Patterns in Distributed Real-Time Scheduling

Lisa Cingiser DiPippo, Victor Fay-Wolfe,
Jiangyin Zhang, Matthew Murphy,
Priyanka Gupta
The University of Rhode Island
Kingston, RI USA 02881
{dipippo, wolfe, zhang, murphym,
guptap}@cs.uri.edu

Abstract

This paper describes patterns in distributed real-time scheduling. It begins with an overview of the patterns in the paper, and a view of the pattern language made up of the patterns described. The paper goes on to describe each of the patterns in Alexandrian form.

1 Introduction

Many real-time distributed applications such as avionics, on-line stock trading, and military command and control systems require that scheduling be performed across the system to ensure that the quality of service (QoS) requirements specified by the application are met. Middleware in a distributed real-time system (DRTS) can provide the infrastructure and mechanisms required to perform the necessary scheduling. In such a distributed system, each endsystem may provide its own local scheduling mechanisms. However, the middleware across the system must take into account the entire system, and provide coordinated scheduling information to the individual endsystems, which will enforce the globally determined scheduling decisions. The global scheduling decisions that are made in a DRTS include where to allocate service requests, how to provide scheduling information to local endsystems, and how to handle overload that can cause QoS failures in the system. For example, in a command and control application in which sensor information must reach a decision-maker, and then possibly a shooter, global end-to-end QoS requirements must be coordinated by the middleware, and enforced on the endsystems involved.

From this type of application, distributed real-time scheduling patterns have emerged. While many patterns may be applied in performing scheduling at the distributed level, this paper concentrates on those patterns that specifically involve global decision-making that considers the entire distributed system. In this section of the paper we give a brief overview of the distributed real-time scheduling patterns. We also provide a view of the relationships among these patterns through a pattern language. Sections 2-4 describe the pattern details in Alexandrian [1] form. Section 5 concludes by summarizing and by placing the patterns described here into a larger context with other patterns in distributed real-time systems.

1.1 Patterns Overview

The distributed real-time scheduling patterns described here are: *Service Request Binding*, *Local Enforcement*, and a group of overload management patterns. *Service Request Binding* allows middleware to match a request for service by an application with a provider of the service using an analysis of the QoS specifications of the requestor, and the QoS capabilities of the provider. *Local Enforcement* provides a way for globally determined scheduling decisions to be enforced on local endsystems with varying implementation mechanisms.

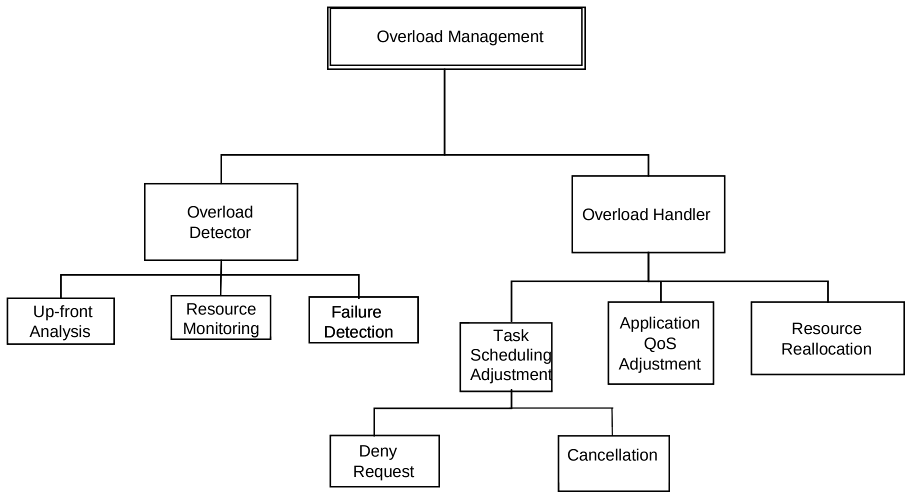


Figure 1 - Overload Management Pattern Hierarchy

The overload management patterns discussed here are a group of related patterns, as depicted in Figure 1. Overload is defined to be a situation in which QoS timing specifications cannot be met. For example, if an endsystem has been scheduled to execute a set of tasks for which it cannot meet the deadlines of all of the tasks, overload has occurred. Further, in an end-to-end application overload on one endsystem can propagate to other endsystems involved in the application if a task on one endsystem is dependent upon another task on an overloaded node. The overload management patterns described here are divided into two types. Figure 1 depicts the categorization of the overload management patterns. Overload detection involves determining when and where overload has or might occur. There are three specific patterns in this category: *Up Front Analysis*, *Resource Monitoring*, and *Failure Detection*. *Up Front Analysis* detects overload before it occurs by performing a priori analysis whenever a new request for resources enters the system. Alternatively, *Resource Monitoring* detects overload by analyzing system conditions as they are occurring. *Failure Detection* detects overload after a QoS failure has occurred.

