JavaSymphony, a Programming Model for the Grid*

Alexandru Jugravu[1] and Thomas Fahringer[2]

[1] University of Vienna, Institute for Software Science, Liechtensteinstr. 22, A-1090 Wien, Austria
[2] University of Innsbruck, Institute for Software Science, Technikerstr. 25/7, A-6020 Innsbruck, Austria

Abstract. In previous work, JavaSymphony has been introduced as a high level programming model for performance-oriented distributed and parallel Java programs. We have extended JavaSymphony to simplify the development of Grid applications written in Java, allowing the programmer to control parallelism, load balancing, and locality at a high level of abstraction. In this paper, we introduce new features to support the widely popular workflow paradigm. The overall idea is to provide a high level programming paradigm that shields the programmer from low level implementation details such as RMI, sockets, and threads. Experiments will be shown to demonstrate the usefulness of JavaSymphony as a programming paradigm for the Grid.

1 Introduction

Numerous research projects have introduced class libraries or language extensions for Java to enable parallel and distributed high-level programming. However, most approaches tend towards automatic management of locality, parallelism and load balancing which is almost entirely under the control of the runtime system. Automatic load balancing and data migration can easily lead to performance degradation. JavaSymphony, on the other hand, is a programming paradigm for wide classes of heterogeneous systems that allows the programmer to control locality, parallelism and load balancing at a high level of abstraction.

In previous work, we describe the JavaSymphony programming paradigm[1], implementation[2] and we evaluate the usefulness of our system for several parallel and distributed applications[3]. In this paper we describe important JavaSymphony mechanisms for controlling parallelism, load balancing, and locality, which are crucial for the Grid. Moreover, we introduce new features to simplify the development of Grid workflow applications at a high level, including a user interface to build the graphical representation of a workflow process, an XML-based description language for workflow applications, library support for the definition of

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 18–25, 2004.

the workflow and its components, and a specialized scheduler for workflow applications. We describe the components of our workflow model, which includes activities, links, conditional branches, and loops.

We also demonstrate these features for an existing JavaSymphony application, which is suitable to be modelled by using a workflow.

The rest of this paper is organized as follows. The next section discusses related work. In Section 3 we briefly introduce the main features of the JavaSymphony programming paradigm, showing also how they are related to Grid computing. Section 4 presents the new framework for developing JavaSymphony workflow applications, and illustrates an experiment to translates an existing application into a workflow application using our tool. Finally, some concluding remarks are made and future work is outlined in Section 5.

2 Related Work

Java's platform independent bytecode can be executed securely on many platforms, making Java an attractive basis for portable Grid computing. Java offers language elements to express thread based parallelism and synchronization.

Therefore, its potential for developing parallel and distributed applications was noticed long before the Grid computing turn out to be relevant for computer scientists. Many projects tried to use the huge potential of the Internet by supplying Java-based systems for web-computing (e.g. Bayanihan[4], Javelin[5], JaWS[6]). Such systems are commonly based on volunteer computing and/or on a three-tier architecture with hosts, brokers and clients.

Other systems are oriented towards cluster computing (e.g JavaParty[7], ProActive[8]) and offer high-level APIs for distributing Java objects over a computing resources of a cluster of workstations or a dedicated SMP cluster.

The Grid community cannot ignore the advantages of using Java for Grid computing: portability, easy deployment of Java's bytecode, component architecture provided through JavaBeans, a wide variety of class libraries that include additional functionality such as secure socket communication or complex message passing, etc. Moreover, Java Commodity Grid (CoG) Kit[9] allows access to the services provided by the Globus toolkit(http://www.globus.org).

