Abstract - Networked heterogeneous systems pose a great challenge for parallel and distributed applications. Middleware provides a critical link between the vast resources and the application domain that simplifies development, provides robust and reliable access to resources, helps optimize resource utilization, and facilitates the generation of stable distributed software. In this paper, we discuss a layered framework for distributed systems middleware, called Delmon®, which supports parallel and distributed programming models, tools, and applications on heterogeneous systems. Distributed systems middleware provides multiple layers of abstractions to hide the details of the distributed system and facilitate the efficient utilization of such systems. From a software engineering point-of-view, such a layered middleware approach and the separation of concerns improve the development and management of parallel and distributed applications in many ways. The proposed model represented by Delmon is a general three-layer middleware that provides the services required to support parallel and distributed models and applications in clusters and heterogeneous systems. The layers are: the resource-specific services, the self-organized run-time environment, and the parallel and distributed programming models. We also demonstrate the benefits of this well-defined layered approach that offers different levels of services and functionalities.

Keywords: Distributed systems middleware, software engineering, cluster computing, distributed systems, separation of concerns, software stability.

1 Introduction

Although technological (hardware) advancements have made it possible to construct clusters [6] composed of large numbers of nodes and clusters of clusters to form still larger pools of resources, there is no software that is mature enough to support systems of such magnitude, in terms of management, development, and deployment. In many cases, the vast resources in large clusters and even the grid [13] are not efficiently utilized due to the high costs and complexity of developing efficient tools and applications that can fully utilize them. Recently some efforts are being made to introduce some form of uniformity to the chaotic development efforts. In this paper we offer a general middleware framework that introduces a uniform set of middleware layers separating the various concerns and non-functional requirements involved in developing effective applications for clusters and heterogeneous systems. Such uniform framework provides opportunities for easy development, deployment, reuse, and maintenance of efficient distributed systems and programming models. As a result, the layered approach provides the needed stability of the framework by separating the different concerns and functionalities of each layer form the others.

The distributed systems middleware, Delmon, that we introduce here strives to provide a general framework for middleware services that support the development, deployment and management of distributed and parallel programming models, tools, and applications. This framework separates the different functional and non-functional requirements in the system into three well-defined middleware layers. This approach helps simplify the development process by removing many of the system-related concerns and providing abstracted APIs to handle the common functions needed. In addition, the distributed software constructed using this framework will be easy to maintain and capable of accommodating changes in the different layers without affecting the whole system. This also results in higher software stability and reliability.

1 Delmon is the ancient name of Bahrain, which is an archipelago of 33+ islands in the Arabian Gulf. The reason for using this name is the striking resemblance in the concept. See http://cse.unl.edu/~jaljaroo/delmon_1.htm

© 2003 IEEE
In this paper, we first discuss basic concepts and challenges in distributed computing and its requirements related to software engineering in Section 2. Section 3 identifies the different advances in (systems) technology and in the software development paradigms that resulted in improvement of the distributed software development process and software reliability, reusability, and stability. The proposed multi-layered middleware framework is described in Section 4 and Section 5 concludes the paper.

2 Background and Challenges

Distributed software development involves many components such as distributed and heterogeneous resources, varying operating environments, and diverse models and tools used, among many others. In this section, we discuss some of the important components of the distributed system and the challenges they impose on the distributed software development process and the attributes and quality of the resulting software.

2.1 Challenges in distributed environments

Distributed and parallel application development involves many intricate technical details that developers must attend to in order to build robust and efficient applications. The first issue is handling the distribution among available resources (e.g. clusters, networks of computers, and the grid) in terms of standards used, resources ownership, efficiency of allocation, and risks involved. Another challenging issue is the handling of the heterogeneity of these resources that are of varying architectures, capacities, specifications, interfaces and operating environments. This leads to a major challenge of the development process, which is high complexity. It is not realistic for a developer to consider and efficiently handle all these issues. Generally, application developers are not well acquainted with systems issues and need to concentrate their efforts on the application functionality rather than the underlying system details. Similarly, system developers need not be concerned with the details of applications that may be implemented on these systems to accommodate for their individual needs. Such requirements and complexities will drive up the costs of developing and maintaining the systems and reduce their effectiveness. In addition, incorporating too many specific requirements in the systems will compromise the usability and stability of the systems, thus resulting in even higher costs and increased efforts to maintain and upgrade them. Moreover, there are the issues of performance and reliability, which often conflict with each other. In many cases, developers may sacrifice some reliability to reduce the overhead in favor of high performance. Finally, and to make things even worse, there is no clear-cut techniques or methods that can be used to measure non-functional requirements or pervasive attributes of the distributed applications [5].

