Target Manager – A Resource Provisioning Service for Enterprise Distributed Real-time and Embedded Systems

Nilabja Roy, Nishanth Shankaran, and Douglas C. Schmidt

Vanderbilt University, Nashville TN, USA 615-343-8197

{nilabjar, nshankar, schmidt}@dre.vanderbilt.edu

Abstract. Middleware is increasingly used to develop and deploy components in enterprise distributed real-time and embedded (DRE) systems. A key challenge in these systems is devising resource management algorithms that deploy application components properly onto target nodes. To provide an accurate view of system resource utilization, these algorithms need runtime monitoring of resources. Runtime monitoring and allocation of resources is also needed to make redeployment or reconfiguration decisions triggered by various factors, such as failures, attacks, overloads, or changes in quality of service (QoS) requirements. DRE systems with a diverse range of applications can therefore benefit from a common resource provisioning service capable of monitoring resource data and enabling proper resource allocation in a timely manner.

This paper provides two contributions to the study of runtime resource provisioning for enterprise DRE systems. First, it describes the challenges in developing *Bulls-Eye*, which is an open implementation of the OMG standard Target Manager specification that provides a reusable service for provisioning distributed resources in enterprise DRE systems. Second, it presents the results of experiments that applied Bulls-Eye to the multi-layer resource management subsystem of a shipboard computing environment. Our results show that provisioning resources at runtime in a DRE system via Bulls-Eye simplifies resource management and helps automate adaptations in the face of dynamic changes in operating conditions.

Keywords: Resource Provisioning, Component Technology, Dynamic Resource Management, CORBA Component Model.

1 Introduction

Resource Provisioning challenges of component-based enterprise DRE systems. Applications in the domain of enterprise distributed and real-time embedded (DRE) systems, such as shipboard computing environments, satellite constellations, and surveillance and reconnaissance systems, are characterized by stringent quality of service (QoS) requirements and operate in dynamic and resource-constrained environments. The operating modes of these systems may dynamically vary in response to changes in policies or input loads, and they often execute across heterogeneous platforms. Certain enterprise DRE system characteristics, such as their longevity and complexity, motivate the use of component-based development. In this context, components are units of implementation and composition that have well-defined QoS requirements and resource consumption profiles. In enterprise DRE systems, applications consist of groups of domain-related tasks that can be implemented by parameterized and executable software components using QoS-enabled component middle-

ware platforms, such as OMG Lightweight CORBA Component Model (CCM) [3] and PRiSM [14].

Although component technologies can help enhance software reuse, maintenance, and extensibility [19], they also introduce new *deployment and configuration* challenges [15] stemming from the need to shield applications and users from the complexities of heterogeneous and dynamically changing hardware/software environments. The process of deploying enterprise DRE systems involves creating a *deployment plan* that allocates available computing and communication resources (e.g., memory, CPU, and network bandwidth) to the components and establishes connections between them. To prepare an effective deployment plan, the DRE system needs to know the resources available in the target domain so that resource consumption profiles of the components can be mapped properly to the available computing nodes and communication links. It is also important to track resource usage at runtime so that components can be redeployed and/or reconfigured to adapt to changes in application operating conditions caused by policy changes or failures, which must be detected quickly so the system can adapt with minimum disruption.

One way to address these challenges is to create *a resource provisioning service* that (1) monitors the resources available in the target domain, (2) supplies this information to human and/or automated planners who prepare a deployment plan using the current resource profile, (3) dynamically allocates resources to deployed components and releases resources when the components are terminated, and (4) facilitates component redeployment and reconfiguration based on resource availability and constraints. Developing such a resource provisioning service for enterprise DRE systems is hard due to the need to handle platform heterogeneity, ensure responsiveness and scalability, and enable dynamic updates within time constraints.

This paper describes the design and application of *Bulls-Eye*, which is an implementation of the Lightweight CCM Target Manager specification [6] that is tailored to the needs of enterprise DRE systems. In particular, we designed Bulls-Eye to optimize its CPU and I/O usage to provide fast/predictable access to resource information and enable its use to provision enterprise DRE systems with a range of QoS requirements. The resulting object-oriented framework has been integrated with the *Component-Integrated ACE ORB (CIAO)* (www.dre.vanderbilt.edu/CIAO), which is an open-source implementation of Lightweight CCM that has been applied to several enterprise DRE systems, including a shipboard computing system and a prototype of a NASA science mission.

The remainder of this paper is organized as follows: Section 2 describes a case study that motivates the need for a resource provisioning framework in shipboard computing systems; Section 3 discusses the structure and functionality of the Bulls-Eye Target Manager; Section 4 explains the design challenges that we overcame while developing Bulls-Eye and applying it to the shipboard computing domain; Section 5 summarizes the results of experiments that measures the overhead of Bulls-Eye and demonstrates its utility in the context of a prototype shipboard computing system; Section 6 compares our work on Bulls-Eye with related research; and Section 7 presents concluding remarks and outlines our lessons learned during this project.

2 Case Study: An Enterprise DRE System for Shipboard Computing

This section describes the structure and functionality of the *Multi-Layer Resource Management* (MLRM) subsystem for shipboard computing that we use as our running case study in the paper to motivate our work on Bulls-Eye. A shipboard computing environment is a metropolitan area network of computational resources and sensors that provides on-demand situational awareness and actuation capabilities for human operators, and responds flexibly to unanticipated runtime conditions. To meet such demands in a robust and timely manner, the shipboard computing environment uses services in the MLRM subsystem to (1) bridge the gap between shipboard applications and the underlying operating systems and middleware infrastructure and (2) support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization. To support the accelerated operational tempo in modern shipboard computing systems, the MLRM software must adapt in response to dynamic conditions for the purpose of utilizing the available computer and communication resources to the highest degree possible to meet changing mission needs.

