, efficient and stable way. The reaction is typically a change in the parameters of the algorithm.

There are many ways in which algorithm parameters can be updated, using information fed back from the system. In this paper a simple update loop, based on the measured task execution time is used. The assumption is that it is the response time of

the system that is important from the user viewpoint. The loads of the internal nodes is of no concern to the user.

The algorithm classifies the response times $t_{task}$ for tasks into three categories: good, fair or bad.

The algorithm uses the following parameters: an interval length $t$, a user defined deadline $t_{deadline}$ to categorize finishing tasks, an increase condition $c_i$, a decrease condition $c_d$, counters for the task categories $x_{good}, x_{fair}$ and $x_{bad}$, measured task system time $t_{task}$, update results variable $R$. $c_i$ is the ratio of tasks classified as good required for the mechanism to allow more tasks into the system. $c_d$ is the equivalent ratio of tasks classified as bad required to decrease the number of tasks allowed into the system.

1. Time is divided into intervals of length $t$ seconds.
2. During interval $N$, $x_{good}, x_{fair}$ and $x_{bad}$ are updated for each finishing tasks as follows:

| Measured $t_{task}$ | Updated parameter |
|---|---|
| $0 \leq t_{task} \leq 0.5 \cdot t_{deadline}$ | $x_{good} := x_{good} + 1$ |
| $0.5 \cdot t_{deadline} \leq t_{task} \leq t_{deadline}$ | $x_{fair} := x_{fair} + 1$ |
| $t_{deadline} < t_{task}$ | $x_{bad} := x_{bad} + 1$ |

3. At end of interval $N$, calculate parameters for interval $N + 1$. Set $R = 0$.
   (a) If $\frac{x_{good}}{x_{good}+x_{fair}+x_{bad}} > c_i$ then $R := R + 1$.
   (b) If $\frac{x_{bad}}{x_{good}+x_{fair}+x_{bad}} > c_d$ then $R := R - 1$.
   (c) Update algorithm parameter according to table (note that $R = 0$ if both (a) and (b) are true above):

| Parameter | $R = 1$ | $R = 0$ | $R = -1$ |
|---|---|---|---|
| $P_{N+1}(admit)$ | $P_N(admit) + 0.01$ | $P_N(admit)$ | $P_N(admit) - 0.01$ |
| $t_{gap,N+1}$ | $t_{gap,N} \cdot 0.95$ | $t_{gap,N}$ | $t_{gap,N} \cdot 1.05$ |
| $W_{N+1}$ | $W_N + 1$ | $W_N$ | $W_N - 1$ |

In all cases there are max and min values that the parameters cannot go beyond.

## 6.3 Simulation parameters

The sample system is a model of a five node network ($N_1 - N_5$) with six object sets $o_1 - o_6$ . Two different tasks, $T_1$ and $T_2$ arrive at the gateway node each according to a Poisson process with rates $\lambda_1$ and $\lambda_2$ respectively.

The tasks are defined by:

$T_1 = ([o_1, o_2, o_1, o_3, o_6, o_3, o_1], t_{arr} + 0.5sec, 1)$

$T_2 = ([o_1, o_2, o_1, o_3, o_4, o_3, o_1, o_3, o_5, o_3, o_1], t_{arr} + 0.5sec, 1)$

where $t_{arr}$ is the arrival time of the task.

Table 1 summarizes other simulation parameters used.

| | Value |
|---|---|
| Marshalling time, $t_m$ | 1ms |
| Unmarshalling time,$t_u$ | 1ms |
| | |
| Increase condition, $c_i$ | 0.50 |
| Decrease condition, $c_d$ | 0.05 |
| Length of update interval | 1s |

Table 1 Parameter settings

Table 2 below contains object distribution and respective probabilities used for load balancing. A dash means that this type of object is not available on this node. The object distribution is assumed to be static, with no object migration or further replication taking place. A random load balancing scheme is used, with the probabilities given in the table.

For example, a task of type $T_1$ is executing in the system, and an object in $o_1$ is about to make a method invocation on an object in $o_3$ for the first time of the task. The random load balancing scheme with the probabilities shown distributes invocations to object set $o_3$ evenly on nodes $N_2, N_3$ and $N_4$. Say that $N_2$ is chosen. Then all further invocations to $o_3$ will also be sent to $N_2$.

| *Object* | $N_1$ | $N_2$ | $N_3$ | $N_4$ | $N_5$ | $t_{o,m}$ |
|---|---|---|---|---|---|---|
| $o_1$ | 1.0 | — | — | — | — | $1ms$ |
| $o_2$ | — | 0.50 | 0.50 | — | — | $4ms$ |
| $o_3$ | — | 0.33 | 0.33 | 0.34 | — | $2ms$ |
| $o_4$ | — | — | 0.5 | 0.5 | — | $8ms$ |
| $o_5$ | — | — | — | — | 1.0 | $10ms$ |
| $o_6$ | — | — | — | 1.0 | — | $3ms$ |

Table 2: Object distribution and probabilities used for load balancing.