Overload handling involves determining how to respond to the detection of overload. This paper describes three ways in which overload can be handled: *Task Scheduling Adjustment*, *Application QoS Adjustment*, and *Resource Reallocation*. *Task Scheduling Adjustment* handles overload by adjusting the task scheduling parameters, such as priority, deadline and importance. This can be implemented by *Deny Request*, which does not allow execution of a request that could potentially cause overload. Task scheduling adjustment can alternatively be implemented by *Cancellation*, in which one or more tasks has its scheduling parameters adjusted in order to relieve the overload. *Application QoS Adjustment* allows the application to specify how to handle the overload. *Resource Reallocation* handles overload by moving execution of a task on an overloaded node to another, less loaded node.

1.2 Pattern Language

The pattern language that illustrates the relationships among the distributed real-time scheduling patterns described in this paper is depicted in Figure 2. These relationships are further explored in the full descriptions of the patterns (Sections 2-4).

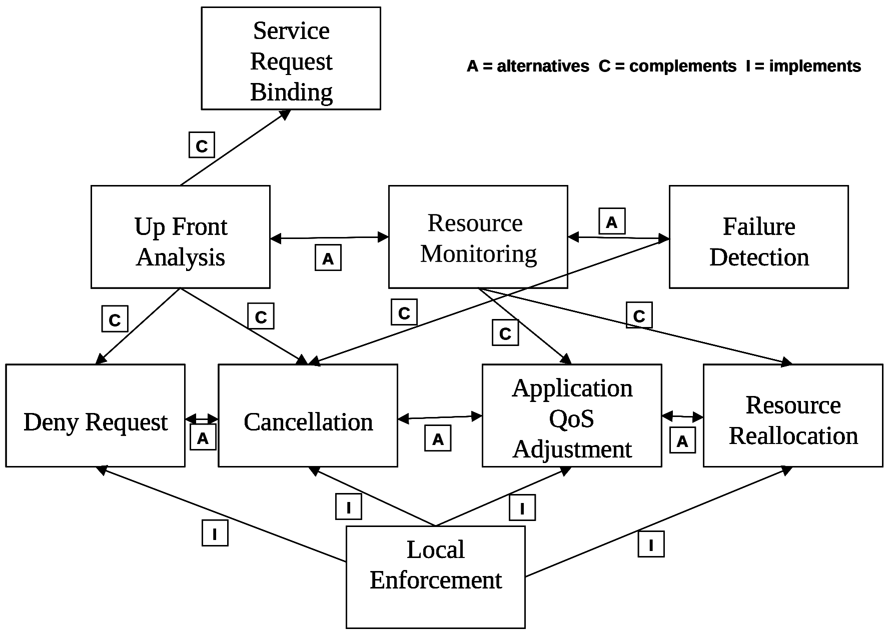


Figure 2 - Distributed Real-Time Scheduling Pattern Language

2 Service Request Binding Pattern

In a DRTS, it is possible that several specific services may be offered by more than one provider. An application that requires such a service must have some way of finding the location of the provider that will best meet its Quality of Service (QoS) needs. The *Service Request Binding* pattern provides a solution that allows applications to “find” the appropriate service provider to which to bind.

◊◊◊

When more than one provider offers a requested service, a choice must be made as to which provider will best meet the requirements of the requesting application.

One of the challenges of large-scale distributed real-time systems involves the need for entities to learn about each other. In client/server systems, the client must find a server from which it can request service. Further, the client may impose timing QoS requirements on the response from the server. Similarly, in a real-time multi-agent system, certain agents provide services that other agents may require, with specified QoS requirements. Consider a simple example in which a client wishes to print a document within a specified amount of time in a network with more than one printer. Instead of the client having to explicitly be aware of all printers and their current capabilities (e.g. pages per minute, color/no color, current queue length), the client should be able to inform the middleware of its service requirements, including timing requirements, and then allow the middleware to choose the appropriate printer. This both simplifies the client development and enhances system flexibility.

In general, an application in a DRTS may require that some service be performed on its behalf and respond within a specified deadline. If there is more than one provider that can offer the required service, then the application should make the request to the provider that can best meet the deadline of this application, while considering other execution in the system as well. This does not necessarily mean that the application should request the "fastest" service or from the provider with the lightest load. This decision should be made in the context of the entire system by considering the QoS requirements of all applications and the QoS capabilities of all nodes in the system.

Provide a middleware mechanism where service providers register their QoS capabilities, and applications use the mechanism to request their required QoS. The middleware analyzes the requests in the context of the entire distributed system and then binds each application request to the provider that it determines to best meets the overall QoS requirements of the system.