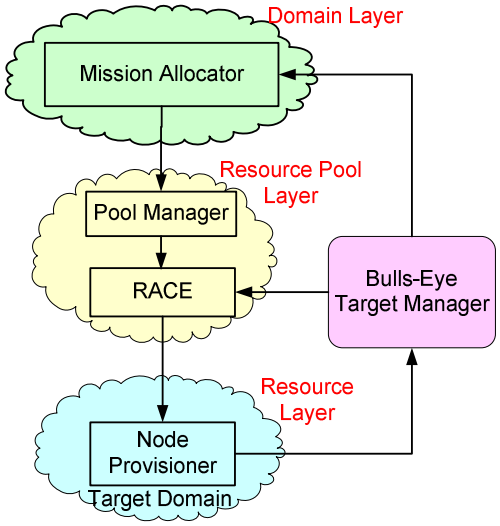

Figure 1. Layered Architecture of the MLRM Subsystem

The MLRM subsystem described in this paper consists of the three layers shown in Figure 1. The command and policy inputs flow in a top-down manner and correspondingly the resource status information moves in a bottom-up fashion. At the top is the *Domain Layer*, which consists of the *Mission Allocator*. This allocator collects command and policy inputs and passes them onto the *Resource Pool* Layer, which represents a set of computing resources managed by a *Pool Manager*. The Pool Manager in turn interacts with the *Resource Allocation and Control Engine (RACE)* [13], which a reusable framework that separates resource allocation and control algorithms from the underlying middleware deployment, configuration, and control mechanisms

so that different algorithms can reuse common middleware mechanisms to (re)deploy components onto nodes and manage the node's resources among competing applications. The bottom layer is the *Resource Layer*, which contains the entire set of hardware elements in the shipboard computing environment, known as the *Target Domain*. Each node in turn contains a *Node Provisioner* that receives commands from RACE to create and destroy applications on the node.

The MLRM subsystem is built using the *Component-Integrated ACE ORB (CIAO)*. CIAO combines *Lightweight CCM* [5] mechanisms (such as standards for specifying, implementing, packaging, assembling, and deploying components) and *Real-time CORBA* [7] mechanisms (such as thread pools and priority preservation policies). The MLRM subsystem has scores of different types and instances of CCM components written in ~500,000 lines of C++ code and residing in ~1,000 files developed by five teams at different locations (dtsn.darpa.mil/ixodarpatech/ixo_FeatureDetail.asp?id=6).

The scale, complexity, longevity and multiple QoS requirements of a shipboard computing environment necessitates that its components be deployed and allocated using effective resource management techniques [5]. This requirement, in turn, motivates the need for accurate information on resource availability in the domain. The Bulls-Eye Target Manager shown in Figure 1 serves this functionality for the MLRM subsystem by providing runtime information on resource usage that helps RACE optimize component allocation and meet end-to-end QoS requirements.

Bulls-Eye is used during initial system deployment when RACE runs algorithms to allocate components to the appropriate nodes in a resource pool. These algorithms interact with Bulls-Eye to obtain information regarding resource utilization in the target domain. This data is used to produce a deployment plan needed to deploy the system via DAnCE, which is CIAO's implementation of the OMG Deployment and Configuration (D&C) specification [1]. The D&C specification standardizes many aspects of deployment and configuration for component-based distributed systems, including component configuration, component assembly, component packaging, package configuration/deployment and repository management of component implementations. Bulls-Eye is also used at runtime to extract dynamic resource availability data and update component implementations dynamically, e.g., in response to damage or to handle changing workload levels.

A particularly important function of the resource allocation and control algorithms in the MLRM subsystem is the (re)deployment and (re)configuration of components based on their operational context. For example, a shipboard computing environment may need to switch rapidly from crew entertainment mode to ship defense mode, which necessitates updating and/or migrating many computing services. Bulls-Eye therefore provides mechanisms to retrieve the resource availability data across the entire target domain by monitoring and dynamically updating component resource usage. RACE uses data provided by Bulls-Eye and the requirements of each component to generate an optimized deployment plan and to ensure that the components allocated conform to the characteristics of each node's hardware, OS, middleware, and programming language(s), which can be highly diverse.

3 The Design of the Bulls-Eye Target Manager

Bulls-Eye is a resource provisioning service designed to enable software developers and applications in enterprise DRE systems to (1) retrieve a list of the initial available resources in a target domain, thereby enabling the preparation of a deployment plan fulfilling the allocation and connection requirements of each component, (2) allocate resources for a particular deployment plan and release resources when the components or the entire deployment is removed, (3) obtain runtime resource available in the system, and (4) dynamically update the resource consumption data. This section describes how the structure and functionality of Bulls-Eye supports these capabilities in the context of the Lightweight CCM Target Manager specification.

3.1 Structure of Bulls-Eye

Figure 2 shows the architecture of Bulls-Eye, which consists of a CORBA interface specified in the Target Manager specification. Bulls-Eye is comprised of two parts: (1) a centralized[1] service, known as the Target Manager core (*TM-core*) used by applications and system services allocate and release resources as needed and (2) multiple monitors (*TM-Monitor*) distributed across the domain that perform resource monitoring and update the TM-core's model of the amount of resources available at any point in time.

