Chapter 1

Applying Optimization Principle Patterns to Component Deployment and Configuration Tools

1.1 Introduction

Distributed, real-time and embedded (DRE) systems are an important class of applications that share properties of both enterprise distributed systems and resource-constrained real-time and embedded systems. In particular, applications in DRE systems are similar to enterprise applications, *e.g.*, they are distributed across a large domain. Moreover, like real-time and embedded systems, applications in DRE systems are often mission-critical and carry stringent safety, reliability, and quality of service (QoS) requirements.

In addition to the complexities described above, deployment of application and infrastructure components in DRE systems incurs its own set of unique challenges. First, applications in DRE system domains may have particular dependencies on the target environment, such as particular hardware/-software (*e.g.*, GPS, sensors, actuators, particular real-time operating systems, etc.). Second, the deployment infrastructure of a DRE system must contend with strict resource requirements in environments with finite resources (*e.g.*, CPU, memory, network bandwidth, etc.).

Component-Based Software Engineering (CBSE) [15] is increasingly used as a paradigm for developing applications in both enterprise [2] and DRE systems [34]. CBSE facilitates systematic software reuse by encouraging developers to create black box components that interact with each other and their environment through well-defined interfaces. CBSE also simplifies the deployment of highly complex distributed systems [37] by providing standardized mechanisms to control the configuration and lifecycle of applications. These mechanisms enable the composition of large-scale, complex applications from smaller, more manageable units of functionality, *e.g.,* commercial off-the-shelf components and preexisting application building-blocks. These applications can be packaged along with descriptive and configuration metadata, and made available for deployment into a production environment.

Building on expertise gleaned from the development of *The ACE ORB* (TAO) [31]—an open-source implementation of the *Common Object Request Broker Architecture* (CORBA) standard—we have been applying CBSE principles to DRE systems over the past decade. As a result of these efforts, we have developed a high-quality open-source implementation of the OMG *CORBA Component Model* (CCM), which we call the *Component Integrated ACE ORB* (CIAO) [17]. CIAO implements the so-called *Lightweight CCM* [22] specification, which is a subset of the full CCM standard that is tuned for resource-constrained DRE systems.

In the context of our work on applying CBSE principles to DRE systems, we have also been researching the equally challenging problem of facilitating deployment and configuration of component-based systems in these domains. Managing deployment and configuration of component-based applications is a challenging problem for the following reasons:

- **Component dependency and version management.** There may be complex requirements and relationships amongst individual components. Components may depend on one another for proper operation, or specifically require or exclude particular versions. If these relationships are not described and enforced, component applications may fail to deploy properly; even worse, malfunction in subtle and pernicious ways.

- **Component configuration management.** A component might expose configuration hooks that change its behavior, and the deployment infrastructure must manage and apply any required configuration information. Moreover, several components in a deployment may have related configuration properties, and the deployment infrastructure should ensure that these properties remain consistent across an entire application.

- **Distributed connection and lifecycle management.** In the case of enterprise systems, components must be installed and have their connection and activation managed on remote hosts.

To address the challenges outlined above, we began developing a deployment engine for CIAO in 2005. This tool, which we call the *Deployment and Configuration Engine* (DAnCE) [7], is an implementation of the OMG *Deployment and Configuration* (D&C) specification [25]. For most of its history, DAnCE served primarily as a research vehicle for graduate students developing novel approaches to deployment and configuration, which had two important impacts on its implementation:

- As a research vehicle, DAnCE's development timeline was largely driven by paper deadlines and feature demonstrations for sponsors. As a result, its tested use cases were relatively simple and narrowly focused.
- Custodianship of DAnCE changed hands several times as research projects were completed and new ones started. As a result, there was often not a unified architectural vision for the entire infrastructure.

These two factors had several impacts on DAnCE. For example, narrow and focused use-cases often made evaluating end-to-end performance on real-world application deployments a low priority. Moreover, the lack of a unified architectural vision combined with tight deadlines often meant that poor architectural choices were made in the name of expediency, and were not later remedied. These problems were brought into focus as we began to work with our commercial sponsors to apply DAnCE to larger-scale deployments, numbering in the hundreds to thousands of components on tens to hundreds of hardware nodes. While the smaller, focused uses cases would have acceptable deployment times, these larger deployments would take unacceptably long amounts of time, on the order of an hour or more to fully complete.

In response to these problems, we undertook an effort to comprehensively evaluate the architecture, design, and implementation of DAnCE and create a new implementation that we call *Locality-Enabled DAnCE* (LE-DAnCE) [27, 26]. This chapter focuses on documenting and applying optimization principle patterns that form the core of LE-DAnCE to make it suitable for DRE systems. Table 1.1 summarizes common optimization patterns [35], many of which we apply in LE-DAnCE. An additional goal of this paper was to supplement this catalog with new patterns we identified in our work on LE-DAnCE.

Table 1.1: Catalog of Optimization Principles and Known Usecases in Networking [35]

Title	**Principle**	**Examples from Networking**
Avoiding Waste	Avoid obvious waste	zero-copy [28]
Shifting in Time	Shift computation in time (precompute, lazy evaluation, sharing expenses, batching)	copy-on-write [1, 21], integrated layer processing [5]
Relaxing Specifications	Relax specifications (trading off certainty for time, trading off accuracy for time, and shifting computation in time)	fair queuing [33], IPv6 fragmentation
Leveraging other Components	Leverage other system components (exploiting locality, trading memory for speed, exploiting hardware)	Lulea IP lookups [6], TCP checksum
Adding Hardware	Add hardware to improve performance	Pipelined IP lookup [14], counters
Efficient Routines	Create efficient routines	UDP lookups
Avoiding Generality	Avoid unnecessary generality	Fbufs [8]
Specification vs Implementation	Don't confuse specification and implementation	Upcalls [16]
Passing Hints	Pass information like hints in interfaces	Packet filters [19, 20, 10]
Passing Information	Pass information in protocol headers	Tag switching [29]
Expected Use Case	Optimize the expected case	Header prediction [4]
Exploiting State	Add or exploit state to gain speed	Active VC list
Degrees of Freedom	Optimize degrees of freedom	IP trie lookups [30]
Exploit Finite Universes	Use special techniques for finite universes	Timing wheels [36]
Efficient Data Structures	Use efficient data structures	Level-4 switching

The remainder of this chapter is organized as follows: Section 1.2 provides an overview of the OMG D&C specification; Section 1.3 identifies the most significant sources of DAnCE performance problems (parsing deployment information from XML, analysis of deployment information at run-time, and serialized execution of deployment steps) and uses them as case studies to identify optimization principles that (1) are generally applicable to DRE systems and (2) we applied to LE-DAnCE; and Section 1.4 presents concluding remarks.