3 JavaSymphony Preliminaries

JavaSymphony is a 100% Java library that enables the programmer to specify and control locality, parallelism, and load balancing at a high level of abstraction, without dealing with error-prone and low-level details (e.g. create and handle remote proxies for Java/RMI or socket communication). The developers of distributed application can use the JavaSymphony API to write distributed Java applications. A typical JavaSymphony application registers with the JavaSymphony Runtime System (JRS), allocates resources (machines where JRS is active), distributes code as Java objects among the resources, and invokes methods

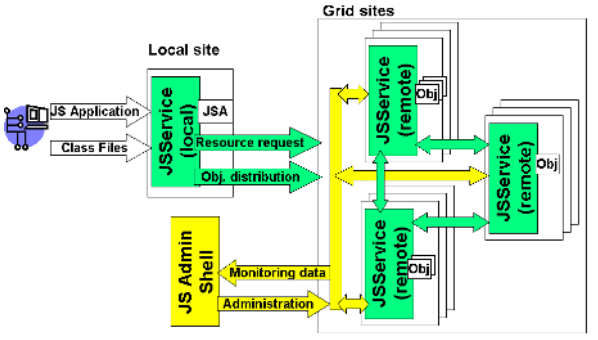


Fig. 1. JavaSymphony on the Grid

of these objects. The objects may also communicate with each other through remote method invocation. Finally the application un-registers and is removed from the JRS's list of active applications.

Dynamic Virtual Distributed Architectures (called VAs) allow the programmer to define the structure and the characteristics of a heterogeneous network of computing resources (e.g. CPU type, speed, or machine configuration), and to control mapping, load balancing, migration of objects and code placement. JavaSymphony also offers possibilities to specify, control and query the properties of the computing resources. We call these properties *constraints*, which can be static (e.g. operating system name, available memory/disc space, java version, etc.) or dynamic (e.g. free memory, load, free disk space, network bandwidth and latency, etc.)

JavaSymphony remote objects (JS objects) are used to distribute objects onto virtual architectures. Under JavaSymphony one can distribute any type of Java objects, by encapsulated them in JS Objects. The programmer is not forced to implement specific interfaces and to encode special methods. Thus any Java object can be used in parallel applications without adding a single line of code. The mapping of the JS objects can be explicit, on specific level-1 VAs (single resources), or controlled by constraints associated with the resources. JavaSymphony applications use the distributed objects by remotely invoking their methods.

Other programming features offered by JavaSymphony include a variety of remote method invocation types (synchronous, asynchronous and one-sided); (un)lock mechanism for VAs and JS objects; high level API to access a variety of static or dynamic system parameters, e.g. machine name, user name, JVM version or CPU load, idle time, available memory; selective remote class-loading; automatic and user-controlled mapping of objects; conversion from Java conventional objects to JS objects for remote access; single-threaded versus multi-threaded JS objects; object migration, automatic or user-controlled; distributed event mechanism; synchronization mechanisms and persistent objects. More details about the JavaSymphony programming constructs can be found in [1,3].

The JavaSymphony Runtime System (JRS) is implemented as an agent based system, with agents running on each computing resource to be used by JavaSymphony applications (see [1,2]). We intend to implement the JS agents as Grid services, running on various grid resources (see Figure 1). The JS applications correspond to Grid distributed applications. The resources are requested using VAs defined in the JS application. JS constraints provide support for managing, querying and monitoring the parameters of the resources. The JS objects play the role of Grid submitted Java jobs. In addition, the distributed components of an application may freely communicate each with other, a feature which is currently not supported or hard to implement for typical Grid applications. Other features like migration, events, synchronization may be extremely useful for collaborative Grid applications.

4 Grid Workflow Applications under JavaSymphony

We define a Grid Workflow application as a set of one or more linked activities, which collectively realize a common goal. Information (files, messages, parameters, etc.) is passed from one participant to another for action, according to a set of procedural rules and the whole process is using a Grid infrastructure.

JavaSymphony programming paradigm is flexible enough to allow the implementation of a large range of distributed applications, including workflow applications. However, the developer usually has to manage the resources, build Java objects, and control the mapping of these objects onto resources. For better performance, the developer also must incorporate a scheduling strategy adapted to his particular applications. All these issues require a significant programming effort.

