CHARACTERIZATION AND DEVELOPMENT OF DISTRIBUTED, ADAPTIVE REAL-TIME SYSTEMS

A thesis presented to the faculty of the

Russ College of Engineering and Technology

In partial fulfillment

of the requirements for the degree

Master of Science

Toni Marinucci

March 2005
This thesis entitled

CHARACTERIZATION AND DEVELOPMENT OF DISTRIBUTED, ADAPTIVE REAL-TIME SYSTEMS

by

Toni Marinucci

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Lonnie R. Welch
Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
Development of real-time resource management system often entails utilizing similar software engineering solutions as other resource managers. However, the developers of these resource managers may reengineer solutions, not knowing that a solution already exists. First time developers of this type of software may not even know what types of problems they will encounter. Each of these situations slows the development process. This thesis examines the challenges in determining what software is needed to build a real-time resource management system. After working with several different types of resource management systems, real-time and otherwise, it is clear that many of these systems use similar approaches in managing their resources. Research in the field of real-time systems unveils that some such approaches are documented as patterns, but many of the existing patterns focus on in depth details about real-time system management, requiring that the reader already have an abundance of knowledge on the topic. The aim of this thesis is to step back and examine the higher-level components of these systems and the interaction of these components and document them as patterns. These patterns allow newcomers to real-time systems to understand the problems incidental the topic. Furthermore, these patterns may aid the expert by presenting the information in a different way. The common components of the systems researched that are found in a number of distributed and real-time systems are essential pieces in building such systems,due to the sheer number of systems in which they are found. Here, they are presented in pattern format.

Approved: Lonnie R. Welch

Professor of Computer Science
Acknowledgments

Since I began working on my Master’s degree, and this thesis in particular, there are quite a few people to whom I have become indebted due to the support and encouragement they have provided me throughout this process. I will begin by thanking my adviser, Dr. Lonnie Welch for offering me a research assistant position and giving me the opportunity to pursue this degree. Throughout the last few years, he has provided me with an innumerable amount of opportunities and experiences, guidance and support. Along with him, I am grateful to the members of my thesis committee, Dr. David Juedes, Dr. Chang Liu, and Dr. Valerie Young, as well as many of the other faculty members at the School of EECS at Ohio University.

I also owe a great deal of thanks to my husband, David Heightland, and the rest of my family - Mom, Dad, Erin and Dustin - for their love, emotional support, and encouragement during the last three years. I am not sure I could have endured this journey without them.

One of the first opportunities I had was to work with some folks at NASA’s Goddard Space Flight Center. They include Barbara Pfarr, who has been quite an inspiration to me as one of the few women I had the chance to work with in this field, Greg Greer, and Brian Goldman.

This thesis has been influenced by probably what most would consider an unusual number of people. Initial concepts for this work appeared at the PLOP and OOPSLA conferences in 2002, and at those conferences, those papers were shepherded. I received very valuable input on learning to write patterns from folks like Doug Schmidt and Chris Gill. Other people whose shepherding comments helped to form this work into its current state include Lisa DiPippo, Doug Niehaus, Richard Gabriel, Linda Elkin, Linda Rising, Bob Hanmer, Paul Rubel, Joe Cross, and Joe Loyall. I know there are plenty of others whom I have forgotten, and for that I apologize.

There are a number of other people at Ohio University who have helped me along in one way or another that I would like to mention and thank. These people include Dr. Costas Vaziliadas, Candace Stewart, Tammy Jordan, April Vale, Sandi Mathews, Erin Ross, Corinne Biehl, and last, but in no way least, my peers in the CIDDS lab.

Thanks to you all!
Contents

Approval Page 2
Abstract 3
Acknowledgments 4
List of Figures 6
1 Introduction 7
   1.1 Existing Management Software for Real-Time Systems . . . . . . . . . . . . . . 8
   1.2 Load Balancing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   1.3 Existing Solution Patterns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   1.4 Pattern Frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   1.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2 Real-Time System Characterization 18
3 Engineering Patterns 28
   3.1 Hardware Profiling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   3.2 Software Profiling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
   3.3 System Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
   3.4 QoS Monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
   3.5 Resource Monitoring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
   3.6 Allocation Policing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   3.7 Adaptation Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
   3.8 Allocation Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
   3.9 Allocation Enactment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
   3.10 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4 Terminology 46
References 46
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Welch’s Taxonomy Uses Kiviat Diagrams to Characterize a Real-Time System (a) and its Environment (b)</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>An example of “fit” in a Kiviat Diagram</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>The Kiviat Diagram for Characterizing a Real-Time System’s Control Function used by Welch’s Taxonomy</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>The Behavior Kiviat Diagram for Characterizing a Real-Time System used by Welch’s Taxonomy</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>The Timing Requirement Kiviat Diagram for Characterizing a Real-Time System used by Welch’s Taxonomy</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>The Task Relation Kiviat Diagram for Characterizing a Real-Time System used by Welch’s Taxonomy</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>The Form of Adaptation Kiviat Diagram for Characterizing a Real-Time System used by Welch’s Taxonomy</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>The Characterization for Data Arrival used by Welch’s Taxonomy</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>The Environment Dynamics for a Real-Time System used by Welch’s Taxonomy</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>Characterizing the Environment Workload using Welch’s Taxonomy</td>
<td>26</td>
</tr>
<tr>
<td>11</td>
<td>The interaction of the patterns</td>
<td>28</td>
</tr>
<tr>
<td>12</td>
<td>An example of a hardware profiler from [1]</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>An example of a software profiling process from [2]</td>
<td>32</td>
</tr>
<tr>
<td>14</td>
<td>An example of a system specification from [3]</td>
<td>34</td>
</tr>
<tr>
<td>15</td>
<td>An example of a QoS monitor from [4]</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>An example of a resource monitor from [5]</td>
<td>38</td>
</tr>
<tr>
<td>17</td>
<td>An example of an allocation policer from [6]</td>
<td>39</td>
</tr>
<tr>
<td>18</td>
<td>An example of the results of adaptation analysis in [7]</td>
<td>42</td>
</tr>
<tr>
<td>19</td>
<td>An example of an allocation optimizer</td>
<td>43</td>
</tr>
</tbody>
</table>
1 Introduction

Developing a distributed real-time system (RTS) can be a daunting task. Many characteristics of the system must be considered prior to building it, and knowing what characteristics to consider is sometimes difficult. For example, how does one characterize the criticality of the tasks to be performed or capture the urgency of meeting the performance requirements of the tasks? Once these characteristics are determined and assessed, mechanisms that enable the requirements of the real-time system to be met according to its characteristics must be built and incorporated into the system.

Although a small array of publications exist that address issues in developing real-time systems, it can be seen in publications cited in the following literature review that the solutions to the problems deal with specific intricacies of systems. Further, the authors of these solutions assume that the audience knows and understands where such a solution would reside within a system. While this information may be helpful at some later point during the career of a real-time system developer, not an abundance of information can be found to tell that developer where to start. The aim of this document is to do just that: provide direction to the software engineer that is new to developing real-time systems. The following pages provide patterns that have been developed to remedy the problems of accurately characterizing a real-time system and then building technologies that enable that system to meet its requirements. Those who wish to gain a better understanding of what real-time systems are and how they systems work and even experienced developers may find this information useful. By no means is this a comprehensive guide to building real-time systems. In fact, the development solutions are primarily aimed at real-time systems that operate in dynamic environments (see section 2) and that must continuously adapt to that environment. In such systems, there may be many fluctuating factors that affect how the system may be required to perform, where in a static real-time system, these factors can be planned, greatly easing the development of the system. In addition, these solutions apply to real-time systems that are distributed as well.

An example of a system using many of the solutions presented in the following pages is DeSiDeRaTa [5]. When the author began her research in distributed, adaptive real-time systems (DARTS), the DeSiDeRaTa software had been developed by a handful of researchers at two separate universities. At that point, the software was becoming more and more difficult to maintain due partly to the number of people who programmed the software but mostly due to the number of added features which the original authors of the program had never intended. Once it was time to “re-engineer” the software, a number of groups of related tasks became apparent. These tasks helped the researchers to realize that these types of tasks had been seen in other systems related to this one [8, 9, 10]. At the time of this writing, DeSiDeRaTa is undergoing yet another re-engineering phase based on the patterns presented here. Throughout this paper, it will be clear that the solutions presented here for building dynamic, adaptive real-time systems are those that have been used successfully over and over again in these types of systems. To effectively approach the problem, the remainder of this chapter will provide a review of literature in the area of distributed and real-time systems and the resource management of such systems. This literature review will provide the readers with background information supporting the author’s claim of the abundance of similar solutions used in dynamic, adaptive real-time systems or similar systems. First, existing real-time system management software will be examined in section 1.1. Section 1.2 takes a close look at load balancing solutions since these solutions share many characteristics to those of distributed real-time systems. Lastly, section 1.3 provides a limited list of existing solution patterns related to dynamic, adaptive real-time systems.
1.1 Existing Management Software for Real-Time Systems

Since one of the two primary topics for this document is creating and/or utilizing software that will ensure that quality-of-service (QoS) requirements of real-time systems are met, it is important to examine systems that already address this task. Each of the software systems discussed in the literature reviewed in this section contain one or more solutions used for managing the resources of dynamic real-time systems, exemplifying that such solutions are similar and are used repeatedly.

**DQM.** The *Dynamic QoS Manager* [11] moderates the resource usage between the operating systems and the applications. This moderation is accomplished by lowering the operating level on applications when these applications are performing poorly, increasing the availability of resources to the poorly performing applications. Similarly, DQM can also raise the operating levels when the system is under loaded, too.

The authors of [11] cite two problems that prompt their solution:

> “the resource management system of the underlying operating system does not account for the precise requirements of applications and ... the applications cannot adequately execute under a range of resource availability.”

So, throughout the paper, they attempt to answer three questions: (a) Is it possible to program applications to use different operating levels? (b) Is it possible to write a QoS manager that controls the operating levels of applications to maximize the performance of real-time applications based on availability of resources? and (c) Is a general purpose operating system sufficient for a QoS manager, or are other features needed?

