Dynamic Resource Management Architecture Patterns

Lonnie R. Welch
Toni Marinucci
School of EECS
Ohio University
Athens, OH
welch@ohio.edu
tmarinuc@cs.ohiou.edu

Michael W. Masters
Paul V. Werme
Naval Surface Warfare Center
Dahlgren, VA
MastersMW@NSWC.NAVY.MIL,
WermePV@NSWC.NAVY.MIL

FOCUS TOPIC: PATTERNS AND PATTERN LANGUAGES FOR DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

I. Introduction

"In 1985 the Internet connected 2000 computers. At the start of this century the Internet connected over 37 million computers. Future networks will connect at least a billion users and will be more complex – they will connect sensors, wireless modems and embedded devices[1]."

To address the challenges presented by such tremendous growth, it is important to identify software patterns and to develop appropriate pattern languages for the domain of distributed, real-time computing. Distributed computing refers a pool of networked computers that can be used for solving complex problems. To harness the computational power of a pool of networked computers, resource management (RM) software that dynamically allocates computing resources to programs has emerged. Furthermore, technologists have produced RM software that allows real-time systems (systems that must perform in a timely manner) to meet their performance constraints in dynamic environments (such as defense [6] and space [5]) by (1) monitoring real-time performance and (2) reallocating resources as needed to provide adequate real-time performance [1, 2, 3].

This paper presents patterns (i.e., solutions to commonly occurring design problems [4]) that have emerged in resource management software. The specific patterns discussed are as follows:

- Resource instrumentation: monitoring the status (e.g., availability and utilization) of hardware resources, such as computers and networks.
- Software performance monitoring: monitoring the status (e.g., performance) and resource needs of software systems.
- Allocation planning and management: determining how and when to allocate hardware resources to software systems.
- Resource control: performing the allocation of hardware resources to software systems.

Each of these patterns is detailed in the remainder of this paper; the patterns are characterized according to the following pattern template (described in [4]): (a) name of pattern; (b) summary of the problem and forces that give rise to it; (c) a solution in terms of collaborations with participating classes; and (d) summary of benefits and consequences of using the pattern.

The remainder of the paper is organized as follows. Sections III-VI details the patterns found in resource management software. Section II presents a UML-based model of an adaptive resource management architecture, which is elaborated to describe the patterns. The domain, in which the patterns have been identified, is characterized by a reference architecture in Section VII.

II. Computing and network resource management architecture

This section contains a UML-based model of an adaptive resource management architecture. The patterns are defined in the remainder of the article by refining the UML model.

The architecture (see Figure 1) of our resource management (RM) middleware consists of five major subsystems: User Management, Allocation Management, Real-Time System Management, Resource Instrumentation and Control, and Specification File Management.

[1] From the Information Technology Office of the Defense Advanced Research Projects Agency.