1.2 Overview of DAnCE

The OMG D&C specification provides standard interchange formats for metadata used throughout the component-based application development lifecycle, as well as runtime interfaces used for packaging and planning. These runtime interfaces deliver deployment instructions to the middleware deployment infrastructure via a *component deployment plan*, which contains the complete set of deployment and configuration information for component instances and their associated connection information. During DRE system initialization this information must be parsed, components deployed to physical hardware resources, and the system activated in a timely manner.

This section presents a brief summary of the core architectural elements and processes that must be provided by a standards-compliant D&C implementation. We use this summary as a basis to discuss substantial performance and scalability problems in DAnCE, which is our open-source implementation of the OMG *Deployment and Configuration* (D&C) specification [25], as outlined in Section 1.1. This summary is split into three sections: (1) *the DAnCE runtime architecture*, which describes the daemons and actors that are present in the system, the (2) *data model*, which describes the structure of the "deployment plans" that describe component applications, and (3) the *deployment process*, which provides a high level overview of the process by which a deployed distributed application is realized.

1.2.1 Runtime D&C Architecture

The runtime interfaces defined by the OMG D&C specification for deployment and configuration of components consists of the two-tier architecture shown in Figure 1.1. This architecture consists of (1) a set of global (system-wide) entities used to coordinate deployment and (2) a set of local (node-level) entities used to instantiate component instances and configure their connections and QoS properties. Each entity in these global and local tiers correspond to one of the following three major roles:

- **Manager**. This role (known as the *ExecutionManager* at the global-level and as the *NodeManager* at the node-level) is a singleton daemon that coordinates all deployment entities in a single context. The Manager serves as the entry point for all deployment activity and as a factory for implementations of the *ApplicationManager* role.

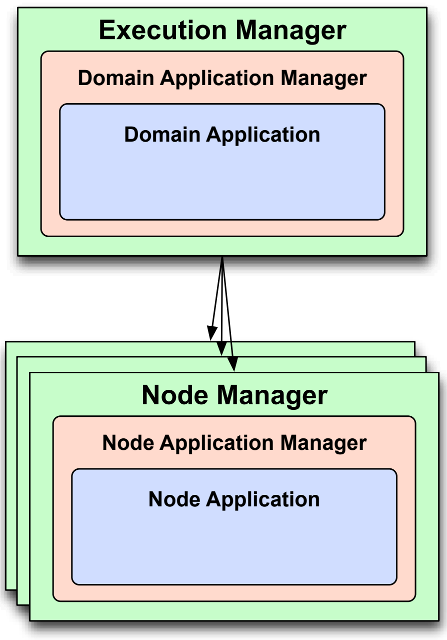

Figure 1.1: OMG D&C Architectural Overview and Separation of Concerns

- **ApplicationManager**. This role (known as the *DomainApplication-Manager* at the global-level and as the *NodeApplicationManager* at the node-level entity) coordinates the lifecycle for running instances of a component-based application. Each ApplicationManager represents exactly one component-based application and is used to initiate deployment and teardown of that application. This role also serves as a factory for implementations of the *Application* role.

- **Application**. This role (known as the *DomainApplication* at the global-level and the *NodeApplication* at the node-level entity) represents a deployed instance of a component-based application. It is used to finalize the configuration of the associated component instances that comprise an application and begin execution of the deployed component-based application.

1.2.2 D&C Deployment Data Model

In addition to the runtime entities described above, the D&C specification also contains an extensive data model that is used to describe component applications throughout their deployment lifecycle. The metadata defined by the specification is intended for use as

- An interchange format between various tools (*e.g.*, development tools, application modeling and packaging applications, and deployment planning tools) applied to create the applications and

- Directives that describe the configuration and deployment used by the runtime infrastructure.

Most entities in the D&C metadata contain a section where configuration information may be included in the form of a sequence of name/value pairs, where the value may be an arbitrary data type. This configuration information can be used to describe everything from basic configuration information (such as shared library entry points and component/container associations) to more complex configuration information (such as QoS properties or initialization of component attributes with user-defined data types).

This metadata can broadly be grouped into three categories: *packaging*, *domain*, and *deployment*. Packaging descriptors are used from the beginning of application development to specify component interfaces, capabilities, and requirements. After implementations have been created, this metadata is further used to group individual components into assemblies, describe pairings with implementation artifacts, such as shared libraries (also known as dynamically linked libraries), and create packages containing both metadata and implementations that may be installed into the target environment. Domain descriptors are used by hardware administrators to describe capabilities (*e.g.*, CPU, memory, disk space, and special hardware such as GPS receivers) present in the domain.

1.2.3 OMG D&C Deployment Process

Component application deployments are performed in a four phase process codified by the OMG D&C standard. The *Manager* and *ApplicationManager* are responsible for the first two phases and the *Application* is responsible for the final two phases, as described below:

1. **Plan preparation.** In this phase, a deployment plan is provided to the *ExecutionManager*, which (1) analyzes the plan to determine which nodes are involved in the deployment and (2) splits the plans into "locality-constrained" plans, one for each node containing information only for the corresponding node. These locality-constrained plans have only instance and connection information for a single node. Each *Node-Manager* is then contacted and provided with its locality-constrained plan, which causes the creation of *NodeApplicationManagers* whose reference is returned. Finally, the *ExecutionManager* creates a *Domain-ApplicationManager* with these references.

2. **Start launch.** When the *DomainApplicationManager* receives the start launch instruction, it delegates work to the *NodeApplication-Managers* on each node. Each *NodeApplicationManager* creates a *Node-Application* that loads all component instances into memory, performs preliminary configuration, and collects references for all endpoints described in the deployment plan. These references are then cached by a *DomainApplication* instance created by the *DomainApplication-Manager*.

3. **Finish launch.** This phase is started by an operation on the *Domain-Application* instance, which apportions its collected object references from the previous phase to each *NodeApplication* and causes them to initiate this phase. All component instances receive final configurations and all connections are then created.

4. **Start.** This phase is again initiated on the *DomainApplication*, which delegates to the *NodeApplication* instances and causes them to instruct all installed component instances to begin execution.

1.3 Applying Optimization Principle Patterns to DAnCE

This section examines three of the most problematic performance problems we identified when applying DAnCE to component-based applications in a large-scale production DRE system. We first describe a case study that highlights many of these performance challenges. We then identify the causes of performance degradation and use this discussion to present optimization principles, which are guidelines that may be applied in other situations and applications to remedy or prevent performance problems.