First, Humphrey et al. discards the common assumption that an application can perform adequately with any amount of resources, stating that this idea does not seem feasible. Instead, the authors wrote a multimedia application with various operating levels corresponding to various resource availability and corresponding quality. They demonstrate this concept with examples, and show that although the resources vary, the application can operate with degraded, although satisfactory quality. Next, the authors explain the dynamic mediation between the applications and the operating system to ensure that the resource demands do not exceed availability by adjusting the operating levels of the applications. The performance is measured in much the same way that DeSiDeRaTa does [5]. Each iteration through the application, it is determined if the deadline is met, and then, using the number of times the deadline is met, it is decided how to react. One of the key differences between the DQM approach and most resource management approaches, including DeSiDeRaTa’s, is that DQM mediates the resource needs based on the current system resources instead of ensuring that all of the QoS requirements can be met prior to starting the real-time application on the system. The authors detail an experiment using replicas of the same application, and they measure the performance of this application by several factors. In the conclusions, they found another question that needed an answer: If an application performs poorly, should DQM reduce the operating level of that application, or the operating levels of those performing well? Finally, the paper is concluded by comparing and contrasting their approach with those of other researchers, making the research complete. The authors did not seem to try to be saying that their approach was the best, and they readily admitted where improvements were needed.

The work presented by Brandt et al. in [8] is an extension of previous research and development of DQM. This paper answers a lot of questions that Humphrey et al. left open. For example, in
Humphrey’s paper, the researchers had not determined whether to adjust the execution level of the application that was not meeting its QoS or of another application. In Brandt’s paper, it is clear that the research includes several algorithms for deciding this issue, but focusing on an algorithm called “Proportional”, which uses the CPU/benefit ratio to determine which application’s execution level to adjust. This approach to resource management gives another example of the need for resource monitoring and a means for managing QoS.

DeSiDeRaTa and SWARM. The DeSiDeRaTa (DSDRT) approach to resource management is presented by Welch et al. in [5]. This source provides a basic reference for the work done by the author and other student researchers during the pursuit of her Master’s Degree at Ohio University. Currently, DSDRT provides real-time system developers the ability to specify the hardware configuration on which the real-time system shall execute, information about the performance, including software profiles, of the constituent applications of the real-time system, track the performance of the real-time applications during execution, and adapt to a dynamic environment during execution. The development of this software has aimed to find solutions to unsolved problems in the realm of dynamic, adaptive real-time system.

One such dynamic, adaptive real-time system that utilized DSDRT is ITOS, the real-time system used as a basis of the SWARM project as presented in [12]. (SWARM stands for sensor web adaptive resource management.) In particular, the paper discusses the concept of a sensor web and what the benefits of resource management will provide to the application and the project. The idea is to enable a sensor web to take advantage of the resource management capabilities in order to improve the computing power of these satellites by splitting the computations among the satellites participating in the sensor web. Some of the features that the SWARM project will provide are the ability to start and stop a real-time system, ITOS, instrument ITOS in order to monitor its performance, and make ITOS a distributed system. The DeSiDeRaTa resource manager is used in this project. This publication gives an example of a real-time system that is used in real-life, thus providing a basis for what types of solutions that can, and do, work. This is valuable information for this type of research, because many of the papers in this area are strictly theoretical in nature; their solutions have never been applied.

KURT Linux. The need for an operating system that can deal with hard and soft real-time constraints, as well as a hybrid constraint, called firm real-time, motivates the research by Srinivasan et al. in [10]. Based on the open source, freely available Linux operating system, KURT-Linux was developed to satisfy the constraints presented by many firm real-time systems. The first modification to Linux is refining the temporal resolution of the operating system; many firm real-time applications at a more frequent resolution that is used in the conventional operating system. Instead of interrupting the CPU at a fixed rate, the timer interrupts the CPU in time to process the earliest scheduled event. So, instead of having second resolution, KURT-Linux is capable of microsecond resolution. They also explain how they simulated the second resolution needed by several kernel processes. The second modification applied to Linux is the KURT scheduler, which requires real-time applications to explicitly state the times at which events need to occur. KURT scheduling also introduces real-time modes, focused and mixed. In focused mode, the real-time processes have full access to the operating system. In mixed mode, the real-time processes are scheduled along with non-real-time processes. The authors continue with an explanation of how these modes are handled in the operating system. The remainder of the paper examines experiments conducted with UTIME,
the KURT modification to the system clock, and with the KURT scheduler. This source is primarily focused on scheduling real-time applications, not with the overall needs of a real-time system manager (aside from the scheduling standpoint).

1.2 Load Balancing

At first thought, it may not be immediately obvious how load balancing is related to real-time systems. One may think of load balancing in the way that DQM adjusts the operating levels of an application, but the work presented here does not manage resources in that manner. Instead, here concerns are with balancing the load of the execution of one or more applications across a distributed computing environment. Since load balancing algorithms are quite similar to those of real-time system resource managers, the examination of such systems is relevant.

**MOSIX.** The MOSIX operating system (OS) [13] is designed for cluster computing. The functionality that MOSIX provides is global (i.e., cluster-wide) resource allocation that distributes the workload dynamically to use cluster-wide resources efficiently and transparently. The techniques used to distribute the work are load balancing and load sharing, which use some form of remote execution and preemptive process migration. The paper emphasizes that no node has to know everything about the other system resource states. Some of the main characteristics of MOSIX include probabilistic information dissemination algorithms, preemptive process migration, dynamic load-balancing, memory ushering, efficient kernel communication between the process and its home node, decentralized control and autonomy, and scaling considerations that ensure the system runs equally well on different scales of configurations. The remainder of the paper evaluates, in not much detail, the performance of the MOSIX algorithms, and then, the performance of parallel applications, which are using MOSIX. The most interesting aspect of the MOSIX paper is its “main characteristics” listed above. They utilize a resource usage collection algorithm that does not poll each node, which varies from the technique of DeSiDeRaTa. The polling is done at regular intervals; each node sends its available resource information to a random subset of nodes in the cluster. It is stated that this random-ness helps to ensure stability, but not explained. Another interesting aspect of the MOSIX OS is that nodes can join and leave the cluster without disrupting operation. This capability is not present in the current version of DeSiDeRaTa. The techniques presented in MOSIX could potentially be applied to real-time computing, but in this paper, it is only discussed as a tool for distributed computing. This source utilizes several concepts commonly used in real-time computing. In general, this paper is helpful in explaining some techniques used in load balancing, a crucial part of a dynamic, adaptive real-time system.

**Condor.** Condor [9] is software that aims to utilize under used CPU processing capabilities of workstations on a network. The authors motivate their research by citing that, in their department, 70% of available gross CPU processing time goes unused, but some “heavy users” do not have the processing capacity they need within their own workstation. Some of the goals of their efforts are to move background processes transparently to the users; if a background job fails, it should be restarted automatically; and the processing needed to implement these features need to require little resource usage.

In order to track where available resources are and which jobs need to be run, a hybrid approach between centralized and decentralized scheduling is used. The software has a central manager as-
signs capacities to the workstations, but each workstation is responsible for scheduling its own jobs. Condor assesses when a workstation is idle (i.e. not in use currently by a user) and transparently invokes background jobs to run on the workstation. Once the user begins using the workstation, the central manager notifies this machine that it has fewer resources available; and, any background processes running on this machine are stopped, state is saved, and the process may be allocated to another machine. The paper contains experimental results of using Condor. This approach, as outlined in the paper is not applied to real-time systems. However, some of the goals are present in distributed real-time systems.

1.3 Existing Solution Patterns

The type of solutions being presented in this document are not new to the real-time system problem domain. Here, some literature including patterns in real-time systems is presented. These patterns focus on problems in scheduling, QoS information and enforcement, monitoring hardware and software performance, and allocation of resources to applications. The literature in this section is a representative sample of what is available, yet the types of problems solved with the existing solutions has not been exhausted with the following list. In reading this section, it can be seen that these patterns assume that the readers already have a hefty amount of knowledge in the problem domain and focus on the intricacies of real-time system management. A higher-level view of how to manage these systems may be useful.

Scheduling. Gill *et al.* discuss the fact that in distributed systems, scheduling occurs at several levels, the kernel level, the end system middleware level (e.g., ACE), and the distributed middleware level (e.g., QuO) [14]. The paper begins with a list of forces that are present at one or more of the levels, and then, beginning with the kernel, lists the patterns that occur at each level, referring to the forces that create the problem. The patterns at the kernel level are planned scheduling, priority-driven scheduling, share allocation, hierarchical scheduling, masking interrupts, and synchronous locks. At the end-system level, patterns include request pacing, request propagation, request partition, strategic request recording, and strategy composition. The distributed level patterns are distributed scheduling service, distributed scheduling, distributed resource consistency control, global to local priority mapping, global load allocation, and global overload management. The last section of Gill’s paper discusses the problem of rationalizing scheduling across levels of scale with the patterns application level quality of service adjustment and distributed temporal coherency. These patterns capture the well known scheduling problems in dynamic, distributed real-time systems. They discuss the problems of mapping priorities across several levels of depth, predicting real-time performance of entire distributed applications, how to construct requests for QoS requirements to ensure timeliness across the entire system, among others. However, at times, it is not clear that all the patterns are “scheduling” patterns. For instance, Global Load Allocation and Global Overload management seem not to be scheduling issues, but allocation issues. If the authors meant these patterns to encompass scheduling and allocation, then, their intent was not obvious. Another aspect of the paper that is unclear is how these patterns help to increase cross-level integration. There were only two patterns on this topic, and an example would have been useful. Still, the intended goal of the paper was captured well in the introduction:

"Many systems design paradigms, which concentrate their attention at a single level of system abstraction fixing the mode of interaction of one level with an adjacent one,
inevitably limits the extent of cross-level integration available. This paper presents a pattern language for increasing coordination of resource management across multiple levels of architectural scale.”