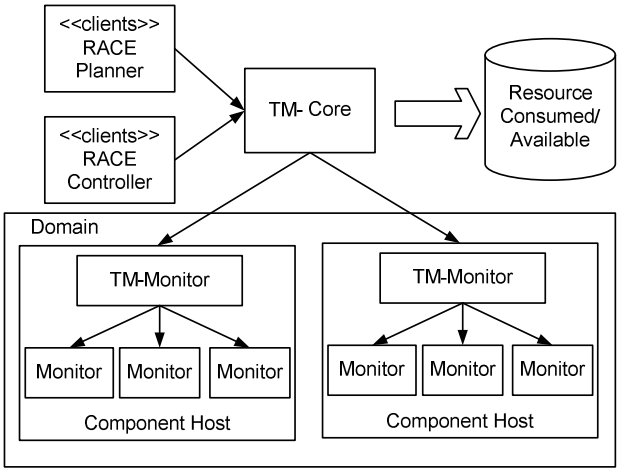

Figure 2. The Bulls-Eye Target Manager Architecture

The Domain comprises of all the elements of a target environment comprising of nodes, interconnects between them, bridges connecting between interconnects and the set of resources belonging to them. A *Domain* is a logical concept wherein a single resource or node element can be part of more than one target domain. *Domains* are therefore structured hierarchically, and a top-level domain may contain other do-

[1] There is only one logical instance of the TM-core in the domain, though it can be replicated to enhance availability and prevent a single point of failure.

mains. Each Domain will have a TM-Core accumulating the resource information for the associated target domain.

The TM-core provides a standard set of operations that applications and system services can use to provision available resources *statically* (i.e., prior to system launch) as well as *dynamically* (i.e., during system runtime) in the form of a generic structure known as the *DomainStruct* [4]. This structure describes the contents of the entire target environment by composing data related to available nodes in the network, the connections between nodes, connection between networks, the shared resources among them, and the resources for each element.

A TM-Monitor is placed on each logical node in the target domain and monitors the resource usage in that node. The TM-Monitor periodically update to the TM-core, with the current resource utilization/availability on that node. Upon receiving th updated, the TM-Core aggregates the data received with previous data and updates its content.

Bulls-Eye maintains a top-level *Domain* element that contains all the elements of a target domain and is uniquely identified by a universally unique identifier (UUID). This *Domain* element is designed so that all possible domain elements can be incorporated, which alleviates the need to create separate structures for different types of resource, such as processor, memory, storage, and/or network bandwidth. This design also makes client code flexible by alleviating the need for any specific type of resource in the domain since it can handle all the varieties of resource elements present in the domain.

The TM-Monitors collect data pertaining to their sub-domain and updates the TM-Cores with fresh data. Clients are interested in data across different sub-domains, so the data from different TM-Monitors need to be aggregated and presented uniformly. In order to avoid latency issues, the distributed monitors push in only the data that changed from the previous update. This data is aggregated with the remaining domain data which is already present.

3.2 Functionality of Bulls-Eye

Bulls-Eye provides the following standard Target Manager operations that can be invoked by clients to provision system resources:

- **Querying static resources.** Developers or planner applications can use `getAllResources()` to obtain the initial static resources in the target domain. This operation returns the *Domain* structure that contains the entire domain resource in a hierarchical fashion.
- **Querying dynamic resources.** Dynamic time resource availability can be returned by `getAvailableResource ()`. This operation returns the same *Domain* structure as above, except thta the resources reflect their remaining capacity.
- **Committing resources.** A planning application can call to `createResourceCommitment()` to commit (i.e., allocate) resources for a particular deployment plan. This operation creates a `ResourceCommitmentManager` that can be used to commit and release resources for a specific plan. A pool of resources can

be specified when a call to `createResourceCommitment()` is made or can be allocated after it is created. An exception is raised if a requested resource cannot be committed.

- **Releasing resources.** When an application or a component in an application is deleted all the resources allocated to it must be released so they can be reallocated when new applications are deployed. Resources can be released by an application by calling `releaseResources()` on the associated `ResourceCommitmentManager`. When a `ResourceCommitmentManager` is itself deleted via `destroyResourceCommitment()`, all remaining committed resources are released automatically.

- **Updating dynamic resource data.** The domain data in the TM-core can be updated via `updateDomain()`. The updated information is passed in the form of *Domain* structure, which is a subset of the higher level domain structure. An enumeration called *DomainUpdateKind* can be passed to tell Bulls-Eye whether the subset should be added, deleted, or updated.

The Bulls-Eye Target Manager functionality plays a key role in the deployment and configuration of enterprise DRE systems. On startup, it reads a configuration script containing the resources present in the domain. The script is prepared by a human or automated domain administrator who understands the initial domain contents, such as nodes, the interconnects linking them, and the resources contained in them and available for application usage, such as processor capacity, memory capacity, and disk capacity. The TM-Monitor is used to monitor components on a host is collocated and started together with its associated NodeManager, which is an entity defined by the OMG D&C specification and implemented by CIAO as a daemon process running on each host.

At startup, the TM-core is passed the subset of the *Domain* tells the TM-Monitor which resources to monitor on the host. The TM-Monitor then checks the *Domain* information and reports any discrepancies (such as the hard disk capacity being smaller than the initial domain description or the node is single processor node instead of a dual processor) to the TM-Core. Once Bulls-Eye is up and running, it can be used by the clients to make the above queries about the domain, e.g., RACE components can extract domain related information for preparing a deployment plan. Any entity willing to deploy plans in the domain will need to commit resources through Bull-s Eye to successfully deploy and ultimately run applications.

4 Resolving Bulls-Eye Design Challenges

Although the CCM specification defines the interface and the functionality of the Target Manager service it does not prescribe any design details. We were therefore faced with a number of design challenges when implementing Bulls-Eye. This section describes the key design challenges we encountered, presents our implementation solutions, and outlines how we applied these solutions to the shipboard computing applications supported by the MLRM subsystem described in Section 2.