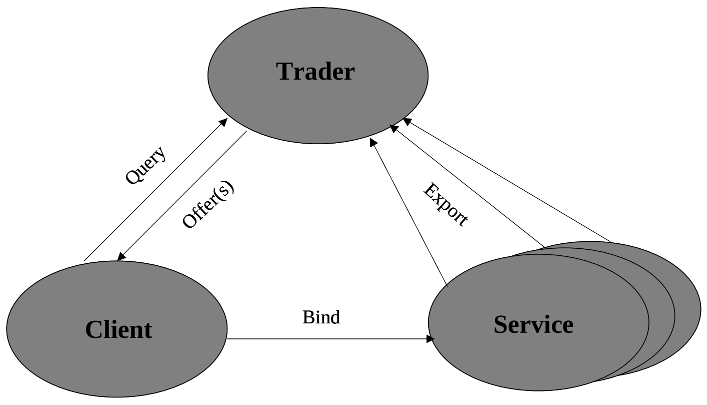


Figure 3 – Service Request Binding Pattern

◊◊◊

Figure 3 illustrates the Service Request Binding pattern where a Trader service is used to provide bindings between clients that have requests with QoS requirements and servers that provide services. Some instances of this pattern include [2], [3], and [4]. The *Service Request Binding* pattern uses the *Up Front Analysis* pattern to make decisions about good bindings. *Up Front Analysis* can determine whether or not a particular binding choice will cause a node in the system to become overloaded. The *Service Request Binding* pattern is also related to the *Resource Reallocation* pattern in that initial bindings may meet requirements as they are known at the time of a client's request. However, as requirements change, or the system changes, reallocation may be necessary.

3 Local Enforcement Pattern

Optimal distributed scheduling often requires the use of global scheduling information and coordinated scheduling enforcement throughout the distributed system. Final scheduling enforcement is usually made by endsystems using techniques that are local to each endsystem. Unfortunately, given the heterogeneity of possible endsystems in a distributed system, there is usually no single notion of local enforcement. For instance, many endsystems use a form of priority-based local enforcement, but even these endsystems often have different valid priority ranges and cardinality. The *Local Enforcement* pattern allows the middleware to transform globally-determined scheduling information for a task into scheduling information that is appropriate for the local endsystem on which it executes.

◊◊◊

Global scheduling decisions in a real time distributed system must be enforced on local endsystems that may have heterogeneous local scheduling enforement techniques.

Endsystems use scheduling information to order task execution, but the implementation details vary among endsystems. Many systems assign all tasks a given priority within some valid priority range. For example, RT CORBA 1.0 allows for 32,767 different priorities, but the real-time operating systems that lie on the individual endsystems may allow far fewer priorities [5]. This problem is not constrained to endsystems, but extends to other local scheduling enforcement points as well. For example, Differentiated Services (diffserv) routing in IP networks allows packets to maintain priority across the network, but only allows up to 64 possible priority values, and usually less [6]. Furthermore, certain systems may order priority in ascending order, while others order priority in descending order.

Given a global set of scheduling parameters such as deadline, period, and importance, different endsystems may enforce the global scheduling decisions differently. For example, consider a distributed system where it is determined to be optimal to globally enforce Most Urgent First (MUF [7]) scheduling across the system. One endsystem may provide an operating system level priority assignment based only on the deadlines of the tasks scheduled on that endsystem, while another endsystem, given the same set of parameters, may locally implement the MUF scheduling by setting up thread queues and dispatch the tasks to threads without the use of OS-level priorities. In either case, the distributed system must have some way to provide the right information to the endsystem so that the scheduling policy can be enforced as expected.

Determine globally optimal scheduling parameters. This determination is based upon the global scheduling decisions made for the system. For each endsystem, when possible, install a local enforcement policy that is consistent with the assumptions of the global scheduling. When such an installation it is not possible, have the global scheduler map the global scheduling parameters to the local scheduling parameters, and have the global scheduling take the affect of the mapping into account when making its global scheduling decisions. In the example described above, if it is possible to install an MUF scheduler that enforces globally determined urgency parameters on each endsystem, do so. However, if a local endsystem only uses deadline-based scheduling, map the global urgency to an appropriate deadline for that local endsystem and have any analysis done in the global scheduler comprehend that local enforcement on this endsystem will be deadline-based, not MUF.

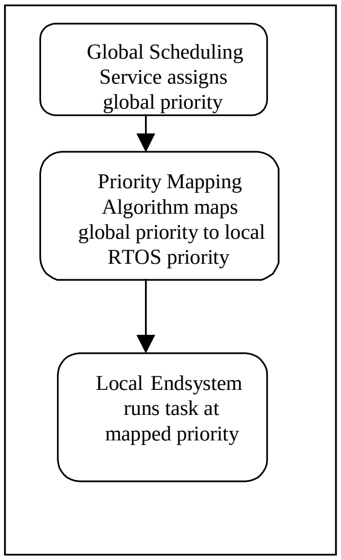

Global Scheduling Service assigns global priority
Priority Mapping Algorithm maps global priority to local RTOS priority
Local Endsystem runs task at mapped priority

Figure 4 - Local Enforcement Pattern

◊◊◊

Figure 4 illustrates the Local Enforcement pattern where a global priority is assigned, but mapped to a local endsystem priority. Instances of the *Local Enforcement* pattern can be found in [8], [9], and [5]. A primary example is the priority mapping provided by RT CORBA 1.0. In these systems, a CORBA priority is determined globally and then mapped to the local priority of the operating systems. Furthermore, RT

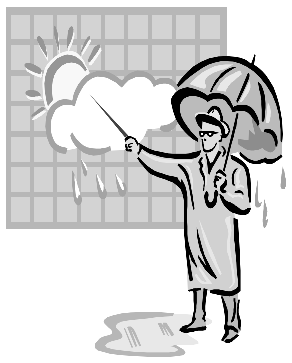
CORBA 1.0 provides an interface to install priority mapping algorithms that are consistient with the global scheduling assumptions.