Gill et al. provides a good set of patterns that have already been identified for dynamic real-time systems. This source serves as a means for comparison of other patterns.

**Quality of Service Information.** In [15], two patterns that are useful for QoS communication are presented, QoS Contract and Snapshot. The first pattern, QoS Contract, has a dynamically changing environment, where the resource needs fluctuate to the extent that there may not be enough resources available to meet all of the need, as the context. The QoS Contract pattern solves the problem by decentralizing the monitoring, analyzing and controlling the QoS information. An example is used to explain that traditional CORBA-based distributed object computing (DOC) middleware is insufficient due to its feature of hiding details such as implementation, transport, and platform. On the other hand, Quality Objects (QuO) is an example of middleware that supports QoS measurement, adaptation, and management. QuO supports these QoS requirements by adding the idea of Contracts, Delegates and System Condition Objects. The Contract specifies the level of service that is expected and/or provided by an object and how to react when the QoS level changes. The Delegate “provides an interface similar to that of the remote object stub [of CORBA], adding locally adaptive behavior in the path of the remote method call based upon the current state of the QoS in the system, as measured by the contract.” And, system condition objects provide interfaces to objects that need to be measured and controlled by the contracts [15]. The Snapshot pattern [15] addresses the problem of needing to have an accurate picture of the system state when making decisions in systems with adaptive behavior. The context of snapshot is a distributed system where the parts of the system may be in different, perhaps unpredictable, locations. Thus, the state information may be distributed through these components. The solution provided in Snapshot creates a set of system conditions, providing an accessor and an aggregator, that collects all of the values from the system conditions. This configuration can guarantee an accurate approximation of the state of the system. This paper provides an example of patterns written for dynamic, real-time systems. Further, the paper addresses two distinct problems encountered in developing adaptive systems: QoS information propagation and saving state.

Also, Loyall et al. discuss QuO [16], but focus on the contract description language (CDL) of QuO. The CDL allows one to specify a contract between a client and an object in an application, which describes the desired QoS of the client and the expected QoS the object will provide. A QuO contract includes:

- A set of operating regions that represent the set of possible QoS states,
- Transitions for each operating region that specify what causes the active region to change,
- System condition objects, which gather values of system resources, object and client state, and are used as part of transitions, and
- Callbacks, which notify the client or object of current information.

The remainder of the section stipulates the structure, syntax and semantics of a CDL contract, with examples. This paper ([16]) exemplifies the need of QuO and its CDL, stating a list of elements that
would not have been able to be controlled in this example without this application programming interface (API), such as notifications between an application and a network session. This paper, in conjunction with Karr’s paper ([17]) provides a complete overview of QuO, and provides another approach to QoS communication, an essential piece of any real-time system.

Finally, Karr et al. discuss QuO and show how it can be applied to a realistic scenario, the UAV scenario [17]. The UAV scenario uses QuO in conjunction with the TAO real-time ORB and the TAO A/V Streaming Service (a CORBA implementation of the A/V Streams Specification). The TAO A/V streaming service lies above TAO and ACE, which handles flow control processing (TAO) and media transfer (ACE). The authors state that the benefit of adaptation enables software systems able to adjust to varying resources and needs that are typical during execution, and the benefit of a framework (e.g., QuO) separates QoS monitoring and requirements from the functional code of the application. More specifically, UAV scenario is as follows: an off board unmanned aerial vehicle (UAV) continually reads an MPEG file, the UAV sends MPEG video to an on-board video distribution process, and the distributor sends the video frames to a number of video display processes. In order to adapt to the varying resources and QoS requirements, three possibilities exist: (1) reduce the amount of data being sent (drop some frames) or reduce the amount of data per frame (reduce image quality), (2) move the distributor to a host that has more network and/or CPU resources, or (3) reserve a portion of network bandwidth (RSVP). Which adaptation technique to apply depends on the specific use of the video and available resources. This can create a large array of scenarios, but QuO has the flexibility to provide information for these different conditions through its contracts. The paper provides an overview of the MPEG compression technique and the tradeoffs associated with various frame rates. These frame rates are specified in the QuO contract, and, based on the current performance (currentFrameRate, a system condition) the contract makes a decision about which QoS can be delivered, thus providing an adaptation mechanism for the video application. Overall, this paper provides a good example of how QuO can be used in a real-time distributed computing environment. The authors were careful to express limitations of their approach and to explain why they chose the route they did.

Aside from QuO, other approaches to QoS communication exist. Cross and Schmidt describe the need to define QoS needs and to connect applications to QoS control mechanisms, as well as provide a mechanism for conforming to QoS requirements [3]. The Quality Connector pattern begins with an example of the CORBA Event Service and describes three desired properties of this service: ability to determine whether a given implementation of Event Service will support the application successfully, ability to port the application from one implementation of Event Service to another, and ability to modify the Event Service implementation and still support the application correctly (by meeting the QoS requirements). Also, Cross and Schmidt clearly define the problem and its context before presenting the solution. The solution consists of four parts: solution, structure, dynamics, and implementation. The structure section describes three components:

1. Static Application Connector, which scans the source code of the application for statements regarding types of service provided. These statements are replaced with methods that specify QoS.

2. Static Infrastructure Connector, which accepts QoS specifications and selects or modifies the middleware components that will be available to the Dynamic Connector.

3. Dynamic Connector, which allocates resources to the application. This component is also
responsible for finding and negotiating resources that can provide service to the application in the requested mode.

The most interesting part of the paper is the known uses section, which provides a few examples of how QoS needs are specified in systems like MINERS and QuO, as well as a thorough comparison of these specifications. The last section, see also, examines how the quality connector pattern is related to the patterns like Interceptor, Proxy, Reflection, and Adapter.

Another alternative to QuO, as discussed earlier in this section, is QoS-enabled DOC middleware, which is distributed processing environment (DPE) middleware that works between the applications and the operating systems, protocols, and hardware [18]. This middleware simplifies and coordinates these components and mediates their inter operation. DOC middleware has four layers: infrastructure middleware (ACE), middleware (CORBA, RMI), common middleware services, and domain-specific services providing the following benefits: strategic focus, effective reuse, and open standards. The publication also presents AQoSA. This API provides network level QoS protocol that has enhanced portability, higher-level QoS specification and increased adaptation capabilities.

The ACE QoS API has been implemented in ACE and it provides “a portable C++ encapsulation of . . . the IntServ Resource Reservation Protocol (RSVP). . . and the RSVP API (RAPI) implementation on UNIX.” AQoSA was developed by identifying common patterns used in existing QoS API’s and then, developing components that “reified” these patterns. Some of the features needed were portability, extensibility, QoS event notification, advanced QoS capabilities, and adaptation capabilities. The implementation of these features included Reactor, Half-Sync/Half-Async, Factory, Bridge, and Decorator patterns. The paper is concluded with a case study that applies AQoSA to the CORBA A/V streaming service.

Monitoring. The monitoring of real-time system’s is the focus of Jahanian’s research in [4]. This is not limited to host and network resource monitoring, but also includes event monitoring, system constraint specification, real-time system monitoring, detection of constraint violations, and operating system support for run-time monitoring, among other related topics. The most interesting aspect of Jahanian’s work is the real-time system monitoring. Keeping track of events, which are broken down into label and transition events, provides a way to determine, for instance, the number of previous occurrences of a particular event. Events need to be specified so that the real-time system knows how to react to what is happening in the system. Next, Jahanian makes a distinction between synchronous and asynchronous monitoring. When the programmer explicitly checks for a constraint in a program and adjusts the computation based on that constraint, this is known as synchronous monitoring. On the other hand, asynchronous monitoring occurs when a constraint is enforced throughout the entire program. These approaches can be handles by embedded or monitored constraints, respectively. Then, the author focuses on the detection of timing violations, followed by bounded event histories, and operating system support for real-time system monitoring. Jahanian [4] provides an in depth explanation of monitoring real-time systems, a topic that most sources do not expend much effort is spent on describing how this is accomplished.

Allocation. Proactive and Reactive Resource Allocation [19] are solutions for two approaches to ensuring QoS constraints are met and maintained. The proactive approach supports situations where critical system changes can be anticipated. This approach requires advance specification of an exact set of circumstances to which the system may have to adapt. Then, the determination of an acceptable reallocation must be performed prior to the actual reallocation. Instead, Cross and
Lardieri’s solution requires only that the set of possible system states be divided into subsets, called modes (e.g., “battle mode”, “only one operational engine mode”, etc.). Next, a monitoring function that measures the system behavior should be implemented. Then, a triggering function must note when a significant change is reported in the system behavior. Finally, the allocation determination function can respond to the information provided to it by the triggering function. The authors add that the reactive resource allocation technique is sometimes used with the proactive resource allocation technique for the modes that do not involve safety and mission critical systems. One of the drawbacks to this approach is that not all modes can be anticipated. On the other hand, an option for situations where changes cannot be anticipated is the reactive pattern. This pattern is used when the state of the system changes dynamically, and cannot be predicted in advance. An example of this type of system is the Dynbench system [20]. The solution requires similar functions as the proactive approach: a monitoring function, a triggering function, and an allocation function. Aside from the allocation function, the other two act the same as they do in the proactive approach. The reactive allocation function differs from the proactive one by compensating for the fact that the adequate allocation cannot be predicted in advance. This paper captures two widely-used approaches to remedy situations where system requirements have been, or soon may be, violated. The author explains these techniques at a very high level, and only cites a couple of places where these techniques are used.

1.4 Pattern Frameworks

The reader will notice that the remainder of this thesis is not presented in a typical manner. This section explains the format in which the remainder of this document is formatted.