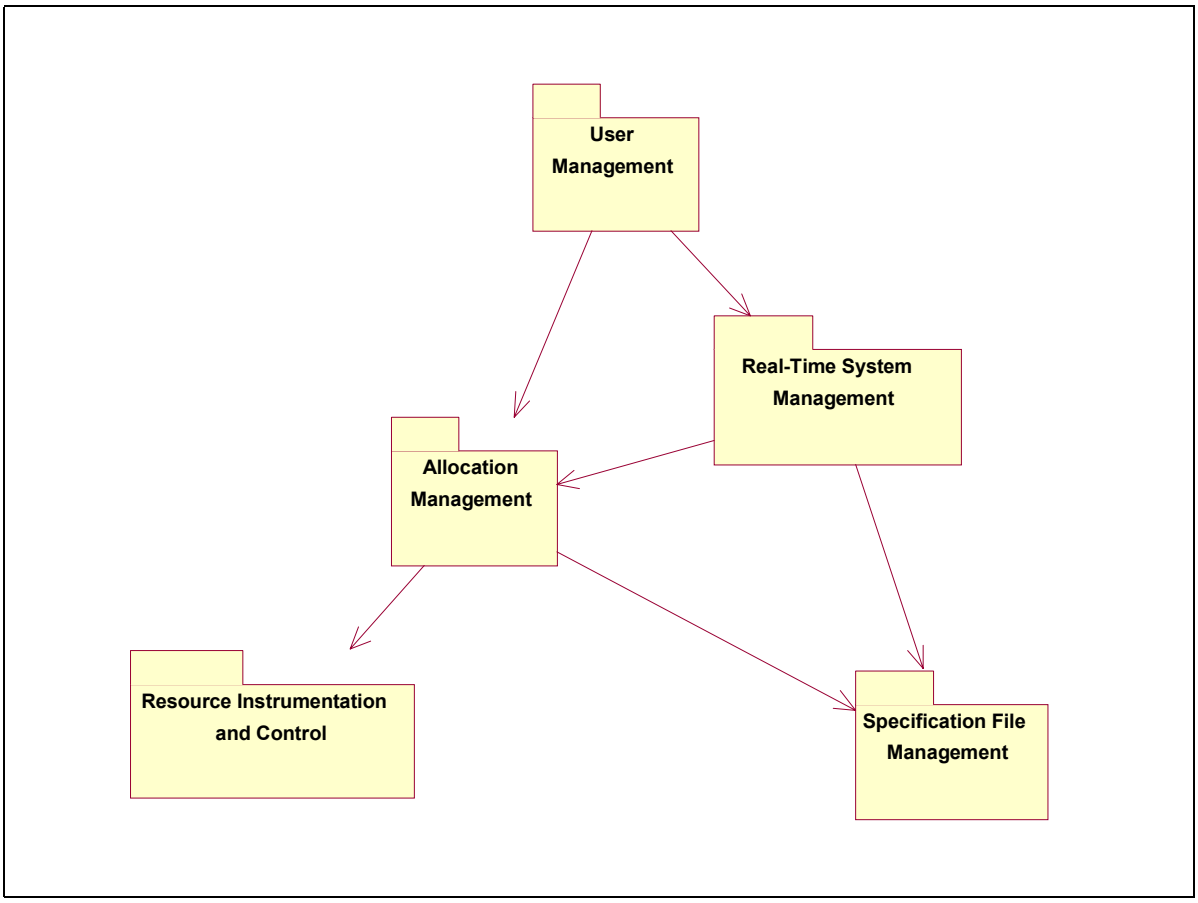

Figure 1. The major subsystems of the middleware and their dependencies.

Specification File Management parses hardware configuration and software specification files; the specification files describe the characteristics of the computing and network resources and the features and real-time requirements of the information system software [1, 2]. The other subsystems use the information from the specification files.

The Resource Instrumentation and Control subsystem has two main purposes. First, its Resource Monitor component is used to gather information about the utilization and availability of the computing and network resources [2, 3, 9]. Second, its Application Control component is used to start and stop the application programs that constitute the information system software.

The User Management subsystem allows an Operator of the RM middleware to give commands to the Allocation Manager to start or stop a real-time system. It also allows the RM Operator to view a real-time system's performance.

The Real-Time System Management subsystem monitors the performance of real-time systems and provides updates to the User Management subsystem. When real-time performance problems are detected, Real-Time System Management performs diagnosis of the causes and of possible resource reallocation actions that could be taken to restore required real-time performance, and reports its findings to the Allocation Management subsystem.

Allocation Management uses Resource Instrumentation and Control to (1) gather information about the resources, and (2) start and stop application programs. The resource information is used to maintain a feasible allocation (one in which all real-time requirements are met) and that provides optimal utility to all information systems under its control.