1.3.1 Overview of the SEAMONSTER Platform

An example DRE system that revealed significant performance issues with DAnCE was a collaboration with the University of Alaska on the *South East Alaska MOnitoring Network for Science, Telecommunications, Education, and Research* (SEAMONSTER) platform. SEAMONSTER is a glacier and watershed sensor web hosted at the University of Alaska Southeast (UAS) [11]. This sensor web monitors and collects data regarding glacier dynamics and mass balance, watershed hydrology, coastal marine ecology, and human impact/hazards in and around the Lemon Creek watershed and Lemon Glacier. The collected data is used to study the correlations between glacier velocity, glacial lake formation and drainage, watershed hydrology, and temperature variation.

The SEAMONSTER sensor web includes sensors and weatherized computer platforms that are deployed on the glacier and throughout the watershed to collect data of scientific interest. The data collected by the sensors is relayed via wireless networks to a cluster of servers that filter, correlate, and analyze the data. Effective deployment of data collection and filtering applications on SEAMONSTER field hardware and dynamic adaptation to changing environmental conditions and resource availability present significant software challenges for efficient operation of SEAMONSTER. While SEAMONSTER servers provide significant computational resources, the field hardware is computationally constrained.

Field nodes in a sensor web often have a large number of observable phenomena in their area of interest. The type, duration, and frequency of observation of these phenomena may change over time, based on changes in the environment, occurrence of transient events in the environment, and changing goals and objectives in the science mission of the sensor web. Moreover, limited power, processing capability, storage, and network bandwidth constrain the ability of these nodes to continually perform observations at the desired frequency and fidelity. Dynamic changes in environmental conditions coupled with limited resource availability requires individual nodes of the sensor web to rapidly revise current operations and future plans to make the best use of their resources.

To address these challenges, we proposed to transition the data collection and processing tasks to a middleware platform built on top of the CIAO and DAnCE middleware described in Section 1.1 and 1.2, respectively. We developed a run-time planner [18] that analyzed the physical observations of the sensor nodes. Based on that information—as well as the operational goals of the network—the planner generates deployment plans describing desired software configuration.

Using DAnCE to apply the deployment changes requested by the run-time planner, however, revealed a number of shortcomings in its performance. These shortcomings were exacerbated by the limited performance of the field hardware, relative slowness of the network linking the nodes, and the stringent real-time requirements of the system. Each of these shortcomings is described below.

1.3.2 Optimizing Deployment Plan Parsing

1.3.2.1 Context

Component application deployments for OMG D&C are described by a data structure that contains all the relevant configuration metadata for the component instances, their mappings to individual nodes, and any connection information required. This deployment plan is serialized on disk in a XML file whose structure is described by an XML Schema defined by the D&C specification. This XML document format presents significant advantages by providing a simple interchange format for exchanging deployment plan files between modeling tools [13].

For example, in the SEAMONSTER case study this format provided a convenient interchange format between the planning front end and the deployment infrastructure. This format is also easy to generate and manipulate using widely available XML modules for popular programming languages. Moreover, it enables simple modification and data mining by text processing tools such as perl, grep, sed, and awk.

1.3.2.2 Problem

Processing these deployment plan files during deployment and even runtime, however, can lead to substantial performance penalties. These performance penalties stem from the following sources:

- XML deployment plan file sizes grow substantially as the number of component instances and connections in the deployment increases, which causes significant I/O overhead to load the plan into memory and to validate the structure against the schema to ensure that it is well-formed.

- The XML document format cannot be directly used by the deployment infrastructure because the infrastructure is a CORBA application that implements OMG *Interface Definition Language* (IDL) interfaces. Hence, the XML document must first be converted into the IDL format used by the runtime interfaces of the deployment framework.

In DRE systems, component deployments that number in the thousands are not uncommon. Moreover, component instances in these domains will exhibit a high degree of connectivity. Both these factors contribute to large plans. Plans need not be large, however, to significantly impact the operation of a system. Though the plans were significantly smaller in the SEAMONSTER case study described above the extremely limited computational resources meant that the processing overhead for even smaller plans was often too time consuming.

1.3.2.3 Optimization Principle Patterns in Parsing Configuration Metadata

There are two general approaches to resolving the challenge of XML parsing outlined in Section 1.3.2.2.

1. Optimize the XML-to-IDL processing capability. DAnCE uses a vocabulary-specific XML data binding [38] tool called the *XML Schema Compiler* (XSC). XSC reads D&C XML schemas and generates a C++-based interface to XML documents built atop the *Document Object Model* (DOM) XML programming API. DOM is a time/space-intensive approach since the entire document must first be processed to construct a tree-based representation of the document prior to initiating the XML-to-IDL translation process. Since deployment plan data structures contain extensive internal cross-referencing, an alternative to DOM including event-based mechanisms to process deployment plans, such as the *Simple API for XML* (SAX), would not yield substantial gains either.

The C++ data binding generated by XSC creates a number of classes (based on the content of the XML schema) that provide strongly-typed object-oriented access to the data in the XML document. Moreover, this interface leverages features of the C++ STL to help programmers write compact and efficient code to interact with their data. The general process for populating these wrappers is to 1) parse the XML document using a DOM XML parser; 2) parse the DOM tree to populate the generated class hierarchy. In order to enhance compatibility with STL algorithms and functors, XSC stores its data internally inside STL container classes.

Initial versions of the XSC data binding were highly inefficient. Even relatively modest deployments numbering as few as several hundred to a thousand components would take nearly half an hour to process. After analyzing the execution of this process using tools such as Rational Quantify revealed a very straightforward problem: the generated XSC code was individually inserting elements into its internal data structures (in this case, `std::vector`) in a naive manner. As a result, exorbitant amounts of time was spent re-allocating and copying data inside these containers for each additional element inserted.

Below we present specific guidelines that developers must be aware of:

- *Be aware of the cost of your abstractions.* High level abstractions, such as the container classes that are available in the C++ STL can greatly simplify programs by reducing the need to reproduce complex and error-prone lower level (largely boilerplate) code. It is important to characterize and document (when writing abstractions) and understand (when using them) what hidden costs may be incurred by using the higher level operations provided by your abstraction.

- *Use appropriate abstractions for your use case.* Often, there is a choice to be made between abstractions that provide similar functionality. An example may be the choice between `std::vector` and `std::list`; each presents its own advantages. In XSC, `std::vector` was initially used because we desired random access to elements in the data binding; the cost was extremely poor performance when parsing the XML document due to poor insertion performance. Our use case, however, only required sequential access, so the much better insertion performance of `std::list` was in the end much more desirable.

By understanding the specific requirements of the particular use case of our generated XML data binding — in particular that most nodes are visited a single time and can be visited in order — we are able to apply the pattern

Expected Use Case through the application of two other optimization patterns. The *Avoiding Generality* pattern is applicable in this case because we consciously avoid generality by generating the data binding without random access containers. We then chose to use the most efficient data structure (*Efficient Data Structures* pattern) to satisfy that lack of generality.