Many problems have already been solved in the dynamic, real-time systems domain and described in pattern format. A pattern is an easy-to-understand, accepted format for detailing a problem within a context, the forces that give rise to the problem, and the solution to the problem. Patterns are used in software development to simplify the complex language and frameworks associated with software development concepts. Additionally, patterns are useful teaching tools, allowing newcomers to an area, such as real-time systems, to understand the problems incidental to a particular area. Furthermore, patterns aid the experienced developer in understanding how related concepts fit together. When a group of patterns are presented with the intent that they work together, as presented in this thesis, this is known as a pattern language. Figure 11 demonstrates an interaction of the patterns in that chapter, or how the patterns can be used as a language.

Several types of pattern frameworks exist, each of which are appropriate for certain types of tasks. This section examines only three different frameworks, Alexandrian, GoF, and POSA, and determines which framework is best for representing patterns in real-time systems. These frameworks were chose for examination among an array of others because they are more commonly used in the real-time systems community.

The Alexandrian Framework. Christopher Alexander is often recognized as a pioneer of the art of writing patterns. In his book [21], he details a framework that he uses for describing patterns that appear in geographical regions, communities, and buildings. The framework includes seven elements. First, there is a picture, which is an “archetypal example of the problem.” This is useful for giving a reader, who is not familiar with the problem, a clue of what the problem is. The second element is the introductory paragraph, which is used to set the context of the problem. The next element is the headline, which states the problem briefly. Following the headline is the body, which
is the motivation of the pattern, and the longest part of this framework. The body explains the problem; why it is a problem by citing background and empirical evidence. The body is followed by the solution. The solution gives instructions to the reader for how to build the pattern. The solution is concluded with a diagram that shows the solution as a diagram. Finally, the closing paragraph describes those patterns that are needed to complete this pattern. The remainder of this book is divided into three main sections: Towns, Buildings, and Construction. Beginning at the highest level of identifiable community, independent regions, the authors describe situations that occur often in the most functional communities. And, to some degree, they describe patterns that make communities optimal in terms of spatial, political and social organization. Progress continues in this manner, breaking the problem down into smaller and smaller pieces, until an individual building within a community is reached. At this point, a similar approach is taken, describing optimal layout for a building, and then, in the last section, for constructing a building.

**The GoF Framework.** Design Patterns [22] is divided into two sections, an introduction to design patterns and a catalog of design patterns for object-oriented design of software. The pattern framework commonly known as GoF, which stands for Gang of Four, referring to the authors of the book is used. This reference is often referred to as the book on patterns for object-oriented design. The book also, like Alexander’s, adheres to a specific framework in which it describes patterns. The format includes the following: Pattern name and classification, where the classification refers to whether the pattern’s scope applies to a class or an object, and whether that class’s or object’s purpose is for creating objects, for maintaining structure, or for determining behavior. The next section is the Intent, followed by also known as, which lists any other well known names for the pattern. The motivation and applicability sections give an example and situations where the pattern should or should not be applied. The structure section provides interaction diagrams to illustrate the collaborations among other classes and objects. The participants section gives the classes and objects that participate in the pattern as well as their responsibilities. The collaborations section explains how the participants of the previous section work together. The consequences section lists the trade-offs of using the pattern, and the implementation section gives hints and techniques for implementing the pattern. Sample code provides code segments that exemplify the pattern. The known uses section gives examples of real world systems that use the pattern. Finally, the related patterns section explains which design patterns are related to the one being described, and lists the differences between them. The rest of the first section of the book explains how to choose and use design patterns. The second part of the book catalogues all of the design patterns using the format presented in the first half of the book.

**The POSA Framework.** The POSA framework is so named as an acronym of its book title, *Pattern-Oriented of Software Architecture* [23]. The POSA framework seems quite sparse at first glance, having only a context, a problem, and a solution. Closer examination of POSA allows one to see that, although only three elements are listed to the framework, patterns in the book have the following sections: an abstract, which is a concise statement of the solution. The next section provides an example of the problem, followed by the context of the problem, which is an explanation of the situation that gives rise to the problem. The problem section states what the issue of focus is and what the forces are that must be balanced. The solution is stated and is followed by the structure, dynamics, and implementation sections. These sections complete the solution through an explanation of how the solution fits together, an examination of situations in which the solution can apply, and implementation details, including code segments. The following section is example resolved, which shows how the solution applies to the example presented at the beginning of the
pattern. Variations of the solution are listed in the variants section of the framework. The known uses section allows one to list other incidents that utilize the same solution. The consequences section provides a list of liabilities and benefits associated with using the pattern. Finally, the see also section lists patterns that are related to the one presented. Aside from describing the framework, this book serves as an introduction to the idea of patterns, explaining the concepts of patterns and pattern systems, as well as how patterns can be used in software engineering. Additionally, the book provides a history of patterns and the pattern community. Aside from the introductions and explanations, a series of architectural patterns, design patterns, and idioms, which are low-level patterns are presented.

The format used in this thesis is an abbreviated POSA, having the abstract, context, problem, solution and consequences. This decision was made because these patterns were to be published in a book that was being edited at the same time as this thesis was being completed.

1.5 Summary

Throughout this section, a variety of literature has been presented that deals with existing real-time resource managers, existing solutions for load balancing that exist for non-real-time environments, and existing patterns that examine problems and their solutions in the real-time systems domain. It has been shown that, while these technologies and solutions may be suitable for a range of purposes, none of them address the larger-scoped problem of deciding which solutions are appropriate for a given real-time system or how develop the appropriate technology if necessary.

The remainder of this document is organized as follows: chapter two details a procedure for assessing and characterizing a real-time system and chapter three describes technologies that may be useful in engineering real-time systems, specifically dynamic, adaptive real-time systems. Finally, chapter four contains a list of definitions for terminology used throughout the document.
2 Real-Time System Characterization

The Real-Time System Characterization pattern addresses the problem of determining the type and environment of the real-time system. This problem is especially difficult because there are so many aspects of real-time systems, and certain aspects can affect others.

Example

When building a house, an abundance of options exist for nearly every aspect of the house, from structural materials to aesthetic finishing touches. However, limitations on these options exist, depending on certain design decisions. For example, the size of the studs used in the walls affects options for insulation, doors, and windows. A similar situation occurs when building real-time systems. Certain design aspects can affect which solutions are suitable for effectively managing the real-time system and ensuring that quality-of-service requirements are met. A real-time system that has hard deadlines have different set of scheduling algorithms suitable to ensure deadlines are met than those real-time systems that have soft deadlines.

Context

- Any real-time system that must be managed to ensure performance constraints are met.
- Any specific algorithm or tool for managing real-time systems.

Problem

In general, real-time systems must perform their tasks in a timely manner. For example, imagine two real-time systems, one that is responsible for producing a handle for a refrigerator door and another that is responsible for tracking the airspace surrounding an aircraft carrier. The timeliness requirements vary greatly between these real-time systems. In the first system, not performing a task before a specified deadline may result in a defective product or not meeting the daily quota. In the other system, the result could result in the destruction of the aircraft carrier and its personnel. Clearly, the severity of the timeliness is greatly different in these systems. Still, there are other considerations for characterizing a real-time system. The general purpose of a real-time system can range from gathering data to completing a process. Then, after determining the purpose of the system, what about the behavior of the system? Does it perform periodically, event-driven, or both? At this point, it is clear that there are a wide variety of factors by which to characterize the real-time system. Further, how can it be determined if and how one factor affects the value of some other factor? Characteristics such as these can easily be seen in any real-time system. Real-time systems have a variety of characteristics, such as workload and criticality. Once a RT system is accurately characterized, then how does one match an appropriate technology to it?

Solution

Use a characterization tool to characterize not only the real-time system itself, but also the the real-time system’s execution environment. Welch’s Taxonomy [24, 25] is an example of such a characterization tool. This approach uses Kiviat diagrams (see Figure 1) to plot the characteristics of each facet of the real-time system. Once all points are plotted, a line is drawn between the points,
creating a polygon. Once this has been done, apply the characterization process to the technology in consideration for managing the real-time system. It can be determined that the technology is suitable for the real-time system if the resulting shape for the technology can fit inside the resulting shape for the real-time system. Alternative approaches to Welch’s taxonomy for characterizing real-time systems may be found in [26] and [27]. To emphasize the array of characteristics to consider during the characterization process, Welch’s taxonomy is presented in detail, as an example. Welch’s Taxonomy has two aspects, as seen in Figure 1. There are five axes to characterizing the real-time system, and three axes for characterizing the environment in which the real-time system executes. The remainder of this section focuses on these axes.

Figure 1: Welch’s Taxonomy Uses Kiviat Diagrams to Characterize a Real-Time System (a) and its Environment (b)
Sidebar: Kiviat Diagrams

Kiviat diagrams, also known as radar charts, are used for pictorially representing data having multiple dimensions. Each of the axes of the diagram represents one dimension, and the values of a particular dimension are marked equidistantly along the axis with the “smaller” values near the center and the “larger” values proceeding toward the end of the axis. Once all dimensions have been selected, a single line is drawn connecting the value at each dimension, thus producing a polygon.

The process described in this section should be repeated for the technology being used to facilitate achieving the requirements (characteristics) of the real-time system. If the polygon for the characterization fits inside the polygon for the technology, then the technology is suitable for achieving the requirement.

Below is an example. Part a of Figure 2 shows a hypothetical evaluation of a technology intended for utilization with the real-time system characterized in part b. Clearly, their characterizations are not exactly the same. Part b shows hypothetical characterization of real-time system having periodic behavior, dependent task relations, adaption using slack, and an initiation control function. (The timing requirement has been omitted from this characterization due to its own multidimensional quality. This means that timing requirements must be characterized and compared for a fit at their own level.) In part c of the figure, the shapes obtained by connecting the values of the respective characterizations are compared, and the polygon from part b fits inside the polygon from part a. Since polygon for the real-time system fits inside the polygon for the intended technology, then this technology is suitable for achieving the requirements of the real-time system.