The *Local Enforcement* pattern is related to the overload handling patterns described in Section 4 because each of the handling decisions must be enforced on the local endsystem.

4 Overload Management Patterns

In this section we describe the patterns for overload management. Recall from the hierarchy in Figure 1 that there are two main categories of overload management patterns: Overload Detection and Overload Handling patterns. Sections 4.1-4.3 discuss the Overload Detection patterns (*Up Front Analysis*, *Resource Monitoring*, *Failure Detection*). Sections 4.4-4.7 discuss the Overload Handling patterns (*Deny Request*, *Cancellation*, *Application QoS Adjustment*, *Resource Reallocation).*

4.1 Up Front Analysis Pattern



Given a particular request for resources in a DRTS, overload can occur if there are not enough resources to fulfill the request. Further, this overload can cause other failures across the system. For example, if the execution of a new task on one node causes another task to miss its deadline, then any other tasks across the system that depend on the failed task may also fail. For this reason, it is often necessary to avoid overload and not let it occur. The *Up Front Analysis* pattern allows a distributed real-time system to determine if an overload might occur by using analysis techniques.

◊◊◊

Up front detection of overload in a DRTS is necessary if the system requires that overload not occur.

In a DRTS with tight timing constraints across the system, missing one deadline can lead to a cascading effect of missing other deadlines across the system. For example, if the first task in an end-to-end chain of tasks misses its deadline, the rest of the tasks in the chain are in danger of also failing. Some systems provide mechanisms to detect these kinds of failures (see Section 4.2) and then react to them. However, the reaction to such failures must itself be scheduled and can cause further deadlines to be missed. This type of unpredictable behavior is not tolerable in systems with tight timing constraints and mission-critical tasks. In systems where uncontrolled failure is not acceptable, it is necessary to be able to predict when overload will occur. Another class of systems in which overload cannot occur are systems where recovery from overload-induced failures is not possible. For instance, in manufacturing applications, and in systems that affect their environment in general, it may often be either costly or impossible to achieve recovery. In these systems it is often better not to start a task than to allow the task to start and then fail.

Use existing knowledge of the system, including timing constraints, resource capabilities and scheduling techniques, to analyze the system. Predict when and where the overloads could happen and provide results of the analysis to the system to allow it to determine how to handle the overload. The goal is to determine dynamically if an incoming task can be scheduled in the existing system without causing overload to occur, thus causing some tasks to possibly miss deadlines.

In a static, hard real-time system, a priori analysis is often used to determine the schedulability of the entire system. Analysis techniques such as rate monotonic analysis [11] and earliest deadline first scheduling analysis [11] are well-known ways of determining system schedulability. In a dynamic DRTS, similar analysis techniques can be used as online "acceptance tests" when a new task requests resources [11].

Up front analysis can provide a predictable means of determining when and where overload can occur. This kind of technique can be pessimistic in that often worst case execution times are used to analyze the system. When the worst case does not occur, resources are wasted. However, in situations where it is better to be predictable than to fully utilize all resources, up front analysis is necessary.

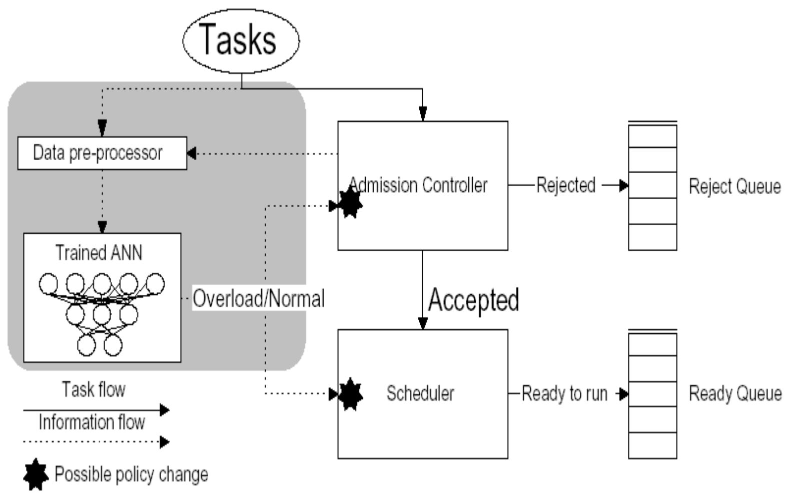


Figure 5 – A system using *Up Front Analysis* in the Admission Controller module [10].