Challenge 1: Integrating the Heterogeneous API of Multiple Platforms

Context. The domain of DRE systems typically consists of multiple platforms across the target environment. Each platform has it's own platform-specific application programming interfaces (APIs) that provide current resource data. For example, in Unix/Linux we can get the resources used up by each process such as processor, memory, byes sent/received information from the /proc system file-structure, but for Windows a DLL needs to be loaded that provides an API for querying resource consumption data.

Problem → Integrating the heterogeneous API of multiple platforms. The data returned by the platform-specific API have their own structures, units and, semantics. There must be some type of conversion algorithm that interpret this data in a common way so that the proper resource management decisions can be made. Moreover, the resource utilization information provided to clients of Bulls-Eye should be consistent, i.e., use similar units/structures. Otherwise, the users of the data will need to convert them manually, which is tedious, error-prone, and can yield redundancy in conversion logic. Ideally, it should be the responsibility of the middleware to convert disparate data into a uniform consistent form that can be readily used by clients.

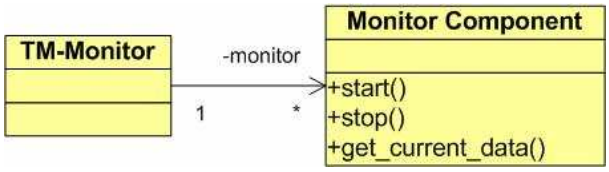

Fig. 3. Using the Adapter Pattern in Bulls-Eye

Solution → Use the Adapter pattern to adapt diverse API. To mitigate the problem of diverse resource management APIs in Bulls-Eye, we used the Adapter pattern [2], which converts non-standard APIs that extract resource data into the standard interface defined by the Target Manager specification. The implementation of this interface converts the platform-specific data into a uniform type for storage and distribution to clients of Bulls-Eye.

The extraction of resource consumption data is tricky and the accurate value depends upon the usage of a number of optimizations. We used some of the points mentioned in [20] appropriate to our solution such as "keeping /proc open between reads" and "reading data in a block rather than individual characters." The data also depends upon the processor architecture as also hardware configuration, for example in a single processor Linux machine it is easy to collect the CPU consumption data from /proc file system, but if there are multiple processors or new technology (such as hyper-threading) used the extraction of the same data becomes complex.

Applying the solution to the MLRM case study. The MLRM Node Provisioner spawns applications on each host. It uses CIAO's Node Manager to start up the components that make up a particular application. These Node Managers contain the instances of TM-Monitor for the designated host. During system startup, the TM-Monitor loads the component suitable for the corresponding platform using CIAO's *Repo-*

Man [16] implementation of the Lightweight CCM Repository Manager specification. These components collect low-level data, convert it to the standard structure Bulls-Eye expects, and the TM-Monitor then corresponds with the component using the standard interface and collects the required data.

Challenge 2: Providing a common access point to provision resources in a domain

Context. Enterprise DRE systems are often distributed across dozens or hundreds of entities. The entire application environment is hierarchically arranged with a top level domain containing sub-domains in it which in turn contains computing nodes connected via many routers and interconnects along with their resources. Any planner specific to a domain will require information of resources contained in the entire domain.

Problem → Accessing data through a common access point. The resource utilization/availability data of all such different entities need to be provided through a single point common location for the users to make use of the data. Otherwise, the client has to parse through hierarchically arranged domains and collect data, and merge it in the right way to make use of it. This is error-prone and is time-consuming, which also means that the client may be working with stale data while the resource condition may have changed.

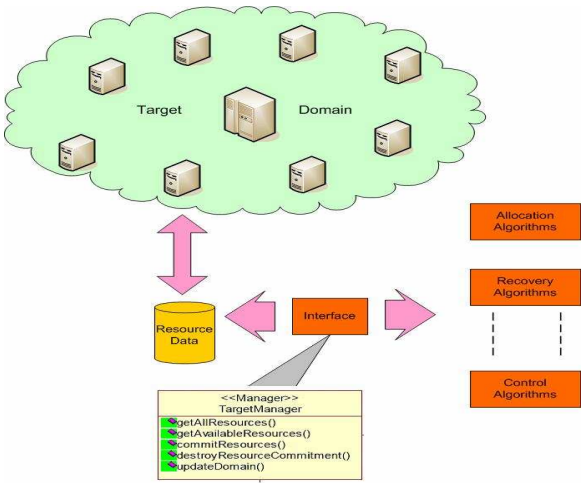

Figure 4. Providing a Common Access Point to Domain Resource Data

Solution → Use distributed monitors to collect data across the domain. To solve the above problem of providing a common access point for data in the domain, we use TM-Monitors across the domain located in nodes in the target domain and a single instance of the server (TM-Core), which is a service in one node in the current domain. There can be multiple instances of this setup as replica to increase reliability. These monitors communicate with the central server and send data updates at a periodic interval (configured externally). The monitors in turn make use of the platform-

independent adapters described above to extract the resource information. Thus the solution above ensures that the monitoring process is carried out in a scalable way in a hierarchical fashion while, at the same time providing a common access point for all resource data in the domain.

Applying the solution to the MLRM case study. As mentioned before the Node Provisioners take the help of Node Managers in order to start-up the applications in each host. The TM-Monitors also start up along with the Node Managers and receive a *Domain* structure specifying which resources it needs to monitor. Using this configuration it starts monitoring the resources and sends back data to the centralized TM-Core.