![Figure 2: An example of “fit” in a Kiviat Diagram](image)

Control Function

Typically, a real-time task may be characterized as either assessing, initiating, or guiding. An assessor contains a data source, a data set, and a data assessor. The data source, such as a sensor, periodically produces a set of data elements. The data elements are evaluated by the data assessor, which then decides if some action should be taken in response to the data received. An initiator, in
contrast to the periodic behavior of the assessor, is event-driven. The initiator has an event source, an event, and an event handler. The arrival of an event causes the event handler to plan and initiate an action in response to the event received. The guiding role is also activated by an event. Once activated, this type of system performs assessment and provides control for the responding action whenever assessment determines that control is needed. Finally, guiding is deactivated by an event.

Characterize the role of the real-time tasks as either assessment, initiation or guidance (see Figure 3).

Figure 3: The Kiviat Diagram for Characterizing a Real-Time System’s Control Function used by Welch’s Taxonomy

Behavior

Three types of behaviors are common among real-time systems: periodic, transient, and transient-periodic or hybrid (see Figure 4). A periodic behavior means that a process is activated at regular intervals. A process that is activated by sporadic events is known as transient behavior. The hybrid of these two behaviors means that a process can be triggered by transient events, executed at regular intervals when it is activated, and/or deactivated by action completion.

Characterize the behavior of the real-time system as either periodic, transient, or transient-periodic.

Timing Requirement

The timing requirement for a real-time system is multifaceted. Not only must one characterize how strict the timing requirement is, which is probably the most obvious aspect of timing, but also, it is important to capture the level of abstraction to which that strictness applies, for example. Other aspects of the timing requirement include the complexity and the granularity of the timing.

The strictness is based on the percentage of deadlines met for the hard, firm and soft classifications. Legitimate strictness levels are hard, firm, soft, importance, utility, and hybrid. A timing requirement is hard if 100% of the deadlines must be met, firm if a specified percentage window must be met, and soft if a maximal percent of deadlines must be met. The importance strictness is assigned if the deadlines are met based on relative importance. The utility strictness is based on
some benefit or time-value function. A hybrid strictness is some combination of hard/firm/soft with importance or utility with importance.

The abstraction of the timing requirement addresses the granularity of the entire real-time system. At the finest level, the abstraction may be at the instruction level. At increasingly larger levels, the abstraction of the timing may apply at the method level, the object level, the task level, or the task group level.

The complexity of the timing requirement is either single or multiple. A single requirement means that, if the real-time system has periodic behavior (see section 2), the cycle has a deadline, and if the real-time system has transient behavior, it has a completion time. On the other hand, a multiple (or hierarchical) requirement means that the behavior of the real-time system is transient-periodic and has both a cycle deadline and a completion time.

The granularity of the timing requirement pertains to the granularity of the deadline or completion time. The values for granularity are millisecond, second, minutes, or hours.

Characterize the timing requirement for the real-time system by determining its strictness, level of abstraction, complexity, and granularity. Classify strictness as one of hard, firm, soft, importance, utility, or hybrid. Classify the level of abstraction as instruction, method, object, task, or task group. Classify the complexity as either single or multiple. Finally, classify the granularity as millisecond, second, minute, or hour (see Figure 5).

**Task Relations**

Two types of task relations exist: independent and dependent (see Figure 6). Independent task relations are self-explanatory, however, dependent task relations come in several types of relationships. Dependent task relations may be total or partially ordered, may have initialization dependencies, and/or may have communication relationships.

Determine whether the real-time system has independent or dependent task relations.
Some real-time systems must adapt to a dynamic environment to meet their performance requirements. Methods for adaptation include resource allocation, degree of concurrency, slack, and precision. Resource allocation involves attempting to make more resources available to the real-time task. This approach may tell the competing portions of the real-time system to release some of the resources they are utilizing, allowing more for the component that is in need of them. Another method of adaptation enables a task, or process, to start a duplicate task, thus increasing the tasks degree of concurrency. The intention is that the multiple copies of the task can execute concurrently, accomplishing more computations, since the task is consequently occupying more processor time. Simply put, this is the same idea as getting a job done faster if more than one person is working on it. Also, adaptation can be accomplished by adjusting the slack of the task. Slack is the difference between the current execution time and the required execution time (to achieve the desired performance) of the process. For example, allowing greater slack in a real-time video stream may allow a server to provide video to a larger number of users (although the users may notice a delay in the arrival of frames). A similar type of adjustment can be made in the precision of the calculations of
the task. Precision affects multiple-fidelity algorithms, which give better answers as they process more information. As an example, a student may detail the steps to solve a series of equations on his homework, where time is presumably ample, but may simply write down answers on a time-limited test.

Describe the way the real-time system adapts as one of precision, slack, concurrency, or resource allocation. If the system falls into more than one category, choose the category that lies furthest out on the Kiviat diagram (Figure 3.5).

![Figure 7: The Form of Adaptation Kiviat Diagram for Characterizing a Real-Time System used by Welch’s Taxonomy](image)

**Data Arrival**

This section begins the portion of the characterization process that deals with the environment in which the real-time system will execute.

The performance of a real-time task depends on external, or environmental, factors. One such factor is the regularity and amount of data. In a static environment, nothing changes; the rate and amount of data is the same throughout the lifetime of the process’s execution. On the other hand, in a dynamic environment the amount of data, as well as the arrival rates may vary. Between these two extremes are time-variant stochastic and time-invariant stochastic. When the input size varies and can be characterized by a distribution and the arrival rate is the same, the data arrival is time-invariant stochastic, but if the input size distribution and the arrival rate varies, the environment is time-variant stochastic. The key here is whether or not the data size can be characterized by a distribution.

Characterize the execution environment for the real-time system as one of dynamic, time-variant stochastic, time-invariant stochastic, or static (see Figure 3.7).

**Environment Dynamics**

Not all real-time system execute in static environments. In these cases, the environment dynamics are characterized as one of constant, gradually changing, bursty, or hybrid. Characterize the environment dynamics as *constant, gradually changing, bursty, or hybrid* if the real-time system
Workload

Workload for a real-time system has several dimensions, which can be seen in Figure 3.8. Do we characterize workload by the size of the data stream, or by how often the data arrives? What about other factors such as the contents of the data stream and the rate at which data arrives? The fact is that each of these questions is important for characterizing the workload for a real-time system.

The first dimension concerns the size of the data stream. If the data stream will always be the same size, it is constant. At the other extreme is a dynamic data stream, where the data stream size fluctuates in an unpredictable manner. Between these extremes are three more possible data stream characterizations. The data stream is characterized as set if the size of the stream is one of a set of predetermined scalar values. If the data stream size can be any value within a continuous range, it is called interval. A distribution data stream size characterization occurs when the size of the data stream adheres to a statistical trend, such as Gaussian or Poisson.

Next, the event arrival rate of the workload must be characterized. Similar to the data stream size, there are two extremes, static and dynamic, for non-changing event arrival rate and changing event arrival rate, respectively. Again, event arrival rates can be characterized as set, interval, or distribution.

Third, the elements of the data stream must be characterized. When the stream elements are always the same, they are homogeneous. If the elements can be any of some set of predetermined...
possibilities, they are characterized as set. When the elements can be any within some continuum with defined endpoints, the elements are characterized as interval. When the elements of the data stream fall into a statistical curve, they are characterized as distribution. If the elements are not known, they are characterized as unknown.

Lastly, the period of the workload shall also be considered. When the period of the workload is always the same, it is fixed. The period can be dynamic, too. It can be characterized as set when the period will be one of a set of predetermined values, interval when the period can be any value within a continuous range, distributed when the period adheres to a statistical trend, or unconstrained if the period cannot be predetermined.

Characterize the workload of the real-time system on each of the four dimensions. Characterize data stream size and event arrival rate as constant, set, interval, distribution, or dynamic. Characterize the stream elements as homogeneous, set, interval, distribution, or unconstrained. Finally, characterize the period of the workload as fixed, set, interval, or unconstrained.

![Figure 10: Characterizing the Environment Workload using Welch’s Taxonomy](image)

**Consequences**

This pattern, Real-Time System Characterization has the following benefits:

- The developers of real-time systems are better aware of all dimensions involved in the characterization of a real-time system.

- This characterization process enables developers to better match technologies to the real-time system based on these characteristics, ensuring a better overall performance of the real-time system.

- This process can be performed prior to development or after the system is already in place. For the development of new systems, it is recommended that this process is applied to the system prior to design.

However, this pattern also has the following liability:

- Improper characterization can lead to problems in meeting QoS requirements.
Summary

Throughout this section, Welch’s taxonomy was used to exemplify the number of aspects of a real-time system that may need to be characterized in order to ensure that the real-time system is able to meet the specified performance requirements. To summarize this pattern, the real-time system itself must be characterized by its timing requirement, behavior, task relationships, forms of adaptation, and control function. Further, the execution environment of the real-time system is also characterized by examining the data arrival, the dynamics, and the workload.
3 Engineering Patterns

At this point, the Real-Time Characterization pattern has described a classification process for real-time systems. Real-time systems are characterized according to the systems themselves as well as the environment in which they will execute. This section presents solution patterns for a particular class of real-time system. In building a real-time system, these patterns work well for systems that are dynamic and distributed, and each pattern clearly states which types of real-time systems for which it is appropriate. Throughout this section, these patterns often refer to one another.

Figure 11 shows how these patterns contained in this section interact. Briefly, the Hardware and Software Profiling as well as the QoS and Resource Monitoring patterns produce useful data for performing the Allocation Policing function. Similarly, the System Specification holds information that, in conjunction with the Allocation Policing, allows Performance Analysis to be completed. From this point, the Allocation Policing information can be fed into the Allocation Optimization, or may be fed directly into the Allocation Enactment. Each of the boxes in Figure 11 will be explained in detail throughout this section.

3.1 Hardware Profiling

The Hardware Profiling pattern assesses the capabilities of the host(s) and network, hereafter referred to as a “cluster”, on which a real-time system executes. This pattern plays a key role in generating the information necessary to alleviate the issues that arise when heterogeneous computing and networking components are present in the cluster.