◊◊◊

This pattern provides analysis results for the *Deny Request* and the *Cancellation* patterns in order to determine if it is necessary to deny the new request or to cancel an existing task. The *Up Front Analysis* pattern is also useful to the Service Request Binding pattern to help determine whether a particular binding of client to server is schedulable. Figure 5 illustrates the Up Front Analysis pattern where an admission controller analyzes the system and a new request to determine if it is schedulable. Examples of this pattern appear in [10], [11] and [12].

4.2 Resource Monitoring Pattern

Unlike the *Up Front Analysis* pattern of Section 4.1, the *Resource Monitoring* pattern detects the potential for overload before or as it occurs. That is, it can compare monitored results with defined thresholds to detect that overload will occur if not action is taken. We do not describe this pattern here because it is described in [13]; we include it in this paper for completeness.

4.3 Failure Detection Pattern

In a system in which it is not necessary or feasible to perform up front analysis, or specific QoS monitoring, it is necessary to know when and where overload has occurred so that it can be handled. The *Failure Detection* pattern detects overload after it has occurred and caused a system QoS failure, such as missed deadlines, or degradation of service in a DRTS.

◊◊◊

In a system in which overload can occur, there must be some way to detect it and report it so that it can be handled.

In order to maintain the specified levels of QoS required by an application, failures in the system should be detected so that they can be handled. Further, detection of failures will allow the system to know where future adjustments might be necessary to eventually reduce the number of future QoS failures.

Provide a mechanism to detect QoS failure. Allow applications to specify QoS requirements such as deadlines and periods. The failure mechanism should provide a means to determine if the requirements have been met. For example, in [7] a mechanism is provided that can detect timing failures and provide information about the failure to the failure handler. CORBA and many other run-time systems provide a timeout mechanism that can detect deadline violations.

This solution differs from the *Resource Monitoring* pattern [13] in that the *Resource Monitoring* pattern can intelligently analyze the performance of the DRTS and determine when it is necessary to act. This may be in reaction to some performance threshold being hit that indicates that overload will occur if no action is taken soon. The *Failure Detection* pattern allows the failure to occur and then reports it to be handled. This is a more "optimistic" pattern than both *Up Front Analysis* and Resource Monitoring because it assumes that no action is required unless the failure actually occurs.

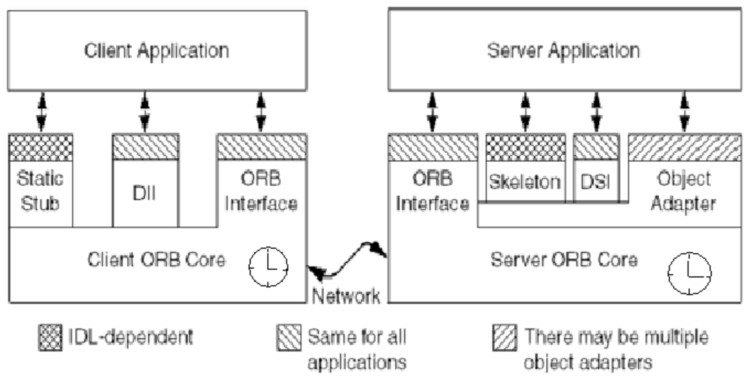


Figure 6 – CORBA Time-Out uses the *Failure Detection* Pattern [14]

◊◊◊

Figure 6 illustrates the Failure Detection pattern as used in a CORBA Time-Out. The *Failure Detection* pattern provides information to the *Cancellation* pattern to determine which tasks to cancel, when. Examples of this pattern appear in [15], [16] and [17].

4.4 Deny Request Pattern

In a DRTS in which recovery is too costly or not an option, and overload should not occur, there needs to be a simple way to maintain the QoS performance of the system. The *Deny Request* pattern addresses overload by keeping tasks from starting if they cause an overload.

◊◊◊

If the potential for overload has been detected up front, and recovery from failure or QoS degradation is not an option, the overload should not be allowed to occur.

In many DRTSs the timing constraints on the applications can be too tight to allow for recovery from QoS failures during system execution. It is often the case that maintaining the performance of the existing system is more important than allowing new tasks to enter the system that might cause overload to occur. For example in sensor networks where existing tasks form intricate dynamic relationships with each other, it might not be safe to terminate existing tasks, but denying an entering task may be allowed. In such systems it is not feasible to wait for overload to occur and then fix it, or to cancel an existing task in order to allow a new task to execute. There must be a way to maintain the integrity of the system if a newly requested task will not fit.

Provide a mechanism that can deny a request if there are not enough resources to service the request. In a DRTS that employs this pattern, it is assumed that overload detection uses an up front analysis technique (Section 4.1) to determine if a newly requested task will cause overload. When a request is denied, the client making the request can decide to delay the request and resubmit it at a later time. However, if the request has a timing constraint that would preclude it from being delayed, it will simply not execute at all. In this case the client may be required to do exception handling of its own.