2.2 Requirements of parallel and distributed applications

The use of distributed systems such as clusters [6] or the grid [13] implies explicit handling of distribution issues. Many applications require huge resources and tremendous processing power, storage capacity, and network resources. Examples of such applications are web servers [7], network file storage [15], medical image processing and visualization [14], and distributed data mining [28]. When such an application is developed for a single-processor machine such as a personal computer or for a multi-processor shared-memory machine, the contents and changes are directly visible to all the processing units, thereby simplifying the development. However, in a distributed environment, the machines (nodes) are physically disjoint, thus they have no direct way of sharing and exchanging information.

Distributed and parallel applications require the system to support a number of functions so that the applications can be deployed and executed correctly and efficiently. Parallel and distributed programming models and tools provide these functions through their APIs such that the developers need only know the specifics of the model or the tool used. However, these models and tools have many basic common requirements that are directly related to the distributed environment they support, as described below.

1. Simultaneous loading of user programs onto the remote participating machines in the system.
2. Resource allocation and management, and user job scheduling.
4. Job and thread naming and support for multiple jobs and users.
5. Availability of a user command shell to monitor and control the jobs.
6. Providing distributed synchronization mechanisms.
7. Providing dynamic group management and communication mechanisms.

These requirements, as described in details in [3] and [2], can be implemented in different ways to provide the necessary tools and APIs for the programming model developer to build the models and tools. However, each model or application will also have its own set of functionalities that need to be implemented as part of the model itself. Another problem is the varying levels of friendliness and usability that can be offered to developers at the different levels such as the system, tools, programming models, and applications. As a result of all these requirements, the software complexity increases. Thus structured and well defined approaches are needed to simplify development, facilitate maintenance and reuse, and increase reliability and stability.
2.3 Middleware

To manage the complexity of distributed systems and simplify the development process, middleware is used [4], [8], [9]. As defined in [4], "Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system." Using middleware, well-defined APIs are utilized to hide many of the details of the distributed environment, thus relieving the developer from dealing with them. The development process is thus simplified and becomes more structured. Middleware services also provide reusable components that can easily be used in many distributed software systems, thus reducing the development costs and efforts. In addition, the utilization of middleware provides mechanisms to integrate independent applications to function together as components of a larger distributed system [24].

Communication middleware such as Sockets [27], RMI [29], MuniSocket [19] [20], RPC [26] and CORBA [21] provide abstractions to hide the details of the communication process. RMI, RPC and CORBA provide higher-levels of communication abstractions than Socket. Socket provides computer-to-computer communication, RMI provides Java object to remote Java object communications, RPC provides remote procedure call form program to remote program in the same language, and CORBA provides distributed object interactions (object-object communication); however, the objects can be written in different languages. (For example, a C object can call a method in a remote Java Object)

The middleware we describe in this paper is a special type that handles more than the communication details. The proposed distributed systems middleware, called Delmon, represents the basic functions that tie all available computing resources including processors, networks, and storage at one level to with the distributed development tools, models and applications at another. Furthermore, Delmon provides the necessary APIs to transparently access the distributed resources without the need to know the actual details of the resources locations and connectivity. Since Delmon is intended and designed for distributed environments, the middleware itself is also distributed among the different resources, thus providing a single system image for the applications. Furthermore, the Distributed systems middleware can utilize many of the existing communication-oriented middleware such as Sockets, RMI, or CORBA. It involves various objectives and concerns that provide different types of functional and non-functional requirements for the applications utilizing it. For example, requirements such as fault-tolerance, reliability, resource management, scheduling can be incorporated into the middleware to benefit the applications above. In addition, this model allows for easy reuse of the middleware components to build different distributed models and applications, while the original applications remain unchanged.