Many distributed applications follow a well-defined pattern, and therefore many of the above-mentioned programming issues could be automated. We are particularly interested in automatic resource allocation and scheduling. In recent times, workflow applications became quite popular in Grid community and many research and industry groups proposed standards to model and develop workflow applications and built workflow definition languages or schedulers for workflow applications[10,11,12,13]. On the other hand the workflow applications may support automatic resource discovery, allocation and scheduling.

Motivated by these aspects, we have considered to support the developing of workflow application in JavaSymphony. This support consists of:

A graphical user interface for building the structure of a workflow, specify constraints associated with the resources or characteristics of the components (called activities) of the workflow.

A specification language for describing the workflow and its components. We use an XML based language and we intend to keep it simple. A JavaSymphony workflow application will be automatically generated using the description of the workflow and the code for its activities.

Library support for workflow applications consists of a set of classes used to standardize the workflows and their activities in JavaSymphony.

A specialized scheduler which automatically finds resources (as VAs), maps components onto resources and runs the distributed computation, according to the rules defined for the workflow.

We have fully implemented the specification language and the GUI. Our GUI uses the same graphical engine as Teuta[14]. We are currently working on the library support and the scheduler for JavaSymphony workflow applications.

4.1 Workflow Model

Our tool is used to build the graphical representation of a workflow as a graph. We have followed the terminology and specifications proposed by the Workflow Management Coalition[15] and used several types of basic workflow elements:

Activities. The activities perform the computational parts of a workflow. They are represented as vertices of the associated graph. Each activity will be associated with a Java class that extends the *JSActivity* abstract class provided by JavaSymphony library. Instances of the associated classes will be placed onto computing resources where they will perform specific computation.

Dummy Activities. The dummy activities represent a special type of activities. They are supposed to perform evaluation of complex conditional expressions that may influence the scheduling of the workflow, but they require minimal computing power. They will not be placed onto distributed computing resources, but will run locally within the scheduler.

Links. Between the vertices of the associated graph, there may be directed edges, which represent the dependency relation (dataflow or control flow dependency) between the activities. Sending data from one activity to another implies that there is also a control flow dependency between the two. Therefore we may consider that all links are associated with dataflow dependencies.

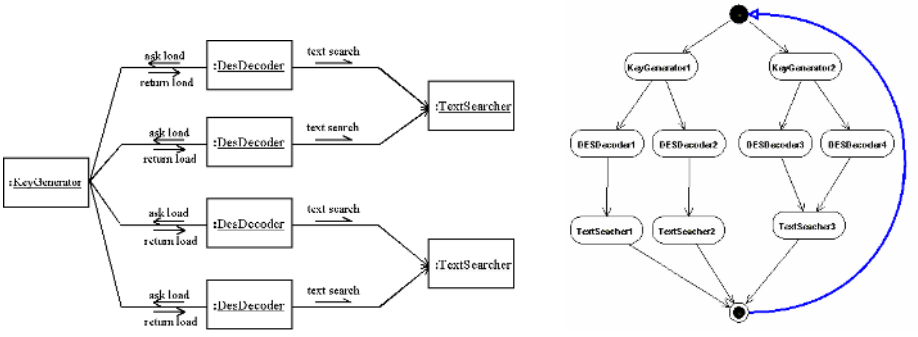
Initial and Final States. Each workflow has an entry and an exit point, which we call **initial state**, respectively **final state**. These states are not associated with computation. They are also used to mark subworkflows of a workflow. Each subworkflow has a unique initial state and a unique final state.

Conditional branches. The execution plan of a workflow is dynamically changed using conditional branches. In our model, a conditional branch has one entry (link from an activity to the conditional branch) and two or more exits. The successors of the conditional branch correspond to entry points of subworkflows. When the execution reaches the conditional branch, a single successor will be chosen and the rest of them will be omitted in the execution.