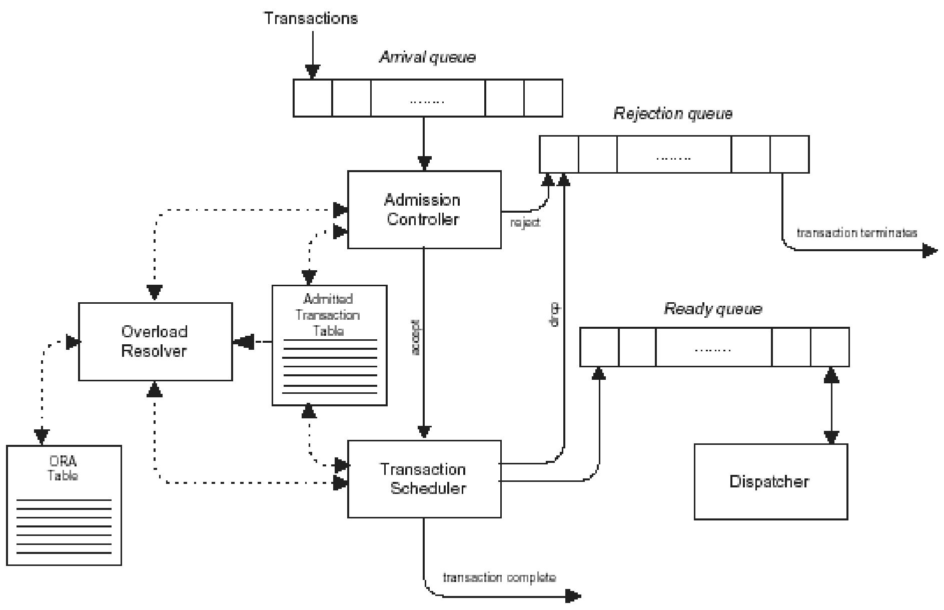
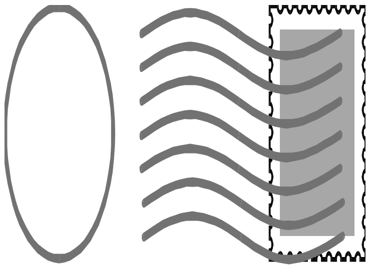


Figure 7 - An overload management in RTDBS using *Deny Request* (from [18]).

◊◊◊

Figure 7 depicts an illustration of the Deny Request pattern in a real-time database system. The admissin controller determines if a new transaction can be scheduled. If not, the request is denied, and it is placed in a rejection queue. Examples of this pattern appear in [18], [11] and [12]. Figure 3, from [18], shows the pattern is used in overload management in a real-time database system.. The *Deny Request* pattern receives information from the *Up Front Analysis* pattern to determine if the requested task should be denied.

4.5 Cancellation Pattern



In a DRTS where overload can occur and it is feasible to terminate or change existing tasks, the *Cancellation* pattern allows for tasks to be either removed from the system, or to have their scheduling parameters changed. This is done when new tasks may contribute more value to the systems than one or more existing tasks.

◊◊◊

In DRTS's where overlaod can occur and new tasks may contribute more value than existing tasks, there must be some way to terminate or adjust the scheduling parameters of existing tasks.

In general, in a DRTS, if overload is either a potential problem (predicted) or an actual problem (occurred) handling the overload is necessary. The *Deny Request* pattern described in Section 4.4 always chooses to not allow the new request to begin execution. However, in many systems tasks can have varying levels of importance, and it may be unacceptable to deny a very important new task in order to allow the currently executing tasks to continue to run. For example, new tasks representing the detection of new threats may be more important to execute than existing tasks. Also, in systems where up front analysis is not used, it is not feasible to use the *Deny Request* pattern to handle overload because there is no way to know if the current request will be schedulable. In both of these cases, it may be necessary to either remove one or more tasks from the system, or to adjust the scheduling parameters of existing tasks in order to alleviate the overload.

Provide a mechanism to allow a task to cancel itself, or to cancel another task. The mechanism should include a technique for deciding which tasks should be cancelled. For instance, if an overload is detected using the *Failure Detection* pattern (Section 4.3), then the *Cancellation* mechanism should cancel the task that failed. On the other hand, if the potential for deadline violation is detected by the *Up Front Analysis* pattern (Section 4.1), the *Cancellation* mechanism can choose from among all of the tasks in the system that are affected by the overload. It can use information about the tasks and about the system to determine which task(s) to cancel. This information can include information such as the relative importance of the tasks, the remaining execution time of the tasks, the time to recover from cancellation, and the dependencies among the tasks in the system. Canceling a low importance task to allow a more important task execute may be increase the value obtained by the system. However, if the cancelled task has almost completed, or has several tasks that depend on its completion, then it might not be the best choice for cancellation. The *Cancellation* mechanism should provide algorithms for choosing the tasks to cancel given various system circumstances.