2. Preprocess the XML files for latency-critical deployments. While optimizing the XML to IDL conversion process yielded conversion times that were tractable, this step in the deployment process still consumed a large fraction of the total time required for deployment. This yet-unresolved overhead could be avoided by applying another optimization principle pattern:

- *When possible, perform costly computations outside of the critical path.* In many cases, the result of costly procedures and computations can be pre-computed and stored for later retrieval. This is especially true in cases such as the XML deployment plan, which is unlikely to change between when it is generated, and when the application deployment is requested.

This optimization approach is applying optimization pattern *Shifting in Time* by shifting the costly conversion of the deployment plan to a more efficient binary format outside of the critical path of application deployment. In applying this pattern, we first convert the deployment plan into its runtime IDL representation. We then serialize the result to disk using the *Common Data Representation* (CDR) [23] binary format defined by the CORBA specification. The SEAMONSTER on-line planner could take advantage of this optimization by producing these binary plans in lieu of XML-based deployment plans, significantly reducing latency.

The platform-independent CDR binary format used to store the deployment plan on disk is the same format used to transmit the plan over the network at runtime. The advantage of this approach is that it leverages the heavily optimized de-serialization handlers provided by the underlying CORBA implementation. These handlers create an in-memory representation of the deployment plan data structure from the on-disk binary stream.

1.3.3 Optimizing Plan Analysis

1.3.3.1 Context

After a component deployment plan has been loaded into an in-memory representation, it must be analyzed by the middleware deployment infrastructure before any subsequent deployment activity is performed. This analysis occurs during the plan preparation phase described in Section 1.2.3. The goal of this analysis is to determine (1) the number of deployment sub-problems that are part of the deployment plan and (2) which component instances belong to each sub-problem.

As mentioned in Section 1.2.3, the output of this analysis process is a set of “locality-constrained” sub-plans. A locality-constrained sub-plan contains all the necessary metadata to execute a deployment successfully. It therefore contains copies of the information contained in the original plan (described in Section 1.2.2).

The runtime plan analysis is actually conducted twice during the plan preparation phase of deployment: once at the global level and again on each node. Global deployment plans are split according to the node that the individual instances are assigned to. This two-part analysis results in a new sub-plan for each node that only contains the instances, connections, and other component metadata necessary for that node.

The algorithm for splitting plans used by our DAnCE implementation of the D&C specification is straightforward. For each instance to be deployed in the plan, the algorithm determines which sub-plan should contain it and retrieve the appropriate (or create a new) sub-plan data structure. As this relationship is determined, all metadata necessary for that component instance is copied to the sub-plan, including connections, metadata describing executables, shared library dependencies, etc.

1.3.3.2 Problem

While this approach is conceptually simple, it is fraught with accidental complexities that yield the following inefficiencies in practice:

1. **Reference representation in IDL**. Deployment plans are typically transmitted over networks, so they must obey the rules of the CORBA IDL language mapping. Since IDL does not have any concept of references or pointers, some alternative mechanism must be used to describe

the relationships between plan elements. The deployment plan stores all the major elements in sequences, so references to other entities can be represented with simple indices into these sequences. While this implementation can follow references in constant time, it also means these references become invalidated when plan entities are copied to sub-plans, as their position in deployment plan sequences will most likely be different. It is also impossible to determine if the target of a reference has already been copied without searching the sub-plan, which is time-consuming.

2. **Memory allocation in deployment plan sequences**. The CORBA IDL mapping requires that sequences be stored in consecutive memory addresses. If a sequence is resized, therefore, its contents will most likely be copied to another location in memory to accommodate the increased sequence size. With the approach summarized above, substantial copying overhead will occur as plan sizes grow. This overhead is especially problematic in resource-constrained systems (such as our SEAMONSTER case study), whose limited run-time memory must be conserved for application components. If the deployment infrastructure is inefficient in its use of this resource, either it will exhaust the available memory, or cause significant thrashing of any virtual memory available (both impacting deployment latency and the usable life of flash-based storage).

3. **Inefficient parallelization of plan analysis**. The algorithm described above would appear to benefit greatly from parallelization, as the process of analyzing a single component and determining which elements must be copied to a sub-plan is independent of all other components. Multi-threading this algorithm, however, would likely not be effective because access to sub-plans to copy instance metadata must be serialized to avoid data corruption. In practice, component instances in the deployment plan are usually grouped according to the node and/or process since deployment plans are often generated from modeling tools. As a result, multiple threads would likely compete for a lock on the same sub-plan, which would cause the "parallelized" algorithm to run largely sequentially. While parallelization has historically been viewed as non-applicable to resource-constrained DRE systems (such as SEAMONSTER), the advent of multi-core proces-

sors in single-board computers is motivating more parallelism in these environments.

1.3.3.3 Optimization Principle Patterns in Analysis of Deployment Plans

This performance challenge could potentially be resolved by applying the *Specification vs Implementation* pattern, and leveraging some of the same optimization principles described earlier for the XSC tool, especially *being aware of the cost of abstractions*, and *using appropriate containers for the use case.* For example, pointers/references could be used instead of sequence indices to refer to related data structures, potentially removing the need to carefully rewrite references when plan entities are copied between plans. Likewise, an associative container (such as an STL map) instead of a sequence could store plan objects, thereby increasing the efficiency of inserting plan entities into sub-plans.

While these and other similar options are tempting, there are some inherent complexities in the requirements of the D&C standard that make these optimizations less attractive. Since this data must be transmitted to other entities as part of the deployment process, using a more efficient representation for analysis would introduce yet another conversion step into the deployment process. This conversion would potentially overwhelm any gains attained by this new representation.

A more attractive result is to apply a different set of optimization principles to this problem, outlined below:

- **Cache previously calculated results for later use.** This is an example of the patterns *Shifting in Time* and *Exploiting State.* It is possible to perform a simple pre-analysis step to pre-calculate values that will be more time consuming to perform later. In this case, iterating over the plan first to determine the final sizes necessary to contain the calculated sub-plans and cache that state for later use.

- **Where possible, pre-allocate any data structures.** As a result of the additional state gleaned through the pre-analysis step described above, we can apply the *Avoiding Waste* and avoid gratuitous waste by pre-allocating the sequences which were previously being re-allocated each time a new plan element was discovered.

- **Design your algorithms to take advantage of parallelization.** While this can be seen as an application of the *Adding Hardware*, this pattern speaks more to taking advantage of intrinsic properties of hardware such as word size caching effects. Moreover, this pattern speaks to adding special purpose hardware to perform specialized calculations.