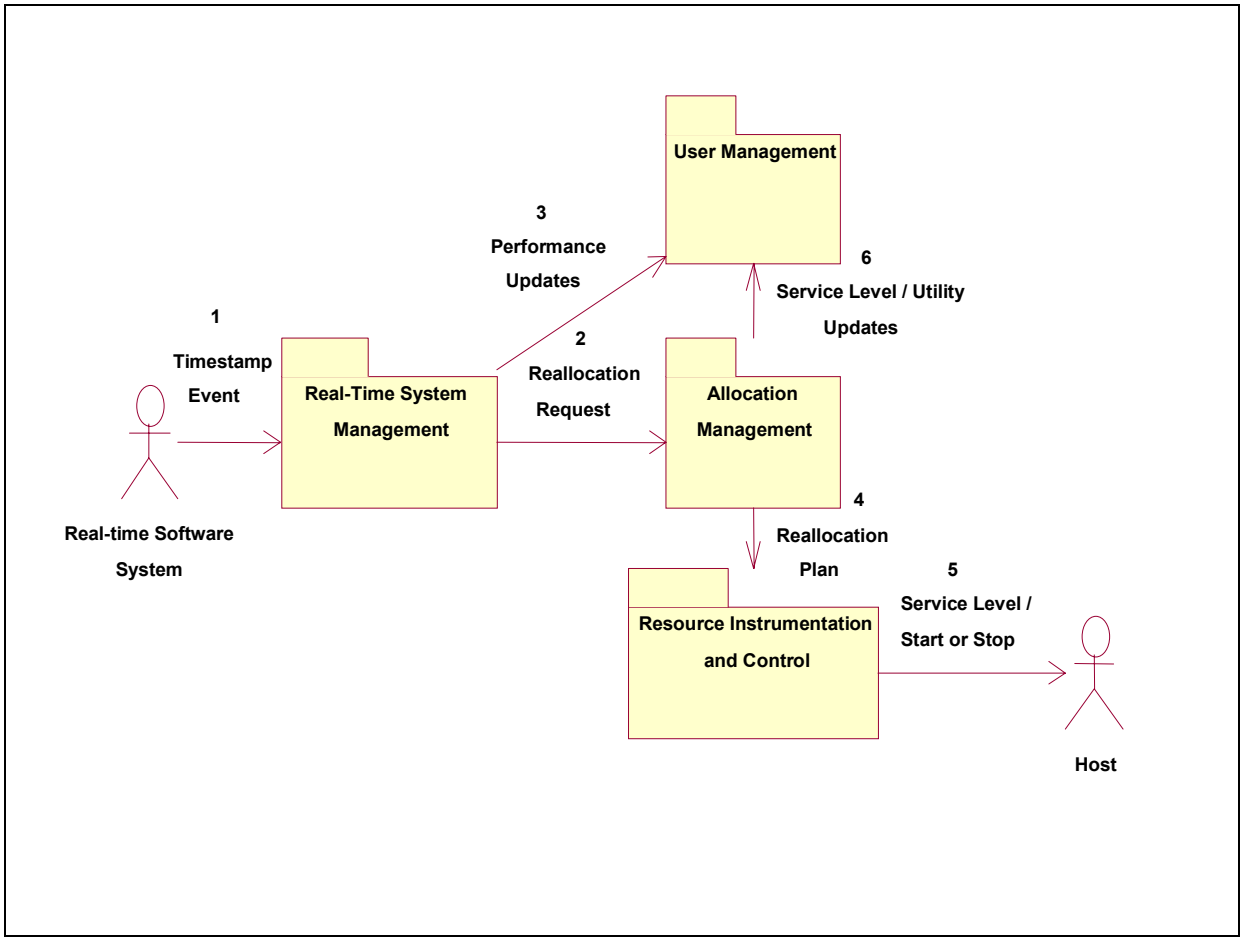

Figure 2. The subsystem collaboration diagram for the *maintain feasible allocation* use case.

The most critical use case of our software system, *maintain feasible allocation*, is illustrated in Figure 2. Real-time systems report their performance data to Real-Time System Management, which monitors real-time performance and requests that Allocation Management reallocate resources if a real-time performance problem is detected. Allocation Management creates a reallocation plan and uses Resource Instrumentation and Control to execute the plan.

III. Resource instrumentation

The resource instrumentation and control subsystem is subdivided into two service packages, *resource monitor* and *application control*. This section describes the *resource monitor* subsystem, which represents the *resource instrumentation* pattern, and section VI describes the *application control* subsystem, which describes the *resource control* pattern.

A. Summary of the problem and the forces that give rise to it

The resource instrumentation pattern solves the problems of collecting and maintaining information about the recent state of computing and network resources. The types of resources include hosts, networks and devices. The state of a resource includes its status (on/off) and its utilization, as well as several device-type-specific attributes, including the following:

- Host: context switching rate, free memory, etc.
- Network: expected latency, available bandwidth, collisions, etc.

Additionally, the instrumentation pattern solves the problem of maintaining historical information, calculating trends, forecasting future resource states, analyzing stability, and diagnosing causes of problems.

The desire to find good solutions to the problem of allocating computing and network resources to distributed systems give rise to the need for the resource instrumentation pattern. Typical goals of allocation include load balancing, schedulability analysis, system certification, and response time prediction; each of these requires knowledge of resource state.

B. A solution in terms of a collaboration with participating classes

This section provides a UML analysis model (see [4]) showing the realization of the use case in terms of classes and their collaborations.