***Cancellation* of a task may be implemented in various ways. When a task has already missed its deadline, which has been detected using the *Failure Detection* pattern, termination of the task maybe**

the best way to implement *Cancellation*. However, in a system that has predicted overload based on *Up Front Analysis*, the prediction may be pessimistic and the overload may not actually occur at all. Rather than terminating the task(s) chosen by the *Cancellation* mechanism, it may be better to change the scheduling parameters instead. This can manifest itself in lowering the priority of a cancelled task, or lengthening its deadline or its period. The Cancelled tasks will not meet their timing constraints in the worst case, but may be able to complete if there is more slack in the schedule than was predicted by the *Up Front Analysis*.

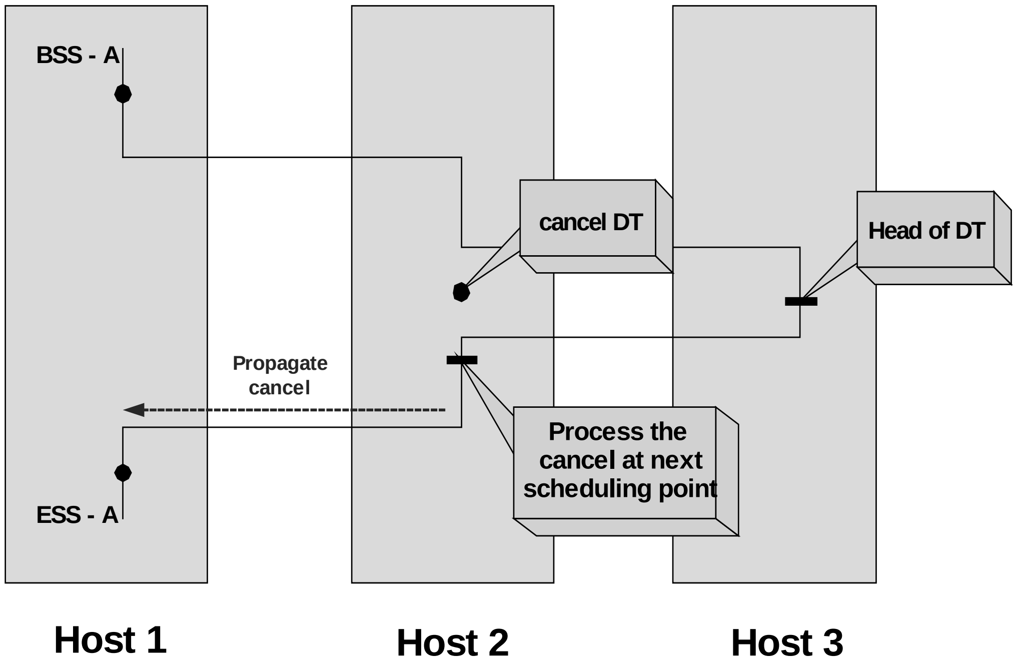


Figure 8 - Real-Time CORBA Distributable Thread using *Cancellation* (from [19]).

◊◊◊

Figure 8 illustrates the Cancellation pattern as specified in Real-Time CORBA 2.0. A distributable thread can be cancelled during its execution. The cancellation is processed at the next scheduling point in its execution. Examples of this pattern appear in [19], [20], [21] and [22]. Figure 4, from [19], shows how the *Cancellation* pattern is used in the Real-Time CORBA 2.0 Distributable Thread architecture. The *Cancellation* pattern can use analysis information provided by the *Up Front Analysis* pattern to determine which tasks to cancel. It may also use information provided by the *Failure Detection* pattern to cancel a task with a QoS failure.

4.6 *Application QoS Adjustment*

The *QoS Adjustment* pattern has the *application* negotiate to adjust the QoS requirements of tasks in an intelligent way when overload has occurred. The pattern is described in [23], we include it here for completeness.

4.7 *Resource Reallocation*

The *Resource Reallocation* pattern has the system reallocate resources to alleviate overload. In this pattern QoS is not necessarily adjusted, nor are tasks cancelled, instead the system attempts to find a better resource allocation, perhaps moving tasks to less loaded nodes, that will still maintain the original QoS specified by the application. We do not describe the pattern here, it is described in [13]. We include it here for completeness.

5 Conclusions

This paper has presented a pattern language for scheduling in a DRTS. These patterns provide a framework for middleware developers to use when building a system with global QoS requirements that must be enforced on local endsystems. While the general description of the context for these patterns involves distributed real-time systems, specific details about the particular applications will result in different traversals of the pattern language in Figure 2. For example, in a system that cannot tolerate recovery of existing execution, the *Up Front Analysis* pattern might be used along with the *Deny Request* pattern for overload management. On the other hand, if the system has softer constraints, and can afford to be optimistic about failure, it can use the *Failure Detection* pattern with the *Cancellation* pattern to manage overload.