  Taking advantage of multiple general-purpose processors is an 1 important emerging principle. Since multi-core computers are pervasive in desktop and server domains, and are becoming increasingly common even in embedded domains, it is increasingly important to design for this important hardware feature. We therefore propose an additional pattern which we will call *Design for Parallelization*, wherein one optimizes design of algorithms and interfaces for parallelization, shown in Table 1.2.

- **Structure shared data access to avoid necessary use of synchronization.** Synchronization, *e.g.* using mutexes to protect access to shared data, is tedious and error prone to use. Moreover, overzealous use of synchronization can often entirely negate any parallelization of your algorithms. A much more preferable approach is to structure your algorithms to eliminate the need for synchronization entirely; requiring only shared *read* access to data, instead of shared *write* access.

  This optimization principle is not only an important companion to *Design for Parallelization* proposed above, but is also a wise programming practice in general: deadlocks and race conditions caused by incorrect synchronization are pernicious and difficult to diagnose bugs. Indeed, our recent work in software frameworks intended for fractionated spacecraft has proposed a component model that eliminates synchronization from application code entirely [9]. To that end, we propose another optimization pattern which we call *Avoid Synchronization*, wherein one should avoid overzealous synchronization and locking, shown in Table 1.2 below.

These principles can be applied to the algorithm described above to create a version that is far more amenable to optimization; the new algorithm (along with how the above principles influenced the design, is described below.

1. **Phase 1: Determine the number of sub-plans to produce.** In this phase, a single thread iterates over all component instances contained in the deployment plan to determine the number of necessary

sub-plans. When this operation is performed at the global level, it simply requires a constant time operation per instance. When performed at the local level, it requires that locality constraints (described in Section 1.2.2) be evaluated. Since this phase is potentially time consuming the results are cached for later use. This is an example of *Shifting in Time* and *Exploiting State.*

2. **Phase 2: Preallocate data structures for sub-plans**. Using information gleaned in phase 1 above, preallocate data structures necessary to assemble sub-plans. As part of this preallocation it is possible to reserve memory for each sequence in the sub-plan data structure to avoid repeated resizing and copying. Statistics are collected in phase 1 to estimate these lengths efficiently. This is an example of *Avoiding Waste*

3. **Phase 3: Assemble node-specific sub-plans.** This phase of the new analysis process is similar to the algorithm described at the beginning of this section. The main difference is that the cached results of the pre-analysis phase are used to guide the creation of sub-plans. Instead of considering each instance in order (as the original DAnCE implementation did), LE-DAnCE fully constructs one sub-plan at a time, by processing instances on a per-node basis. This approach simplifies parallelizing this phase by dedicating a single thread per sub-plan and eliminates any shared state between threads, except for read-only access to the original plan. It is therefore unnecessary to use any locking mechanism to protect access to the sub-plans. This is an example of *Design for Parallelization* and *Avoid Synchronization.*

The revised algorithm above is a much more efficient implementation of plan analysis, and can show improvement even on the single-core embedded processors that were typical of the SEAMONSTER use-case: the above is far more memory efficient, both in terms of space used and the amount of re-allocation that is necessary. The use of multi-core embedded processors would substantially improve run-time performance over the old algorithm.

1.3.4 Optimization Through Reduction in Serialized Execution of Deployment Tasks

1.3.4.1 Context

The complexities presented below involve the serial (non-parallel) execution of deployment tasks. The related sources of latency in DAnCE exist at both the global and node level. At the global level, this lack of parallelism results from the underlying CORBA transport used by DAnCE. The lack of parallelism at the local level, however, results from the lack of specificity in terms of the interface of the D&C implementation with the target component model that is contained in the D&C specification.

The D&C deployment process presented in Section 1.2.3 enables global entities to divide the deployment process into a number of node-specific subtasks. Each subtask is dispatched to individual nodes using a single remote invocation, with any data produced by the nodes passed back to the global entities via “out” parameters that are part of the operation signature described in IDL. Due to the synchronous (request/response) nature of the CORBA messaging protocol used to implement DAnCE, the conventional approach is to dispatch these subtasks serially to each node. This approach is simple to implement in contrast to the complexity of using the CORBA *asynchronous method invocation* (AMI) mechanism [3].

1.3.4.2 Problem

To minimize initial implementation complexity, we used synchronous invocation in an (admittedly shortsighted) design choice in the initial DAnCE implementation. This global synchronicity worked fine for relatively small deployments with less than $\sim$100 components. As the number of nodes and instances assigned to those nodes scaled up, however, this global/local serialization imposed a substantial cost in deployment latency.

This serialized execution yielded the most problematic performance degradation in our SEAMONSTER case study, *i.e.*, the limited computational resources available on the field hardware would often take several minutes to complete. Such latency at the node level can quickly becomes disastrous. In particular, even relatively modest deployments involving tens of nodes quickly escalates the deployment latency of the system to a half hour or more.

This serialization problem, however, is not limited only to the global/local task dispatching; it exists in the node-specific portion of the infrastructure, as well. The D&C specification provides no guidance in terms of how the NodeApplication should interface with the target component model, such as the CORBA Component Model (CCM), instead leaving such an interface as an implementation detail.

In DAnCE, the D&C architecture was implemented using three processes, as shown in Figure 1.2. The ExecutionManager and NodeManager processes

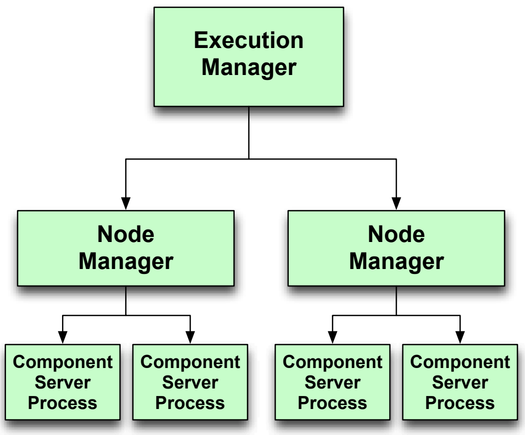

Figure 1.2: Simplified Serialized DAnCE Architecture

instantiate their associated ApplicationManager and Application instances in their address spaces. When the NodeApplication installs concrete component instances it spawns one (or more) separate application processes as needed. These application processes use an interface derived from an older version of the CCM specification that allows the NodeApplication to instantiate containers and component instances individually. This approach is similar to that taken by CARDAMOM [24] (which is another open-source CCM implementation) that is tailored for enterprise DRE systems, such as air-traffic management systems.