As defined here, distributed systems middleware is an interdisciplinary field that involves three major fields in computing (See Figure 1): (1) Systems in the sense that it handles different types of system resources and provides its services at that level. (2) Software engineering in the sense that middleware provides a set of clear APIs at the different levels such that the development process becomes simpler and more focused on the application functionality rather than the underlying details. Besides, using software engineering paradigms improves maintainability, reusability, and stability of the distributed software. (3) Networks in the sense that distributed resources need to communicate efficiently to support all other layers in the system.

3 Meeting the challenges

As discussed so far, many challenges face the development of distributed software. Thus, to develop an efficient, reliable, and stable distributed software, it is necessary to address the challenges and provide acceptable solutions that help realize the desired results.

![Figure 1. Interdisciplinary view of the distributed systems middleware (Delmon).](image)

3.1 Applications demand and systems resources

Current technology advances can provide practically endless resources and immeasurable power by pooling existing, but scattered computing resources, such as processing power, storage capabilities, and memory capacities, among other resources, through network connections, wired and/or wireless. Unfortunately, we have no direct way to seamlessly integrate these scattered resources to solve our most challenging problems. On the other hand, the software and application demand for resources continue to increase substantially, which further strains the available resources. Distributed/parallel programming models, tools, advanced communications, and middleware provide the means to utilize all the
resources. However, the efficiency, reliability and reusability of such approaches remain a major challenge.

3.2 Development for reusability

The functional requirements of distributed software are not the only things to consider; non-functional requirements are equally important. The concept of separation of concerns [23][25] serves as a good basis to analyze and account for some of these requirements. Many researchers and software engineers have tried to address these issues by devising suitable techniques to help make the developments process less complex and more efficient [12]. In addition, they are trying to make the resulting software more robust, reliable and stable. Furthermore, research in software stability [10][11] also provides solid techniques to address some issues in software development. As described in [11], software deteriorates over time mainly due to changes in clientele, technology or information needs. Although many changes should only affect small parts of the software, in most cases the whole software is reengineered to accommodate these changes. This results in fast deterioration of the software and thus its instability. The research in Enduring Business Themes, Business Objects and Industrial Objects provide a model to approach this problem and try to produce more stable software.

3.3 The layered middleware approach

Many techniques have been proposed to achieve software stability, reliability, and reusability. Distributed systems middleware provides a well structured approach to address the issues of increased complexity and multiple other concerns when developing distributed software. However, such middleware can end up being as complex as any distributed system, unless some requirements and services, including the following, are clearly defined and incorporated in the design process:

1. Functional requirements for the different programming models, development tools and applications that will rely on this middleware.
2. The general requirements needed to support the basic operations involved in a distributed system such as inter-process communication, process/thread remote deployment and access, synchronization, etc.
3. Resource access services that directly relate to the resources available. Some examples of these resources are specialized communication networks, special purpose resources, and non-standard types of components, in addition to the standard resources.

Other concerns may also arise as the distributed systems evolve and change. All this increases the complexity of the middleware. Hence, to accommodate these requirements and concerns, it is necessary to approach the middleware in the same way distributed applications are treated by separating its components and concerns and placing them into multiple layers such that each layer will provide specific functions and satisfy a well-defined set of requirements. The next section proposes a general framework of the layered middleware architecture that attempts to meet many of the challenges and provide the different requirements in each layer.

4 The distributed Systems middleware Layers

In this section, we propose a framework for the layered distributed systems middleware. This architecture is based on our thorough study of some distributed parallel programming models, tools, and distributed applications requirements [3] and it provides seamless access to heterogeneous resources. The middleware consists of three layers (see Figure 2) that separate the parallel and distributed applications and the networked heterogeneous resources. Based on a thorough analysis of the distributed and parallel programming models, the main components of the middleware were identified. In addition, by separating the requirements depending on their functionality and dependencies in the hierarchy, we were able to accommodate for future changes in each layer without having to change all the layers or applications in the system. The top layer represents the distributed and parallel applications that built using the middleware services. Some examples are high performance parallel applications, distributed server-based applications such as I-MINDS (an intelligent-agent-based distributed educational system) [17], clustered web servers, and large-scale scientific and engineering simulations. These layers provide these applications with many common and general functions and non-functional attributes.