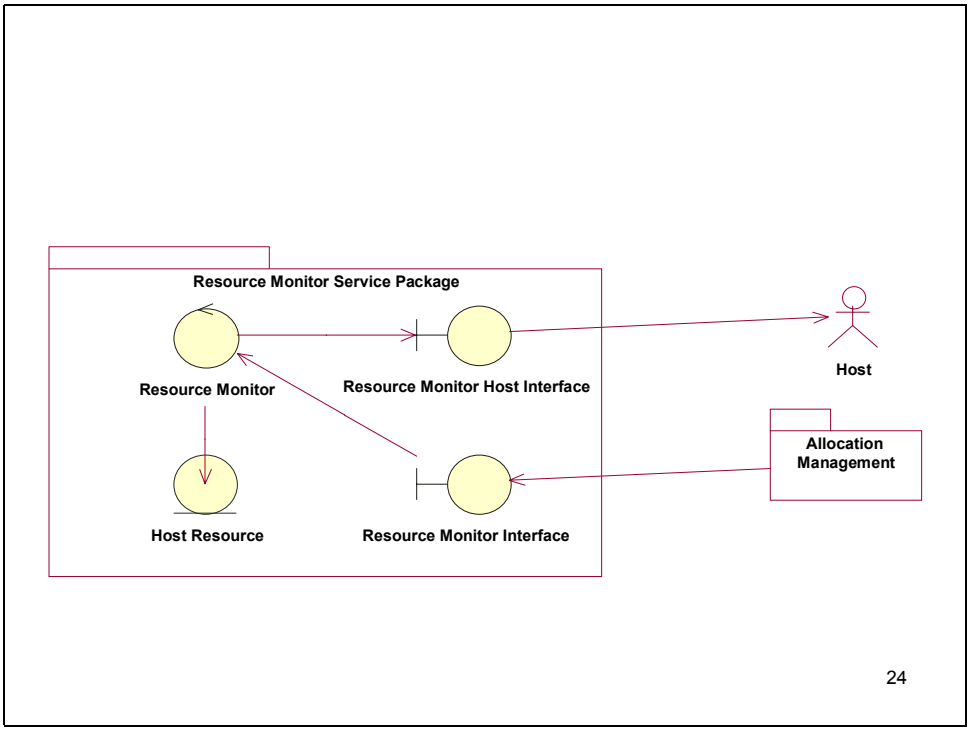

Figure 3. A solution for the *resource instrumentation* pattern.

As shown in Figure 3, the solution for the resource instrumentation pattern involves four classes. The boundary class *resource monitor host interface* gathers state information from a

resource (in this example a *host* resource is shown). The control class *resource monitor* stores the state information as well as any computed metrics in the entity class *host resource*. Queries for information are received and handled by the boundary class *resource monitor interface*.

C. Summary of the benefits and consequences of using this pattern

One benefit of using the resource instrumentation pattern is that it masks heterogeneity of resources. Device-specific and operating system-specific details are hidden in the implementation of the pattern. Another benefit is that many aggregate metrics (such as trend) can be implemented once and reused. Since the pattern is device-specific or OS-specific, implementations of this pattern are not platform-independent.

IV. Software performance monitoring

This section characterizes the *software performance monitoring* use case.

A. Summary of the problem and the forces that give rise to it

The *software performance monitoring* pattern addresses the problem of determining and assessing the performance of distributed real-time subsystems (called paths). One problem it solves is the aggregation of time-stamped events (from the application programs that constitute a path) into a single value that represents end-to-end path latency.

In addition to aggregation, the pattern solves the following problems:

- checking that end-to-end latencies meet performance requirements,
- forecasting future end-to-end latencies, and
- diagnosing causes of poor performance.

This pattern is motivated by the need to accommodate dynamic workload changes; this property makes it necessary to monitor the performance and resource needs of real-time applications. If the workloads of real-time systems were constant, then resources could be statically allocated in a way that permits all real-time constraints to be met.

B. A solution in terms of a collaboration with participating classes

The *real-time system management* subsystem (see Figure 4) provides a solution to the *software performance monitoring* pattern. A real-time software system (or its constituent application programs) provides time-stamped events to the *real-time system interface* boundary class. The *real-time system manager* control class stores the event information in a *real-time system event* class; additionally, it (1) performs checking, forecasting, and diagnosis and stores the result in the *real-time system state* class and (2) provides data to other subsystems.