The DAnCE architecture shown in Figure 1.2 was problematic with respect to parallelization since its NodeApplication implementation integrated all logic necessary for installing, configuring, and connecting instances directly (as shown in Figure 1.3), rather than performing only some processing

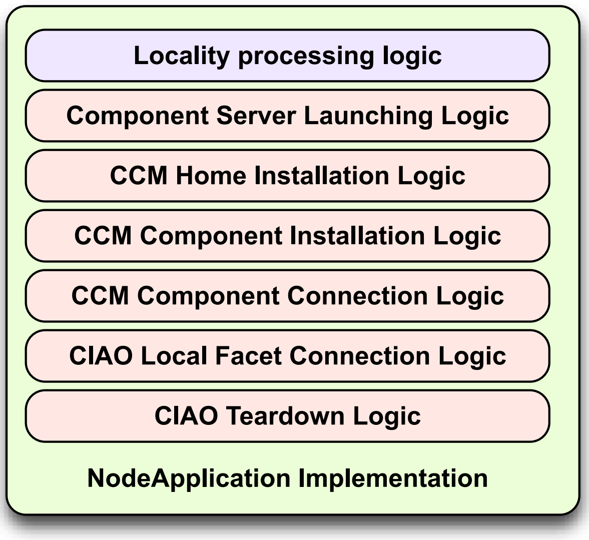
Locality processing logic
Component Server Launching Logic
CCM Home Installation Logic
CCM Component Installation Logic
CCM Component Connection Logic
CIAO Local Facet Connection Logic
CIAO Teardown Logic
NodeApplication Implementation

Figure 1.3: Previous DAnCE NodeApplication Implementation

and delegating the remainder of the concrete deployment logic to the application process. This tight integration made it hard to parallelize the node-level installation procedures for the following reasons:

- The amount of data shared by the *generic deployment logic* (the portion of the NodeApplication implementation that interprets the plan) and the *specific deployment logic* (the portion which has specific knowledge of how to manipulate components) made it hard to parallelize their installation in the context of a *single* component server since that data must be modified during installation.

- Groups of components installed to separate application processes were considered as separate deployment sub-tasks, so these groupings were handled sequentially one after the other.

1.3.4.3 Optimization Principle Patterns in Reducing Serialized Phasing

In a similar vein to the analysis problem described earlier, this is a problem wherein excessive serialization is impacting performance. In this case, however, instead of re-evaluating the algorithmic approach to the *deployment process*, we will re-consider the *architectural design* of the system instead. In order to address the performance challenge in this case, we applied the following optimization principles to DAnCE:

1. **Don't let specifications overly constrain your design.** When implementing a system or software framework according to the specification, it is often natural to model your design along the strictures and implicit assumptions of the specification. It is often possible to architect your implementation in order to introduce architectural elements or behavior that remain within the strictures of the specification. This is an example of both the *Specification vs. Implementation* pattern and the *Degrees of Freedom* pattern.

2. **Maintain strict separation of concerns.** Ensure that your system operates in *layers* or *modules* that interact through well-defined interfaces. This helps to ensure that the state for each layer or module is well-contained, simplifying interactions between logically distinct portions of your applications and making it easier to apply the *Design for Parallelization* pattern. Moreover, ensuring that the state for each layer is self contained helps to apply the *Avoid Synchronization* pattern.

   Moreover, modularizing your software design can often reveal ways that other optimization principle patterns can be applied. As such, we propose another principle pattern, *Separate Concerns*, leveraging separation of concern to modularize architecture (summarized in Table 1.2. Although traditionally a level of indirection may be frowned upon because it could lead to performance penalties, sometimes it can reveal new opportunities or help apply other optimizations.

3. **Ensure that these layers or modules can interact asynchronously.** If the modules or layers in your architecture have interfaces that assume synchronous operation, it becomes difficult to leverage parallel operation to improve performance. Even if the interface is itself synchronous, it is often possible to use other techniques, such as leveraging

abstractions that allow you to interact with a synchronous interface in an asynchronous manner. Avoiding synchronous interactions between is another important application of the *Design for Parallelization* pattern.

Applying these principles at the global level (*e.g.*, the `ExecutionManager` described in Section 1.2.1; the separation of concerns is maintained by virtue of the fact that it and the node-level resources are in separate processes, and likely the different physical nodes. Asynchrony in this context is also easy to achieve, as we were able to leverage the CORBA Asynchronous Method Invocation (AMI) to allow the client (in this case, the global infrastructure) to interact asynchronously with the synchronous server interface (in this case, the node level infrastructure), and dispatch multiple requests to individual nodes in parallel. This is an example of *Degrees of Freedom* in that the specification does not reject the notion of asynchronous interaction between these entities.

Applying these principles in the node level infrastructure, however, was more challenging. As described above, our initial implementation had poor separation of concerns, making it extremely difficult to apply multiple threads of execution in order to parallelize deployment activity at the node level. To support this, we created a new abstraction at the node level that we called the Locality Manager, which was the result of applying the above optimization principles.

Overview of the LE-DAnCE Locality Manager. The LE-DAnCE node-level architecture (*e.g.*, NodeManager, NodeApplicationManager, and NodeApplication) now functions as a node-constrained version of the global portion of the OMG D&C architecture. Rather than having the NodeApplication directly triggering installation of concrete component instances, this responsibility is now delegated to LocalityManager instances. The node-level infrastructure performs a second "split" of the plan it receives from the global level by grouping component instances into one or more application processes. The NodeApplication then spawns a number of LocalityManager processes and delegates these "process-constrained" (*i.e.*, containing only components and connections apropos to a single process) plans to each application process in parallel.

The Locality Manager is an example of the *Specification vs. Implementation* pattern. The specification would suggest that the NodeApplication is the final entity that interacts with the component middleware; by recognizing that our implementation could introduce another layer of abstraction, we've been able to apply a number of other optimization patterns.

Unlike the previous DAnCE NodeApplication implementation, the LE-DAnCE LocalityManager functions as a generic application process that strictly separates concerns between the general deployment logic needed to analyze the plan and the specific deployment logic needed to install and manage the lifecycle of concrete component middleware instances. This separation is achieved using entities called *Instance Installation Handlers*, which provide a well-defined interface for managing the lifecycle of a component instance, including installation, removal, connection, disconnection, and activation. Installation Handlers are also used in the context of the NodeApplication to manage the life-cycle of LocalityManager processes.

The genesis of these installation handlers is an example of the *Degrees of Freedom* pattern; by under specifying the explicit interaction with the component middleware, it has left us free to design our own interaction. In doing do, we have applied the *Separate Concerns* pattern.

Using the Locality Manager to reduce serialized execution of deployment steps. LE-DAnCE's new LocalityManager and Installation Handlers make it substantially easier to parallelize than DAnCE. Parallelism in both the LocalityManager and NodeApplication is achieved using an entity called the *Deployment Scheduler*, which is shown in Figure 1.4. The Deployment Scheduler combines the Command pattern [12] and the Active Object pattern [32]. Individual deployment actions (*e.g.*, instance installation, instance connection, *etc.*) are encased inside an Action object, along with any required metadata. Each individual deployment action is an invocation of a method on an Installation Handler, so these actions need not be rewritten for each potential deployment target. Error handling and logging logic is also fully contained within individual actions, further simplifying the LocalityManager.