Loops. For consistency reasons, the loops in JavaSymphony may be associated only with entire (sub)workflows units, with a single entry (initial state) and a single exit (final state) point. For a (sub)workflow which has a loop associated with it, the entire sequence of activities is executed repeatedly a fixed number of times (for-loops) or until an associated condition is satisfied (until-loops).

4.2 DES Encryption/Decryption Application

In the following, we show how we have used our tool to transform an existing JavaSymphony application into a workflow application. This application



(a) Original application schema (b) DES workflow

Fig. 2. JavaSymphony DES decoding algorithm

implements DES encryption/decryption algorithm[16]. In previous work[1], we have studied the performance of this application in comparison with two other Java-based programming paradigms for concurrent and distributed applications: JavaParty[7] and ProActive[8].

DES encryption/decryption algorithm[16] uses a key of 56 bits, which is extended with another 8 parity bits. DES tries to detect the key that has been used to encrypt a message using DES, based on a "brute-force" approach (every possible key is tested). The assumption is that we know a string that must appear in the encrypted message.

Figure 2(a) shows our original design of the DES decoding algorithm. The *DesDecoder* objects process a space of possible keys, which are provided by one or several *KeyGenerator* object(s). A *DesDecoder* acquires the keys from the *KeyGenerator* through a synchronous method invocation. The *KeyGenerator* keeps track of the keys that have already been generated. After the *DesDecoders* have decoded a message by using their asigned keys, a one-sided method invocation is used to transfer the decoded messages to a *TextSearcher* object, which validates it by searching the known string in the messages.

The most important component of the application is the *DESDecoder* class. One *DESDecoder* plays an active role: it requests new data from a *KeyGenerator* as a *DESJob*, applies the decryption algorithm and sends (buffers of) processed data further to a *TextSearcher*, which (in)validates it.

4.3 DES Encryption/Decryption as a Workflow Application

The DES JavaSymphony application described above is not implemented as a workflow application. There are three types of functional components, which we used, but the data exchanged does not follow a "flow", as for workflow applications. There is a bi-directional communication between *KeyGenerator* and *DESDecoder(s)* (i.e. request and response), and respectively unidirectional be-

tween *DESDecoder(s)* and *TextSearcher(s)* (send data). However we can modify the functional components to build a workflow application.

The activities of the workflow are associated with the three types of objects: *KeyGenerator*, *DESDecoder* and *TextSearcher*. There is workflow precedence between *KeyGenerator(s)* and *DESDecoders*, respectively between *DESDecoders* and *TextSearchers* (see Figure 2(b)). Dataflow precedence overlaps the workflow precedence: There is unidirectional communication from *KeyGenerator* to *DESDecoders* and from *DESDecoders* to *TextSearchers*. The implementation for the three classes has to be changed accordingly.

The graphical representation of the workflow is shown in Figure 2(b) and has been built by using our tool. In our example, there are 2 *KeyGenerator* activities, 4 *DESDecoder* activities, and 3 *TextSearcher* activities represented as vertices. The arrows between vertices represent the links of the workflow (data and control flow). There is no need for conditional branches or dummy activities. The process is repetitive and therefore a loop is associated with the whole workflow.

In addition, the developer has to implement the classes for the activities such that they implement a specific JavaSymphony interface for activities. Using the GUI, the developer associates the activities with these classes. He may also specify input parameters, resource constraints (for the VA generation), and activity characteristics relevant for the scheduling (e.g. estimated computation load, minimum/average/maximum execution time for the activity, priority etc.). For the data links between activities, one may specify link constraints (e.g. bandwidth or latency constraints) or communication characteristics (e.g. estimated communication load), which are relevant for the scheduling.

The GUI produces a file, which describes the workflow (structure, constraints, characteristics of the workflow elements, etc.) in a XML-based workflow specification language . The scheduler will use the workflow description file to map activities of the workflow onto computing resources, to enact these activities and to control communication between them. The scheduler is currently under development.