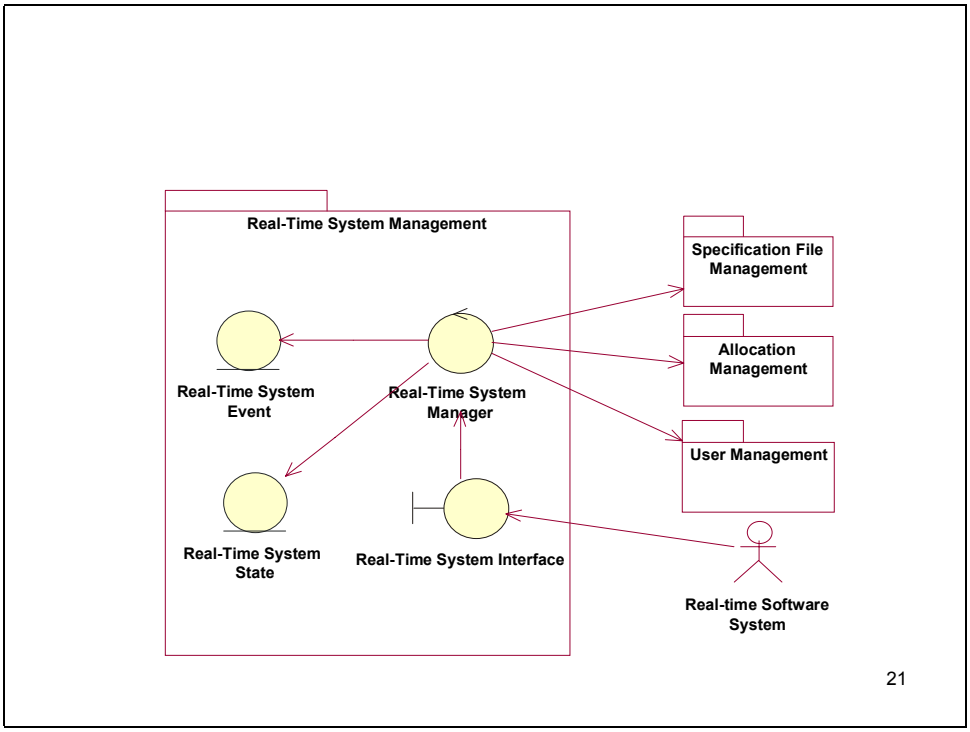

Figure 4. A solution for the *software performance monitoring* pattern.

C. Summary of the benefits and consequences of using this pattern

Use of the *software performance monitoring* pattern results in several benefits. The communication with application programs is masked; communication protocols can be customized for different applications and encapsulated within the *real-time system interface* class. Additionally, functions for checking, forecasting and diagnosis can be implemented once and reused. However, this pattern assumes that the real-time software system contains the ability to send time stamps to real-time system management.

V. Allocation planning and management

The *allocation planning and management* pattern is described in this section.

A. Summary of the problem and the forces that give rise to it

The *allocation planning and management* pattern solves the problem of dynamically mapping software (the demand space) to hardware resources (the supply space) in a way that satisfies all constraints (e.g., real-time requirements) and optimizes the utility that accrues from the system as a result. The software entities to be mapped include systems, subsystems/paths, application programs and connections between application programs. The hardware includes CPUs, memory, network components, bandwidth, devices, and power.

The need for the pattern stems from dynamic environments (e.g., war-fighting and space), dynamic workloads (e.g., unknown event arrival rates and unknown input sizes for data-driven processing), and hardware failures.

B. A solution in terms of a collaboration with participating classes

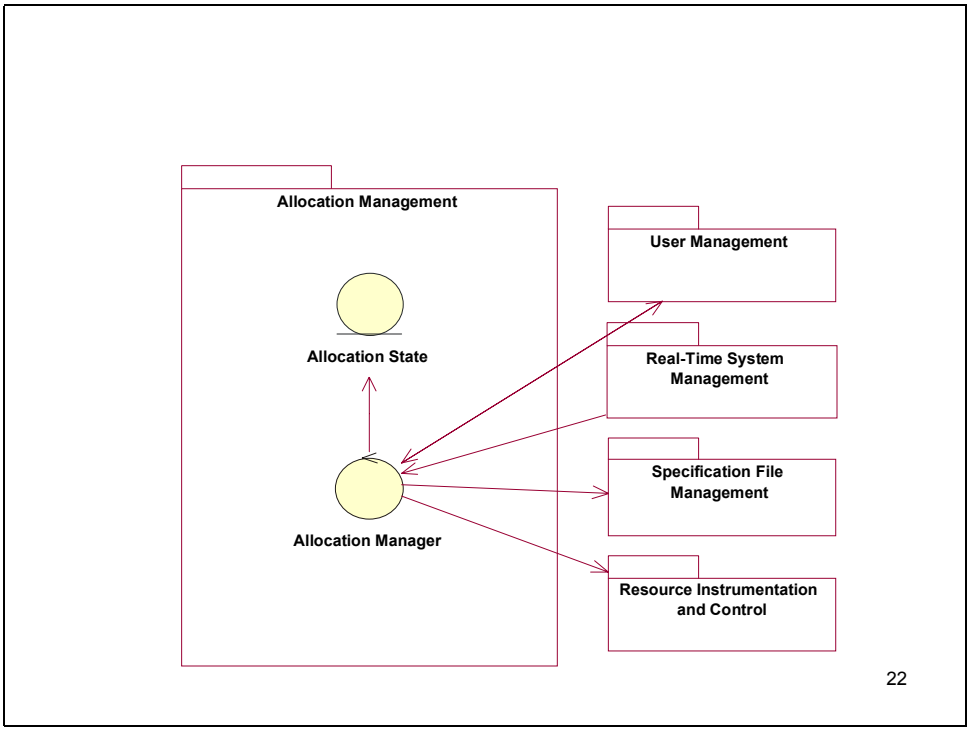

Figure 5. A solution for the *allocation planning and management* pattern.