At the other end, below the middleware is the collection of distributed resources including the different types of machines such as clusters, servers, multi-processor machines, networks, the storage devices, etc. All these resources may have similar or different operating environments (e.g., Unix-based, Linux, and MS Windows operating systems) and may have different network topologies and connections from local area networks to the Internet. Furthermore, the communication protocols, authorization/authentication mechanisms and requirements, access control policies will also vary from one set of resources to another.
The parallel and distributed programming models include the message-passing [16] and the newly introduced object-passing [18] models; the distributed shared memory [22] and shared object models; the parallel/multithreaded applications; and the transparent or seamless parallelization, in addition to various distributed applications and tools. Any of these models and tools provides distributed and parallel application developers with some mechanisms to write their applications. However, each model has its advantages and disadvantages in terms of the development effort, the explicit/implicit distribution/parallelization requirements, and the execution efficiency of the resulting applications [3]. In addition, each of these models provides a specific set of APIs that developers can use without worrying about the underlying technical details. As an example, JOPI [18] (Java Object-Passing Interface) APIs provide methods to send, receive, and broadcast objects among processes. It provides synchronization mechanisms to control the flow of the parallel execution. However, the model is not responsible for issues such as remote thread deployment, job scheduling, and resource management, which are handled by the next layer.

4.2 The run-time support

The run-time support layer provides common run-time functionalities needed by the models in the first layer. Although programming models vary in their approaches and APIs, they all need common services essential for their operations. Such functions, as discussed in Section 2.2, include but not limited to: remote class/process deployment and execution, communication primitives, resource management tools, and job scheduling and control tools. To illustrate the operations and functions this layer provides, we will briefly discuss our prototype implementations reported in [1] and [2], where self-organized distributed agents are used to handle the details of communicating among and managing the resources available. For example, when a parallel application is built using JOPI, the developer need not be concerned with issues such as how the processes will be scheduled, where will they execute, and how they will be moved to the remote machines. JOPI is implemented using the APIs of the run-time layer such that it could easily utilize the functions at this layer to satisfy the applications' requirements. Thus, effectively, the run-time environment transparently handles these details for the programming model. Similarly, other models and tools can take advantage of the APIs to implement their functions.

4.3 The resource specific layer

The resource specific layer provides enhanced services for the other layers to improve the execution environment and facilitate better performance. Within this layer, specialized protocols, middleware services, and mechanisms specific to some types of resources are implemented and made available through a unified set of APIs to other layers. One example of such API in this layer is MuniSocket [19] [20], which provides high bandwidth transmission between machines with multiple network interfaces. MuniSocket provides its services for any machine that has more than one network interface installed and connected. This service is integrated into this layer such that communications initiated at the top layers can directly take advantage of it. Other examples include distributed storage modules, enhanced I/O, and device access and control functions. According to this architecture, when a new technique or function is added to this layer, it will be available to the other layers without having to modify or reengineer them to accommodate these changes.

4.4 The advantages and drawbacks

Our distributed system middleware enjoys the advantages of middleware such as providing clear and well-defined APIs to access the various resources and avoid the need to handle too many of the underlying systems details. In addition, our middleware simplifies the development process by separating the application's functional requirements from the systems and distribution requirements that can be provided by the middleware. Furthermore, the existence of the middleware layers reduces the need to change the applications to accommodate changes that occur in the system. Using multiple layers within the middleware framework provides a number of distinguishing characteristics:

1. A better understanding of the different services that can be made available for each layer and how these services will interact with services in the other layers.
2. An organized framework of layered services that can be efficiently utilized by applications to achieve high performance over a heterogeneous environment.

3. A well defined separation between the parallel and distributed programming models and tools and the underlying run-time support services, which allows for easy adaptation to changes in the models or the run-time environment.

These characteristics give rise to even more benefits to the system:

1. *Easier and faster development process* for services or programming models in any of the layers with the well-defined APIs of that layer. For example, with a well-defined parallel programming model and APIs, the developer will be able to easily write parallel applications using the model's APIs without having to spend time and effort on the common functions provided by the lower layers.

2. *Extensible infrastructure* that can expand to support increasing number of resources and applications, while preserving the defined APIs available to the other layers. Since each layer is responsible for a specific set of functions, any changes or expansions in the system can be easily handled in the concerned layer only. For example, adding more hardware such as processors, network interfaces, or storage devices is handled exclusively at the resource specific layer. Another example is when additional functions are needed at the run-time support layer, say by adding an advanced module for scheduling or resource discovery, the added modules will directly utilize the APIs of the lower layer as necessary, while providing new services to the upper layer through the original APIs.