Individual actions (*e.g.*, install a component or create a connection) are scheduled for execution by a configurable thread pool. This pool can provide user-selected, single-threaded, or multi-threaded behavior, depending on application requirements. This thread pool can also be used to implement more sophisticated scheduling behavior, *e.g.*, a priority-based scheduling algorithm that dynamically reorders the installation of component instances based on metadata present in the plan.

The LocalityManager determines which actions to perform during each particular phase of deployment and creates one Action object for each in-

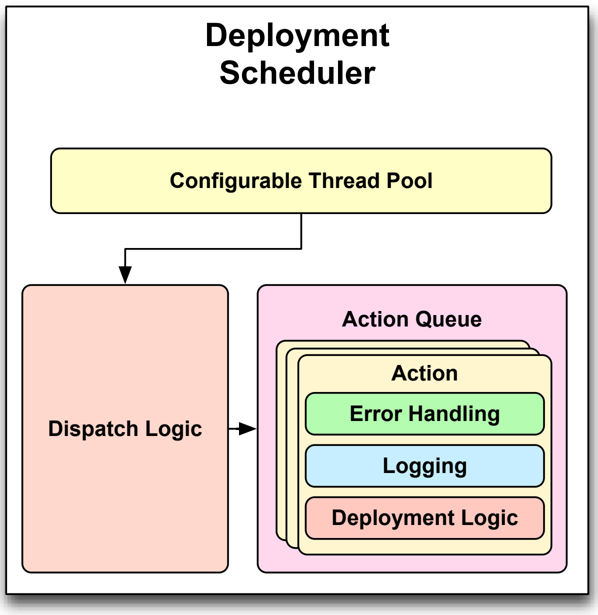

Figure 1.4: DAnCE Deployment Scheduler

struction. These actions are then passed to the deployment scheduler for execution while the main thread of control waits on a completion signal from the Deployment Scheduler. Upon completion, the LocalityManager reaps either return values or error codes from the completed actions and completes the deployment phase.

To provide parallelism between LocalityManager instances on the same node, the LE-DAnCE Deployment Scheduler is also used in the implementation of the NodeApplication, along with an Installation Handler for LocalityManager processes. Using the Deployment Scheduler at this level helps overcome a significant source of latency whilst conducting node-level deployments. Spawning LocalityManager instances can take a significant amount of time compared to the deployment time required for component instances, so parallelizing this process can achieve significant latency savings when application deployments have many LocalityManager processes per node.

Taken together, the dynamic re-ordering of deployment events and parallel installation of LocalityManager instances is a promising approach to improve deployment latency in the SEAMONSTER domain. By attaching high priority to critical deployment events, such as the activation or change in configuration of a sensor observing a present natural phenomena, DAnCE can help ensure that critical mission needs are met in a timely fashion. Moreover, the parallelism enabled by this design can reduce latency by allowing other LocalityManager instances to execute while one is blocked on I/O as it loads new component implementations, or by taking advantage of newer multicore embedded processors.

1.4 Concluding Remarks

This chapter provided an overview of the *Deployment And Configuration Engine* (DAnCE), an implementation of the OMG *Deployment and Configuration* specification. As a research tool, DAnCE was used to demonstrate novel techniques for the deployment and configuration (D&C) of component-based applications in DRE systems. While its performance was satisfactory for the narrow and focused demonstrations required for publications and demonstration, its performance was not satisfactory when applied to larger-scale production DRE systems. A number of factors, including changing architectural ownership and the demo-focused nature of DAnCE's development, caused a number of poor design choices early on to become entrenched in its architecture and design, seriously impeding performance.

A typical use case of DAnCE, in this case the *South East Alaska MOnitoring Network for Science, Telecommunications, Education, and Research* (SEAMONSTER) platform, was described to highlight many of the optimization opportunities present in DAnCE. Motivated by this use case, this paper described how we applied a catalog of optimization principles from the domain of networking to re-evaluate and re-engineer the design and implementation of DAnCE to remedy the deficiencies outlined above. In addition, we described three additional optimization principles: dealing with parallelization, synchronization, and separation of concerns. These additional patterns—in conjunction with those described in the initial catalog—were used to develop LE-DAnCE, which substantially improved the performance and reliability of DAnCE. A summary of the original pattern catalog, along with our additions, is shown in Table 1.2. Likewise, a thorough quantitative discussion of the performance enhancement results is described in [26].

Based on our experiences applying the optimizations described in this chapter to LE-DAnCE and observing the results, we have learned the following lessons:

- **Taking advantage of parallelization is a critical optimization opportunity.** As multicore processors become a standard feature of even embedded devices, it is critically important that algorithms and processes be designed to take advantage of this capability. When optimizing algorithms and processes for parallelization, be judicious in applying synchronization since improper use of locks can cause parallel systems to operate in a serial fashion, or worse, malfunction in subtle ways.

- **When possible, shift time consuming operations out of the critical path**. While our optimizations to the plan analysis portion of the D&C process (described in Section 1.3.3) were effective in reducing the total deployment latency for large scale deployments, additional improvement is possible by further applying the *Shifting in Time* pattern Like the XML parsing problem described in Section 1.3.2, the result of this operation is likely fixed at the point that the XML plan is generated. This process could be similarly pre-computed and provided to the D&C infrastructure for additional latency savings. Passing these pre-computed plans (both for the global split and the local split) would be an example application of the *Passing Hints* optimization pattern.

- **Serialized execution of processes is a major source of performance problems in DRE systems.** Executing tasks in a serial fashion when designing distributed systems offers significant conceptual and implementation simplicity. This simplicity, however, often comes with a significant performance penalty. Often, the additional complexity of asynchronous interaction is well worth the additional complexity.

- **Lack of clear architectural and technical leadership is detrimental to open-source projects.** Developers often contribute to an open-source project to solve a narrow problem and leave soon after. Without clear leadership, poor architectural and technical decisions made by individual contributors eventually snowball into a nearly unusable project.

TAO, CIAO, and LE-DAnCE are available in open-source form from `download.dre.vanderbilt.edu`.