The *allocation planning and management* pattern is realized by a control class (*allocation manager)* that utilizes information contained in the entity class *allocation state.* The allocation manager performs constraint checking and allocation optimization analysis in response to reallocation triggers. Information from *allocation state* is used to evaluate specific candidate allocations. Upon discovering a suitable new allocation, the *allocation manager* class employs the pattern to carry out its decision.

C. Summary of the benefits and consequences of using this pattern

Benefits of using the pattern include the following:

- flexibility (of allocation),
- adaptability,
- graceful degradation,
- survivability, and
- scalability.

A consequence of employing the pattern is that on-the-fly performance analysis must be performed.

VI. Resource control

The *resource control* pattern is described in this section.

A. Summary of the problem and the forces that give rise to it

The resource control pattern addresses the problem of modifying the way that software components are allocated to hardware resources. The pattern allows for components to be started, stopped and moved. Additionally, the pattern monitors the status of the components and provides a notification service that allows clients to be notified when a component unexpectedly terminates.

The force that gives rise to the pattern is the need to dynamically affect how resources are being used and the need to provide survivability to software components.

B. A solution in terms of a collaboration with participating classes

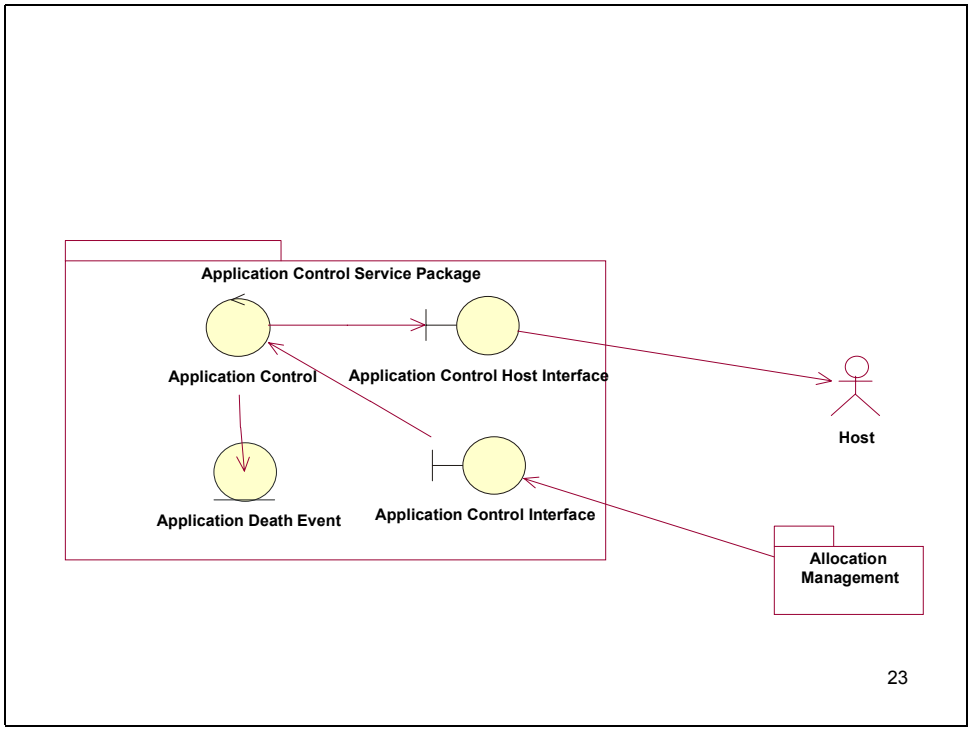

Figure 6. The *resource control* pattern.

Figure 6 shows a solution for the *resource control* pattern. The *application control interface* class accepts requests to control components and provides notifications of unexpected component terminations. The *application control* control class dispatches commands to hosts via the *application control host interface* class and records unexpected component terminations in the *application death event* class.

C. Summary of the benefits and consequences of using this pattern

A benefit of using this pattern is that control of heterogeneous hardware resources and software components can be handled via a uniform interface.

The use of this pattern implies interaction among some of the patterns presented here. For example, in the case that the software performance monitoring pattern forecasts or detects a real-time constraint violation, it must communicate with the allocation planning and management pattern. In turn, the allocation planning and management pattern may instruct the resource control pattern to reallocate resources if the violation may be resolved.