3. *Reusability of the systems components is increased.* With the multiple layers, new programming models, distributed applications and tools can be built using existing modules of the system instead of building it fully from scratch. For example, a new distributed shared object model can be implemented using the APIs from the run-time layer in addition to some of the APIs available in JOPI, which may be more abstract than those in the run-time layer. The possible mix and match in using the APIs and the functions at the different layers provide a flexible source of components and functions for the developer to use, instead of creating all the functions again.

4. *Stability of the resulting distributed and parallel applications is increased.* When systems are built completely from scratch and do not utilize a layered approach, changes at any level will necessitate the reengineering of the whole application to accommodate for these changes. Using the layered approach, concerns are separated into the different layers and dependencies among them are reduced to the level of well-defined APIs between the layers. In this case, when changes are needed, they are limited to a single layer and would not have a direct impact on any of the other layers as long as the APIs are carefully maintained. By avoiding frequent changes to the system and applications, we increase their stability, thus extending their lifespan.

Although it is very clear how advantageous the multi-layered middleware is, some drawbacks must be recognized and addressed to determine the appropriateness of the multiple layers, or the right number of such layers if determined appropriate. The main drawback stems from the overhead imposed by each layer. It is well understood that the more layers placed between the application and the resources, the higher penalty they incur in terms of response time and overall performance. However, since much of the overhead is inevitable due to the distributed nature of the resources, some overhead may be worth the benefit gained from the layered approach. One simple example is the Java virtual machine (JVM), which imposes some overhead on the application built to run on it due to the interpretive nature of execution. However, because of this same feature, Java is machine independent, thus, it solves the problems of platform heterogeneity. In a more complex situation, many more factors need to be considered to answer the questions of, for example, “what type of applications will be using the middleware?” “what are the most important attributes (e.g., performance, reliability, stability, fault-tolerance, and scalability) of these applications?” and “how often will the applications change or require enhancements in the resources?” On the other hand, answers to these questions will determine a suitable approach to the middleware design.

The distributed systems middleware described here provides many advantages that in most cases outweigh the overhead imposed by the structure. In addition, the implementation is optimized to avoid as much overhead and unnecessary delays as possible. Furthermore, the APIs provided allow distributed application developers to bypass one or more layers for the sake of high performance. However, when a distributed application is implemented with reliability as its main concern, then it will be more appropriate to tolerate the additional overhead imposed by layers for the sake of reliability.

5 Conclusion

In this paper, we defined distributed systems middleware as a three-layer middleware that ties the heterogeneous resources at one end to the parallel and distributed applications at the other. It hides the details of each layer from the layers above it and provides well-defined APIs to access each layer. The three layers and their functionality were determined based on:
1. A thorough investigation of the current programming models for parallel and distributed applications to identify their common characteristics.
2. Identification of the common requirements that must be available for such models to operate efficiently.
3. Experimental evaluations and observations of the behavior and performance of the prototype implementations of Delmon, including JOPI and the run-time layer, and MuniSocket, along with other supporting functions and components.

Based on the experimental evaluations and the software analysis from the software engineering point-of-view, it was found that the proposed distributed systems middleware framework provides many advantages such as simplifying the distributed and parallel applications development, simplifying testing and maintenance of the system, facilitating reusability and extensibility, and increasing reliability and stability.

Our current implementation and experimentations have provided us with many insights into how this approach can be enhanced and optimized. In addition, many of the concepts introduced here apply to many other areas of research such as peer-to-peer computing, mobile computing, and autonomous systems. Therefore, we plan to start first by investigating the opportunities of enhancements such as optimizing the different functions of each layer, providing standard API definitions in each layer to simplify maintenance, reuse and extensibility, and providing more specialized functions at each layer to support more requirements as necessary. In addition, at a later stage, we plan to start investigating the applicability of this model for autonomous systems.

Acknowledgement

This project was partially supported by a National Science Foundation grant (EPS-0091900) and a UNL academic priority grant (PRISM). We would also like to thank the members of the secure distributed information (SDI) group and the research computing facility (RCF) at the University of Nebraska-Lincoln for their support.

References