Table 1.2: Catalog of Optimization Principles and Known Usecases in LE-DAnCE

Title	**Principle**	**Examples from LE-DAnCE**
Avoiding Waste	Avoid obvious waste	Pre-allocate memory when parsing deployment plans.
Shifting in Time	Shift computation in time (pre-compute, lazy evaluation, sharing expenses, batching)	Pre-convert deployment plan to binary format, *potentially pre-compute plan splits.*
Relaxing Specifications	Relax specifications (trading off certainty for time, trading off accuracy for time, and shifting computation in time)	*Potentially pre-compute plan splits.*
Leveraging other Components	Leverage other system components (exploiting locality, trading memory for speed, exploiting hardware)	(n/a)?
Adding Hardware	Add hardware to improve performance	(n/a)
Efficient Routines	Create efficient routines	XML-IDL Data Binding
Avoiding Generality	Avoid unnecessary generality	Optimize plan parsing
Specification vs Implementation	Don't confuse specification and implementation	LocalityManager
Passing Hints	Pass information like hints in interfaces	*Potentially used to pre-compute plan splits*
Passing Information	Pass information in protocol headers	(n/a)
Expected Use Case	Optimize the expected case	XML-IDL Data Binding
Exploiting State	Add or exploit state to gain speed	Pre-allocate child plans during plan analysis.
Degrees of Freedom	Optimize degrees of freedom	LocalityManager Installation Handlers
Exploit Finite Universes	Use special techniques for finite universes	(n/a)
Efficient Data Structures	Use efficient data structures	Optimize XML-IDL data binding
Design for Parallelization	Optimize design for parallelization	Process child plans in parallel
Avoid Synchronization	Avoid synchronization and locking	Unsynchronized access to parent plan during plan analysis.
Separate Concerns	Use strict separation of concerns to modularize architecture	Locality Manager

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tavanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pages 93–112, Atlanta, GA, June 1986.

[2] Anatoly Akkerman, Alexander Totok, and Vijay Karamcheti. Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. In *3rd International Working Conference on Component Deployment (CD 2005)*, pages 17–32, Grenoble, France, November 2005.

[3] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[4] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[5] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[6] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Archi-*

tectures, and Protocols for Computer Communication, pages 3–14, New York, NY, USA, 1997. ACM Press.

[7] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, November 2005.

[8] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the* 14^{th} *Symposium on Operating System Principles (SOSP)*, December 1993.

[9] Abhishek Dubey, William Emfinger, Aniruddha Gokhale, Gabor Karsai, William Otte, Jeffrey Parsons, Csanad Czabo, Alessandro Coglio, Eric Smith, and Prasanta Bose. A Software Platform for Fractionated Spacecraft. In *Proceedings of the IEEE Aerospace Conference, 2012*, pages 1–20, Big Sky, MT, USA, March 2012. IEEE.

[10] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, pages 53–59, Stanford University, California, USA, August 1996. ACM Press.

[11] D. R. Fatland, M. J. Heavner, E. Hood, and C. Connor. The SEAMONSTER Sensor Web: Lessons and Opportunities after One Year. *AGU Fall Meeting Abstracts*, pages A3+, December 2007.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[13] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt, Andrey Nechypurenko, Jeff Gray, Nanbor Wang, Sandeep Neema, Ted Bapty, and Jeff Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002. ACM.

[14] Jahangir Hasan and T. N. Vijaykumar. Dynamic pipelining: Making IP-lookup Truly Scalable. In *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–216, New York, NY, USA, 2005. ACM Press.

[15] George T. Heineman and Bill T. Councill. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, Reading, Massachusetts, 2001.

[16] Norman C. Hutchinson and Larry L. Peterson. Design of the x-Kernel. In *Proceedings of the SIGCOMM '88 Symposium*, pages 65–75, Stanford, Calif., August 1988.

[17] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO, Vanderbilt University.

[18] John S. Kinnebrew, William R. Otte, Nishanth Shankaran, Gautam Biswas, and Douglas C. Schmidt. Intelligent Resource Management and Dynamic Adaptation in a Distributed Real-time and Embedded Sensor Web System. Technical Report ISIS-08-906, Vanderbilt University, 2008.

[19] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Conference*, pages 259–270, San Diego, CA, January 1993.

[20] Jeffrey C. Mogul, Richard F. Rashid, and Michal J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *Proceedings of the* 11^{th} *Symposium on Operating System Principles (SOSP)*, November 1987.

[21] M. Nelson and J. Ousterhout. Copy-on-Write For Sprite. In *USENIX Summer Conference*, pages 187–201, San Francisco, CA, June 1988. USENIX Association.

[22] Object Management Group. *Lightweight CCM FTF Convenience Document*, ptc/04-06-10 edition, June 2004.

[23] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 2: CORBA Interoperability*, OMG Document formal/2008-01-07 edition, January 2008.

[24] ObjectWeb Consortium. CARDAMOM - An Enterprise Middleware for Building Mission and Safety Critical Applications. `cardamom.objectweb.org`, 2006.

[25] OMG. *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 edition, April 2006.

[26] William Otte, Aniruddha Gokhale, and Douglas Schmidt. Efficient and Deterministic Application Deployment in Component-based, Enterprise Distributed, Real-time, and Embedded Systems. *Elsevier Journal of Information and Software Technology (IST)*, 55(2):475–488, February 2013. <ce:title>Special Section: Component-Based Software Engineering (CBSE), 2011</ce:title>.

[27] William R. Otte, Aniruddha Gokhale, and Douglas C. Schmidt. Predictable Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, CBSE '11, pages 21–30, New York, NY, USA, 2011. ACM.

[28] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions of Computer Systems*, 18(1):37–66, 2000.

[29] Y. Rekhter, B. Davie, E. Rosen, G. Swallow, D. Farinacci, and D. Katz. Tag Switching Architecture Overview. *Proceedings of the IEEE*, 85(12):1973–1983, December 1997.

[30] Sartaj Sahni and Kun Suk Kim. Efficient Construction of Multibit Tries for IP Lookup. *IEEE/ACM Trans. Netw.*, 11(4):650–662, 2003.

[31] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, and Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[32] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, 2000.

[33] M. Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, New York, NY, USA, 1995. ACM Press.

[34] Dipa Suri, Adam Howell, Nishanth Shankaran, John Kinnebrew, Will Otte, Douglas C. Schmidt, and Gautam Biswas. Onboard Processing using the Adaptive Network Architecture. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, June 2006.

[35] George Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices.* Morgan Kaufmann Publishers (Elsevier), San Francisco, CA, 2005.

[36] George Varghese and Tony Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. *IEEE Transactions on Networking*, December 1997.

[37] Jules White, Brian Dougherty, Richard Schantz, Douglas C. Schmidt, Adam Porter, and Angelo Corsaro. R&D Challenges and Solutions for Highly Complex Distributed Systems: a Middleware Perspective. *the Springer Journal of Internet Services and Applications special issue on the Future of Middleware*, 2(3), December 2011.

[38] Jules White, Boris Kolpackov, Balachandran Natarajan, and Douglas C. Schmidt. Reducing Application Code Complexity with Vocabulary-specific XML language Bindings. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, 2005.