## 6.4 Simulation cases

Three performance perspectives were investigated. First the impact of the simple overload protection mechanisms on the throughput was compared to the case when no protection was used. Second, the algorithms were tested for fairness. Fairness was considered to be how available resources were used when the mix of tasks arriving was changed. Third, the system's reaction to traffic transients was investigated.

In all cases a Poissonian input process was used.

### 6.4.1 Case A: Throughput

The throughput of successful tasks was investigated. Successful tasks are tasks that fall into category 4 as described in section 4.3 above. The throughput of successful tasks is what is usually called *Goodput* [7].

75% of the tasks were of type 1 and 25% of the tasks were of type 2.

### 6.4.2 Case B: Fairness

In this case the arrival rate $\lambda$ was kept constant, but the task mix was changed. This meant that the load on the system changed, since the two task types had different processing requirements.

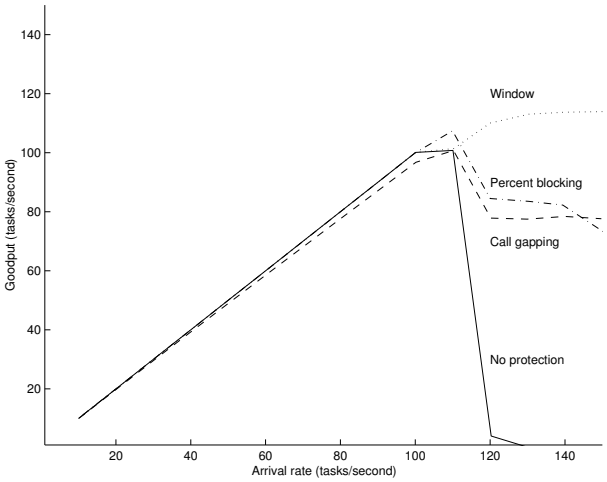


Figure 3: Throughput of succesful tasks ("Goodput") as function of arrival rate.

The objective of the investigation was to see whether the overload protection mechanism had any impact on the mix of tasks leaving the system.

Arrival rate was kept constant at $\lambda = 120s^{-1}$. The task mix was varied from 0% to 100% of task 1 and the rest of task type 2.

### 6.4.3 Case C: Traffic transient reaction

In this case the mean response time for the system was investigated when a traffic transient increase appeared. The task mix was kept constant, with 75% of the tasks were of type 1 and 25% of the tasks were of type 2. A period of 600 seconds were simulated. For the first 200 seconds, the arrival rate was kept at $\lambda = 100^{-1}$. At time 200s the arrival rate was changed to $\lambda = 120^{-1}$, which made the system overloaded. At time 400s the arrival rate was changed back to $\lambda = 100^{-1}$.

## 6.5 Simulation Results

### 6.5.1 Case A: Throughput

Figure 3 shows the throughput of successful tasks (goodput) as a function of total arrival rate $\lambda$. Throughput is severely degraded for the case with no overload protection (solid line), while call gapping (dashed line) and percent blocking (dash-dotted line) are very similar, with call gapping being slightly better than percent blocking at higher rates. The window mechanism (dotted line) gives the best throughput. All three protection algorithms are much better than a system without any protection.

The reason for the window mechanism being more successful is that it is less sensitive to small statistical fluctuations than percent blocking or call gapping. With a given window size, the number of tasks allowed into the system is kept at a level that the system can handle, given its timing constraints.

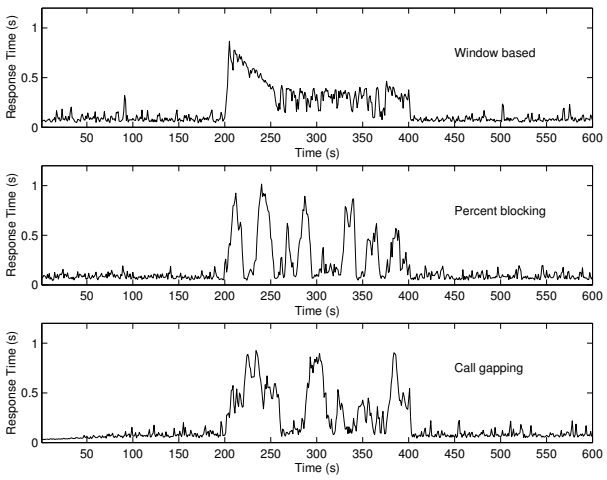


Figure 4: Mean response time for transient case

### 6.5.2 Case B: Fairness

The results from the simulations were the same for all three mechanisms. The throughput of successful tasks changed, but the ratio of arriving tasks of type 1 and exiting tasks of type 1 was constant.