5 Conclusions and Future Work

JavaSymphony is a system designed to simplify the development of parallel and distributed Java applications that use heterogeneous computing resources ranging from small-scale cluster computing to large scale Grid computing.

In contrast with most existing work, JavaSymphony allows the programmer to explicitly control locality of data, parallelism, and load balancing based on dynamic virtual distributed architectures (VAs). These architectures impose a virtual hierarchy on a distributed system of physical computing nodes.

JavaSymphony is implemented as a collection of Java classes and runs on any standard compliant Java virtual machine. No modifications to the Java language are made and no pre-processors or special compilers are required.

In this paper we have presented a framework for developing Grid workflow applications in JavaSymphony. We consider that JavaSymphony offers a very

suitable programming paradigm for developing Java-based Grid applications. We plan to further investigate the applicability of JavaSymphony in the context of Grid computing and implement an automatic scheduler for JavaSymphony workflow applications.

References

1. Jugravu, A., Fahringer, T.: Javasymphony: A new programming paradigm to control and to synchronize locality,parallelism, and load balancing for parallel and distributed computing. to appear in Concurency and Computation, Practice and Experience (2003)
2. Jugravu, A., Fahringer, T.: On the implementation of JavaSymphony. In: HIPS 2003, Nice, France, IEEE (2003)
3. Fahringer, T., Jugravu, A., Martino, B.D., Venticinque, S., Moritsch, H.: On the Evaluation of JavaSymphony for Cluster Applications. In: IEEE International Conference on Cluster Computing (Cluster2002), Chicago USA (2002)
4. Sarmenta, L.F.G., Hirano, S., Ward, S.A.: Towards Bayanihan: Building an extensible framework for volunteer computing using Java. In ACM, ed.: ACM 1998 Workshop on Java for High-Performance Network Computing, New York, NY, USA, ACM Press (1998)
5. Alan, M.N.: Javelin 2.0: Java-based parallel computing on the internet (2001)
6. Lalis, S., Karipidis, A.: Jaws: An open market-based framework for distributed computing over the internet (2000)
7. Haumacher, B., Moschny, T., Reuter, J., Tichy, W.F.: Transparent distributed threads for java. In: 5th International Workshop on Java for Parallel and Distributed Computing ,Nice, France, April 22-26, IEEE Computer Society (2003)
8. Baude, F., Caromel, D., Morel, M.: From distributed objects to hierarchical grid components. In: International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November, Springer Verlag, Lecture Notes in Computer Science, LNCS (2003)
9. Laszewski, G., Foster, I., Gawor, J., Lane, P.: A java commodity grid kit. Concurrency and Computation: Practice and Experience **13** (2001)
10. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Systems, S., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services (bpel4ws). Specification version 1.1, Microsoft, BEA, and IBM (2003)
11. Erwin, D.W., Snelling, D.F.: UNICORE: A Grid computing environment. Lecture Notes in Computer Science **2150** (2001)
12. The Condor Team: (Dagman (directed acyclic graph manager)) http://www.cs.wisc.edu/condor/dagman/.
13. Krishnan, S., Wagstrom, P., von Laszewski, G.: GSFL : A Workflow Framework for Grid Services. Technical Report, The Globus Project (2002)
14. Fahringer, T., Pllana, S., Testori, J.: Teuta: Tool Support for Performance Modeling of Distributed and Parallel Applications. In: International Conference on Computational Science. Tools for Program Development and Analysis in Computational Science., Krakow, Poland, Springer-Verlag (2004)
15. WfMC: Workflow Management Coalition: http://www.wfmc.org/ (2003)
16. Wiener, M.J.: Efficient DES key search, technical report TR-244, Carleton University. In: William Stallings, Practical Cryptography for Data Internetworks. IEEE Computer Society Press (1996)