Example

Generally speaking, a distributed system consists of two or more computing units that communicate among one another via at least one network connection. All computers are not created equal. They vary widely based on intended usage, storage capacity, memory access speed, and, perhaps most significant, the capability of the CPU. Generally, desktop personal computers are less robust than servers, since usually a server does a much greater volume of processing than a desktop system. Even in comparing desktop systems to one another, a wide range of options exist.

Similarly, networks vary in their performance abilities. Many are familiar with the difference in performance of a 56Kbps Internet connection and a T3 Internet connection. They vary in their
speed and the amount of data they can carry. Clearly, enforcing real-time requirements of a system would require a basis for comparing these variations to have any guarantees on the performance of the system.

**Context**

- A real-time system enabled to distribute the execution of its constituent processes across non-similar hosts.
- A real-time system that is not distributed.
- A real-time system that communicates across heterogeneous network links.

**Problem**

Often, the constituent applications of a distributed real-time system can be divided so that all or part of any of those applications may run on a separate host than the rest of the system or application. This ability enables those applications to have access to more host and network resources than if the entire system was executed on one host, which is often a desirable trait of a real-time system. Even in the case that the real-time system is not distributed, a measure of the performance of the host on which it will execute is useful. But with a variety of host and network resources available, we need some means of determining each resource’s performance capacity. The following forces must be balanced:

- A real-time system may execute in a distributed environment, meaning the processes of the real-time system may be enabled to execute on separate hosts.
- All host resources in a particular cluster must be assessed for their performance capacity.
- All network resources in a particular cluster must be assessed for their performance capacity.
- A reliable performance assessment of the resources must be known prior to execution of a real-time system on those computing and networking resources in order to make guarantees about the performance of the real-time system.

**Solution**

Perform host profiling on each host in the cluster on which the real-time system may execute. Host profiling assesses the performance abilities of a host in comparison with a reference host. A specific example of host profiling is called a SPEC rating. SPEC is the Standard Performance Evaluation Company whose “mission is to establish, maintain, and endorse a standardized set of relevant benchmarks and metrics for performance and evaluation of modern computer systems” [28]. The SPEC benchmarks consist of a series of 18 different programs designed to measure the CPU, memory and compiler performance. The performance of the host being profiled is compared to that of the reference host, and SPECfp and SPECint are the two performance metrics produced. SPECfp is a measure of the a host’s floating point performance, and SPECint is a measure of a host’s fixed point performance. Other examples of hardware profiling are found in [29] and [1].
Also perform network profiling to assess the speed and latency of the network connections. Generally, network resources consist of the network interfaces on each host, the network devices (e.g., routers, switches), and the links that connect the hosts and devices. For the network interfaces and devices, consultation of the vendor documentation for the device will provide useful metrics like maximum bandwidth, forwarding latency, duplex capacity, and queuing scheduling. Link latencies can be assessed using utilities like ping [30].

![Diagram of communication system](image)

Figure 12: An example of a hardware profiler from [1]

**Consequences**

The Hardware profiling pattern provides the following benefit:

- The capabilities of each host are known.
- The capabilities of the network resources are known.

However, the Hardware Profiling pattern also maintains the following liability:

- Each resource, host and network, must be profiled individually. Therefore, this process should be performed prior to execution of the real-time system.

### 3.2 Software Profiling

The workload of dynamic real-time systems fluctuates throughout the duration of execution, making difficult the ability to make performance guarantees. The Software Profiling pattern addresses this difficulty by providing a method for measuring the resource usage of the dynamic real-time system.

**Example**

In the classic UAV (Unmanned Aerial Vehicle) scenario [17] (see page 12), the application supplying the video to the base station may be performing under a variety of conditions. To summarize, the UAV (unmanned aerial vehicle) surveys a target from a high altitude and at some distance from the safety of a base station, recording the activity of the target. The video recording is then transmitted back to the base station. Depending on the urgency of the situation, the acceptable rate at which the frames of the video arrive varies. The higher the frame arrival rate, the more resources used. For example, if the UAV is not surveying enemy targets, the frame rate may be low, and requires less application resources than if the frame rate is very high.
**Context**

- Real-time systems executing in a dynamic environment.

**Problem**

Ensuring dynamic real-time system requirements are met is a difficult task. Knowing the amount of processing, memory and communication required by a real-time system to operate under a particular set of circumstances provides a foundation for the solution of the problem, but collecting and computing such information at runtime would consume even more resources. Further, to perform effective schedulability analysis and resource allocation online, resource requirement information must be available during the execution of the application.

The following forces must be resolved:

- In dynamic real-time systems, the applications can use varying amounts of computing resources
- To ensure real-time performance requirements, information about the expected resource usage is needed.
- An average-case or worst-case scenario is generally inadequate for characterizing an accurate amount of resource usage.

**Solution**

Measure resource usage as a function of the workload. Generate resource usage profiles by gathering resource usage and processing time information at different workloads. First, determine a range of workloads (e.g., set the minimum equal to one, and the maximum equal to the workload that causes the host on which profiling is being performed to have 70% CPU utilization). Next, determine the minimum sample size that meets error and accuracy requirements (e.g., use the statistical Z-test [31]). Then, collect N profile samples at the determined workload. Finally, determine if enough profile points have been gathered to accurately predict execution time for this workload in the specified range (e.g., perform prediction by using piecewise linear interpolation between points and check errors of the predictions). If not, divide the interval in half, and gather N profile samples at the midpoint of each subinterval. This process is explained fully in [2]. Other examples of software profiling can be found in [32], [33], [34], [35], and [36].

**Consequences**

The Software Profiling pattern provides the following benefits:

- Information about resource consumption in various scenarios is available so that schedulability analysis and online resource allocation may be performed.
- This information obtained enables the proactive resource allocation, foretelling a problem with the current allocation and compensating for it before QoS requirements are violated.

The Software Profiling pattern presents the following liabilities:
• Profiling an adequate range of scenarios takes time.

• Overlooking a possible scenario can negatively affect the performance of a real-time system.

3.3 System Specification

Each real-time system has a series of properties that must be available for the real-time system manager and possibly other parts of the system. These properties may include the location of needed files, the quality-of-service requirements, and the software profile, among others. The System Specification pattern provides a method for specifying properties for those components that need it.

Example

Suppose a person is purchasing anti-virus software for their computer that can be configured to automatically download virus updates. During the initial purchase, certain information will have to be provided to ensure the correct version is purchased for the intended computer. For example, the computer user will need to specify on which operating system the software must run and perhaps even the processor architecture of the computer receiving the software. Without such specifications, the software may not even run on the computer. Next, to configure the automatic download of updates, the computer user may have to specify the geographic location of the computer as well as the speed of the connection the computer has to the Internet. This information about the computer’s location and connection may impact the efficiency of the download. Similar to these configuration specifications, real-time systems have specifications that may vary depending on the environment in which it will execute. The hosts in the cluster may be running both Linux and Windows operating systems, but the real-time system may only be able to execute on the hosts using the Linux operating system. When the user is asked to specify geographical location, performance capability is being taken into account; the closer the download site, the faster the resulting download. In real-time systems, performance can be a very important factor in measuring quality of service. Although the
computer user usually cannot require a specific download speed, generally a real-time system must specify its performance requirements.

**Context**

- A real-time system that is distributed, where host characteristics vary.
- A dynamic real-time system whose resource usage fluctuates depending on system load.
- A real-time system that has performance constraints that may or may not vary during the lifetime of execution, which must be specified.

**Problem**

Each real-time system has unique characteristics. These characteristics may consist of performance requirements, operating system dependence, replication limits, quality-of-service information, and the location of files needed by the real-time system. Sometimes several applications make up a real-time system, where potentially each application has its own set of specifications. The question is “how can this information be communicated to the real-time system manager?” Sure, this information could be programmed into the components that need it, but then, if and when this information changes, the components must be rebuilt, at a minimum. The following forces must be resolved:

- Specification information of the real-time system must be able to be edited easily, since that information could change on a somewhat frequent basis.
- All types of relevant information, such as quality-of-service requirements, must be able to be provided in the specification to ensure proper control and adjustment of system performance.

**Solution**

Develop a text-based format for specifying the properties of the real-time system. The format should allow accurate and unambiguous specification, and should be stored in a file. It is helpful for the format to include syntax for comments, so that developers may describe the information that is contained in the specification. A specification may need to include the location of executables, the communication path among the constituent applications of the real-time system, hardware profiling information, software profiling information, quality-of-service thresholds (e.g., frame rate, data processing rate), quality-of-service deadlines, and other quality-of-service specifications such as utility and importance. If the system supports replication of processes, the system specification can be used for details about replication limits. If the system is distributed and process may be migrated from one host to another, the list of eligible hosts for those processes to execute could be included in the specification, especially if the cluster contains hosts running heterogeneous operating systems. Examples of this pattern can be found in [16], [4], [37], [3], [38], [39], [40], and [41].

**Consequences**

The System Specification pattern provides the following benefits:

- Needed real-time system properties are collected into a single location.
The approach is easily customizable and adaptable. The files may be edited as needed, without having to change program code each time something changes.

The System Specification pattern has the following liabilities:

- File reading can be time-consuming and an undesirable task during the execution of the real-time system.
- If the specification must be read during the startup of the real-time system, changes to the system specification may require that the real-time system is stopped and restarted to be detected.

3.4 QoS Monitoring

The QoS Monitoring pattern solves the problem of collecting information about the QoS performance of the real-time system and providing the information to the software analyzing the allocation of the real-time system.