Challenge 3: Presenting data to clients with fast response time in uniform structure

Context. An enterprise DRE system can have many resources that are present in various forms of composition. For example, a target domain may consist of X hosts, each host can consist of Y elements, e.g., one host can have a sound card connected to it while another may have a video card. The data from different sections of the domain need to be presented in a uniform and aggregated form for the clients to use the data for effective resource management. The data also needs to be relevant, the changes in the domain need to captured with very low response time so that it is useful for clients to make use of the data meaningfully.

Problem → Providing aggregated data of entire domain with fast response time. In a typical application scenario there can be numerous domain elements, data related to all these elements can be huge and there can be significant latency in transfer of such information. Data updates from distributed monitors will reach the central server separately and will pertain to only a specific section of the entire domain. These updates need to be merged along with multiple such updates to the main data store, parsing and trying to find the corresponding data store for a particular resource can be costly and can significantly slow down the response time. Thus, there is a need for an efficient and scalable algorithm to handle the data merge.

Solution → Combination of heap-sort and timer based aggregation algorithm. To solve the above problem, Bulls-Eye uses a combination of two approaches, I) it optimizes the data uploaded to the TM-Core to minimize unnecessary CPU and network processing by maintaining a cache of the last update sent to the TM-Core. Whenever it gets fresh data from the underlying component it compares the data received with the cached data. It only sends a data update if there is any difference. For example, when memory resource is monitored, if a particular reading informs the TM-Monitor that the memory usage has not changed from the last update to the TM-Core then there is no update sent to the TM-Core.

II) It uses a combination of a heap-sort algorithm and a timer based aggregation mechanism is employed. Heap sorting gives an O (log n) time complexity in the worst case. We label each resource entity with a unique identity and place the identity along with the pointers to the actual data structure in a heap. There is also a timer which fires at regular intervals (configured externally).Once updated data from the monitors is received, it is stored in a cache. On the firing of the timer, the cache is examined for

any outstanding data update. The resource entity id of the update is searched in the heap and its corresponding data structure is updated in constant time. This gives us an O (log n) time complexity in the worst case. The TM-Monitor optimizes the data uploaded to the TM-Core to minimize unnecessary CPU and network processing by maintaining a cache of the last update sent to the TM-Core. Whenever it gets fresh data from the underlying component it compares the data received with the cached data. It only sends a data update if there is any difference. For example, when memory resource is monitored, if a particular reading informs the TM-Monitor that the memory usage has not changed from the last update to the TM-Core then there is no update sent to the TM-Core.

Applying the solution to the MLRM case study. The operational context of a shipboard computing environment evolves continuously, e.g., it needs to satisfy changing mission requirements and adapt to transient overload and failure in the nodes. Such changes provoke a reaction in the control algorithms that drive the dynamic update or the partial or complete redeployment of the system. In order to achieve this, current domain resource data should be available. The specialized aggregation algorithm used by Target Manager (1) improves the responsiveness of the Target Manager and allows it to collaborate faster with clients (such as the MLRM subsystem and its applications) and (2) helps reduce the costs associated with redeploying and updating the system, thereby enabling more CPU and I/O processing to be spent performing mission tasks and meeting system deadlines.

Challenge 4: Using Multiple Configurable Monitor Components to Extract Variety of Data

Context. There are many types of elements in a typical target domain for enterprise DRE systems. Each element can have its own monitor component supplying its resource usage. These separate monitors could also be developed by multiple vendors, e.g., in some platforms there can be vendor supplied software component providing the processor consumption data (Windows) while in others a developer may need to write code to access the data (Linux/Unix), and yet other applications may want to use specialized third-party hardware monitoring utilities.

Problem → Using multiple configurable monitor components to extract variety of data. Bulls-Eye's TM-Monitor communicates with the underlying components to extract data. Since there can be different elements attached with a host, it must keep track of each element to be monitored along with its component. For example there can be a component monitoring CPU and memory usage, while another can monitor the usage of a I/O and disk space. There is also the need to swap displays for a particular type of resource, e.g., when there is an upgrade of a display with its latest version.

Solution → Initial Domain data configured with resource element and component name. The initial Domain data sent to TM-Monitor is configured with the name of the element to be monitored along with its' component name, which is done by the Domain Administrator before the startup of Bulls-Eye. The resources that need to be tracked are initially configured. The Strategy pattern is used here to load and unload multiple components for the same resource elements. All the components confirm to a

particular interface, containing life-cycle activities and data supplying operations. The component is loaded by the TM-Monitor and it maintains a map of resource element to component name. The TM-Monitor makes a call to each component to start and stop the component and also periodically and gets their current data. It also combines the data from each component into one single *Domain* structure before uploading it to the TM-Core, which makes it easy to extend Bull-Eye's monitoring capabilities.

Applying the solution to the MLRM case study. The Domain Administrator creates the configuration initially. The elements monitored in each host are included along with their components. During startup the *Domain* data reaches each TM-Monitor in each host. TM-Monitor then loads the component and starts monitoring. In case it fails to find the component, TM-Monitor throws an exception.

5 Experimental Evaluation of Bulls-Eye

This section outlines the testbed that provides the infrastructure for a representative enterprise DRE system from the domain of shipboard computing used to evaluate the performance of Bulls-Eye, describes our experiments, and analyzes the results obtained to evaluate the performance of Bulls-Eye.

5.1 Hardware/Software Testbed

Our experiments were performed on the ISISLab testbed at Vanderbilt University (www.dre.vanderbilt.edu/ISISlab). The hardware configuration consists of three nodes acting as the system domain. The hardware configuration of all the nodes was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1Ghz Ethernet network interface, and 40 GB hard drive. Redhat Fedora Core release 4 operating system running in real-time scheduling mode was used for all the nodes.