### 6.5.3 Case C: Traffic transient reaction

In this case the transient characteristics of the system was investigated. Figure 4 shows the mean response times, measured over 1 second long intervals, for one realization of the simulation. All tasks that finish, whether they meet their deadline or not, are included in the mean response times calculated for each interval.

The diagrams show that the window based mechanism seems to be more stable than percent blocking or call gapping. All mechanisms show a large increase in the beginning of the transient. The time it takes for the mechanism to establish a new steady state depends on how fast the control loop is. In the window based case, the response time decreases as fast as the loop allows it until it ends up on a new mean response time, which is now slightly below the deadline at 0.5 seconds. Both call gapping and percent blocking never attain any new steady state during the transient. A better transient response for call gapping and percent blocking could probably be achieved with better tuned parameters.

# 7 Conclusions

This paper argues for the introduction of overload protection in CORBA systems with time constraints. The main reason to use overload protection is to prevent complete throughput degradation when the load is high. This can be very important when dimensioning systems in an environment where budgets are limited or where the accuracy of traffic estimates is poor.

Three classic simple overload protection mechanisms are added to a CORBA system in this paper. Simulations show that all three mechanisms prevent a complete throughput degradation that happens when no protection is used. To control the mechanisms a control loop is used. The control loop uses comparison of measured response times and the deadlines of the task to update the parameters of the mechanisms. Using the measured response time is useful for two reasons. First, it is what the user of the system sees, and therefore the experienced quality of the system is used to control the system. Second, CORBA systems can be very complex, with many nodes, many objects and quickly changing traffic patterns, where traditional load measurements are both difficult and error-prone.

Simulations show that a window based mechanism performs better than both percent blocking and call gapping. The window based mechanism has both better throughput at high loads and is less sensitive to sudden traffic increases.

# 8 Acknowledgements

This work has been partially sponsored by the Swedish Research Council for Engineering Sciences (TFR) under contract 271-97-203.

# References

[1] E. D. Jensen,*Eliminiating the 'hard'/'soft' real-time dichotomy*, Computing and Control Engineering Journal, February 1997.

[2] O. Othman, C. O'Ryan and D. C. Schmidt, *The Design of an Adaptive CORBA Load Balancing Service*, Distributed Systems Engineering Journal, April, 2001

[3] P. M. Melliar-Smith, L. E. Moser, V. Kalogeraki and P. Narasimhan, *Realize: Resource Management for Soft Real-time Distributed Systems*, Proceedings for the IEEE Information Survivability Conference, SC, January 2000.

[4] O. Kreimen and J. Kramer, *Methodical analysis of adaptive load sharing algorithms*, IEEE Transactions on parallel and Distributed systems, Vol. 3, No. 6, Nov. 1992.

[5] T. Kunz, *The influence of different workload descriptions on a heuristic load balancing scheme*, IEEE Transactions on Software Engineering, vol. 17, no. 7, July 1991.

[6] U. Körner and C. Nyberg, *Overload Control in Communication Networks*, GLOBECOM '91, Phoenix 1991.

[7] A. W. Berger, *Comparison of Call Gapping and Percent Blocking for Overload Control in Distributed Switching Systems and Telecommunication Networks*, IEEE Transactions on Communications, vol. 39, no.4, April 1991.

[8] TINA Consortia's homepage: www.tinac.com

[9] ACTS Project AC333 MARINER, *Deliverable 9: Project Final Report*, Dublin, 2000.

[10] M. Kihl, *On overload control in a TINA network*, 6th IEE Conference on Telecommunications, Edinburgh, Scotland, 1998.

[11] S. Jordan and P. Varaiya, *Control of multiple service, multiple resource communication networks*, Proceedings of Infocom'91, Bal Harbour, Florida, 1991.

[12] C. McArdle, N. Widell, C. Nyberg, E. Lilja, J. Nyström and T. Curran, *Simulation of a Distributed CORBA-based SCP*, IS&N 2000, Athens, Greece, 2000.

[13] M. Rumsewicz, *Load Control and Load Sharing for Heterogeneous Distributed Systems*, Proceedings of ITC 16, Edinburgh, Scotland, 1999.

[14] L. Kleinrock, *Queuing Systems Volume 1: Theory*, Wiley-Interscience 1975.

[15] M. Gerla and L. Kleinrock, *Flow Control: A Comparative Survey*, IEEE Transactions on Communications, VOl. 28, No. 4:553-574, 1980.

[16] Object Management Group's homepage: www.omg.org

[17] Object Management Group, *Real-time CORBA 1.0*, 1998.

[18] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. Son and M. Marley, *Performance Specifications and Metrics for Adaptive Real-Time Systems*, Proceedings of the 21st IEEE Real-Time Systems Symposium, 2000.