Example

Suppose one is about to purchase a vehicle. The shopper may consider various measures of quality of service of a vehicle to determine which vehicle to purchase. For example, when test-driving the vehicle, the shopper may track the number of miles it can travel with a single tank of gasoline and compute the number of miles per gallon the vehicle achieves. The shopper may desire that a vehicle utilizes a certain number of miles per gallon, where that number may fluctuate depending on whether driving in city traffic or on in freeway traffic. The process of assessing the number of miles per gallon is an example of QoS monitoring. Certain types of real-time systems, especially those that are dynamic, require a similar type of monitoring to ensure that the system is meeting the QoS requirements established for that system.

```
<Proposal>
  <mode>
    <or>
      <ci name="radioVHF" state="onLine"/>
      <ci name="radioUHF" state="onLine"/>
    </or>
  </mode>
  <QoS type="latency">
    <upperPoint sec="1.0" prob="0.99"/>
    <upperPoint sec="4.0" prob="0.9999"/>
  </QoS>
  <load type="interMessageTime">
    <upperPoint sec="1.0" prob="0.0001"/>
    <upperPoint sec="1.0" prob="0.9999"/>
  </load>
  <load type="messageSize">
    <upperPoint bytes="256" prob="1.0"/>
    <upperPoint bytes="32" prob="0.5"/>
  </load>
  <load type="priority">
    <urgency val="10"/>
    <importance val="2"/>
  </load>
</Proposal>
```
Context

- Dynamic, adaptive real-time systems whose workloads and needs change during the course of execution.
- Non-dynamic real-time systems where a monitor of how the system is performing is useful.

Problem

Real-time systems often execute in environments where some factors cannot be controlled, such as when and how many unrelated processes will be spawned during the lifetime of the real-time system. Such factors may become increasingly harder to control in distributed systems. In dynamic real-time systems, the nature of the system is that environmental factors will change, and these factors cannot be known a priori. So, how does a real-time system that executes under such conditions have any guarantees on its performance? Assurance that the real-time system can have the necessary resources to complete its tasks must be provided. To solve this problem, the following forces must be resolved:

- Monitoring the system must not consume excessive resources [19].
- Information gathered must be able to be communicated to other processes that may analyze the information.
- The monitoring and reporting process should be a robust, reusable solution that can be applied to a variety of types of real-time systems.

Solution

Gather and communicate QoS performance information to the application responsible for assessing this information (see Allocation Policing). Insert reporting mechanisms into the real-time system and develop a component to receive and process the reported information. Metrics, like the workload of the process and the CPU and memory utilizations of the process should be gathered. Processing this QoS information may require aggregation of many process-level events into a single QoS metric. For example, in periodic and distributed systems, this aggregation can be simplified by tagging and time-stamping event messages. When monitoring a particular process, or a set of end-to-end processes (e.g., a path), determine the end-to-end latency of that process for one unit of the workload by collecting a timestamp, cycle number, and the event (e.g., start, end). Subtracting the values of the start timestamp from that of the end time stamp to determine the latency. Note, the cycle number is a number from one (1) to n that records the number of times that process has been executed. Examples of this pattern may be found in [42], [16], and [8].

Consequences

The QoS Monitoring pattern provides the following benefits:

- Provides a robust, adaptive solution for gathering QoS performance to notice when needs of the real-time system change.
• Provides a monitoring tool for non-dynamic real-time systems, whose performance may not necessarily need further assessment.

However, the following liabilities also exist:

• Gathering and communicating this information takes time that could otherwise be used by the real-time system

• The types of useful metrics gathered for each real-time system may vary. Ensure that the implementation is generic enough to handle most situations.

---

**Figure 15: An example of a QoS monitor from [4]**

---

### 3.5 Resource Monitoring

The Resource Monitoring pattern assesses the usage and availability of resources throughout the cluster on which the real-time system is executing.

**Example**

To continue with the example in the previous section, the automobile shopper may closely monitor the gasoline gauge on the vehicle, since he may not initially know how far he can travel before needing to refuel. Further, the driver may notice that the car consumes fuel at various rates, depending on the type of driving he is doing (e.g., city driving versus freeway driving). Monitoring the amount of gasoline remaining in the vehicle can be considered a type of resource monitoring for the vehicle. In a similar fashion, dynamic real-time systems whose resource needs fluctuate depending on the situation may need to have their resources monitored to allow reliable resource management.

**Context**

• Real-time systems where it is useful to know resource utilization of CPU, memory, network, and possibly others.
• Dynamic, adaptive real-time systems where resource usage may fluctuate depending on the workload the system is processing.

• Distributed real-time systems where environmental factors may dramatically affect network bandwidth availability.

**Problem**

Intelligent attempts to design an allocation that satisfies the system objectives needs not only the information about the performance of the software, but also the performance availability and state of the hardware resources. Useful information about the hardware resources may include resource status (e.g., off, on, standby) and utilization, as well as device-type-specific attributes, including context switching rate and free memory for host resources, and for network resources, the expected latency, available bandwidth, and collisions. This information should be reported to an application that can perform some analysis on it (see Allocation Policing). The following forces must be resolved:

• Monitoring the resources must not consume excessive resources [19].

• The hardware resource information gathered must be able to be communicated to other processes that may analyze the information.

• The monitoring and reporting process should be a robust, reusable solution that can be applied to a variety of types of real-time systems.

**Solution**

Provide a means of obtaining information about the host and network resources and tracking the historical utilization values and trends for the resources. Provide aggregate load indices for resources, such as a host load index that combines normalized values for CPU utilization, context switching rate and available memory. Such values can be obtained by exercising operating system utilities. Examples of the resource monitor pattern can be found in [43], [44], [45], and [46].

**Consequences**

The Resource Monitoring pattern provides the following benefit:

• Provides a robust, adaptive solution for gathering resource usage.

However, the following liabilities also exist for the Resource Monitoring pattern:

• Gathering and communicating this information takes time that could otherwise be used by the real-time system

• The types of useful utilization measurements may vary depending on the hardware on which the real-time system will execute. Ensure that the implementation is generic enough to handle most situations.
3.6 Allocation Policing

The Allocation Policing pattern provides a reusable solution to the problem of determining how well the real-time system is currently performing in comparison to the required QoS specification, allowing for a proactive approach to resource management.

**Example**

Surveillance systems are installed to monitor an area, such as a parking garage. Assuming a guard is watching the surveillance, if she notices a suspicious person appearing to vandalize a car, she will dispatch another guard to try to arrest the person, because such action is not acceptable. (This does imply that if the guard herself left the surveillance post to perform the arrest, she is no longer watching the surveillance and another problem may go unnoticed. If she dispatches another guard to deal with the situation, surveillance continues, but more resources are being used.) When a person is doing nothing wrong, she will not bother them, because their actions are acceptable. This type of scenario can be related to RT systems. It is important to ensure that the applications are performing acceptably (e.g., meeting their QoS requirements) and to notify someone (e.g., another program) when performance is unacceptable, so that the situation can be remedied.

**Context**

- Dynamic real-time system where performance can fluctuate, depending on environment, and advance knowledge of the workload is not known.

**Problem**

Occasionally, a real-time system must be evaluated for its performance, especially if it is dynamic. Some examples of dynamic real-time system include those in which environmental factors can consume resources unexpectedly, the workload of the real-time system may change, requiring more resources than originally allocated for it, a host or network link may fail. In any of these cases, no
tool is built in to the real-time system so that the manager can know that a problem exists. More than likely, a system will fail to meet QoS requirements before a problem is detected. Consequently, little opportunity exists for a successful proactive approach to managing resources for the real-time system. So, it may be wise to reevaluate the allocation if it is noticed that the workload has changed significantly or that the execution time is steadily increasing. Similarly, if it is noticed that the system is unstable, the processes for improving the allocation should be triggered. To solve this problem, the following forces must be resolved:

- Problems in meeting QoS requirements must be detected as soon as possible.
- Careful assessment of the problem must be made in order to not propagate a cycle of violation, notification, reallocation.
- A set of “signs” can indicate that a problem may exist. A real-time system manager must realize these situations and act on them intelligently.

**Solution**

Create an algorithm that can analyze performance and workload data pertaining to the real-time system, as well as that of the hardware resource performance. This policing function should monitor environmental-related conditions and trigger analysis of the allocation whenever conditions have changed sufficiently to warrant reanalysis. Consider the trade off of accuracy of analysis versus the overhead involved in achieving the accuracy. The rate at which the policing function triggers reanalysis, the triggering rate, affects both aspects of the trade off. If analysis is never triggered, the result is comparable to an offline approach, like RMA [47], which perform poorly in dynamic environments. On the other hand, if analysis is performed continuously, the analysis will be very accurate, but the overhead may further obstruct the performance of the system. Aside from RMA [47], other examples of allocation policers are found in [48] and [49].

![Figure 17: An example of an allocation policer from [6]](image)

**Consequences**

The Allocation Policing pattern has the following benefits:

- Constant surveillance of performance allows for better overall performance by preventing more QoS violations.
Policing can alleviate instability of individual processes or the real-time system as a whole.

However, the following liability exists, too:

- The constant surveillance of the system detracts from the resources that could be allocated for the real-time system to use.

### 3.7 Adaptation Analysis

The Adaptation Analysis pattern solves the problem of improving the current allocation, enabling real-time system tasks to better meet their QoS requirements.

#### Example

Dynbench [20] is a distributed real-time system that simulates a radar and tracking system that may be found on a naval ship or destroyer. Dynbench simulates the detection and evaluation of potential threats, also known as *tracks*, in a perimeter surrounding the destroyer. Tracks must be evaluated as either threats or non-threats within a specified time limit as a quality-of-service requirement. If a track has been determined to be a threat, the system simulates the targeting and destroying of the threat, again with a specified quality-of-service deadline. So, when more tracks are present in the perimeter and need to be evaluated, the workload of the system is higher. Further, the higher the number of threat tracks, the higher the amount of processing required by the targeting and destroying processes. The DSDRT system exemplifies a need to adapt to processing demands in order to ensure QoS requirements are met.

#### Context

- Dynamic, adaptable real-time systems where an increase in slack is not detrimental to the overall execution of the system.

- Dynamic, adaptable real-time systems where workload varies.