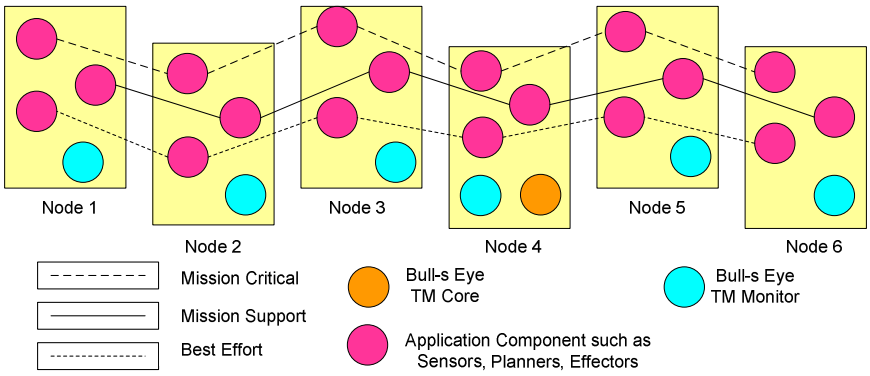

Figure 5. Operational Strings in the Testbed

Figure 5 shows our representative enterprise DRE system test configuration, which was composed of three operational strings [13], each containing six application components. The application components were implemented using work load generators [18]. Real-time QoS properties and requirements of these operational strings are specified by their relative priority and end-to-end deadline, respectively. The three

operational strings were composed of one mission-critical, one mission-support, and one best-effort operational string. The mission-critical operational string was configured with the highest priority, followed by the mission-support and best-effort operational strings. An end-to-end deadline of 500 ms was specified for the mission-critical operational string.

To evaluate the utility of Bulls-Eye, we deployed the mission-critical operational string followed by the best-effort operational string, which was then followed by the mission-support operational string. At each node within the domain, Bulls-Eye monitored the net processor utilization, as well as processor utilization per each component. We also monitored the end-to-end execution time of the mission-critical operational string. Since Bulls-Eye is implemented as a component, we also monitored the resource utilization of Bulls-Eye to determine the overhead of Bulls-Eye itself.

In conjunction with Bulls-Eye, the *Resource Allocation and Control Engine (RACE)* [13] was used in our experiments to ensure end-to-end execution time of the mission-critical operational string was below its end-to-end deadline. RACE enables DRE system developers to configure allocation and control algorithms depending on the characteristics of applications being deployed and enables the use of multiple algorithms without needing to handcraft the mechanisms used to configure the algorithms. It also deploys the application components to various nodes within a resource pool using specialized allocation algorithms. Inputs to RACE include (1) end-to-end deadline of mission-critical operational string and (2) runtime resource utilization information, which was provided by Bulls-Eye.

5.2 Analysis of Results

This section presents results from running the experiment described above on our ISISlab testbed. We used end-to-end execution time of the mission-critical operational string as a metric to evaluate the utility of Bulls-Eye and the resource utilization by Bulls-Eye as a measure of the infrastructure overhead. Resource utilization information collected by Bulls-Eye for the six nodes in the domain is shown in Figures 6-A, 6-B, 6-C, 6-D, 6-E, and 6-F as a function of time. End-to-end execution time of the mission-critical operational string is shown in Figure 7 as a function of time. Since Bulls-Eye was deployed on node 6, Figure 6-F also captures the overhead of the Bulls-Eye infrastructure.

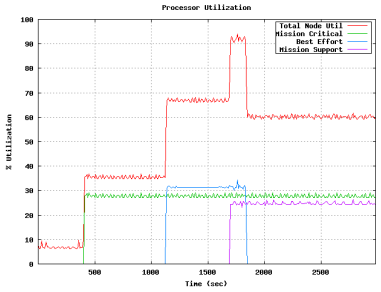

Figure 6-A. Node 1

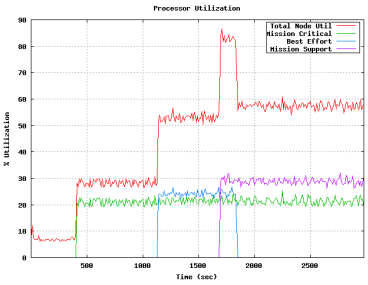

Figure 6-B. Node 2

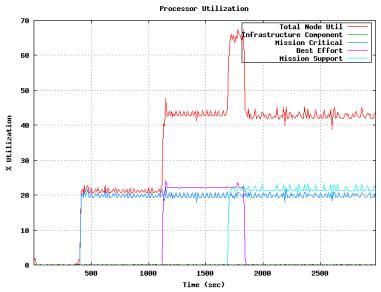

Figure 6-C. Node 3

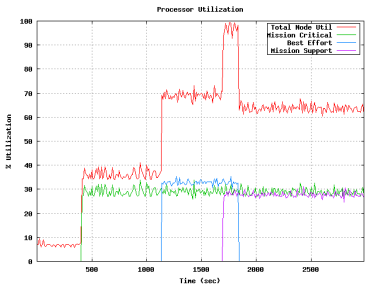

Figure 6-D. Node 4

As shown in Figure 6-A, 6-B, 6-C, 6-D, 6-E, and 6-F, when the mission-support operational string is deployed at the 1,800th second, the net processor utilization of the nodes increased above the RMS recommended utilization setpoint of 0.7 [17]. At the same time, as shown in Figure 7, the end-to-end execution time of mission-critical operational string increased above it deadline of 500 ms. This result indicates that the increase in execution time of the mission-critical operational string results from over-utilization of system resources (CPU).