VII. Domain reference architecture

This paper deals with the domain of large, distributed real-time systems that have execution times and resource utilizations which cannot be characterized *a priori*. (The motivation for our work is provided in part by the characteristics of *combat systems*, as described in [7].) There are several implications of these characteristics: (1) demand space workload characterizations may need to be determined *a posteriori*, and (2) an adaptive approach to resource allocation may be necessary to accommodate dynamic workload changes. Thus, this paper considers approaches for dynamically managing distributed computing resources by continuously computing and assessing QoS and resource utilization metrics that are determined *a posteriori*. This section defines the domain by presenting an adaptive distributed system reference architecture [8] that is suitable for such an approach. This reference architecture provides the capabilities and infrastructure needed to construct multi-component, replicated, distributed real-time systems that negotiate for distributed computing resources to achieve desired QoS levels.

Figure 7 depicts the distributed system reference architecture. The diagram shows the functional architecture structure needed to support distributed application programs for a computer containing a client application. This structure is repeated throughout the computers of the distributed system; in particular, the computer hosting the server application contains a comparable structure. Also executing somewhere in the distributed system's collection of computers is a set of QoS management components that interact with the computers, network components and applications of the distributed system to provide QoS management.

Besides the applications themselves, the components of the reference architecture consist of four primary types: operating systems, network services, high-level communication and state management middleware services, and resource management services. The operating system services are those needed to support real-time applications. The network services provide low-level network communications and time management capabilities. The middleware components provide the ability to communicate in accord with three high level communication models: publish-subscribe; group programming, with associated ordered multicast and state data synchronization services; and distributed object programming. The resource management services consist of computing resource and application status monitoring and reporting services, QoS negotiation services, and program control services.

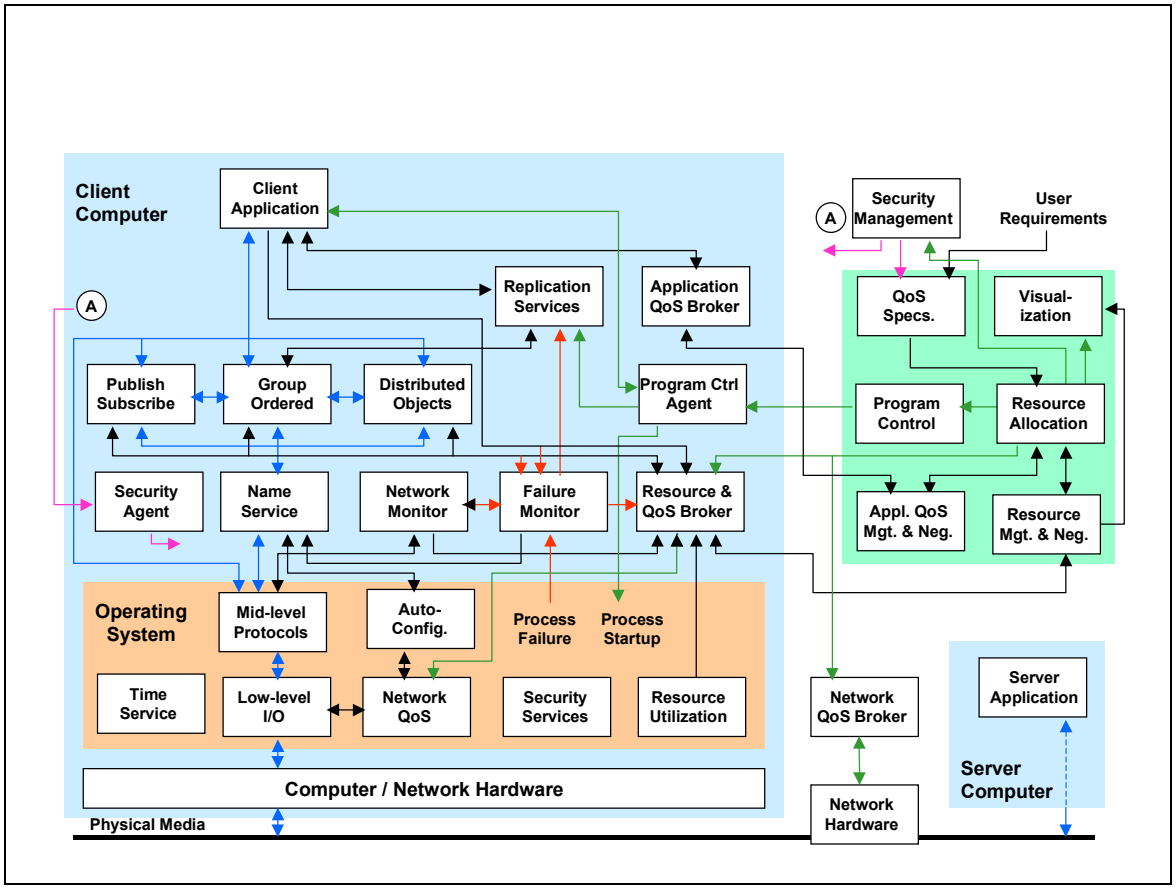

Figure 7. A distributed system reference architecture.