#### Problem

Generally, improving upon the current system performance is beneficial to the execution of the system. While the current allocation of resources to real-time system processes may not be “bad”, it may be helpful to know the answers to questions like: What if process X was moved to host B? Will replicating process Y improve the systems performance? and Host A is not doing anything, is there a process that can be moved there? So, an estimate of how the real-time system will perform when an allocation is enacted is especially important in systems that must contend for resources. Such an ability would provide a method of creating a contingency plan. In reactive systems, an alternative solution must be available quickly. In proactive systems, the ability to see the results of the reallocation may allow quicker action in preventing QoS violations. But, having a contingency plan is only useful if that plan works by meeting the QoS requirements of the system. To accomplish these desires, the following forces must be resolved:

- Modeling contention for resources among the constituent real-time system tasks must be determined without affecting the current allocation of resources.
• Performance estimates must be computed quickly to be effective. Dynamic systems can change workloads quickly. To adapt effectively, the assessment of a new allocation should complete before the system changes so much that this new allocation is no longer helpful.

• Suitable allocations must not worsen the current QoS.

Solution

Predict the real-time QoS by modeling CPU contention. The CPU contention accounts for the individual application $a$ in the real-time system, processing a specific (e.g., current) workload $w$, on a specific host $h$, $C_{pred}(a, w, h)$. A path is a series of one or more applications where the subsequent application must wait for the output from the previous applications. For each application in the path, values for the performance of the application under a workload, $C_{profile}(a, w, h)$, can be collected according to the Software Profiling pattern. The software profile ($C_{profile}$) multiplied by the SPEC rating for that specific host $SPEC(h)$, such as the Hardware Profiling pattern explains.

The CPU contention should take into account the queuing delay $D_{pred}$, which can be estimated using the CPU utilization percentage (CUP), multiplying $C_{pred}$ times the CUP. To summarize, we have:

$$C_{pred}(a, w, h) = C_{profile}(a, w, h) \times SPEC(h)$$

$$D_{pred}(a, w, h) = C_{pred} \times CUP$$

, and latency is

$$\lambda_{pred}(a) = C_{pred}(a, w, h) + D_{pred}$$

CPU contention may also take into account the probability that task executions will overlap, summing the $C_{pred}$ for each application in a path and dividing the sum by the Least Common Multiple among maximum arrival rates and periods of all tasks assigned to $h$ divided by the maximum arrival rate or period of $a$, $T(a)$. Doing so, $D_{pred}$ now is:

$$D_{pred}(a, w, h) = C_{pred}(a, w, h) \times CUP(h) / \left( \sum C_{pred}(a, w, h) \times (LCM/T(a)) \right)$$

Once the contention is modeled for each application in the real-time system, verify this new allocation is feasible. Is $\lambda_{pred}(a)$, the latency estimation of application $a$, less than $\lambda_{req}(a)$, the required latency of $a$? (The required latency may be stored as part of the System Specification.) If the predicted value is less than or equal to the required value, the allocation is feasible. Examples of this pattern may also be found in [47], [50], and [7].

Consequences

The Adaptation Analysis pattern has the following benefits:

• In reactive RT systems, this method generates an alternative allocation, which may be saved for when it is needed.

• In proactive RT systems, an alternative plan is generated and ready to be enacted.

However, the following liabilities are present:
Figure 18: An example of the results of adaptation analysis in [7]

- The performance estimates produced may not be effective if the system changes before a new allocation is enacted.
- In proactive systems, if a new plan uses a resource that becomes unavailable, must generate a new allocation, perhaps converting to a reactive response mechanism.
- A feasible allocation solution is obtained, but one may consider using the Allocation Optimization pattern to optimize the solution.

### 3.8 Allocation Optimization

The Allocation Optimization pattern provides a reusable solution for finding allocations that not only meet the QoS requirements, but also attempt to be the best feasible allocation.

**Example**

Assume that a person is saving money for retirement. Suppose the retirement plan her company offers allows her to contribute a certain dollar amount each year to her retirement plan based on her pay, and she contributes that amount. Some formula forecasts that she will need a particular dollar amount in the retirement fund on the day she retires to continue living in the lifestyle to which she is accustomed. Now, suppose that she received a raise, but still continues to contribute the same amount of money to her retirement plan as she did prior to receiving the raise. To optimize the amount of money she has on retirement day, she should have maximized her contribution when she got the raise. A similar type of problem arises in real-time systems. Often, the solution for “right now” is applied, but this solution does not plan for the future, potentially costing the system more allocations than if a better solution applied in the first place.

**Context**

- Real-time systems whose allocations of processes to resource change throughout the execution lifetime.
Problem

The system should have an allocation that is not only feasible, but is also attempts to be a best-effort allocation given the current configuration of resources. Before carrying out a suboptimal allocation, which is likely to require reallocation sooner than that of an optimal one, allow the system to improve the allocation to gain better performance. The solution for this problem requires that the following forces are resolved:

- Attempts to improve the current allocation should not use excessive amounts of resources. The goal is to improve the allocation but not to create a need for another reallocation due to the amount of resources used in the improvement process.

Solution

Provide functionality to improve the current allocation. The goal is to avoid performing an excessive number of reallocations, so improving upon a solution that is simply "feasible" is beneficial. Some methods for allocation optimization include load balancing, maximize the minimum slack of all hosts in the environment, and adjusting the utility of an application. Load balancing, for example, can improve an allocation by distributing the workload of a distributed real-time system’s constituent applications across the hosts in the execution environment. This approach can lead to less contention for resources, providing more resources to the applications to help meet QoS requirements. Examples of this pattern are found in [51], [52], and [53].

Figure 19: An example of an allocation optimizer

Consequences

The Allocation Optimizer has the following benefit:
Finding a better allocation now may prevent an incident of the somewhat costly job of finding another allocation in the near future.

However, the following liability is present:

- The optimization algorithm can take time that, in cases, may be better spent enacting the current solution.

3.9 Allocation Enactment

The Allocation Enactment pattern solves the problem of executing a new allocation plan.

Example

Change is a fact of life. If humans are not able to respond to such changes, survival is not likely. Options exist for what to do in the face of change; we can adapt to it or try to ignore it. Generally, not all people will react in the same fashion, they each react in a way that is suitable for themselves. Just as people are inherently equipped to function and react to a change, real-time systems must also be able to react to change in order to continue executing successfully.

Context

- A real-time system whose allocation of processes to resources can change throughout the lifetime of execution.
- A dynamic real-time system whose workload will fluctuate throughout the lifetime of execution.
- A distributed real-time system that can reallocate its applications to hosts with more resources available.

Problem

During the lifetime of execution for a real-time system, a wide variety of factors may affect the allocation. In an adaptive system, the real-time system may be equipped to create and remove replicas of one or more of its constituent processes, or to change the precision of the solutions being produced. In RT systems that are distributed, the real-time system may be enabled to distribute its constituent processes across a number of hosts in the cluster. Throughout the execution, distributed processes may be stopped on one host and resumed on another. Adaptive, distributed real-time system could employ any or all of these methods. Note, the methods of reallocation described here is not intended to be exhaustive. The following forces must be resolved:

- For a real-time system to be managed, it must have the functionality to enable allocation changes to be carried out.
- A method for carrying out the adaptation for adaptive real-time system must exist.
- Distributed real-time systems must have the ability to distribute their workload.
Solution

Provide methods for controlling the real-time system. For example, provide mechanisms for process migration and replication; this can be implemented most efficiently within an operating system, but may also be implemented using common process control services. Another common mechanism is to adjust the resource usage of the real-time system by changing the fidelity of one or more of its algorithms; this can be accomplished by having the real-time system provide a control interface, as in DQM [8]. Additional examples of this pattern can be found in [54], [9], [37], [44], [46], and [55].

Consequences

The Allocation Enactment pattern has the following benefits:

- Real-time systems that are dynamic are enabled to adapt to new conditions and continue to meet QoS requirements.

However, the following liabilities are also present:

- Frequent changes of allocation may bring a point of diminishing returns; a new allocation may be better than the previous, but the overhead in changing the allocation is more costly than enacting it.

3.10 Conclusion

After working with a number of distributed and real-time systems, two observations were made. The first observation was that these systems, which did similar types of work, were developed using similar components, as discussed in Chapter 1. They used similar approaches to solve similar problems. The second observation was that real-time systems are complicated pieces of software, and if the system was distributed and/or dynamic, they were even more complicated. Describing such systems accurately is a hard problem that must be solved in order to ensure real-time performance. The large number of facets that can be used to characterize real-time systems was exemplified using Welch’s Taxonomy, one of several existing models of classifying real-time systems. Once a real-time system is accurately characterized, that information should be used in designing solutions customized for that particular system. In Chapter 3, a number of solutions were described for different parts of a real-time system: gathering initialization information in the Hardware and Software Profiling and the System Specification pattern, assessing whether the system’s performance is satisfactory using the Allocation Policing pattern, finding an improved allocation with the Adaptation Analysis and Allocation Optimization patterns, and finally executing a new allocation using the Allocation Enactment Pattern. These solutions are highly dependent on which type of real-time system is being developed, and should be chosen according to how the system was characterized in Chapter 2.

After these observations and the formulation of these patterns, these pattern’s solutions have been applied in the re-engineering of the DeSiDeRaTa resource management system and recommended for use in developing new real-time management solutions.
4 Terminology

**real-time system** Systems in which the correctness of the system depends on the results of the computation as well as the time at which the results are produced [26].

**distributed system** A software system that is made up of one or more applications that can execute on a set of hosts in a client-server fashion.

**cluster** A term used to describe the set of eligible hosts and the network that connects the hosts on which a distributed system executes.

**adaptive** As applied to real-time systems, refers to the fact that the system is dynamic and enabled to compensate for changes in workload while continuing to meet its performance requirements.

**quality of service (QoS)** A requirement for the performance of the real-time software, such as execution deadline, user-perceived benefit, or image resolution.

**technology** The software, which may be commercial off-the-shelf (COTS), or engineered in-house, used for remedying a particular problem, as presented in a particular pattern.

**component** A software process or application. A real-time system may consist of one or more processes or applications, or components.

**kiviat diagram** See sidebar in section 2.
References


[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