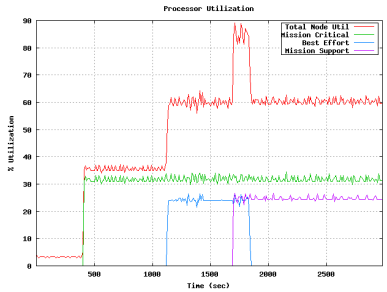

Figure 6-E. Node 5

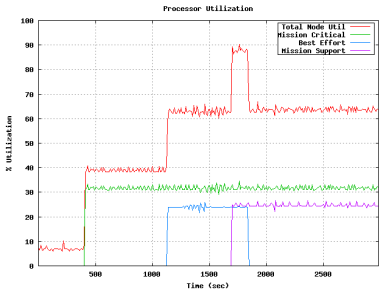

Figure 6-F. Node 6

The resource utilization information collected by Bulls-Eye serves as the input to RACE and triggers RACE to perform adaptive system control modifications, such as modifying operating system priority, scheduler class, and/or tearing down lower priority operational strings.

In our experiment, RACE tears down the best-effort operational string to meet the QoS requirements of higher priority mission-critical operational strings. As a result of these adaptive control actions, the end-to-end execution time of the mission-critical operational string is once again below its deadline, as shown in Figure 7. Figure 6-F shows that the infrastructure overhead due to Bulls-Eye itself is insignificant compare to the network resource utilization.

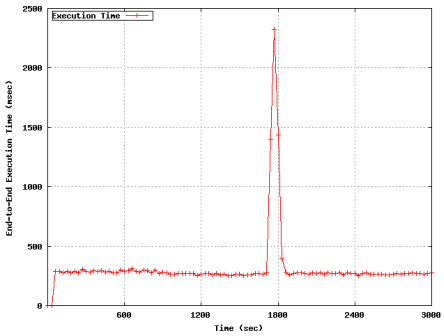

Figure 7. End-to-End Execution Time of the Mission-Critical Operational String

Without a resource provisioning service like Bulls-Eye, over-utilization of system resources could go unnoticed. Resource utilization information is a key input to any control framework. A control framework for enterprise DRE system, such as RACE, requires resource utilization information regarding multiple types and instances of resources from the domain. A resource provisioning framework such as Bulls-Eye is therefore essential to effective adaptive resource management for enterprise DRE systems.

6 Related Work

This section compares our work on Bulls-Eye with related work in the domain of resource provisioning.

The CMU Resource Monitoring System (ReMoS) [8] is service that allows network-aware applications to obtain relevant information about the bandwidth and latency of a specific flow, where flow is an application level connection between a pair of computation nodes. It also answers queries about the network topology. ReMoS uses two abstraction levels: explicit management of resource sharing and statistical measurement. Its flows abstraction captures the communication between nodes and its topologies abstraction provides a logical view of network connectivity. ReMoS measurements are made at the network level, so it provides information for use in sharing of resources. Bulls-Eye, in contrast, focuses on the resource availability for component assemblies, rather than the network level. This focus requires the aggregation of data into a single unit so that decisions regarding whole units/assemblies can be taken. Resource provisioning, synchronizing multiple planners, and matching of component requirements to target domain availability are other key concerns for Bulls-Eye, which acts as a common service for resource provisioning at multiple layers.

The BBN Resource Status Service (RSS) [9] is another multi-layer resource monitoring service. RSS consists of monitors (known as “condition objects”) that are distributed to hosts in a network and which communicate with each other to acquire the required data. In addition, RSS aggregates data of various resources, such as processor load average and memory consumption. Whereas the RSS is based on a non-standard interface, Bulls-Eye supports the OMG Lightweight CCM Target Manager specification, which defines standard interfaces that third-party providers can use to

integrate their monitoring mechanisms. The Lightweight CCM Target Manager specification (and thus Bulls-Eye) also supports resource provisioning by providing a common point to commit and release resources for different plans deployed, which is not supported by RSS.

The Globus Toolkit [10] provides a number of resource provisioning services that focus on monitoring, management, scheduling, and coordination of different computations in a computing grid. It also has tools for transmitting and managing large amounts of data useful to grid-based applications. Bulls-Eye does itself does not manage applications of a distributed environment (relying on other services in CIAO, such as DAnCE and RACE for these capabilities), but instead focuses on the collection, aggregation, and presentation of resource information in a timely manner. Bulls-Eye focuses on deployment and configuration of component-based applications and has features to support real-time QoS policies for mission-critical DRE systems.

[11] proposes an integrated architecture for managing dependencies uniformly in distributed component-based systems. It allows developers to present dependencies between components; instantiates component based applications and manages hardware resources in the distributed system. For this purpose, it has a *resource management service* which is similar to Bulls-Eye in that it uses distributed monitors to acquire local status information and aggregates the information on a central server. [11] focuses largely on the allocation and running of the applications, however, whereas Bulls-Eye is built more generically and supports standard interfaces for plugging in multiple types of resource monitors. [11] also does not deal with the resource provisioning aspects supported by Bulls-Eye.

[12] implements a resource monitoring service similar to Bulls-Eye, but with a focus on collecting resource data to create a forecasting model that provides process schedulers with resource trends so that they can schedule more efficiently. Our Bulls-Eye approach is different in that it collects resource information at runtime at a finer-grained level i.e., it collects data for each participating process and thread and feeds it to sophisticated framework such as RACE which uses multiple allocation and control algorithms [13]. These algorithms can then (re)deploy and (re)configure the applications with the goal of maintaining stringent Qos requirements. Since Bulls-Eye is targeted for enterprise DRE systems, it focuses on the latency of the data collection and a standard interface to make it available to automated resource management framework, such as RACE.