The patterns in this language clearly do not stand alone. They are closely related to other patterns involved in scheduling in a distributed system. A larger pattern language is described in [24]. Here, these distributed real-time scheduling patterns are put into context with patterns for global resource allocation, scheduling on endsystem middleware, and scheduling at the operating system level.

References

[1] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[2] A. Uvarov, L. Cingiser DiPippo, V. Fay-Wolfe, and P. Gupta, Slack-Time Driven Distributed Real-Time Dynamic Binding, submitted to *Journal of Parallel and Distributed Computing*, April 2003.

[3] G. Parr, I. W.K.Ho, A. Marshall, D H.T.Chieng, A Software Agent Brokering Environment for ATM Real-Time Network Resource Allocation, *Online Seminar on Embedded Software, Open Symposium for Electrical Engineers (OSEE)*. 27 Feb 2001. www.osee.net.

[4] H. O. Rafaelsen, F. Eliassen,. Trading and Negotiating Stream Bindings, In *Proceedings of Middleware'2000,* New York, 3-7 April 2000, http://citeseer.nj.nec.com/rafaelsen00trading.html.

[5] OMG, *Realtime CORBA*. Electronic document at http://www.omg.org/docs/orbos/98-10-05.pdf.

[6] Sourceforge, Differentiated Services for Linux, http://diffserv.sourceforge.net/.

[7] D. Stewart, P. Khosla, Real-Time Scheduling of Dynamically Reconfigurable Systems, *Proceedings of the IEEE International Conference on Systems Engineering*, Dayton Ohio, pp. 139-142, August 1991.

[8] Craig Rodrigues, Yamuna Krishnamurthy, IrfanPyarali, Pradeep Gore, Using Prioritized Network Traffic to Achieve End-to-End Predictability, *Real-Time and Embedded Distributed Object Computing,* June 15-18, 2002, Arlington, Virginia, USA.

[9] L. DiPippo, V. F. Wolfe, L. Esibov, G. Cooper, R. Johnston, B. Thuraisingham, J. Mauer, *Scheduling and Priority Mapping for Static Real-Time Middleware*, *Real-Time Systems Journal,* 20, 155-182, 2001.

[10] R. Steinsen, Predicting Transient Overloads in Real-Time Systems using Artificial Neural Networks, University of Skovde, Technical Report #HS-IDA-MD-99-006, 1999.

[11] J. Liu, *Real-Time Systems,* Prentice Hall, 2000

[12] Sven Gestegård Robertz, *Flexible automatic memory management for real-time and embedded systems*, Licenciate thesis, Dept. of Computer Science, Lund University, May 2003.

[13] Toni Marinucci, Lonnie Welch, Patterns for Developing Adaptive, Distributed Real-Time Systems, submitted to *PLoP 2003*.

[14] OMG, Common Object Request Broker Architecture, http://www.omg.org.

[15] J. Hansson, S. Son, J. Stankovic, S. Andler, Dynamic Transaction Scheduling and Reallocation in Overloaded Real-Time Database Systems, *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98),* Hiroshima, Japan, IEEE Computer Society Press, 1998.

[16] C. Gill, D. Schmidt, R. Cytron, Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing, *IEEE Proceedings Special Issue on Modeling and Design of Embedded Systems,* October 2002.

[17] Douglas C. Schmidt Steve Vinosk,i Object Interconnections: An Overview of the OMG CORBA Messaging Quality of Service (QoS) Framework (Column 19), C++ Report, March.

[18] J. Hansson, S. Son: Overload Management in RTDBs. *Real-Time Database Systems: Architecture and Techniques,* Kluwer Academic Publishers, 2001.

[19] OMG, Realtime CORBA Dynamic Scheduling, ptc/01-08-34, http://www.omg.org/cgi-bin/apps/doc?ptc/01-08-34.pdf.

[20] C. Gill, D. Schmidt, R. Cytron, Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing, *IEEE Proceedings Special Issue on Modeling and Design of Embedded Systems,* October 2002.

[21] D. Hong, Real-Time Transaction Scheduling: a Cost-Conscious Approach, thesis, D. Hong, Theodore Johnson, Sharma Chakravarthy, Real-Time Transactin Scheduling: A Cost Conscious Approach, SIGMOD Conference 1993: 197-206.

[22] C. Montez, J. Fraga, R. Oliveira, An Adaptive Model for Programming Distributed Real-Time Applications in CORBA, WSTR'98, Rio de Janeiro, RJ, Brazil, May 1998.

[23] J. P.Loyall, P. Rubel, M. Atighetchi, R. Schantz, J. Zinky, Emerging Patterns in Adaptive, Distributed Real-Time, Embedded Middleware, *OOPSLA 2002 Workshop on Patterns in Distributed Real-time and Embedded System* , Nov. 2002.

[24] C. Gill, D. Niehaus, L. DiPippo, V. Fay Wolfe, and L. Welch, Mapping a Multi-Level Scheduling Pattern Language to Distributed Real-Time Embedded Applications, *OOPSLA 2002 Workshop on Patterns in Distributed Real-time and Embedded System* , Nov. 2002.