Taken together, these services allow distributed applications to perform allocated processing functions, to communicate with each other, to perform fault detection and recovery activities, to perform load sharing activities, and to negotiate resource utilization and QoS requirements with the underlying distributed system infrastructure. This architecture has been partially implemented and successfully employed for dynamic management of QoS and distributed computing resources within a Navy distributed computing testbed [2,9] and within a NASA satellite constellation testbed [10]. Features of the testbeds include real-time mission critical computing, fault tolerance and scalability (a description of the testbed is contained in [2]).

The reference architecture is currently serving as a roadmap for the construction of a distributed resource management system; a significant portion of the architecture has been realized [2]. In addition to providing guidance for implementation, the authors have found that the architecture is very useful for classifying emerging and existing technology components. Thus, it is used as the starting point for the pattern definitions contained in the remainder of the paper.

VIII. Summary

This paper characterizes important patterns within the domain of dynamic resource management. A reference architecture that characterizes the domain is provided. Additionally, an

architecture that focuses on the sub-domain of computing and network resource management is provided. The architecture is decomposed to provide solutions for the following patterns:

- resource instrumentation,
- software performance monitoring,
- allocation planning and management, and
- resource control.

For each pattern, the paper describes (1) the problem that it address and the forces that give rise to it, (2) solutions to the problem, and (3) the benefits and consequences of using it.

IX. Acknowledgements

This project was funded in part by the DARPA program entitled *Program Composition for Embedded Systems*.

X. References

[1] L. R. Welch, B. Ravindran, B. A. Shirazi and C.Bruggeman, "Specification and modeling of dynamic, distributed real-time systems," in *Proceedings of The 19th IEEE Real-Time Systems Symposium*, 72-81, IEEE Computer Society Press, 1998.

[2] Lonnie R. Welch, Paul V. Werme , Larry A. Fontenot, Michael W. Masters, Behrooz A. Shirazi, Binoy Ravindran and D. Wayne Mills, "Adaptive QoS and Resource Management Using A Posteriori Workload Characterizations," *The IEEE Real-Time Technology and Applications Symposium*, 266-275, June 1999.

[3] Binoy Ravindran, Lonnie R. Welch and Behrooz A. Shirazi, "Management Middleware for Dynamic, Dependable Real-Time Systems," in *The Journal of Real-Time Systems*, 20:183-196, Kluwer Academic Press, 2000.

[4] I. Jacobson, G. Booch and J. Rumbaugh, *The Unified Software Development Process*, 1998, Addison Wesley Longman, Inc.

[5] G. E. Prescott, S. A. Smith and K. Moe, "Real-Time Information System Technology Challenges for NASA's Earth Science Enterprise," in *Proceedings of the International Workshop on Real-Time Mission-Critical Systems*, Dec. 1999.

[6] Lonnie R. Welch, Binoy Ravindran, Robert D. Harrison, Leslie Madden, Michael W. Masters and D. Wayne Mills, "Challenges in Engineering Distributed Shipboard Control Systems," *Work-in-progress - The IEEE Real-Time Systems Symposium*, 19-22, December 1996.

[7] Robert D. Harrison Jr., "Combat system prerequisites on supercomputer performance analysis," in *Proceedings of the NATO Advanced Study Institute on Real Time Computing*, NATO ASI Series **F(127)**, 512-513, Springer-Verlag 1994.

[8] Lonnie R. Welch, Michael W. Masters, Leslie A. Madden, Paul V. Werme, Dave Marlow, Phil Irey and Behrooz A. Shirazi, "A Distributed System Reference Architecture for Adaptive QoS and Resource Management," *Lecture Notes in Computer Science*, **1586**:1316-1326, Jose Rolim et al. (eds.), ISBN 3-540-65831-9, Springer-Verlag, 1999.

[9] Lonnie R. Welch, Paul V. Werme, Behrooz A. Shirazi, Charles D. Cavanaugh, Larry Fontenot, Eui-Nam Huh and Michael W. Masters, “Load Balancing for Dynamic Real-Time Systems,” *Cluster Computing,* **3(2000)**:125-138, 2000.

[10] Toni Marinucci, Anbuselvan Neelamegam, Brett Tjaden, Lu Tong, Lonnie R. Welch, Brian Goldman, Greg Greer, Deepak Kaul, and Barbara B. Pfarr, “Sensor Web Adaptive Resource Manager,” *NASA Earth Science Technology Conference (ESTC-2001)*, August 2001.