7 Concluding Remarks

This paper motivated and described Bulls-Eye, which is an implementation of the Lightweight CCM Target Manager specification we developed to support resource provisioning for enterprise DRE systems. We discussed the design challenges faced when developing Bulls-Eye and applying it to a shipboard computing system and showed how our solutions helped resolve these challenges. We also presented results the results of experiments that show how Bulls-Eye simplifies resource management and helps automate adaptations in the face of dynamic operating condition changes.

The following are lessons learned during our work on Bulls-Eye and its application to the Multi-Layer Resource Manager (MLRM) subsystem case study:

- Building enterprise DRE systems whose operational semantics change frequently necessitates the dynamic monitoring of domain resources and requires a framework to provide resource availability information to enable the automated (re)deployment and (re)configuration of heterogeneous components throughout the system.
- The CCM Target Manager specification strikes an effective balance between flexibility and efficiency by keeping client code considerably simpler and supporting dynamic updates and system (re)deployment and (re)configuration.
- Applying patterns to Bulls-Eye helped ensure that its design used best practices associated with solving recurring problems and leveraging the experience of experienced developers. Patterns applied to Bulls-Eye included Adaptor and Strategy.
- Using efficient aggregation algorithms helped improve overall system performance and also increased the responsive of Bulls-Eye, which in turn led to clients responding to changes in the application operating condition or policy in an effective manner.
- The judicious use of distributing computing of resource data across different stages, helped increase the performance of Bulls-Eye by fully exploiting the computing power of distributed hosts across the target domain and distribute complexity over multiple processors.
- Bulls-Eye plays an important role in the allocation of components to different hosts across the domain. It helps allocation algorithms to come up with a deployment plan which optimizes resource usage. Control algorithms which need to re-allocate components due to changing operating environment also use Bulls-Eye to monitor the running of the application.

The implementation of Bulls-Eye is freely available as open-source software and can be downloaded along with the CIAO, DAnCE, and RACE open-source middleware from www.dre.vanderbilt.edu/CIAO.

8 REFERENCES

1. Deng, G., Balasubramanian, J., Otte, W., Schmidt, D. and Gokhale, A. (2005, Nov), "DAnCE: A QoS-enabled Component Deployment and Configuration Engine," Proceedings of the 3rd Working Conference on Component Deployment. Grenoble, France.
2. Gamma, E., Helm, R., Johnson, R., and Vlissides J., "Design Patterns Elements of Reusable Object-Oriented Software," Addison-Wesley, 1994.
3. Object Group Management (2003, May), Light Weight CORBA Component Model Revised Submission, Ed. OMG Document realtime/03-05-05.
4. Object Management Group: Deployment and Configuration Adopted Submission, OMG Document ptc/03-07-08 edn. (2003).
5. Object Management Group (2002, Aug). Real-time CORBA Specification. Ed. OMG Document formal/02-08-02.
6. D. Schmidt, M. Stal, H. Rohert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Networked and Concurrent Objects*, Wiley and Sons, 2000.

7. D. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma, “Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems,” CrossTalk, Nov, 2001.

8. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, D. Sutherland, "ReMoS: A Resource Monitoring System for Network-Aware Applications" Carnegie Mellon School of Computer Science, CMU-CS-97-194.

9. J. Zinky, J. Loyall, and R Shapiro “Runtime, Performance Modeling and Measurement of Adaptive Distributed Object Applications,” Proceeding of International Symposium on Distributed Object and Applications, DOA 2002, Oct 28-30 2002, University of California, Irvine CA USA.

10. I.Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. Intl. Journal of Supercomputer Applications and High Performance Computing, 11(2):115-128, 1997.

11. F. Kon, T. Yamane, C. Hess, R. Campbell, and M. Mickunas, “Dynamic Resource Management and Automatic Configuration of Distributed Component Systems,” Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, Texas, Jan, 2001.

12. R. Wolski, “Experiences with Predicting Resource Performance On-line in Computational Grid Settings” ACM SIGMETRICS Performance Evaluation Review, Volume 30, Number 4, pp 41--49, Mar, 2003.

13. N. Shankaran, J. Balasubramanian, D. Schmidt, G. Biswas, P. Lardieri, E. Mulholland, and T. Damiano, “A Framework for (Re)Deploying Components in Distributed Realtime and Embedded Systems”, poster paper at the Dependable and Adaptive Distributed Systems Track of the 21st ACM Symposium on Applied Computing, Apr 23 -27, 2006, Dijon, France.

14. W. Roll, “Towards Model-Based and CCMBased Applications for Real-Time Systems,” in Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), Hokkaido, Japan, IEEE/IFIP, May 2003.

15. S. Murat, Bicer, F. Pilhofer, G. Bardouleau, and J. Smith, "Next Generation Architecture for Heterogeneous Embedded Systems", International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA). Jun 2003, Las Vegas, NV, USA.

16. S. Paunov and D. Schmidt, “RepoMan: A Component Repository Manager for Enterprise Distributed Real-time and Embedded Systems,” Proceedings of the 44th ACM Southeast Conference, Melbourne, FL, Mar 10-12, 2006.

17. J. Lehoczky, L. Sha, and Y. Ding, “The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior,” In Proceedings of the 10th IEEE Real-time Systems Symposium (RTSS 1989), Santa Monica, Dec 1989.

18. J. Hill, J. Slaby, S. Baker, and D. Schmidt, “Evaluating Enterprise Distributed Real-time and Embedded System Quality of Service with System Execution Modeling Tools,” Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney, Australia, 16-18 Aug 2006.

19. G. Heineman and B. Councill, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley, Reading, Massachusetts, 2001.

20. C. Smith and D. Henry, “High-Performance Linux Cluster Monitoring Using Java,” Proceedings of the 3rd Linux Cluster International Conference, 2002.